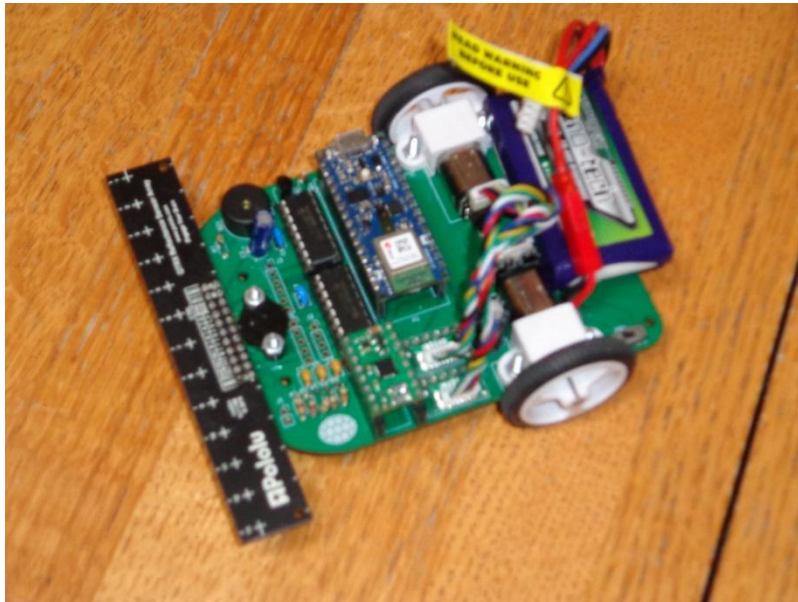


Final Report for  
**ENEE408I: Team 1 (2021)**



Submitted to:

Prof. Gilmer Blankenship

ENEE 408I: Capstone Design Project: Autonomous Control of Interacting Robots  
Fall 2021

Department of Electrical & Computer Engineering  
2410 A.V. Williams Building  
University of Maryland  
College Park, MD 20742

Date: 12/21/2021

Prepared by:

Edward Aaron Bondy (ebondy1@terpmail.umd.edu)

Zachary Breit (zbreit@umd.edu)

Onyemachi Azubuko (oazubuko@umd.edu)



## TABLE OF CONTENTS

<b>Introduction</b>	<b>4</b>
<b>Challenge Overview</b>	<b>4</b>
<b>Proposed Solutions &amp; Teamwork Distribution</b>	<b>5</b>
<b>Machi's Contributions</b>	<b>5</b>
<b>Edward's Contributions</b>	<b>6</b>
<b>Zach's Contributions</b>	<b>6</b>
<b>Approach</b>	<b>7</b>
<b>Hardware Overview</b>	<b>7</b>
<b>Software Overview</b>	<b>8</b>
<b>Procedure</b>	<b>13</b>
<b>Hardware Testing Progress</b>	<b>13</b>
<b>Software Testing Progress – Control &amp; Navigation</b>	<b>13</b>
<b>Basic Motor Control</b>	<b>13</b>
<b>Line Following</b>	<b>16</b>
<b>Junction Identification</b>	<b>17</b>
<b>Turning</b>	<b>18</b>
<b>Maze Traversal</b>	<b>19</b>
<b>Maze Mapping</b>	<b>20</b>
<b>Software Testing Progress – Communication</b>	<b>22</b>
<b>Software Testing Progress – Camera Detection</b>	<b>23</b>
<b>Software Testing Progress – Operating Systems/Source Code</b>	<b>24</b>
<b>Execution/Demo Result</b>	<b>25</b>

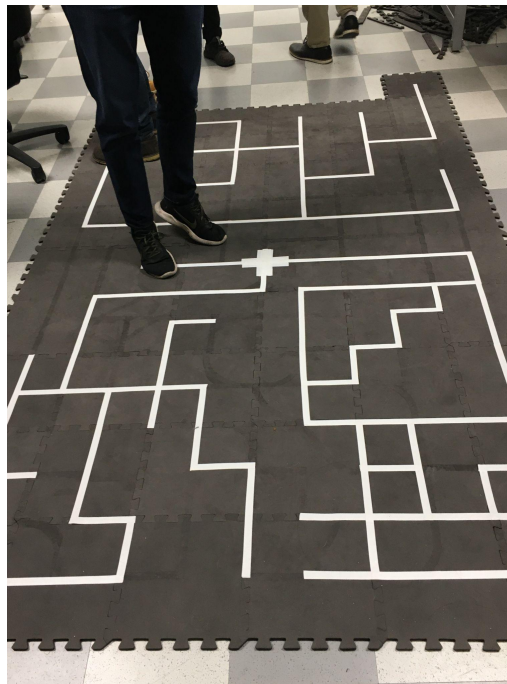


<b>Summary &amp; Future Suggestions</b>	<b>26</b>
<b>Machi</b>	<b>26</b>
<b>Edward</b>	<b>26</b>
<b>Zach</b>	<b>27</b>
<b>Feedback</b>	<b>30</b>
<b>Machi</b>	<b>30</b>
<b>Edward</b>	<b>30</b>
<b>Zach</b>	<b>30</b>
<b>References</b>	<b>32</b>
<b>List of Abbreviations</b>	<b>33</b>

## Introduction

### Challenge Overview

The primary objective of this course was to learn about robotics, engineering, and teamwork by developing a team of three robots that could traverse and collaboratively solve a maze. Each robotic “MicroMouse” was equipped with some basic hardware, such as a line sensor, camera module, and a pair of motors. Additionally, each team had access to three powerful GPU-based computers (NVIDIA Jetson Xavier NXs) that could perform wireless communication and image processing. By researching, analyzing, and breaking down a new problem, our team hoped to learn about many fundamental aspects of robotic design such as control systems, communication protocols, and robotic coordination.



*The maze that our robots solved on demo day (12/8/21).*

On top of simple maze solving, our team also challenged ourselves to develop robots that could run a relay race towards the maze exit. We would start by positioning the three MicroMice at arbitrary points in the same maze. The first mouse would travel until it reached the second mouse. At that point, the first mouse would stop and the second mouse would begin moving. Similarly, when the second mouse reaches the third mouse, the second mouse would stop and the third mouse would begin moving. The third mouse would then find the exit and then leave the maze. Unfortunately, we ran out of time this semester to finish our specialized challenge. However, our team is still proud that we were able to solve the first challenge and create a team of robots that could collaboratively solve any maze.



## **Proposed Solutions & Teamwork Distribution**

The first step towards a solution was understanding the major problems we had to overcome. In the first couple of weeks, initial testing made it clear that we would need to implement a PID controller to balance the left and right motors to keep the MicroMice driving straight, enabling them to follow lines of tape. The next challenge on the path to maze traversal was detecting and identifying junctions. Once a junction was found and classified, the robot would need to turn to a precise angle to continue traveling through the maze, which also required a PID controller.

To implement collaborative maze-solving, our team decided on a leader/follower architecture. The “leader” MicroMouse would traverse the maze first using a right-hand wall-following strategy until it identified the end of the maze. The “follower” MicroMice would then travel along a simplified version of the “leader” mouse’s path through the maze by eliminating any unnecessary turns (e.g., whenever the robot had to backtrack after reaching a dead-end).

Initially, the team decided to route all robot communications through a single Jetson computer. The single greatest problem that appeared during the initial testing of the communications software was connecting one Jetson to all three MicroMice simultaneously, and then remaining connected. To overcome this obstacle, a client/server architecture was developed, in which each MicroMouse was connected to its own client process. In the background, a server process facilitated the transfer of data between the three client processes.

Our team decided to implement direction optimization on the Jetson to avoid burdening the MicroMice with this performance-intensive task. Accordingly, we had to develop a subprocess for the Jetson that simplified the directions that it received from a “leader” MicroMouse client process. This strategy had two benefits: it slightly reduced the amount of data that needed to be exchanged between the client/server processes and it allowed the “follower” MicroMice to travel along more optimized routes.

In order for the “leader” MicroMouse to escape from mazes with loops, we needed to implement loop detection. This capability required accurate maze mapping as a prerequisite so that our robots could determine if they have reached the same node multiple times. After maze mapping and loop detection capabilities were implemented, we could implement a mechanism for loop escape.

With all of these design challenges in mind, our team distributed the work for this project in the following way:

### **Machi’s Contributions**

- Implemented maze mapping
- Added loop detection and escape (using maze mapping data)
- Designed an algorithm for maze traversal



### **Edward's Contributions**

- Built the entirety of the communications protocol. This includes both the robot side and the Jetson side
- Wrote all the Jetson-side Python programming, including the direction optimizer, as well as the startup and shutdown scripts
- Built on the code for blind maze traversal to create the optimized maze traversal using simplified directions based on results of a previous maze run
- Worked with Zach and Machi to integrate the communications software with the control and navigation software
- Built small-scale test maze

### **Zach's Contributions**

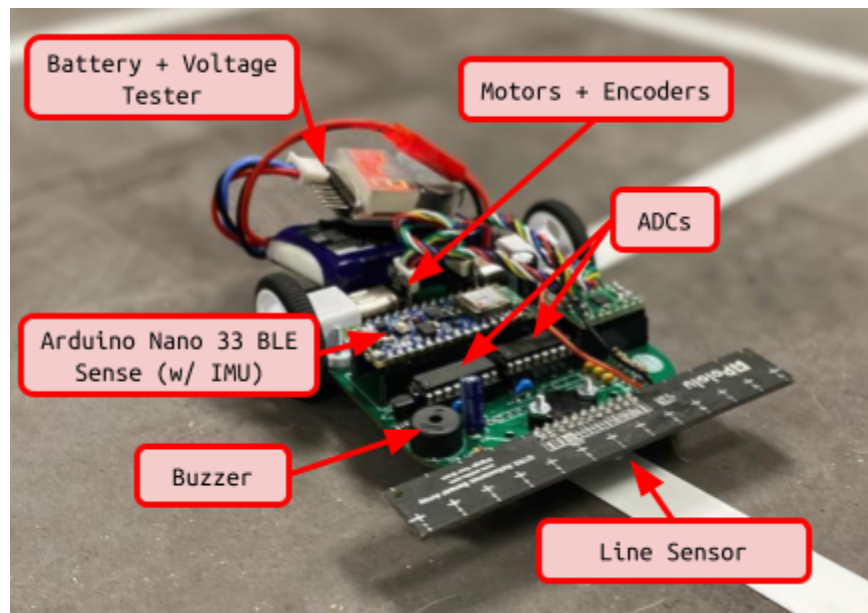
- Programmed the logic for blind maze solving (right-hand wall-following)
- Wrote the velocity controller for the robot's motors
- Tuned PID controllers for reliable turning and straight driving (both with and without tape)
- Designed easy-to-use C++ classes for interfacing with sensors/actuators such as the reflectance array, motors, and the gyroscope
- Created a position estimator to give a decent estimate for the robot's current position and heading in the maze by adding displacement vectors over small time increments
- Converted the code to a non-blocking architecture (super loop) with a Finite-State Machine (FSM)
- Wrote a computer vision pipeline to classify the different types of junctions in a given image
- Helped Edward and Machi integrate communication, maze-mapping, and loop detection into the robot code
- Designed functionality for junction and end-of-maze detection

## Approach

### Hardware Overview

The brain of each MicroMouse is the **Arduino Nano 33 BLE (Arduino)**, which is a microcontroller that interacts with every sensor and actuator on the robot. The Arduino has an onboard **Inertial Measurement Unit (IMU)** with a three-axis accelerometer, gyroscope, and magnetometer, which measure acceleration, rotation, and magnetic fields respectively. Our group only used the IMU for the gyroscope, which we needed to measure angles for precise turning. The Arduino also came equipped with a **Bluetooth Low-Energy Module (BLE Module)**, which we used for robot-to-server communications.

On top of built-in hardware, the Arduino was also connected to many external components through its digital and analog IO pins. The most important external component was an analog **Reflectance Sensor Array (Line Sensor)**, which we used for line following and junction detection. The sensor array had thirteen individual reflectance sensors, which were individually hooked up to an **Analog-to-Digital Converter (ADC)** so that the sensor readings could be accessed in software. The Arduino also commanded two **Brushed DC Motors** that could drive the MicroMice forwards and backwards. These motors had two **Quadrature Encoders** that measured the rotational position of each motor. The Arduino could also power a **Camera Module**, which communicated over radio frequencies (RF) to an external computer. We planned to use the camera module for improved maze mapping and junction detection, but we ran out of time to implement the feature.



*Diagram of the hardware components on the MicroMouse. The photo is courtesy of the [@eceumd](#) Instagram account.*





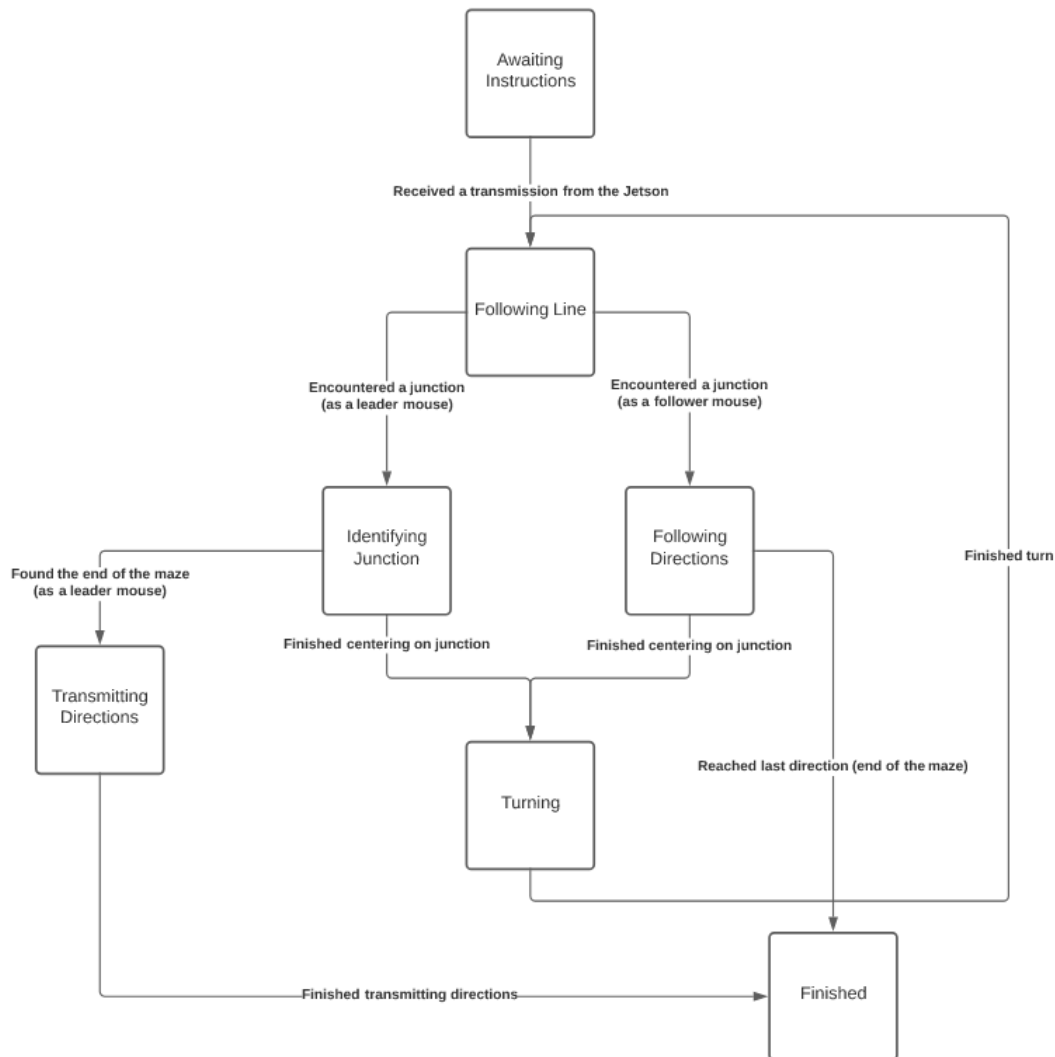
We also had access to three powerful, GPU-based computers. The **Jetson Xavier NX (Jetson)** is a 64-bit computer with a six-core CPU, a GPU, multiple USB ports for attaching peripherals, at least one HDMI port, and hardware for transmitting Bluetooth and BLE signals. The Jetson is easiest to operate when connected to a keyboard, mouse, and monitor. However, it can also function in “headless” mode, in which it is connected to another computer, such as a laptop. While the Jetson can transmit and receive Bluetooth and BLE signals, the MicroMice can only exchange BLE signals. Thus, the team was unable to utilize the Jetson’s Bluetooth capabilities.

The Jetson comes with the software needed to write and run Python programs, more specifically with Python version 3.6. The version of Python is critically important, because, unlike most software languages, Python is not fully backward or forward-compatible. Code that works in one version of Python is not guaranteed to work in another. The Jetson’s CPU has six processing cores, but the Python pre-installed on the Jetson was only able to run Python programs on four of them. We took advantage of the availability of multiple processors to run multiple instances of the client software, one per MicroMouse, each on a different processor core.

## **Software Overview**

The MicroMouse software is written in C/C++ using the Arduino IDE. The code is designed using a finite-state machine (FSM) architecture, which means that the robot can only be in one of a fixed number of states at once. When the correct conditions are met (e.g., a junction is identified), the robot switches into a different state. This architecture allowed sensor polling and motor controller tasks to be executed at a high frequency (~100 Hz) without blocking other processes. An FSM diagram, including the states and the state transition conditions, is included below:





*Finite-State Machine for our MicroMice. There are two primary paths—one path for leader robots and another for follower robots.*

The Arduino code was further separated into classes such as a `PIDController`, `MotorController`, `Gyro`, and more to improve the readability and modularity of our code. Other functions such as maze mapping, loop detection/escape, and BLE communication were integrated directly into the main `Robot.ino` file. With more time, we would have separated these tasks into new files to make the `Robot.ino` file more readable.

The Jetson-side software, which included both a client and a server program, was written for Python 3.6. The client software includes a direction-optimizer that takes in directions data supplied by the “leader” MicroMouse and creates simplified routes for the “follower” MicroMice to traverse.



Below, we provide a brief description of each file used in our project:

Purpose	Filename(s)	Description	Language
MicroMouse Control Software	Robot.ino, Constants.h, Errors.h, Gyro.h, Junction.h, LineSensor.h, Motor.h, MotorController.h, PIDController.h, PositionEstimator.h, Songs.h, StateMachine.h, Stopwatch.h, Utils.h	<p>Arduino-side code for controlling all aspects of a MicroMouse.</p> <p>Robot.ino is the main entry point file. It implements the FSM described in the figure above.</p> <p>The *.h files are either classes or namespaces that encapsulate our robot's sensors and actuators (e.g., the three-axis gyroscope, the motors, the piezo buzzer, etc.).</p> <p>For in-depth descriptions of each file, take a look at the comments at the top of each file on our <a href="#">GitHub</a>.</p>	Arduino C/C++
Jetson-side Client	bluetooth_client.py	The client that sends and receives directions to the MicroMice	Python 3.6
Jetson-side Server	socket_server.py	The server that transfers data between clients	Python 3.6
Startup script	startup.sh	The script that starts up all Jetson-side processes in optimal order	Bash
Shutdown script	shutdown.sh	The script that cleanly shuts down all Jetson-side processes in optimal order	Bash

The Jetson side of the communications protocols consists of one server process and three identical client processes. The server maintains socket connections with each of the three client processes via loopback IP addressing, in which the Jetson connects to itself without routing through an external network. Loopback IP addressing was used to avoid making the server code dependent on successfully creating and maintaining a connection to the local internet network. Each client process was run on a different processing core and conducted communications between a specific MicroMouse and the server process. This scheme made it easier to connect the Jetson to multiple MicroMice simultaneously and prevented problems with one MicroMouse connection interfering with communications with the other MicroMice.

The Arduino side of the communications protocols was designed to be modular so it could be easily inserted into other elements of the MicroMuse control software. The state-independent components consist of global variables (including two characteristics), several callback routines, and the `bluetooth_init()` function. The `bluetooth_init()` function performed setup tasks that were needed to establish smooth communications between the MicroMouse and the Jetson. For those not familiar with BLE, a characteristic is a type of variable designed so its value can be transmitted or received via BLE. These characteristics contained the information that needed to be sent between the Jetson and the MicroMouse. For example, the “directionsCharacteristic” was a list of characters that stored the direction the MicroMouse had traveled after reaching each junction, according to the following mapping.

Character	Direction Driven
‘L’	Left
‘R’	Right
‘S’	Straight
‘B’	Backward (from a dead-end)

*A description of our message format for communicating direction strings over BLE.*

When we transitioned our maze-solving code from a blocking to a finite state machine (FSM) architecture, we needed to add two new states for BLE communications (AWAITING\_INSTRUCTIONS and TRANSMITTING\_DIRECTIONS) and a third that followed optimized directions from another MicroMouse (FOLLOWING\_DIRECTIONS). The MicroMouse was initialized to the AWAITING\_INSTRUCTIONS state, where the MicroMouse established a BLE connection to the Jetson and then requested any (optimized) instructions. The MicroMouse remained in this state until it received a response. If the Jetson had no directions available, it would not send any directions and the MicroMouse had to blindly traverse the maze on its own to find the end. After the MicroMouse received any response, it would transition to the LINE\_FOLLOWING state.

After a MicroMouse performing a blind traversal found the end of the maze, it entered the TRANSMITTING\_DIRECTIONS state. In this stage, the MicroMouse attempted to transmit the directions that it took during that maze run to the Jetson. When the Jetson received these directions, it would change the



value of the MicroMouse's "flagCharacteristic," thereby acknowledging the receipt of the directions. After this confirmation, the MicroMouse transitioned to the FINISHED state.

The FOLLOWING\_DIRECTIONS state allows the MicroMouse to traverse the maze using optimized directions supplied by the Jetson instead of performing a blind walk through the maze. While one might think that the MicroMouse remained in the FOLLOWING\_DIRECTIONS state for the duration of an optimized maze run, this is not the case. Rather, during optimized maze runs, the FOLLOWING\_DIRECTIONS state takes the place of the IDENTIFYING\_JUNCTION state. Thus, the MicroMouse decided how to react to the current junction based on the directions from the Jetson rather than the identified junction type.

## Procedure

### Hardware Testing Progress

Initially, it was necessary to verify that the hardware on the Jetson and each MicroMouse functioned properly before we could start developing our software. We used the test code from the [ENEE408I NOTES EXAMPLES GitHub repository](#) to determine that each MicroMouse was operating properly and to verify a minimal level of hardware functionality. For the most part, our hardware worked without error. Towards the end of the semester, many components began failing. Here's a comprehensive list of components that failed or did not work as expected:

- Zach's LiPo battery drained to dangerously-low voltage levels. Because charging fully-drained LiPo batteries can start a fire, Zach had to get his battery replaced.
- Towards the end of the semester, Zach's piezo buzzer started playing tones more and more faintly, until it was inaudible. This malfunction was caused by a faulty circuit design for the piezo buzzer's transistor. According to Levi (the lab assistant who created the MicroMice), the transistor should have connected to a shunt diode to prevent current from flowing from other parts of the robot into the transistor. This buzzer problem also affected two or three other students' MicroMice.
- Machi's Arduino received a static shock when Levi went to pick it up during the last week of class, so his Arduino had to be replaced.
- Edward's MicroMouse came with a malfunctioning motor that would not spin, so it was replaced with a working part.
- Zach's first Arduino had a problem with the BLE hardware that prevented it from connecting to the Jetson from far away. We suspect that the BLE antenna may have been damaged during navigation testing earlier in the semester. After swapping in a new Arduino, Zach's robot worked flawlessly.
- A flaw in the Arduino gyroscope hardware caused the gyroscope to underreport angular speed (see the section below on **Turning** for more details on this bug).

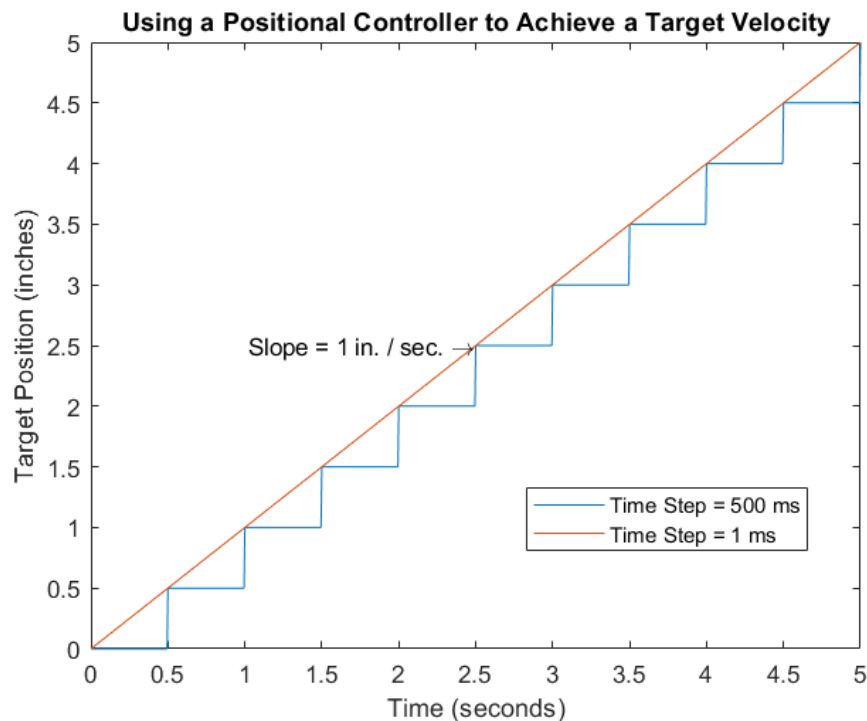
### Software Testing Progress – Control & Navigation

#### Basic Motor Control

Coding the MicroMice to navigate through the maze was one of the most challenging aspects of this project. Initially, we used simple strategies to control the robot, such as sending raw Pulse-Width Modulation (PWM) signals to each motor. However, this scheme was not reliable because the left and right motors can spin at different speeds when given the same PWM command due to manufacturing defects and nonlinearities in each motor. To correct these issues, we created PID controllers that maintain a target velocity for each motor. The code for these velocity controllers can be found in the `MotorController.h` file on our GitHub.

Making a velocity controller was much easier said than done. Unfortunately, our MicroMice had no way to directly measure the speed of each motor, which meant that any velocity PID controller would

need to perform double numeric integration/differentiation (which is prone to accumulated errors). To overcome this obstacle, our team decided to build a simpler positional PID controller using the rotary encoders on each motor. Then, we could use that positional controller to drive at a target velocity by incrementing the positional setpoint at every time step. So long as the positional controller was well-tuned, this scheme would work well. Below is a diagram showing how we achieved velocity control using a positional controller:



*An implementation of a velocity controller using a positional controller. At every timestep, the velocity controller increments the positional setpoint to achieve a target velocity of 1 inch/second. When the time step is large (500 ms), the discrete jumps of the controller are visible and significantly degrade the performance of the controller. However, when the time step is small enough (1 ms in this case), the positional setpoint closely mimics a true velocity controller.*

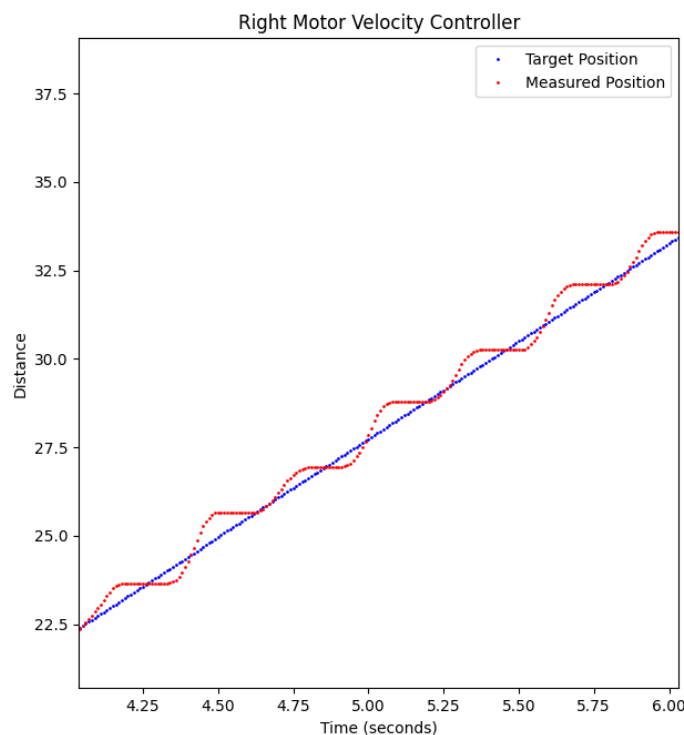
Before we could build the positional controller, we needed to determine how to measure how far the robot traveled using our available sensors. Thankfully, the robots had rotary encoders that measured motor rotation in units of “ticks.” Each “tick” corresponds to a small amount of angular rotation of the motor shaft. Instead of using encoder ticks directly, we measured a conversion factor between ticks and inches by pushing the robot forward on the tile mats (making sure that the wheels did not slip) and observing how far the robot traveled with a measuring tape. We found that there were roughly 84.9 encoder ticks per inch of forward distance.

Now that we could determine how far each motor had traveled in inches, we designed a positional PID controller that could drive the motor to a specified distance. We used the [Arduino-PID-Library](#) to

perform all of the low-level PID calculations. We tuned the PID constants for each wheel independently, using the procedure described in [this StackOverflow post](#). Other tuning strategies, such as the Ziegler-Nichols method, did not work well for our robot.

A PID tuning tip that our team would recommend is to write a function that can tweak PID coefficients by accepting input from the Serial monitor (see our [PIDController::AdjustPIDConstants\(\)](#) function). This function allowed our team to tweak PID constants without having to recompile our code, which was a significant speedup considering the 40+ second upload times for the Arduino. Another helpful tip is to use the Arduino Serial plotter to inspect the setpoint, input, and output signals for PID controllers, which makes the tuning process much simpler.

After we properly tuned our positional controller, we thought that the velocity controller would be simple to implement. However, we encountered a major obstacle with the [Arduino-PID-Library](#) that caused our robot to jerkily maintain the positional setpoint. A graph of this stop-start behavior can be seen below:

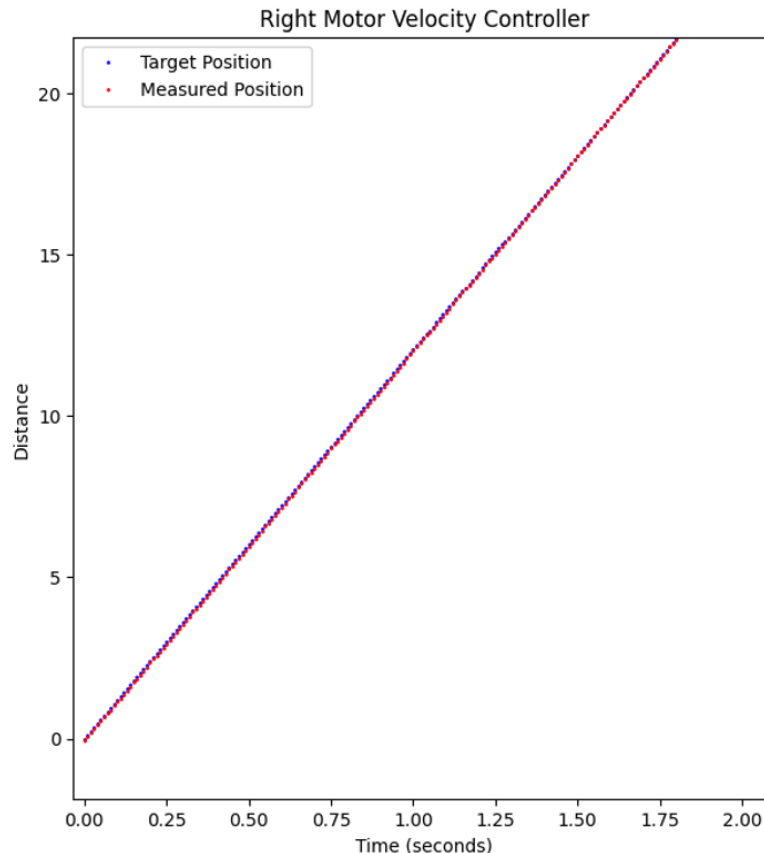


*Here is a graph of our jerky velocity controller. For intervals of 100ms, the motor seems to stop spinning.*

As you can see, the motor would reach its target setpoint temporarily, then stay stationary for around 100 ms (which causes the staircase shape of the red line). Samples were taken every 10 ms (the sample rate of our PID control loop). After two weeks of trial and error, we discovered that this error was



caused by the Arduino PID library's `SampleTime` property. This property controls the minimum amount of time between PID computations. If you try running PID computations more frequently, the output will stay the same until a full `SampleTime` has elapsed. Because we performed a PID computation (and incremented the velocity setpoint) every 10ms, we needed to call `PID::SetSampleTime(10 ms)` at the beginning of our program to fix the issue. After catching this bug, our robot could closely track a reference velocity, as seen below:









*Here is a graph of our velocity controller after correctly setting the sample time of the PID controller to 10 ms.*



## Line Following


Now that our robots could drive both motors at specific velocities, following lines of tape became simple: just drive both motors at the same velocity (e.g., 10 inches/second) and apply a slight correction if the robot drifts off the tape. To measure how “skewed” a robot is relative to the tape, we take a weighted difference between the left and right light sensors. If more of the left sensors see tape, then the robot has a negative skew. If more of the right sensors see tape, then the robot has a positive skew. This scheme worked perfectly during our final test run.

## Junction Identification

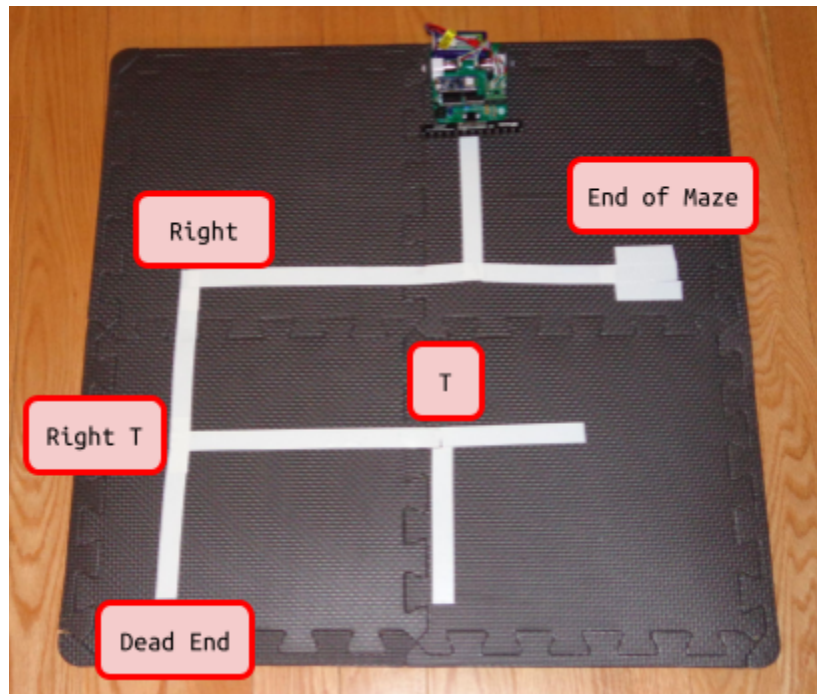
Before the robots could fully solve mazes, we had to implement robust junction identification using the line sensor. We identified the following classes of junctions (the robot enters from the bottom):

Junction Type	Shape	Unambiguous?
Left		✗
Right		✗
Line		✗
Dead End (Empty)		✓
T	T	✗
Plus	+	✗
Right T		✗
Left T		✗
End of Maze (3 pieces of tape)		✓

We labeled some of the maze junctions as “ambiguous,” which means that the robot needs *multiple* line sensor readings to confirm that junction’s true identity. For example, the robot has no way to distinguish between a Right T () and a Right () junction because they will both appear as an abundance of tape on the right. To distinguish between these two cases, the robot takes multiple line sensor readings. The initial sensor reading is taken when the robot identifies any junction that is not a line. After that, the robot centers itself on the junction and takes another measurement.

Taking two line readings to determine a junction type worked well for many cases, but it was prone to errors if the robot misidentified the first junction. For example, when approaching T junctions from a slightly-slanted angle, it was common for the robot to sense a Left () junction instead of a T. Since the robot only considered the very first reading, the robot would always misclassify junctions if it got the first measurement incorrect.

To prevent junction misidentifications, we implemented a line sensor confidence system. Every time the robot asks the line sensor to identify a junction, the software only returns the last “confident” reading. The line sensor only becomes “confident” in a new reading when it has been seen a certain number of times (configured using the `LINE_CONFIDENCE_THRESHOLD` variable in `Constants.h`). Through experimentation, we found that a threshold of 4 consecutive readings worked well in practice. After adding this system, our robot experienced far fewer junction misidentifications.



*In this image, a MicroMouse is positioned at the beginning of a test maze that we used to check the control and navigation software for bugs and errors. A few of the different junction types are labeled on the drawing (all assuming that the robot is approaching the junction from the bottom of the image).*

## Turning

Now that our robots could drive straight, follow lines, and identify junctions, our last major obstacle was turning the robot to precise angles. To accurately measure how far the robot turns, we used the z-axis gyroscope on the Arduino's IMU, which returns the rotational speed of the robot in degrees per second. To convert these speeds into angular displacement, we had to numerically integrate. All of this functionality is contained within our `Gyro.h` file.

One complication with gyroscopes is that they often return a nonzero angular speed even at rest (due to manufacturing defects). To obtain an accurate angular speed, we had to determine the gyro's bias and subtract it from the raw rotational speed. Since this bias is different for each gyro, we designed a calibration script to calculate the sensor bias when the robot boots up (see [Gyro::calibrate\(\)](#)). The calibration procedure works taking the median of many gyroscope measurements while the robot stays still.

While testing out the gyroscope, our group discovered a bug with the Arduino Nanos that caused the IMU's gyroscope to underreport angular speeds. Empirically, we found that the Arduino reports 310 degrees for every 360 degrees of real-world rotation. [Other Arduino users](#) have experienced this bug as



well, so it appears to be a hardware issue or a bug with the IMU's Arduino library. To fix this bug, we simply multiplied every raw sensor reading by the conversion factor  $\frac{360}{310}$ .

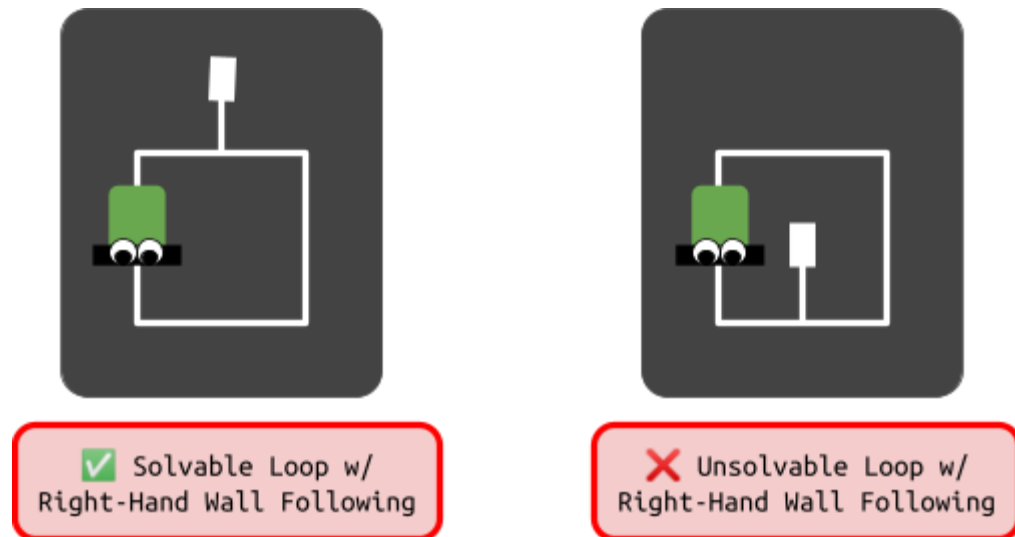
With these adjustments in place, our gyroscope finally returned stable measurements that corresponded to real-world degrees. From there, it was simple to design and tune a PID controller that turned the robot to a target angle.

### Maze Traversal

Our team opted to use right-hand wall-following to traverse the maze. At every junction where we could go right, we would. The table below lists the turn that the robot takes for each junction:

Junction Type	Turn Direction
Left (└ )	←
End of Maze (  )	↑
Line ( )	↑
Left T (└+)	↑
Dead End (Empty)	
Right (┐)	→
Right T (┐+)	→
T	→
Plus (+)	→

The right-hand wall-following technique can solve any loopless maze. Even if there are loops in the maze, a right-hand wall-following robot can solve the maze if the exit is not positioned on the inside of a loop. The diagram below demonstrates the difference between solvable and unsolvable loops:



*A diagram of solvable vs. unsolvable routes. The thick piece of tape represents the end of the maze. In the diagram on the right, a right-hand robot will never opt to take the left turn to exit the maze.*

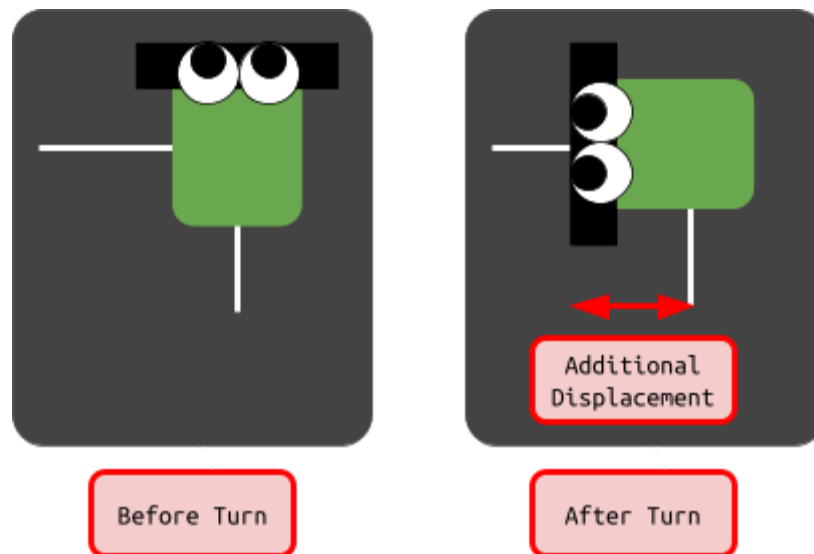
Although we never encountered any mazes in class with unsolvable loops, we still wanted to have code that would allow us. To escape from these troubling loops, we had to map the maze and identify when the robot encountered the same junction twice.

## Maze Mapping

The basic algorithm for maze mapping is as follows. First, create a square 2D matrix that represents the maze. In our case, each entry in the maze map corresponds to a possible junction location, and junctions can only be placed in increments of half a floor tile. Since each floor tile is 30 cm long, our maze cells were  $\frac{30}{2} = 15\text{cm}$  each. The matrix has  $2c + 1$  elements along each dimension, where  $c$  is the total number of cells in the maze. Why do we make the matrix larger than the maze itself? It is a programming trick that allows us to always imagine that our robot starts in the center of the maze map. As the robot traverses the real-world maze, the extra cells in the matrix prevent the robot from overflowing the maze map array regardless of its starting location.

Initially, all values in the maze map are set to -1, indicating that those cells are unexplored. As the robot traverses the maze, it fills in unexplored cells with its current heading ( $0^\circ$ ,  $90^\circ$ ,  $180^\circ$ , or  $270^\circ$ ). Here,  $0^\circ$  corresponds to the starting orientation of the robot. When the robot first boots up, we start the robot in the center of the maze (i.e., set `Maze[NUM_ROWS / 2][NUM_COLUMNS / 2] = 0`). Next, the robot traverses the maze using right-hand wall-following. When a junction is detected, the robot measures how much time elapsed since the last junction was seen. Since the mouse drives along straight lines with roughly constant speed (thanks to the velocity controller), time can be used to determine how many cells the robot traveled. We could have also used encoder ticks to measure this cell count, but time was just as accurate and was easily accessible on the Arduino through the `micros()` function.





Our biggest challenge during maze mapping was determining how many cells the robot traveled. To properly count cells, we needed to know how long it takes for a MicroMouse to drive a single cell. We called this value the `UNIT_SPEED`. Using this value, we can calculate the cell count as `round(time_since_last_junction / UNIT_SPEED)`. While sound in theory, this approach was troublesome in practice. The biggest complication was that our robot travels part-way to the next cell whenever it turns at a junction. Using timing alone makes this additional displacement hard to account for in our calculations. After adjusting our code to account for turns, cell counting became accurate and reliable.



*A diagram that demonstrates why there is an additional displacement each time the robot turns on a point.*

With an accurate maze map, the robot can detect loops by checking when it enters a junction twice from the same starting angle. Whenever the robot detects a loop, the loop-escape system triggers. Our basic strategy for escaping a loop is to temporarily switch from right-handed to left-handed wall-following. The left-handed wall-following scheme is described in the table below:

Junction	Direction Taken	Escaped Loop?
Left (└ )	←	✗
End of Maze (┘)	↑	✗
Line ( )	↑	✗
Left T (└┐)	←	✓
Dead End (Empty)		✗

Right ( $\sqcap$ )	$\rightarrow$	
Right T ( $\vdash$ )	$\uparrow$	
T	$\leftarrow$	
Plus ( $+$ )	$\leftarrow$	

As soon as the robot escapes a loop, it must return to right-handed wall-following because remaining in the left-handed state for too long can sabotage the direction optimization code on the Jetson. Early versions of the loop-escape system switched back after taking any left-handed action. However, this failed in situations where the left and right handed-systems take the same action, such as left (  $\sqcap$  ) junctions where both schemes must go left to proceed. To avoid swapping back too early, the MicroMouse only returned to right-hand wall-following when it took a left-handed action that a right-handed system would not perform (denoted by checkmarks in the table above).

## Software Testing Progress – Communication

Our design used BLE to allow the MicroMice to communicate through the Jetson. Because of this, determining the BLE addresses of all the MicroMice was a critical prerequisite for developing the communications software. It would have been impossible to connect to a specific MicroMouse or exchange data without this information, which we acquired during the hardware tests.

The initial testing of the communications software consisted of connecting all three MicroMice to the Jetson and maintaining this connection for a period of time. Consistently achieving a stable connection proved to be a difficult, yet critical step in the testing and development. In order to achieve consistently stable connections, it was necessary to raise the priority of the client processes and run them on different processing cores. After achieving stable connections for all three MicroMice, the focus of testing shifted to data exchange. This required us to overcome a variety of issues relating to timing and stability. Successfully connecting to the Jetson in order to pass small quantities of data to and from the MicroMice followed.

Additionally, the software was designed so that an operator could manually shut down and then restart a single client process on the Jetson without affecting the other processes. This capability was used to correct problematic BLE connections and to test multi-robot software capabilities with only a single MicroMouse. This allowed one person with one MicroMouse to perform end-to-end testing of a blind maze run followed by an optimized maze run with just a single MicroMouse.

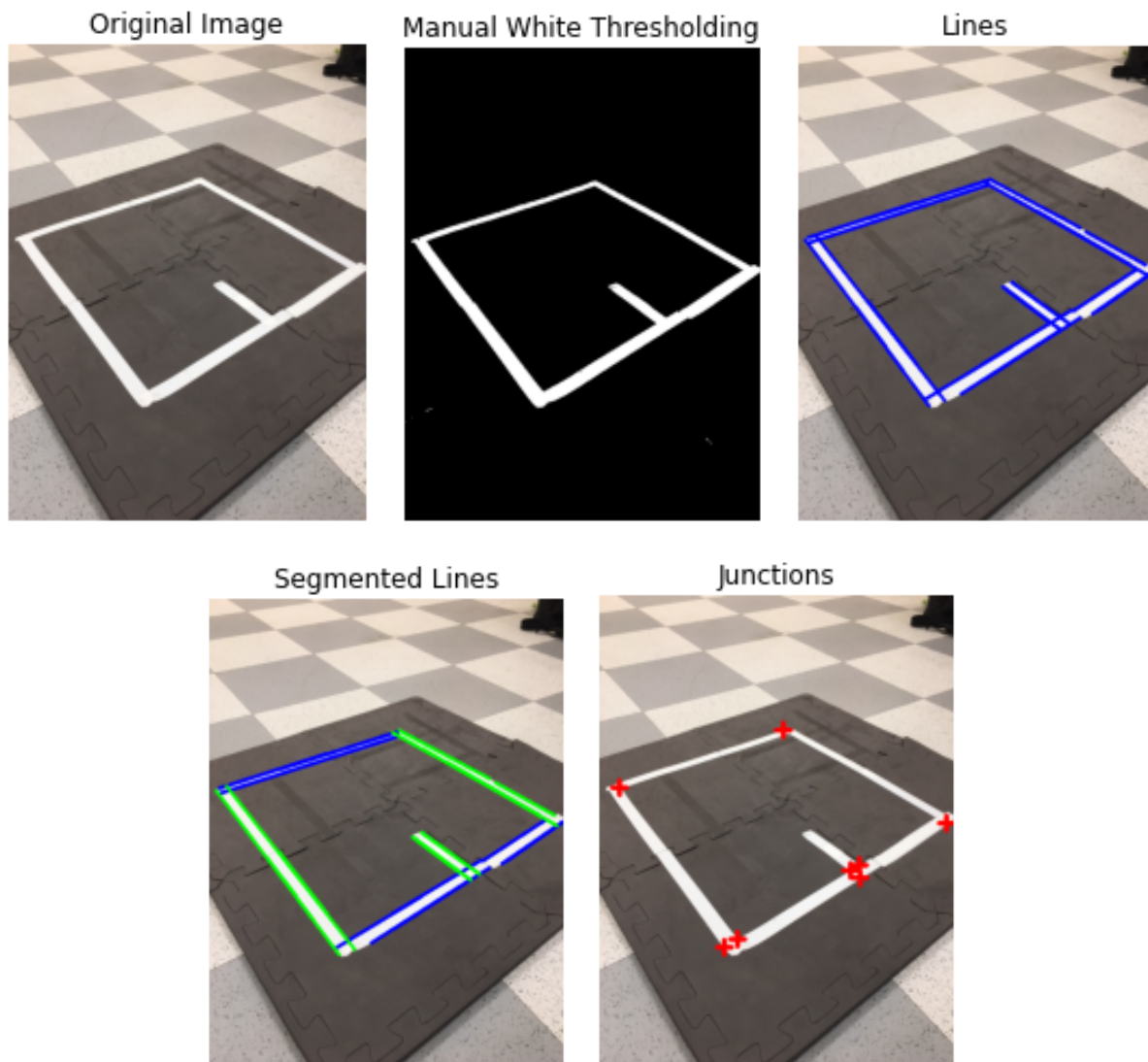
Finally, a series of integration tests had to be performed, as the communications software was painstakingly integrated with the motor control software. The MicroMice were connected to the Jetson, and then repeatedly ran multiple test mazes to ensure that both the communications, and the motor control software worked individually, and did not interfere with one another. This would test the MicroMouse's ability to connect to the Jetson, run the maze, and then send back correct directions. After those capabilities had been debugged, the focus of testing changed to the code to run the maze based on



previously gathered directions. Conversion to an FSM architecture required making significant modifications to the communications and optimized maze running software to conform to the new architecture. These changes were followed by another round of integration testing.

### Software Testing Progress – Camera Detection

Although we never integrated any computer vision code into our final robot, we spent several weeks investigating junction identification using OpenCV. All of our tests can be found in the [computer-vision/line-detection.ipynb](https://github.com/UMD-RC-Team/robotics/blob/master/computer-vision/line-detection.ipynb) Jupyter Notebook on our GitHub. We would recommend that other teams also use Jupyter Notebooks because they provide a simple interface for evaluating and tweaking computer vision pipelines. They are also useful for many other data processing tasks.



*An overview of our computer vision pipeline. The images are ordered from left-to-right, top-to-bottom. In the “Segmented Lines” image, green represents vertical tape and blue represents horizontal tape.*



Our computer vision pipeline starts with an unprocessed image of a maze that is taken from above. First, we apply a manually-calibrated filter to select all of the white pixels in the image. Any pixels that fall within a targeted color range will pass through the filter and everything else is rejected. We perform our filtering in the HSV color space because it is good for selecting highly saturated (white) pixels. The result from our first pipeline stage is a black and white binary image that only contains the lines of tape from the maze.

Next, we apply a Hough Line Transform to this binary image, which identifies any line segments in the image and returns them in an array. Before we can identify junctions using these line segments, we need to group them into clusters of horizontal and vertical lines.

To separate horizontal and vertical lines, we used the k-means clustering algorithm with  $k = 2$ , which separates the line segments into two classes (horizontal or vertical) based on their slopes. K-means clustering works by assuming that there are  $k$  clusters in a dataset and picking  $k$  random guesses for the center points of those clusters. Points are then assigned to different clusters by finding the cluster with the closest center point. At this point, we calculate the mean value of every point in the candidate clusters. These mean values become the new cluster centers in the next iteration of the algorithm. The algorithm continues iterating until the cluster centers settle on stable values. Separating line segments into horizontal and vertical lines is a perfect use-case for k-means clustering because we know ahead of time that there will only be two types of lines in our images.

The final stage of the pipeline identifies intersection points between horizontal and vertical line segments using an efficient algorithm described in [this StackOverflow post](#). If we had more time to work on our computer vision pipeline, we would have classified each of these intersection points as a junction. However, due to time constraints, we were unable to make any more headway on our computer vision tests.

## Software Testing Progress – Operating Systems/Source Code

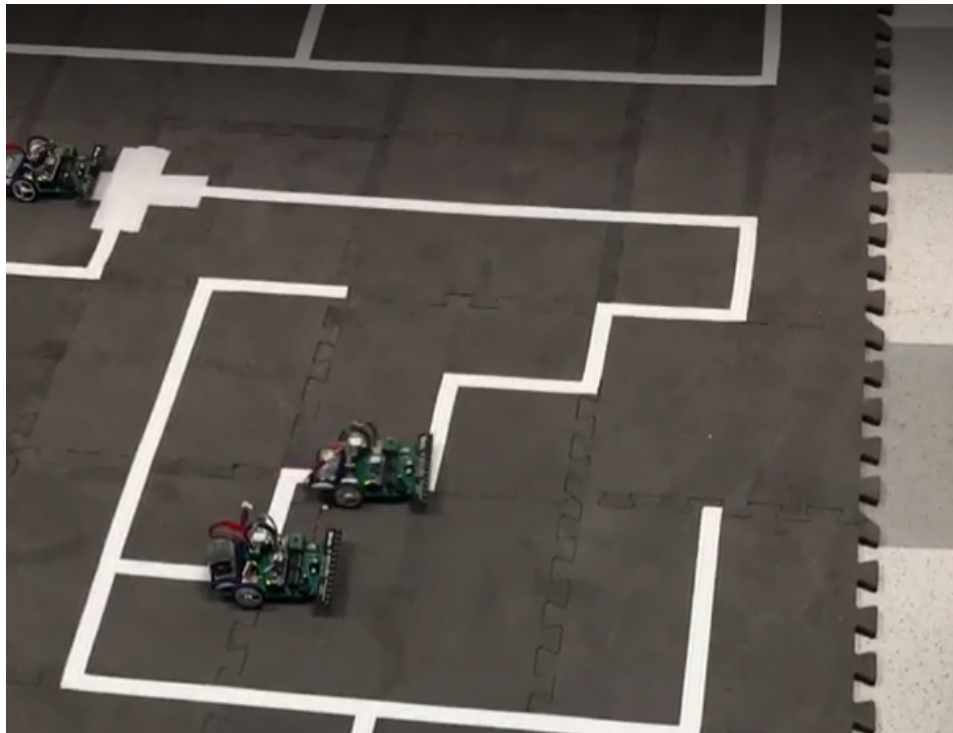
We ran our robot in a simple, bare-metal environment without an RTOS. One of the primary benefits of an RTOS is being able to run tasks at specific frequencies. Since we did not have strict timing requirements for any of our code, an RTOS was not necessary. Instead, we used a “Super Loop” architecture, which means that all of the robot code runs inside a `while` loop that is executed at a very high frequency. All of the code inside the super loop executes in a non-blocking manner (i.e., with no long operations).

The super loop architecture still allowed us to run high-priority tasks, such as motor control calculations and sensor polling, at a high frequency (e.g., 100 Hz). It also avoided the overhead of thread synchronization, which we would have needed if we had used an RTOS. Although our group members were interested in learning about software design in an RTOS, the super loop architecture worked sufficiently for our use case.

## Execution/Demo Result

After a long semester of work, we finally got our MicroMice solving arbitrary mazes (in the final week, no less)! The latest version of our code can be found on our team's [GitHub repository](#).

Here is a [video of our final demonstration](#). In this video, Edward's robot led the charge and traversed the most complicated maze track using a right-hand wall-following strategy. After it found the end of the maze, Edward's robot sent the string of the directions that it took to the Jetson. The Jetson eliminated any loops and unnecessary turns in the maze directions and sent them back to Zach and Machi's robots, which followed the optimized directions and reached the end of the maze!



*Here is a screenshot from our final maze run, where Zach and Machi's robots are following the directions from Edward's mouse. Thanks to the reliable turning code, the two follower robots avoided a collision before reaching the end of the maze.*

At the end of demo day, we also tested our robot's ability to escape loops. A video of a single robot successfully escaping a loop and finding the end of the maze can be found [here](#). We had to start our robot inside a loop to allow loop detection to activate. Otherwise, the robot would simply hug the outer-right walls of the maze and find the end of the maze.



## **Summary & Future Suggestions**

### **Machi**

The final product is slightly different from our proposed solution because we switched from Tremaux's maze traversal algorithm to modified right-hand wall-following. The end product was slightly simpler and worked well for the final maze, so it did not seem to make a difference. One difficulty that we encountered was the issue of integrating the various software functions that we created. We solved it by creating a finite state machine that updates the state periodically. While the final product worked well, it still had some issues because we did not prioritize integration until late into the semester.

Our expectations for the maze changed drastically throughout the semester, and our final product had to adapt in tandem. Ultimately, we completed all four mazes, as was expected. However, due to difficulties with curvilinear line following, I do not believe we would have been able to solve any mazes with curved lines using our current mapping scheme. I recommend that future work be focused on curvilinear maze solving and optimization of paths, both within and outside of the maze.

I have learned a great deal about maze-solving algorithms, general fault detection, and finite state machines. I have also picked up some info on controller design and AI learning. However, my biggest triumph is learning C++. I suspect that will take me further than anything else.

### **Edward**

During this course, I learned several important lessons and gained valuable working experience. The first lesson that became apparent is that any project element that is a prerequisite, or dependency for the completion of another project element should be given priority. It is important to avoid scenarios in which one or more parts of the project being delayed creates additional delays in the development of other project elements. Second, everything is harder and takes longer than you initially estimated. Because of this, it is necessary to plan to complete all work before its assigned deadline in order to cope with the inevitable delays caused by previously unanticipated problems that will inevitably occur. Along the same line of reasoning, it is necessary to assume any and all prototypes will have bugs, errors, and other issues, and budget sufficient time in the work schedule for testing and debugging.

During this course, I gained basic proficiency writing in the Python programming language and learned firsthand about its various idiosyncrasies. One of Python's more useful features is its large selection of available libraries with their prebuilt functions, much like in Java. On the flip side, Python is neither fully backward compatible, nor fully forward compatible. Thus, any program written in one version of Python may or may not run correctly with another version. Additionally, Python uses indentation to define blocks, in contrast to many other programming languages such as C/C++ that use brackets to define blocks. This makes Python the only programming language that I have ever used in which code functions differently, or will not execute at all due to the presence or absence of white spaces, or tabs.



These indentation errors can be difficult to locate, and unpleasant to correct. I am grateful for the opportunity to gain experience with a programming language that is so prevalently used in the industry.

## **Zach**

By the end of this class, I was elated that our robot could drive and turn as reliably as I had hoped at the start of the class. It took *a lot* more effort than I expected to complete certain tasks (e.g., implementing velocity control, reliably identifying junctions, making the gyroscope more reliable), but the work was always rewarding. The lessons that I learned about designing real-world controllers and organizing software for real-time systems will carry with me into the workforce and into my personal projects.

If I had more time to work on the MicroMouse, I would make the following improvements to our robot:

1. Refactor our FSM to a more modular, extensible design. I had worked with Levi to plan out a better structure for the FSM that had a more clear separation between high-level and low-level tasks. An image of this improved architecture can be seen below:



*Our ideal design for the robot's FSM. This structure would have grouped all of the maze-solving decision-making into a single state and prevented duplication of the junction identification code.*

2. Integrate our computer vision pipeline into the robot code. It would have been incredible to see the MicroMice identifying junctions without needing to take multiple line sensor readings.
3. Add collaborative maze mapping to our robots. Rather than having a leader-follower design, it



would have been more efficient to have multiple robots map the maze at once. This modification would require a significant overhaul to the communications and mapping code, but it would allow our robots to more quickly discover the shortest path to the exit.





## **Feedback**

### **Machi**

One aspect that I believe should be improved upon is the quality of the micromouse. We had to write programs and come up with creative ways to bypass the errors that cropped up due to the low quality of the mice. For example, the mice would turn 80 degrees when it was supposed to turn 90 degrees. It would also be ideal if an end of the maze detection program was provided ahead of time in the ENEE408I\_Notes\_Examples repository, or at the very least, if the expectations for the end of the maze are known at the beginning of the semester.

Another great idea is for the GitHub Desktop to be integrated into the course rather than simply GitHub. This makes the integration of specific modules easier within teams. However, what I really liked was that office hours were easily accessible. This made it much easier to test the functionality of the modules we wrote.

### **Edward**

One aspect of this course I am truly thankful for, and think should be retained, is the great hardware support and availability of spare parts provided. Over the course of this semester, our group experienced multiple failures of hardware components. Without rapid access to replacement parts, the resulting delays would have been crippling. I hope that future students will have access to the same level of hardware support provided by the TA and lab assistant this semester.

The Jetson has enough processing power between its six-core CPU, and additional GPU unit to be, for all practical intents and purposes, a super-computer. However, because of the limited IO capabilities of the Arduino, this processing power could not be fully utilized. In order to more fully utilize the Jetson's capabilities, the IO capabilities of the other hardware devices should be improved.

### **Zach**

ENEE408I was my first design course at UMD, and I loved having the freedom to explore new engineering concepts in a lab environment. This class exposed me to a diverse set of topics, from computer vision to communication protocols to low-level robot control. Each of these topics could have been their own semester-long course, so getting to explore all of them in a single class was a privilege. I also loved the loose deadlines for course objectives, which made the class feel like a project that my team could adapt to our busy schedules.

One area where this class could improve is by increasing collaboration between teams. I only began collaborating with other teams towards the end of the class, and I learned so much by helping and receiving help from my classmates. Since all of the teams are solving the same problems, it would be helpful to have more formalized ways to collaborate, especially earlier in the semester. One idea to increase collaboration is to introduce code reviews between teams. During the code reviews, each student could use GitHub to comment, critique, and ask questions about another teams' codebase. This opportunity would allow students to see how other teams are approaching the same problems and to offer



advice to help other teams improve. Another small recommendation is to move away from the Arduino Nanos because of the challenges that we faced this semester with the BLE hardware. Hardware problems aside, ENEE408I has been my favorite class at UMD. I hope that it is available to future ECE students for years to come.



## References

Any reference link(s) that you used in this project.

1. ENEE408I Examples/Notes ([https://github.com/UMD-ENEE408I/ENEE408I\\_Notes\\_Examples](https://github.com/UMD-ENEE408I/ENEE408I_Notes_Examples))
2. ENEE408I Team 1 GitHub ([https://github.com/UMD-ENEE408I/ENEE408I\\_FALL2021\\_Team1](https://github.com/UMD-ENEE408I/ENEE408I_FALL2021_Team1))
3. GitHub: Arduino PID Library (<https://github.com/br3ttb/Arduino-PID-Library>)
4. Wikipedia: Maze Solving Algorithm ([https://en.wikipedia.org/wiki/Maze-solving\\_algorithm](https://en.wikipedia.org/wiki/Maze-solving_algorithm))
5. StackOverflow: Good Strategies for Tuning PID Controllers (<https://robotics.stackexchange.com/questions/167/what-are-good-strategies-for-tuning-pid-loops>)
6. Arduino Forums: Nano 33, heading keeps on dropping while resting: (<https://forum.arduino.cc/t/nano-33-heading-keeps-on-dropping-while-resting/635234/66?page=3>)
7. StackOverflow: How do you detect where two line segments intersect (<https://stackoverflow.com/questions/563198/how-do-you-detect-where-two-line-segments-intersect>)



## **List of Abbreviations**

- ADC = Analog-to-Digital Converter
- BLE = Bluetooth Low Energy
- CPU = Central Processing Unit
- FSM = Finite State Machine
- GPU = Graphics Processing Unit
- HSV = Hue, Saturation, Value Colorspace
- IDE = Integrated Development Environment
- IMU = Inertial Measurement Unit
- IO = Input Output
- IP = Internet Protocol
- LiPo = Lithium-ion Polymer
- MM = MicroMouse
- PID = Position, Integral, Derivative
- PWM = Pulse-Width Modulation
- RF = Radio Frequency
- RHWF = Right Hand Wall Following
- RTOS = Real Time Operating System
- USB = Universal Serial Bus