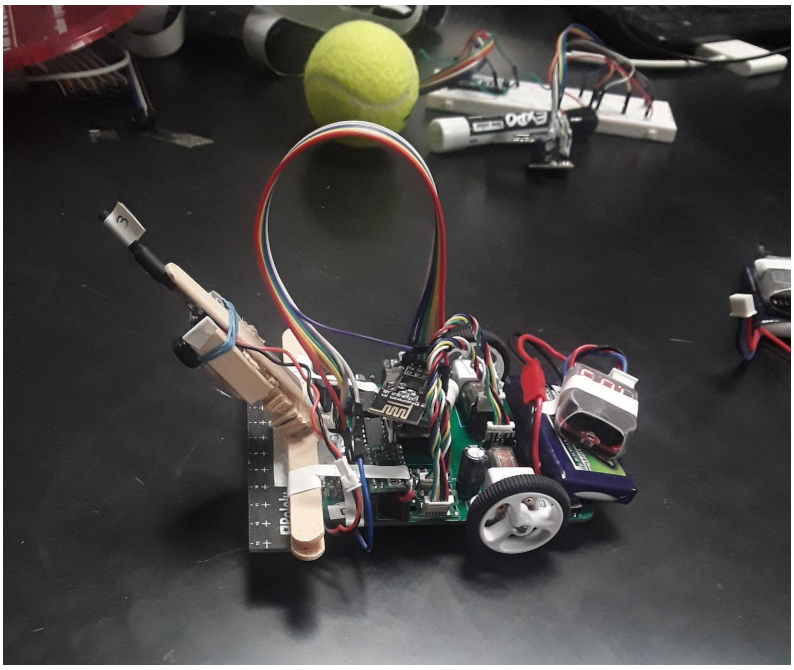
Final Report for

Team 3



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: 12/19/2021
Prepared by:

Caitlin Lee (clee811@terpmail.umd.edu)
Daniel Lee (dlee21@terpmail.umd.edu)
Gillian Lee (gyl37@terpmail.umd.edu)

LIST OF ABBREVIATIONS (OPTIONAL)

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Challenge Overview

The main objective for this class was to design a fleet of three robots that could solve a maze autonomously. The maze was constructed from lines of white tape on black foam squares so a robot, called a MicroMouse, would have to be able to detect and follow the lines. Our spin on the main challenge was that we wanted to use all three mice, networked together, to map out the entire maze, and then find the shortest path through the maze so that all of the mice would be able to solve it. Going into the course, we expected to be able to apply what we'd learned from control systems courses, which had only given us theoretical knowledge, and expand our skills in programming Arduinos. Additionally, based on all of the materials we were provided with at the very beginning, we wanted to learn how to use each of the components themselves, such as the wireless camera and the Jetson, but also learn more about the different ways that each of these items could be used and how to implement them. None of us had ever worked with programming multiple robots to network them together, so we also hoped to learn how to do that. Finally, as with any team project, we wanted to improve our communication skills and improve our abilities to function better as part of a team.

1.2 Proposed Solutions & Teamwork Distribution

To accomplish our design challenge, we wanted to use all three mice to be able to map the maze more quickly (i.e. a divide and conquer mentality). Our original plan was to use Bluetooth to communicate with each mouse to a central Jetson that acted as the central computer to store data sent back by the mice and to send commands to the mice themselves. The mapping was going to be accomplished by implementing a proportional–integral–derivative (PID) controller that would make a mouse drive at a constant velocity, which would allow us to keep track of distances traveled based on time. We were also going to use the camera to detect the lines of the maze to make sure we were following them, and to detect intersections and corners so that we'd record them and turn properly.

After researching, trying things out, and finding out issues with this plan, our final solution was to use a server as the central computer which would store the map and decide what command the mice should follow. These commands would be sent to the mice via radio frequency (RF), achieved by adding RF transceivers to each robot. Daniel designed and coded the entire server, including how to store the map and which commands the mice should follow when mapping and when solving, as well as how to use the RF to send those commands. The mice would drive at a nearly constant velocity by using a PID controller, and would use the infrared (IR) sensor bar already attached to the mice to follow the line when driving straight and when making turns. Gillian designed PID controllers for driving straight and turning. The wireless camera would be used to not only detect intersections and corners, but also measure the distance to intersections so that the mice would drive exactly the right distance to be centered over the corner for making the turn. Caitlin designed the image processing for detecting intersections and measuring distances. More details can be found in Section 2.

2. APPROACH

2.1 Hardware Overview

Each mouse holds an Arduino Nano BLE Sense, two motors, two wheels, two encoders (one for each motor), a line follower IR array, an RF module, FPV camera, and custom camera mount. Additional hardware that was used included an additional RF module to pair with the one on the mouse, a camera receiver module, and a laptop.

The majority of components worked as intended, with any issues being easily remedied with quick fixes or by switching out the component. We had some issues with the rubber tires of the wheels sliding off every once in a while, or the wheels being pushed a bit too far in so as to touch the motor housing or the side of the mouse, both of which led to the wheels not turning as intended, but these were easily fixed one we noticed. The motors had minimal issues, though designing a motor controller was absolutely necessary to guarantee a desired rate of rotation. One motor's gearbox also broke on demonstration day, though that was quickly replaced by Levi with a motor from one of our other mice. The cameras were less than ideal in terms of image quality and would occasionally have unreliable connections with the receiver, but were overall acceptable for our purposes.

The majority of the RF modules did not work as intended, except for one.

2.2 Software Overview

The software was written in Arduino and Python. Arduino was used for functions that directly related to the mouse such as controls and RF communications. Python was used for all other functions such as image processing and the server. The software can be broadly divided into three parts: software running on the server, software running on a laptop, and software running on the mouse.

For our original solution, the idea was that the server would act as the central control for all of the mice networked together. However, as discussed in Section 3.1, with the RF module only working for Daniel's mouse, then we only needed to be able to control his. Central control still occurred in the server code running on the laptop, where the information regarding the explored parts of the maze and the mouse's current position were used to decide what the mouse should do next. The only code uploaded to the mouse was an Arduino script for connecting the mouse's RF transceiver to the laptop's RF transceiver, along with the header file containing all of the movement functions (i.e. driving forward and turning, see Section 3.2 for details). The camera had its own receiver, so it was controlled by a Python script with functions for initialization, intersection identification, and closing (see Section 3.4 for details). Separate Python scripts running on the laptop acted as a middleman, controlling the flow of information between the server, Arduino, and camera.

3. PROCEDURE

3.1 Hardware

For the control aspect of the robot, all of the hardware components that we needed worked properly, except for the one motor gearbox that broke during final testing, as previously mentioned.

In comparison, there were many hardware issues with RF communications. One issue was with the wiring of the RF modules on our mice. When we first started to work with the RF modules, we were not able to get them working at all. Even with the example code provided to us, the RF modules would not connect. The only way we got them to work is by bringing our hands close to the mouse. We later found out that this was due to the wrong wire being connected between the mouse and the RF module. We were able to fix this, but it was frustrating that we'd spent so long trying to figure out the software when we should've been checking the hardware.

Another issue we ran into was our RF modules would not work unless specific steps were taken before connecting. For example, to be able to communicate with Daniel's mouse, we had to turn on the mouse, flash the mouse with the Arduino code, check the serial monitor to verify the RF module initialized, run our RF initialization code on our laptop, and unplug the usb cable to the laptop. If these steps were not followed in this order, the laptop was not able to communicate with the mouse. This made testing our mice take a very long time. If we wanted our mouse to restart a maze, we would have to first go through these steps. Even if we did not change any code, we would have to go through these steps to make the RF modules communicate.

These issues with RF communications were the reason we were only able to have one mouse solve the maze on demo day. Caitlin's mouse was working when we first tested RF communications with example code. However, when we made a lot of modifications to the code, we were not able to get it working on Caitlin's mouse. We checked the wiring and tried replacing the RF module but it still did not work. We decided to continue with just two mice. However, on the Monday before demo day, we found that Gillian's mouse would not connect to the laptop if its battery was on. We tried changing the initialization steps like turning on the battery after the mouse was flashed. We were not able to get it to work. The only way it would communicate with the laptop was when the battery was turned off.

One major issue that we realized we'd need to deal with if we wanted to use the camera was how to mount it on each robot. The camera would heat up very quickly, which one of our classmates, Tin, found out was even hot enough to melt a 3D printed camera mount he made. We needed an actual way to mount the camera both elevated above the robot and at an angle to include less of the background in the camera's view for more reliable intersection detection, and to lessen foreshortening effects for better distance calculation. We wanted to make sure that whatever we added to our robots for this purpose would be easily removed in case it didn't work, but also because if it did work, everything would need to be able to be taken off so the original robot could be reused in the future.

Because of the limitation on workable materials, we had to be a bit creative with how we did this. Caitlin and Gillian spent a night doing some rapid prototyping using popsicle sticks, white glue, and hot glue, cut and stacked to provide the elevation and angling, and some other cut pieces arranged to hold the camera in place. The original camera mount was designed to fasten on the IR sensor bar, since this provided a long, flat surface that would allow the popsicle stick base to be securely attached with rubber bands. After testing the distance calibration and intersection detection, we realized that the camera needed to be angled downwards more, so it'd need to be moved backwards. This generated issues with where and how to attach the base of the mount so that it would be secure, as having to reattach it would mean that the camera wouldn't be at exactly the same angle and position as it was last calibrated at, so Caitlin would need to spend a while recalibrating it again. We decided to use a lot of electrical tape to cover the gap right behind the IR sensor bar, and then fill it with hot glue to provide an elevated and sticky surface that we could then attach the base popsicle stick to. After running the mouse with this design on it several times, the electrical tape ended up coming loose since the camera was leaning over, so we had to take everything off and redo it. This time, we added more tape around the sides to hold it down, plus additional pieces that held the mount base down to counteract the camera wanting to fall over. We didn't have any more problems with the camera falling over and losing the calibration, so this was a stable (and reusable!) camera mount design.

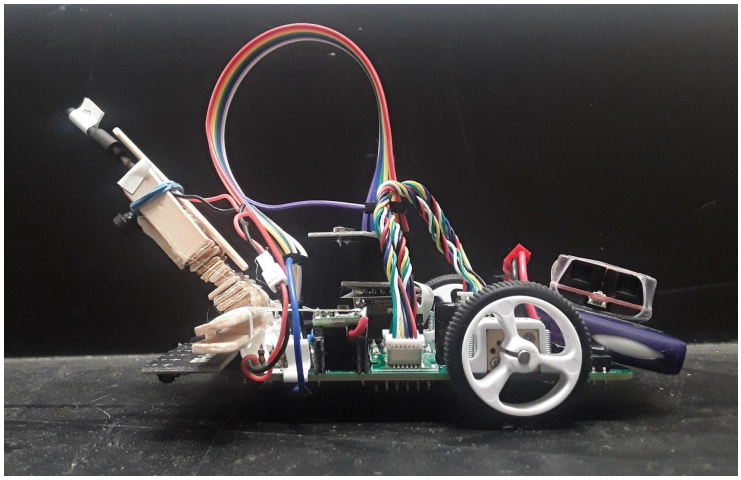


Figure 1. Side view of mouse with camera mount.

3.2 Control & Navigation Software

A PID controller was designed to control the velocity of each robot. Gillian wrote the PID controller from scratch (rather than using the Arduino PID library, mainly because we didn't know about it at the time), initially to control the velocity itself. The controller target input r(t) was the desired velocity. The plant signal, y(t), which was the current velocity, was calculated by recording the change in encoder position from the previous time interval and converting "encoder counts per second" into meters per second. The error e(t) was calculated by subtracting e(t) = r(t) - y(t), and this signal was input into the proportional, integral, and derivative

calculations. Then each of those signals were summed up to generate the control signal u(t) to control the plant (i.e the motor). After implementing it and spending a lot of time trying to tune the proportional, integral, and derivative components without any good success, Levi informed us that the better way to implement velocity control was by using the velocity to set a ramped target position, and implement the PID controller on that target position. In this case, the desired velocity was used to calculate the target position r(t) using position = velocity * time, so by recording the time since the last loop, then the desired change in position could be calculated. In this controller, all positions were calculated as encoder positions, so this desired target position was converted from meters to encoder count, again based on the gear ratio. The plant signal y(t) was the measured encoder position. Again, error e(t) = r(t) - y(t), and this was the signal input to the proportional, integral, and derivative calculations. The components were scaled by their respective coefficients, and the sum of these signals, u(t) was used as the PWM value used to drive the motors. Each motor had its own PID control signal u(t) so each motor would be controlled on its own. The loop was executed with a frequency of about 100 Hz.

There were some issues with the first implementation of this controller where the response position would shoot up really fast past the ramping target position, as shown in Figure # below, which shows the measured current encoder position readings for both motors (i.e. M1 and M2), compared to the ramping target encoder position. After spending a lot of time trying to fix this, Gillian finally decided to reach out to Zach to see if he might be able to help since he was the first to have a really well-implemented PID controller.

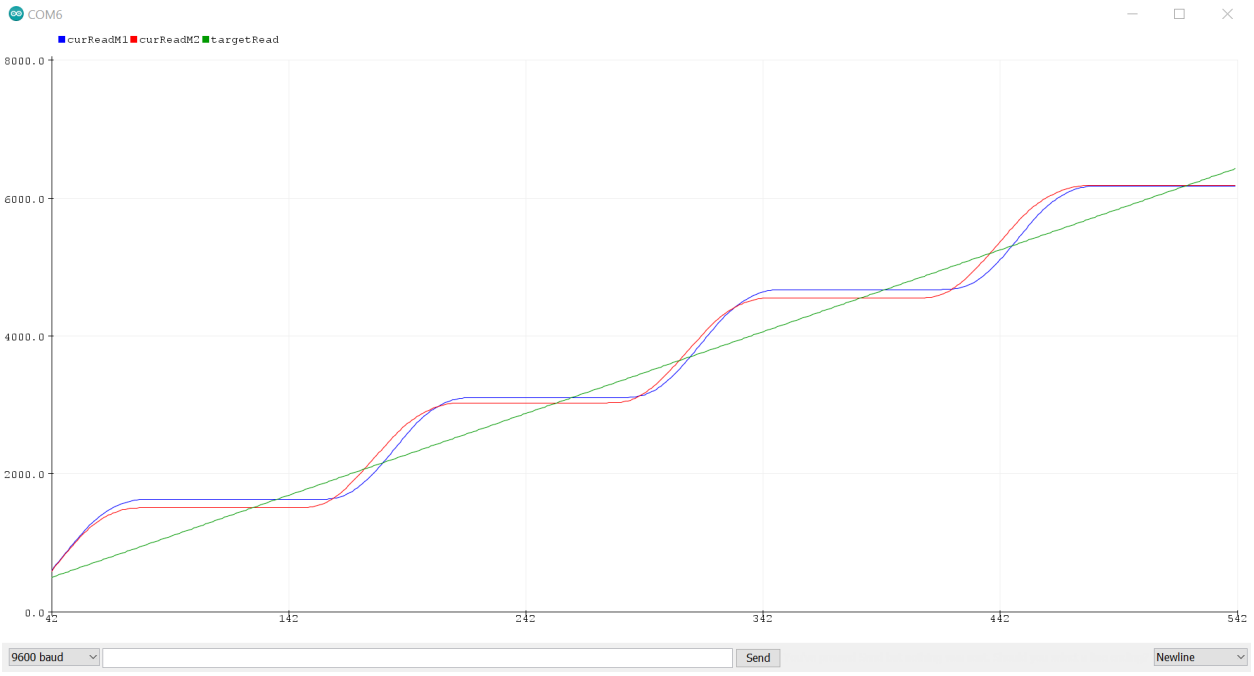


Figure 2. Target (green) and measured (blue and red) encoder responses from both motors controlled by the incorrect positional set-point PID controller.

It turned out that the issue was Gillian had added the controller signal output to the error signal, and used that to set the PWM, rather than just using the PID controller signal output. This meant that the PWM was being increased way too fast past the desired ramped position. After correcting that error, the behavior was much better and the coefficients were tuned pretty well, as shown in Figure #.

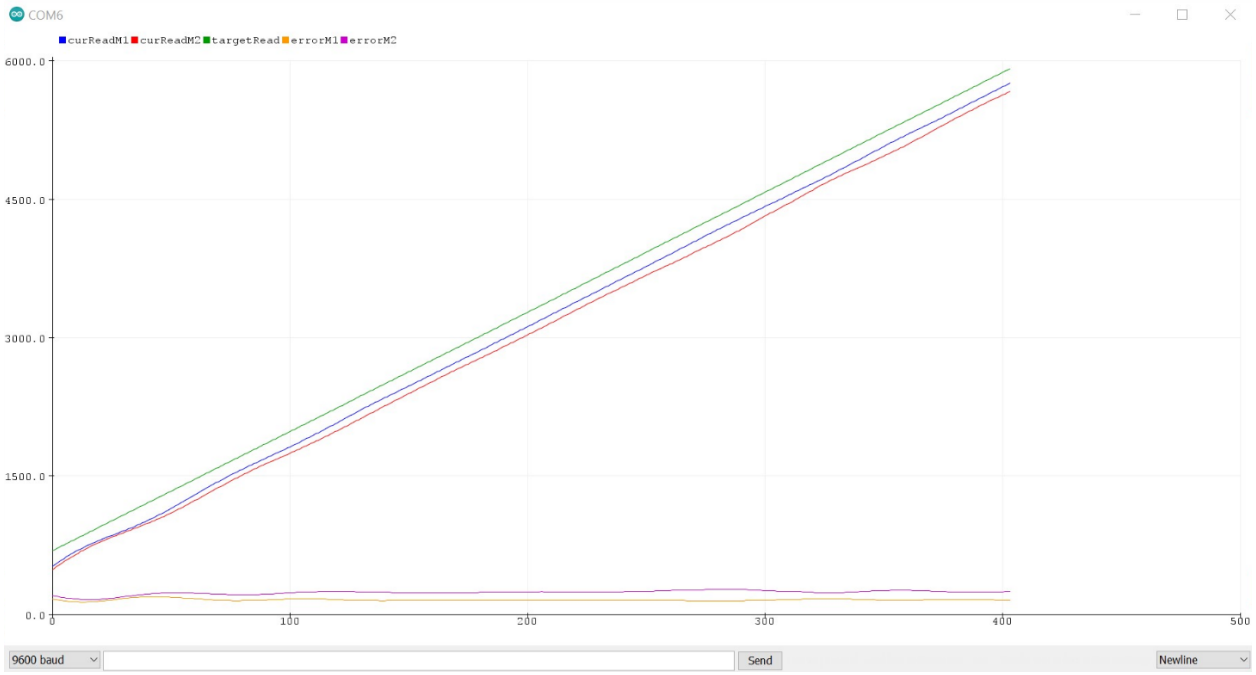


Figure 3. Target (green) and measured (blue and red) encoder responses from both motors controlled by the correct positional set-point PID controller.

The controller was initially tuned on the kitchen floor of Caitlin and Gillian's apartment, and the robot travelled in a very linear path at what looked to be a pretty constant velocity. However, once we were in the lab that week and we tested it on the lab floor and on the foam mats, the robot would curve off to the left and then back to the right to straighten out again, so it would be travelling on a straight path but parallel to the original intended direction. This behavior was very much not desirable. It was only at this point that Gillian remembered that implementing line following capabilities might be helpful (rather than counting solely on the PID controller to always hold the robot driving straight). At the beginning of the semester, we'd implemented line following on all of our robots, where it would just continually drive forward to follow a line. This was implemented by checking which sensor(s) on the IR sensor bar were lit up, where if the ADC value was lower than some threshold, this indicated the white of the tape whereas everything else was the black foam. This code was reused here to create a boolean array corresponding to the sensor numbers so that the lit sensors could be checked in each loop. Then, according to where the tape was, some small amount was added/subtracted from the PWM value generated by the PID controller (i.e. drive the left motor faster and the right motor slower to turn left when the sensor bar indicates the mouse if tilted off to the right, and vice versa). Though the

mouse would be kind of wiggly when it drove forward due to the PWM corrections for line following, it worked to ensure the mouse wouldn't lose the line.

Around a day before the previously mentioned line following stuff was added, Gillian also worked on turning, initially with the plan to do something quick (since this was probably at 2 weeks before the showcase so time was running out) so that meant just trying to drive the motors at some PWM to get a turn to happen. This was not going to be done using the gyroscope since that would've taken some time to figure out how to use and integrate, so the IR sensor bar was used. The idea was that once the sensor bar detected the middle sensor to be lit, then the turn would stop as that would mean the mouse was seeing a line. A small delay was added where if less than 0.2 seconds had elapsed since the turn command started, then the IR sensor bar check would be skipped. This helped with issues where because just fixed PWM values were being used to drive the motors, sometimes the turn would start very slowly so the mouse would end up detecting on the same original line that it was supposed to be turning from.

After getting this to work on one mouse, and then fixing the PID controller with the line following code as previously mentioned, Gillian realized that it would be very time consuming and annoying to figure out which PWM values to use for each motor on each mouse. So, because the straight line PID controller had already been implemented, the new idea was to drive each motor in opposite directions at the same constant speed. After turning the original PID controller into a function, the turns were relatively quickly implemented into their own functions. Other than some small logic issues that needed to be addressed, the most major issue was when the mouse would be approaching a corner or intersection but slightly turned to the side when it was in the middle of doing its line following correction. The mouse would then see the perpendicular tape at the intersection and continue on its slightly turned path because it would continue to detect along the middle IR sensor. This would then place the mouse off center when the next turn command was given, which would then screw up its positioning. To fix this, some code was added to check how many of the light bar sensors were currently lit; if more than 3 sensors were lit, this indicated that the mouse wasn't just seeing one piece of tape for a line. When this was the case, then the previously discussed line following PWM adjustments wouldn't be made; otherwise, they would. Other than this one issue, coding in turning capabilities went pretty smoothly.

While Caitlin was still working with video processing (see section 3.4 below), the hope was that we could get live intersection detection with distance calculation so at a certain point close to an intersection or corner, the drive forward function would be called with the calculated distance to the intersection. Thus, we needed a function that would drive the mouse forward for a desired distance. The original idea here was that we could drive the mouse for a certain amount of time (i.e. distance = velocity * time). It turned out that this was not a good idea because with the implemented line following corrections, then the time that the mouse spent doing the slight turns would stack up and it wouldn't drive far enough. To correct this, the new idea was that a target final encoder position would be calculated by converting distance to encoder counts. A flag variable was added to control the loop. In every loop before each motor was driven at the new PWM value, the current encoder position was checked against the target final position, and

if it was greater than or equal, then the flag would be set to high so that the loop would be exited, and then the motors were set to stop.

After we gave up on video processing, since the camera needed the mouse to be stopped in order to take pictures, we decided we wanted the mouse to drive only on the grid that the maze was built on, which was 15 cm units (i.e. corners and intersections would only occur on a 15 cm grid). When testing out the accuracy of driving 15 cm at a time, we noticed that the mouse would always overshoot, so we added in a scaling factor to the target final position (i.e. the mouse would drive 16 cm on a 15 cm stretch, so we scaled down the target final position by 15/16). There were still some inaccuracies sometimes, possibly due to the motors slipping, so we realized we should have some way to adjust for the distances, especially because the turns were not great when the mouse wasn't perfectly aligned with the center of the corner right between the two wheels. With Caitlin being able to add distance detection on the intersections, then this function became very useful as then the corrected distance needed to be travelled could be specified in the function call.

Finally, to close out this section, we wanted to point out our code organization regarding the controls code. When implementing line detection and following, each mouse had its own threshold value for distinguishing between white tape vs. black foam. Each mouse also had its own 15 cm scaling value. Since each mouse would need to have its own RF sketch uploaded, then it made sense to make a header file with the forward and turning functions in it for each mouse as well so that the functions were abstracted from the main RF sketch running on each mouse, making it easier to read.

3.3 Camera Software

The image processing software was written in Python. The basic algorithm is as follows. The base of the code relies on Canny edge detection and the regular Hough line transform from OpenCV. Once all lines in the image are found, some form of selection criteria is used to identify the lines of interest. Finally, calculations performed on these lines identify the center of the intersection and its distance from the mouse, and the algorithm was eventually developed to classify the type of intersection seen.

The earliest version of the image processing algorithm involved identifying the edges of a simple straight tape path by selecting two lines: the most positive and most negative slopes. The center line between these two edges could then be calculated, and could be used as a real-time indication of the mouse's position over the line. If the mouse was centered straight over the tape, the center line would be vertical and halfway across the image. Deviations in slope and left or right shifts could be used to control the motors according to how the mouse should adjust to achieve a centered position.

The next version added a similar selection algorithm to identify the horizontal edges of an intersecting path. Instead of looking for the most positive and most negative slopes, the slopes closest to zero were taken as the edges of the intersection. Once the four edges of interest were

identified, the points at each corner of the tape intersection could be calculated. From these points, the center of the tape intersection could be calculated. There were issues with extraneous lines from objects in the background being detected, so image cropping was also added to reduce the presence of those unwanted edges. However, this form of intersection detection did not work if the intersection was close to the mouse, as the wide frame of view from the camera meant that as lines moved away from the center they would curve, preventing the Hough line transform from working properly.

To address the distortion, each camera was calibrated to remove the curving and ensure that straight lines appeared straight in the image. This involved printing out a standard checkerboard and taking numerous pictures of the board at various positions and angles in the camera's frame of view. Using these images, OpenCV was able to generate matrices that could then be used to undistort images. As can be seen below in Figure 4a vs 4b, the calibration did not completely straighten out all of the lines, especially those at the edges of the image, but the integrity of the original straight lines is still significantly improved. Figure 4b also shows that calibration and undistortion introduced additional black areas, but these were easily removed by adjusting the cropping that was already being done.

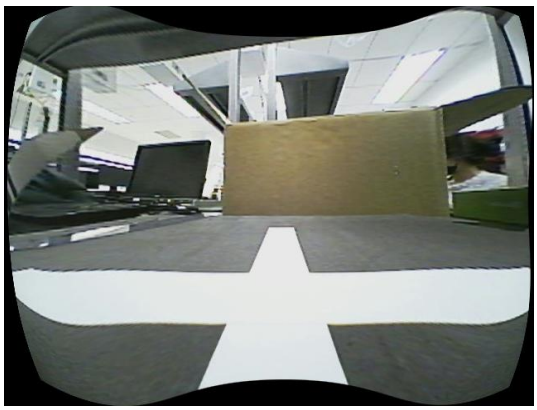
a)  b) 

Figure 4. a) Raw image taken from camera. b) Same image after calibration and undistortion.

Intersection classification was then added. Since the maze is composed of a predetermined set of possible intersections, it was possible to determine the intersection type by sampling points above, below, left, and right of the center of the intersection. While the algorithm worked for some cases, the classification failed too often because of a failure to properly identify the lines of interest, usually the horizontal lines. This would cause an incorrect calculation of the center point of the intersection, resulting in the wrong points being sampled and an incorrect determination of the intersection type. An example is shown below in Figure 5. The algorithm has identified the edges of the horizontal path as being nearly the same line, when one edge should be located on the bottom edge of the horizontal piece of tape.

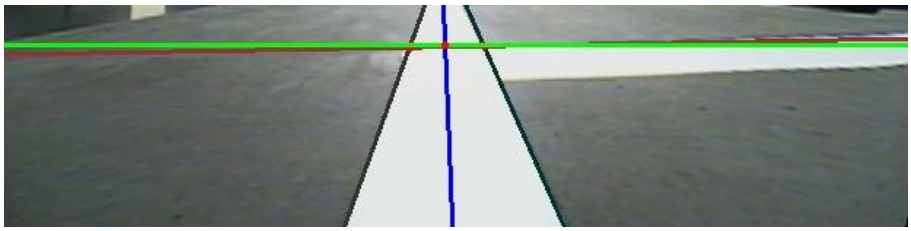


Figure 5. Example of failed horizontal edge identification.

Through strategic Googling, Daniel found that the HoughLines function returns lines in order of confidence. So instead of simply looking for the most vertical and most horizontal lines, the algorithm was modified to traverse through the returned array of lines and save the first two vertical lines and the first two horizontal lines encountered. Ideally, all four lines would be detected, allowing for accurate calculation of the center of the intersection and classification. As the maze was not guaranteed to have an intersection every 15 cm, a case was added to handle the calculation of the "center" and the classification if no horizontal lines were detected. Another case was also added for a dead end, or if Canny and Hough Lines failed to recognize one of the horizontal edges, both of which would identify only one horizontal line. This change to the algorithm significantly improved robustness, allowing intersections to be detected from more angles and varying positions. Another addition on top of this was to calculate the "horizontal" midline between the edges of the intersecting path, and use that as the reference when picking the left and right sample points. Previously, these points were based on the y-coordinate of the intersection center which meant that when the mouse wasn't centered on the tape, the mat might be sampled instead of the tape. The vertical sample points were left alone because the mouse was rarely far enough from the tape for incorrect sampling to happen for those points. An example of intersection identification with all of these improvements is shown below in Figure 6.

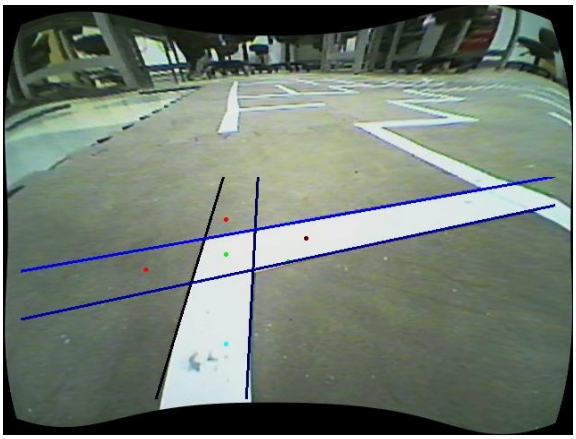


Figure 6. Detected edges and sample points when the mouse is positioned off center.

The final feature of the algorithm was the calculation of the distance from the mouse to the center of the detected intersection. For simplicity, distance was calculated exclusively with

respect to the y-coordinate. Tape pieces were set various distances away from the mouse, the image was undistorted and cropped, and probabilistic Hough Lines was run to get the y-coordinate of each edge. Based on the y-coordinate and the known actual distance of each edge from the mouse, a curve, which ended up being an exponential function, could be fit to the data. Since the camera calibration and undistortion matrices are unique to each camera, and each camera's mount is slightly different, this process was repeated for each camera to get a unique distance equation for each.

Despite these improvements, the image processing was not fast or reliable enough for constant live video analysis. The time necessary for the program to take the picture, run Hough lines, and perform the calculations and analysis meant that the intersection classification would take so long that the mouse would have been way past where the image was taken. The camera quality was also a bit unreliable in that every once in a while the image would glitch out with rainbow colors, weird lighting, or bits of static. So instead of the original plan to rely only on live video analysis for line following and mapping, we decided to make the mouse stop, take a few pictures, and perform the intersection identification on each one.

For integration with the operating system software, all of the camera software was changed into a set of three functions. One function initialized the camera by opening the video feed. Another closed the video and destroyed any open windows. The last contained the bulk of the processing code, and would return a tuple of an int representing the type of intersection seen, and a float for the distance to the intersection in cm.

3.4 Communications Software

Communications can be broken down into two main components: RF communications and server communications. We used RF communications to send movement commands from our laptop to the mouse. Specifically, we were exchanging packets with two integers. One integer was used to send a direction command (e.g. 1 is forward, 2 is turn left, etc.) and the other integer was used to send a distance. These integers were forwarded to the PID controller which used these values to control the next movement of the mouse. After movement was completed, the mouse sent a reply packet to the laptop. The reply packet was created by setting both the command and distance variables to the reply value. The reply value is initialized to 100 and is incremented by 100 after a packet has been received. This is done to send a unique reply to each command from the laptop.

The RF communications code was created by modifying the nRF24 two way comms example from the notes and examples repo. A major change we made to the example code is changing the direction of communication. The example was made so that the mouse could send packets to the Jetson. The Jetson would receive these packets and immediately send a reply. For our server to send commands to the mouse, we needed the opposite behavior. We modified the example by copying the mouse code to the Jetson file and vice versa. Doing this caused a lot of problems. Many of the problems we faced were hardware issues that are listed in section 3.1. The

first major software problem was the mouse was unable to receive commands. Many times when sending commands to the mouse, the mouse would not receive anything but the Jetson would read a reply packet of (0, 0). This was very puzzling behavior because the RF modules are coded to send a series of hexadecimal numbers before the packet. These hexadecimal numbers are called the header and are used to figure out when to start reading the packet in the stream of data from the RF module. For the Jetson to read the reply packet, it must first read the header then it will read the following data and interpret it as the reply packet. This is why the behavior was so puzzling. The mouse did not receive any packets but it must have sent the header because the Jetson was able to read it. This problem is still not fully understood. However, we found a solution that mostly solved this issue. When starting RF communications, we start by sending a packet to the mouse. The Jetson or laptop then tries to continuously read a reply from the mouse. If we receive a packet of (0, 0) for more than 10 seconds, we then send a command again to the mouse. We continue this process until we receive a reply packet of (50, 50). After this initialization code, our RF communications are mostly smooth and the mouse is able to read commands from the laptop.

Another problem we faced with RF communication is that sometimes packets would drop and not reach the mouse. This was very problematic because one dropped packet would cause the mouse to miss a movement command without the central server knowing. This would cause the server's mouse position and the actual mouse position to be out of sync causing the mapping of the maze to be ruined. We tried to fix this issue by verifying the replies sent from the mouse. Since we send unique reply packets for each command, we can verify the unique reply was received before sending a new command. Unfortunately, we ran into many issues with receiving the correct reply. When sending a series of commands to the mouse, the laptop would receive some repeat replies and would drop some replies altogether. For example, (100, 100), (100, 100), (200, 200), (400, 400) could be the replies that the laptop receives after sending the mouse four commands. The expected set of replies is (100, 100), (200, 200), (300, 300), and (400, 400). This means that the reply (100, 100) was repeated and the (300, 300) packet was unsuccessfully delivered to the laptop. Because of this behavior, we were unable to use these replies to verify that the mouse received the command. This problem reminded us of the Two Generals' Problem which does not have a clear solution. One work around we found was to add many sleep() functions in our code. If we increased the time between commands being sent to the mouse, we found that it decreased the number of dropped packets.

For server communications, we used HTTP requests to communicate between our laptop and the AWS server. The code for the server was written in Python using the flask framework. The AWS server was created using Elastic Beanstalk. Using the Elastic Beanstalk command line interface, we uploaded the Python flask code. Elastic Beanstalk created an EC2 instance that ran our code and outputted a URL for us to use to communicate with the server. The Python flask code was used to define functions to associate with certain routes. A route is the text that follows a URL. For example, in 'example.com/help', the URL would be 'example.com' and the route is '/help'. The flask code allowed us to write functions for the server to run when a route was requested by our laptop. The two main routes of our server are '/start' and '/coords'. Our laptop would send a POST request to one of these routes. A POST request allows our laptop to send

information to the server in the form of JSON objects. To reply to this request, the server would run the associated function and would return a set of commands for the mouse to follow. This server reply would also be in the form of a JSON object.

3.5 Operating Systems/Source Code

The server stored maze information in the form of a graph. Each node of the graph would represent an intersection in the maze. Each node would store the intersection's coordinates, the type of intersection, and references to other nodes in the graph. The server also stored information on each of the mice. It stored each mouse's current node, current coordinates, and current heading. All of this information was used to build the maze graph and to give directions to the mouse.

The '/start' route was only used at the beginning of each run. This would initialize the server variables and create a starting node in the graph. Each entrance to the maze was a straight line, so the start() function would always return the forward command to the mouse. The '/coords' route was much more complex. The '/coords' route was requested after the mouse performed some movement. The route is used to create a new node for the mouse's current position and to calculate the mouse's next moves. The '/coords' route takes a POST request that sends the type of intersection that the camera sees. The function begins by determining the intersection type when looking at it facing north. This is the intersection type stored in the nodes. For example if the mouse is faced east and sees a ┬ intersection, this would be stored as a ┤ intersection. The function creates a new node with the current coordinates of the mouse and the type of intersection that we found. This new node is connected to the node that the mouse was previously at. The function then checks if the maze has been fully explored. It does this by iterating through each node in the graph and checking if the required node references exist. For example a ┼ intersection should have all north, east, south, and west node references. If the maze is fully explored, the function will run breadth-first search to find and return the directions to the maze end node. If the maze is not fully explored, then the function will check if the current node is fully explored. If so, the function will run breadth-first search to find and return the directions to the closest node that is not fully explored. If the node is not fully explored, it will return a command to explore one of the unexplored regions of the node preferring left. right, then forward. This decision making algorithm was designed to solve our challenge of mapping the entire maze.

Our decision making algorithm seemed to run well on demo day. However, getting our algorithm to that state was a difficult task. There were many edge cases to handle and it took us a long time to debug. One cause of these issues was because the graph data structure only held intersections. We did not create nodes for units of straight lines on the maze. This was because originally, the robot was not going to move in 15 cm units. We had planned for the robot to move forward until it reached the next intersection. Storing only intersections created many edge cases when using breadth-first search on the graph. Although debugging eventually was enough to fix

these problems, storing straight lines in the graph would have avoided many of our algorithm issues.

Through RF and server communications we were able to communicate between the server and laptop and between the laptop and mouse. To combine these two forms of communication, we created a python module called Middleman. The Middleman module was also needed because we ran the image processing code on the laptop. The algorithm for Middleman goes as follows. First, initialization code for the server, image processing, and RF communications are run. Next, a POST request is sent to the '/start' route. The resulting movement command is forwarded to the mouse. Next is the main loop of the algorithm. First, we run the image processing code to determine the intersection type in front of the mouse. This information is sent to the server through a POST request to the '/coords' route. When the server reply is received, the algorithm will iterate through the list of commands in the reply and send each command to the mouse one-by-one. This loop is repeated until the maze has been solved.

4. EXECUTION/DEMO RESULT

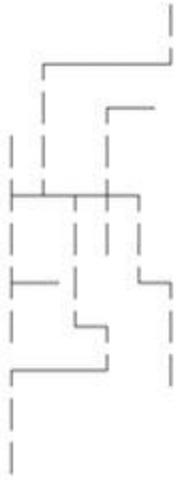
On demo day, we knew that there was no way that we would be able to achieve networking, since as previously discussed, we only had one RF module that would even connect. Our new goal was to get the one working mouse to fully map the maze and then find the end. First, the mouse should reach the end of the maze. Once it does so, if there are parts of the maze that were unvisited and unmapped, it should return to map those parts. Once the maze is fully mapped, the mouse should take the shortest path from wherever it is located after mapping to reach the end.

The four final mazes fell into two categories: 15 cm grid and 30 cm grid. Since our entire control scheme assumed that the grid size was 15 cm with added corrections using the camera and image processing for distance calculations, the mouse had the capability to handle both cases. In testing we had some errors when the mouse had to travel long stretches of only straight path (no intersections), as the mouse would sometimes not travel the 15 cm it was supposed to, so we were concerned about how the 30 cm mazes would be more likely to have longer straightaways where the driving distance wouldn't be corrected for by the camera. Thus, we decided to run the mouse on the 15 cm mazes.

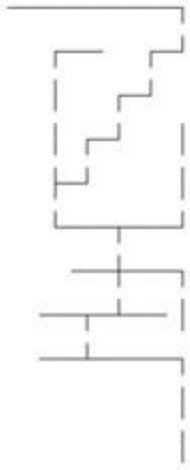
We ran the mouse on both 15 cm mazes. In both demos, the mouse was successful in accomplishing our goal. It reached the end, turned around to map the parts of the maze that were missed, and then traveled to the end again after mapping.

We also attempted a more challenging version of one of the 15 cm mazes, where Khoa added more tape pieces to create closed loops. We believe our mapping algorithm would have been able to handle the loops, but unfortunately the camera's image processing failed and gave an error, so the mouse stopped moving and ended the program. We would have liked to try again after recalibrating the camera, however we ran out of time during that lab session and didn't have the opportunity to.

1st run (starting point was at bottom left of the generated map below):
Video link



2nd run (starting point was at the bottom right of the generated map below):
Video link



5. SUMMARY & FUTURE SUGGESTIONS

Our original plan was to rely on image processing in real time for mapping and controlling the mouse. We wanted to be able to identify an upcoming intersection and track its distance as the mouse travelled towards it. We also wanted to use the paths identified in the image as the references for both turning and driving straight. Our contingency plan in case that did not work out was to only use the camera for intersection classification, and to use the IR array and PID controller for driving and turning. Our additional contingency plan in case the image processing really wasn't working was to purely rely on the IR array. We're very glad to have been able to get the various parts working well enough so that we did not need to go for the contingency contingency plan. As can be seen from previous parts of the paper though, there were a lot of pieces to work on in, as well as a lot of work needed for bringing everything together into a working product.

The second to last week before demo day, we barely had the separate parts working. The image processing needed work, the PID controller had issues, and we had barely touched the RF. The last lab before demo day, we all decided to basically put all of our time into getting the mice working. That Friday, we worked in the lab for about 12 hours, staying extremely late with Khoa and Levi working on the RF. On Saturday we put in about 8 hours, and were able to have our first successful test with all of the pieces combined. On Sunday we put in another 8 hours, mostly working on making improvements to fix the various bugs that came up when testing on the practice mazes. On Monday we worked another 13-14 hours fixing bugs again. Even though time pressure is a great motivator, we definitely suggest really working to follow the guide deadlines from the course introduction, as that spreads out the work more evenly and continuous progress can be made throughout the semester, rather than just the week before the demo.

We would have liked to run the mouse on the maze with loops again to see if it would solve properly without a failure by the camera. Depending on the results of that, future work may involve debugging issues with mapping and maze solving. We also would have liked to try networking with at least one more mouse. The main challenge there would be to get another RF module working. Once that is figured out, then there would obviously need to be testing and some modifications to the central control code, but we don't think this side of the code should be too difficult.

The main thing we learned was that the RF module is absolutely perfect, and was the perfect accompaniment to our absolutely perfect mice.

If we were to do this again, or make suggestions to future students doing a similar project, we would caution against using the camera and RF modules. While our project turned out to be very cool and we were satisfied, it took a lot of work to figure out how to use them and what was going wrong, and we were only able to get one to work. Even then, the RF still didn't function as we expected it to and was unreliable enough that if we built in checks to try and verify that communications were happening as we wanted them to, it would turn into something like the two generals problem. The camera was also a bit unreliable for properly detecting lines. Sometimes the connection would be bad and result in static or weird colors that would mess with edge detection, or the wrong edges would be selected, or the lighting in

the image would make it so that there wasn't enough contrast between the white tape and dark mat for an edge to show up properly.

We would also highly recommend using Elastic Beanstalk to set up any AWS servers. Daniel tried spinning up an EC2 instance manually without the help of Elastic Beanstalk. He ran into many issues with accessing the server from his computer. Elastic Beanstalk eased the process by doing most of the work. After uploading the code using the Elastic Beanstalk command line interface, we received a URL to access the server. AWS also has many tutorials on how to use Elastic Beanstalk for many languages and frameworks other than Python flask.

6. FEEDBACK

Use the RF modules, they’re perfect.

We are so thankful for Levi and Khoa. Literally couldn’t have done it without you.

7. REFERENCES

Github folder with final code used on the mouse:
https://github.com/UMD-ENEE408I/ENEE408I_FALL2021_Team3/tree/main/final_code/daniel