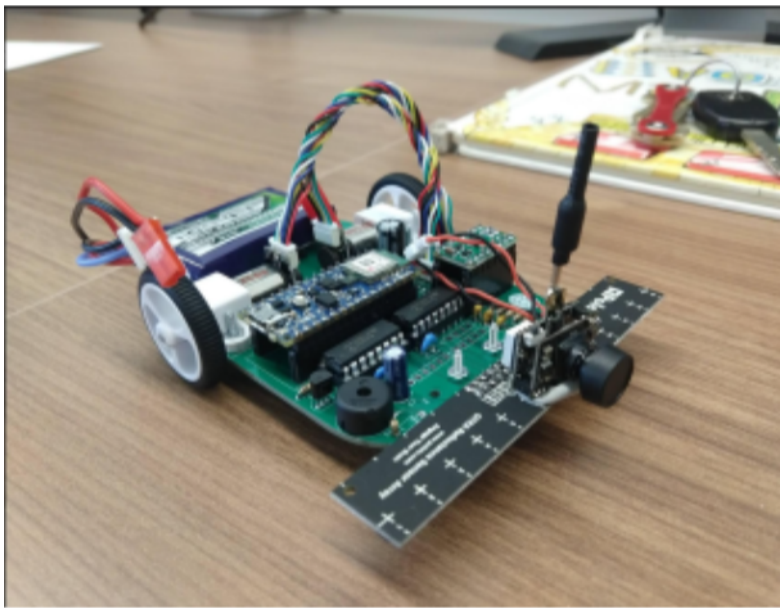
Final Report for

Team 1



Submitted to:

Prof. Shihab Shamma

ENEE 408I: Capstone Design Project: Autonomous Control of Interacting Robots
Spring 2023
Department of Electrical & Computer Engineering
2410 A.V. Williams Building
University of Maryland
College Park, MD 20742

Date: 5/19/2023
Prepared by:

Nicholas Boomsma (nboomsma@umd.edu)
Dilan Cruz-Flores (dilancruzflores@gmail.com)
Adrian Fernandez (afernan8@terpmail.umd.edu)

TABLE OF CONTENTS

1. INTRODUCTION....3
1.1. Challenge Overview....3
1.2. Proposed Solutions & Teamwork Distribution....3
2. APPROACH....6
2.1. Hardware Overview....6
2.2. Software Overview....7
3. PROCEDURE....6
3.1. Hardware Testing Progress....9
3.2. Software Testing Progress – Control & Navigation....9
i. Tracker Navigation....9
ii. Evader Navigation....11
3.3. Software Testing Progress - Communication....13
3.4. Software Testing Progress – Camera Detection....15
3.5. Software Testing Progress – Sound Data....26
4. EXECUTION/DEMO RESULT....29
5. SUMMARY, FUTURE SUGGESTIONS, & FEEDBACK....30
6. REFERENCES....31

1. INTRODUCTION

1.1. Challenge Overview

The main focus of this course is to work with your teammates and complete a challenge which involves sound localization through the use of robots. Specifically, the course challenge involves using robots to track a sound source through the use of microphones. This requires implementing different control algorithms to have the robots successfully track the sound source autonomously. The course focuses heavily on using teamwork to achieve the main challenge of the course. This is reflective of our future as engineers since we will have to work in teams constantly to achieve a variety of technical challenges.

We decided that our challenge would be to play a game of "Marco Polo" with three robots: two "Trackers" and one "Evader". The Tracker robots would each have a microphone connected to them while the Evader robot would have a speaker which would play a sound source. The goal of the game is for the Trackers to use sound localization techniques to hone in on the Evader while the latter is attempting to avoid being caught by the former. In addition, all three robots would be limited to a small arena which would be constructed using Apriltags. There would be seven Apriltags in total: four which would be used to create the arena and three which would be placed on the three robots to calculate their position inside the arena. This would be executed through the use of a camera placed in on a ceiling pointing downwards, allowing us to limit the movement of all three robots to stay inside of the arena.

First and foremost, we expect to learn how to manage a daunting project as a team in order to be as efficient with our time as possible. From a technical point of view, we expect to learn about sound recording, filtering, and analyzing in order to even begin implementing the sound localization algorithm. We also expect to learn how networking works when having the robots and Jetson communicate with each other. Finally, we will learn about how to utilize OpenCV in conjunction with Apriltags. Overall, we believe the most important lesson this class will teach us involves teamwork. Learning how to work together as a team to tackle this challenging problem will be invaluable in our future careers as engineers. It will teach us about time management, communication between teammates, and how to use each other's strengths to ensure each teammate works on his specialty.

1.2. Proposed Solutions & Teamwork Distribution

Before coming up with the algorithms as to how the sound localization and evasion tactics would work, we first had to figure out what needed to be done to get to that portion of testing. We determined that the robot's ability to receive commands from the Jetson, record noise, and maneuver around the arena would serve as the building blocks for completing the challenge. Therefore, we delegated these three tasks among ourselves. As a team, we brainstormed how we would tackle each part. The communication was fairly simple, as we were given sample code for sending and receiving data in the form of UDP packets: WifiUDPClient (for the robots) and udp_server (for the Jetson). When it came to maneuvering around the arena, we knew the best way would be to utilize the OpenCV library and Apriltags to analyze live camera footage and track the locations of each robot. Originally, we were planning on setting up Apriltags along the perimeter of the arena. That way, using the cameras that came with the robots, we could recognize the Apriltag(s) that are in view of each robot, determine the distance

and angle they are away from each tag, and calculate their location based on that. However, there were multiple issues with this. First, the cameras given to us were not of very good quality to the point where they had trouble reading an Apriltag from a yard away. Second, there are many instances in which an Apriltag could not be in frame for a robot, as in if they are near the edge of the arena or their view of a tag is blocked by another robot. Third, the calculations required to figure out the locations of the robots would be difficult and require a lot of time and effort to complete. Therefore, we decided on a better solution: use a higher quality camera and tape it to the ceiling in order to look down on the arena. That way, all Apriltags could be easily viewed and recorded, we would only need four tags to mark each corner of the arena, and we could avoid the complicated calculations when determining each robot's location.

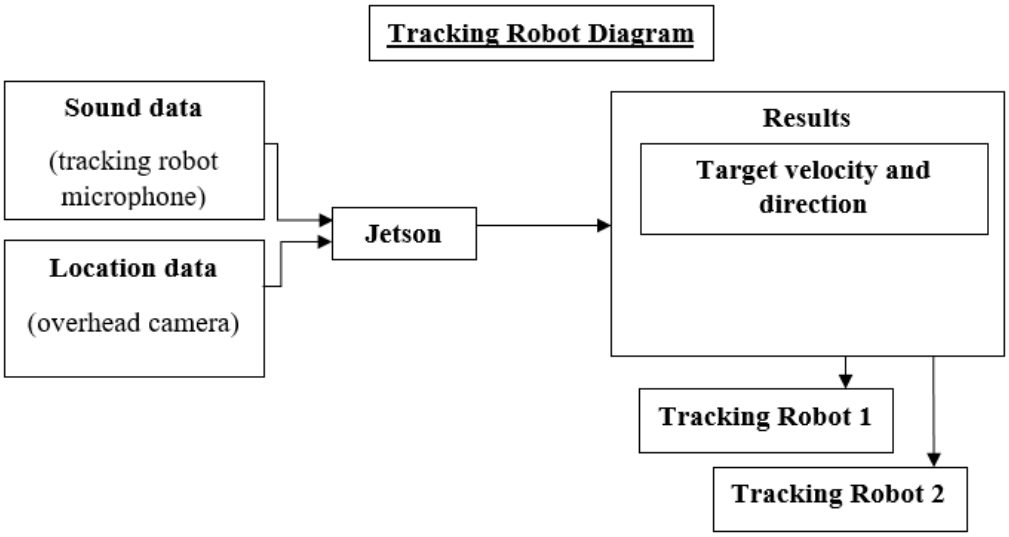
Initially, we discussed using a combination of time delays and sound intensity. During one of the lectures, Prof. Shamma explained how, given a specific delay in time, one could know that the sound source must be located along a parabolic line. We were planning on using that time delay approach in tandem with taking the sound intensity and comparing it with a list of calibration intensities from various distances to estimate where the sound source is located. However, we decided against this in the beginning since we wanted to implement a sound localization approach which used intensity. With the help of Levi we decided to try and implement a version of extremum-seeking control. This involved manipulating the parameter phi from the Trajectory_Tracking code and setting it equal to a sinusoid of the form phi = A*sin(wt-b) - c*(correlation_variable). This ensured the robot would move as a sinusoid and would change direction depending on where the sound intensity was located. However we faced difficulties implementing this and moved on to a second approach. The second approach is detailed in the Software Testing Progress - Sound Data section. In summary, the second approach also relied on sound intensity, but it didn't work since we found the microphones were not directional at all.

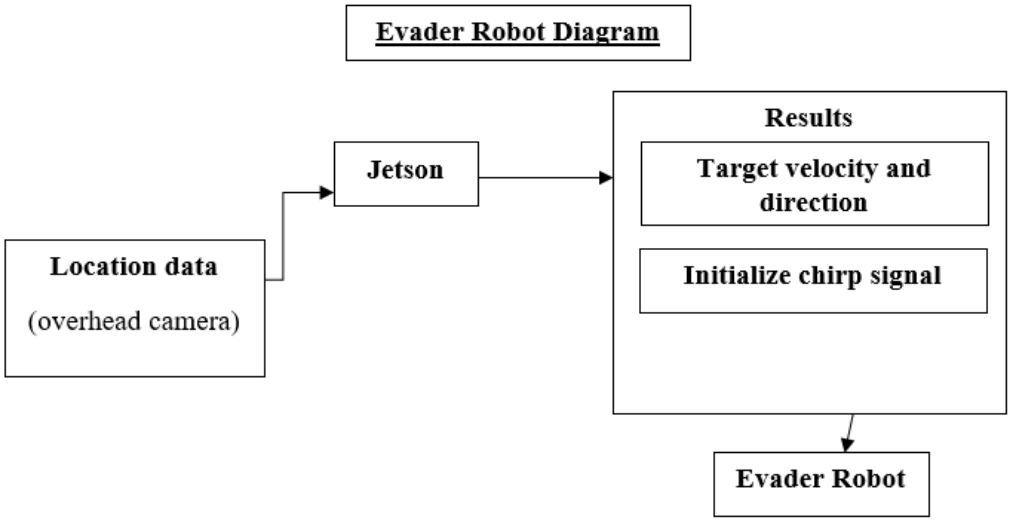
Finally, this led us to our third and final approach, the time delay approach. This was the only sound localization approach which worked as we expected. This approach involved two microphones and a single sound source. The two microphones would be placed on the two tracker robots and the sound source would be placed on the evader robot. The tracker robots would be at different distances from the sound source (evader robot) so there would be a difference in time from when the two microphones received the sound signal. This time delay was calculated by correlating the two different sound signals recorded by the two microphones. We used the sample Python script published by Levi which was invaluable in this implementation. Overall, the third approach was successful in calculating a correct time delay between the two microphones. This approach is covered in more detail in the Software Testing Progress - Sound Data section.

There was an additional task we had to develop: the evasion tactic code. This was the last part we had to worry about, as making sure the sound localization worked in the first place was top priority. With the deadline approaching quickly and numerous other parts not near completion, the development of the evasion algorithm had an emphasis on simplicity. We did not want to spend more time than needed on something we may have never even gotten to use. However, we also wanted an algorithm based around making intelligent and logical decisions rather than simply following a rigid, predetermined loop. So we came up with a solution to split the arena into four equal quadrants in a 2x2 arrangement. A better explanation is located below, but the gist of it is that the Evader's goal was to always be in the quadrant diagonal to the

Tracker. The only exception was if the Tracker and Evader were both in the same quadrant, in which the Evader would move to the most convenient quadrant. At the time of its conception, it was assumed that we would have only one robot doing the sine-intensity approach, not two. When we decided to go with the time delay approach, in which both Trackers would be used, we decided that the evasion algorithm would only worry about the closest of the two.

As for the delegation of tasks, Nick was in charge of figuring out how to send commands between the Jetson and the robots. Adrian was in charge of understanding how to record, filter, and analyze sound. Dilan was in charge of utilizing OpenCV and Apriltags, which would be used for robot movement. Additionally, since the UDP packets were the most straightforward portion to implement (given the sample code) and took the least amount of time, Nick also worked on developing the evasion tactic code.





2. APPROACH

2.1. Hardware Overview

The components on one of our robots included:

1. One Heltec Wifi Kit 32 Microcontroller
2. One 460mAh Battery
3. Two 6V DC Gearbox Motors
4. One Micro Metal Encoder
5. One MPU6050 Accelerometer/Gyroscope
6. One Reflectance Sensor Array
7. One Bluetooth Microphone

The rest of the components included:

1. One HP Laptop
2. Two Bluetooth Microphone Receivers
3. One USB Webcam
4. Six Paper Apriltags
5. One Bluetooth Speaker

Overall, the majority of our components worked well throughout the semester with each component serving a different role. The two bluetooth microphones were mounted on our robots and they were used to record sound data necessary for the sound localization. The two microphone receivers were then connected to the HP laptop for the computer to receive the sound data and run calculations on it. After the calculations were done the data was sent to the robots through the Heltec Wifi Kit 32 in order for the robot to move towards the sound source. The sound source was composed of a 1kHz signal which was played through a bluetooth speaker connected to a phone. This was played in a chirp-like manner since the audio was not continuous. In addition, the USB webcam was connected to the laptop since it was used to record the Apriltag arena and determine the position of the different robots inside the arena. The six Apriltags were necessary for the webcam to determine both the edges of the arena and the position of the robots. In the end, every component was used to achieve our goal.

2.2. Software Overview

The majority of software development was done via Microsoft VSCode. VSCode is a robust interactive developer environment, or IDE That works on all operating systems including Windows, Mac OS, and Linux. Because of this versatility, it became a crucial tool for us to be able to work on the software half of the project as well as have libraries that would help us with both the processing of data collected by the sensors and the control of the robots. Specifically, we use the Platform IO plugin in VSCode to handle the Arduino microcontrollers that were on-board the robots. Initially, we were planning to use the Jetson NX computer to run all of our calculations for this project. However, we had some difficulties connecting the microphones reliably to the board, so we decided to switch to one of our laptops for our final presentation. Despite this, we still used the Jetson for some development of the software. This development was done mainly via Nano, the terminal based text editor.

The two main languages that we used in this project were Python and C++. The C++ language was used for the Arduino side of the project. That is to say, C++ was used to drive the motors on the robots as well as the networking necessary for the robots to both communicate with themselves and with the base station that would do all the calculations necessary based on the sensors. C++ is an object oriented programming language, which provides a clear structure to the program that allows code to be reused multiple times. Python, on the other hand, was used to run the calculations necessary for all of the sensors which included the webcam that was suspended above the arena and the microphones that were to be used for sound localization. Some Python libraries that we used included matplotlib, pupil_apriltags, opencv, numpy, as well as some proprietary files that we received at the beginning of the semester, namely trajectory_utility.

The final demonstration code can be found in our GitHub directory with the following link: https://github.com/UMD-ENEE408I/SPRING2023_Team1/tree/main/Final%20Final. The following table explains the different files within our final directory.

File	Description
Server_Cmd_Center.py	This Python file is the main script we used to integrate all aspects of our project. The file contains the networking code necessary to connect the robots to the computer. In addition, it contains all of the code necessary to run the camera with the apriltags. The file also calculates the time delay and executes the main loop which allows us to continuously scan the apriltag arena and keep calculating the different time delays between the robots.
Trajectory_Utility.py	This Python file contains the multiple functions we created to calculate the tracker robot headings and the evader robot headings. Although we weren't able to integrate most of this code, the work was done. In theory this script would've allowed us to calculate the paths the tracker robots needed to take in order to find the evader robot. At the same time the file calculated the path needed for the evader robot to escape.

sound_delay.py	This Python file contains the sound calculations necessary to obtain the time delay between two different microphones. The file implements a correlation algorithm and alongside PyAudio to analyze the sound data recorded by two different microphones. It compares these two sound signals to calculate which microphone received the sound first. This was applicable to our project since we used a 1kHz chirp signal as our sound source.
mouse_movement.cpp	This C++ file contains the code we uploaded to the two different robots in order to make them move. The file integrates the robot movement and the networking to the laptop. This was done by connecting the robot and the laptop through a Wifi network and sending the laptop calculations to the robot. The robot would then act on these values to force it to move.

3. PROCEDURE

3.1. Hardware Testing Progress

Overall, our hardware testing in regards to the robots went well. All three robots worked as expected and did not give us any major issues. However, we did encounter some robot issues when we were testing the second sound localization approach. One of the robots stopped working during the constant testing with this approach. The robot in question wouldn't turn on when it was connected to the battery. Whenever a robot was turned on the leftmost wheel spun rapidly, but this wasn't occurring anymore. We noticed a burnt smell and we believe the Heltec Wifi Kit 32 Microcontroller on the mouse burnt out. We brought this to the attention of Levi who quickly fixed the problem by replacing the failed component. This was the last of our robot troubles.

One important hardware test involving the Jetson failed and occurred very close to the date of the project demonstration. This test involved the integration of all the code we had written and in order to fully integrate the program. The sound localization code implementing the time delay calculation did not work at all on the Jetson. We don't know why this occurred, but we quickly switched over to the HP laptop on which all of the sound testing had been previously done. This eliminated any issues we faced and we were able to begin integration.

3.2. Software Testing Progress – Control & Navigation

We used the trajectory tracking code as a starting point for our navigation. The lemniscate code was deleted while the rest of the conversion code from target velocity/theta to motor control was left untouched. This allowed us to simply pass in the desired velocity (target_v) and theta (target_theta) to each robot from the Jetson using UDP packets. The only other value that would be passed from Jetson to the robot is the theta value given by the Apriltags (cam_theta). While we were using the gyroscope contained in each robot to determine the direction, we were worried that, over time, the accuracy of the robot's orientation would deteriorate. So we planned on comparing the gyroscope's current orientation to its actual orientation as seen by the overhead camera. If the robot's internal theta value was not equal to that of cam_theta by a certain threshold, the robot's current theta value will be set to the cam_theta value in order to maintain its directional accuracy. Unfortunately, this part of the project was not integrated due to time constraints.

i. Tracker Navigation

The calculations for the Trackers' velocity and directions are calculated in Trajectory_Utility.py. The two Tracker robots are to work together in finding the sound source through the use of time delays. The algorithm for how this works is as follows.

First, the Trackers listen for the sound source, then calculate which of the two heard it first. Next, an equation for a perpendicular line located at the midpoint between the two Trackers is calculated using the current positions of each Tracker. Then, a second equation will be developed, this time being a circle with the location of the Tracker who heard the sound first being the center while the distance between the Trackers is the radius. From there, we use GEKKO, a Python package that can solve systems of equations, to find the intersection point between the two equations. The farther Tracker will then move to that location while the closer

Tracker remains in its current position. The direction for the robots are calculated using the Signed Angle function found in the Trajectory Tracking file. The velocities for the Trackers are either 0.0 if it is the closer of the two or 0.1 if it is further than the other. Once the robots record another beep from the sound source, the calculations will be done again and new velocity and directional commands are sent to them. This can be seen in Figure 3.2.1

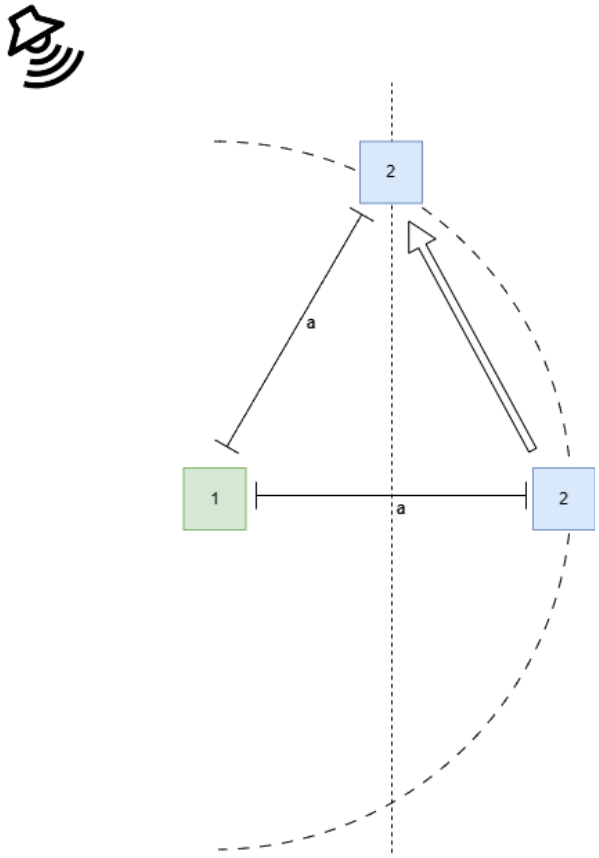


Figure 3.2.1 - Tracker Navigation (Single Iteration)

In this example, Tracker 1 is closer to the sound source than Tracker 2. Therefore, it is the latter that has to move to the new location. It is important to note that there are two locations in which the circle and midpoint line intersect. Since we knew the Trackers would start on one end of the arena while the Evader was located on the other end, we could assume that moving in the positive x-direction would be the optimal choice. So, we always chose the intersection with the greater x-position. Below is a diagram (Figure 3.2.2) that shows how this algorithm works when run multiple times. The numbers next to each arrow represents the number of iterations the algorithm has run through, starting at 1 and ending at 9.

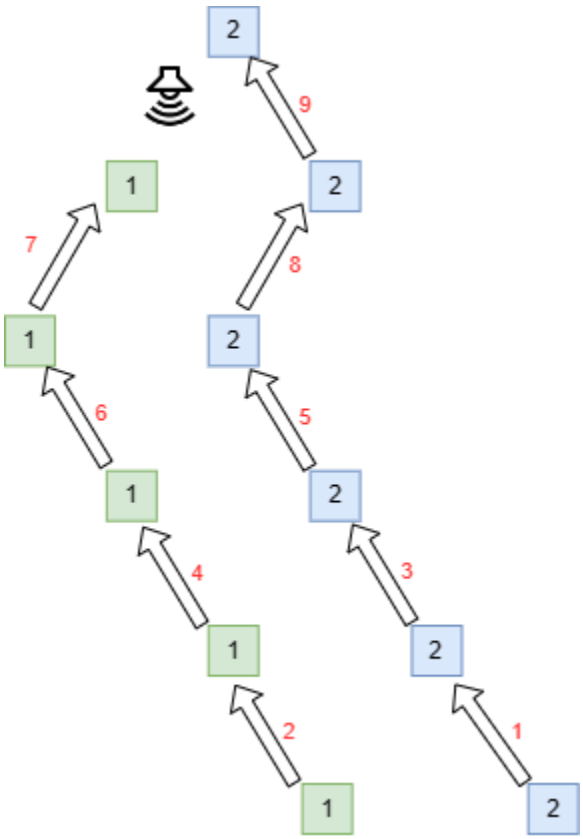


Figure 3.2.2 - Tracker Navigation (Multiple Iterations)

We were also anticipating that, if the robots reached the other side of the arena and went out of bounds, we could instead start choosing the point of intersection with the lower x-position. However, we did not get this far into the implementation, nor did we implement a way for the Trackers to narrow in on the sound source when they were within reach. We had a few ideas for this, most notable having the two robots be close enough that there's little space between them that the Evader could fit through. Another approach would have been for the further robot to move in an inward spiral around the closer Tracker in order to sweep the area.

ii. Evader Navigation

Since this challenge's main focus is Sound Localization, the Evasion tactic was less important to implement. However, we did not want the Evader robot to follow a predetermined track. Instead, we created a simple algorithm that allows the Evader to react to the Trackers' movements. The algorithm works as follows.

First, the arena is split into four quadrants using the four corners (represented by Apriltags). Next, we take the locations of all three robots and determine which of the Trackers is closer to the Evader. We will refer to this as Tracker 1. We determine the quadrant Tracker 1 and the Evader currently resides. From there, the decision making process begins. If Tracker 1 is in

the quadrant diagonal to that of the Evader, the Evader does not move. An example of this can be seen in Figure 3.2.3.

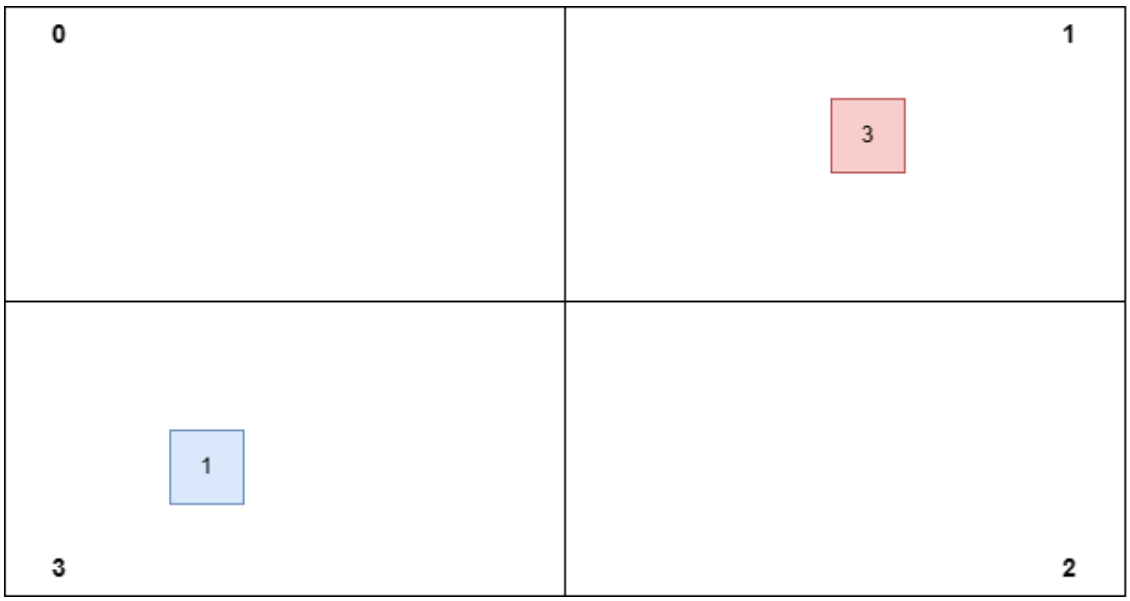


Figure 3.2.3 - Tracker 1 in quadrant diagonal to Evader

If Tracker 1's quadrant is contiguous to the Evader's quadrant, the Evader moves to the other contiguous quadrant not occupied by the Tracker. An example of this is seen in Figure 3.2.4.

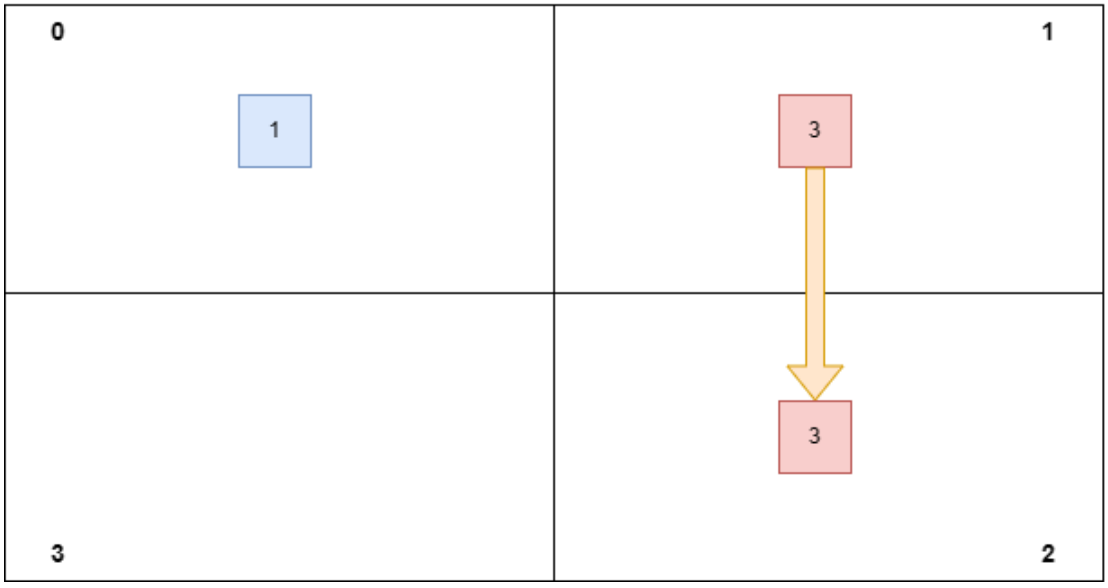


Figure 3.2.4 - Tracker 1 in contiguous quadrant to Evader

If Tracker 1 and the Evader are both in the same quadrant, the Evader moves towards the contiguous block that is safest. This is decided with a basic calculation. First, the distance from each robot to the quadrant border is calculated. Below is an example in which the robots are both in Quadrant 1 (Figure 3.2.5).

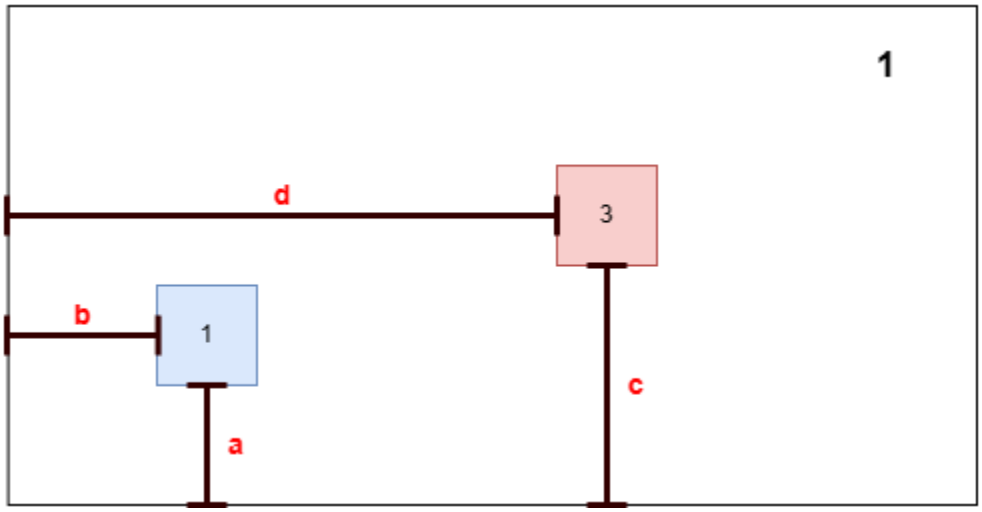


Figure 3.2.5 - Tracker 1 and the Evader are in the same quadrant

From there, we subtract the distances for Tracker 1 from the distance for Tracker 2. To simplify, we will say $m = d - b$ and $n = c - a$. If $m < n$, the Evader will move along the x-axis to the nearby quadrant that is to the left/right of its current position.. If $m \geq n$, the Evader will move along the y-axis to the nearby quadrant that is above/below its current position. In the case shown in Figure 3.2.5, $m \geq n$, so the best approach would be for the Evader to move down to Quadrant 2. Essentially, this approach allows the Evader to determine what quadrant could be reached in the shortest distance while also taking into account how quickly Tracker 1 could also get to the same quadrant. The lesser the value of m/n, the better.

As for where in particular the Evasion robot moves, we decided to make it simple and only move to the middle of each quadrant, which was calculated at the same time as the quadrant boundary lines. The velocity when it moves would be constant, but half as slow (0.05) compared to the Tracker robots, as we wanted to make sure it would be eventually caught. The direction/theta would be calculated using the Signed Angle function from Trajectory Tracking.

This Evasion tactic was thought up during the development stage when we assumed there would be only 1 Tracker using the Sound Intensity approach (described in Section 3.5). That is why, instead of taking into account the locations of both Trackers, this algorithm only worries about the closest one. Unfortunately, we did not have enough time to further develop and implement this approach.

3.3. Software Testing Progress – Communication

The communication between the robots and the HP laptop consisted of sending UDP packets using the Wifi network “408ITerps.” As a starting point, we were given the sample code

WifiUDPClient (for the robots) and udp_server (for the Jetson/HP Laptop). Testing the UDP packet sending/receiving worked as follows.

First, the WifiUDPClient code was copied into the mouse_movement code. We considered (and tried) to create a WifiUDPClient object that could be created in the mouse_movement code and handle all of the sending and receiving of documents in order to make the movement code less cluttered. However, this was too difficult to implement. All of the necessary variables, including the WiFiUDP object, network name/password, and UDP address/port, were instantiated. While the network name and password stayed consistent, the UDP address changed depending on which computer would be used as the server. Additionally, the UDP port would change depending on what robot was being programmed (Tracker1=3333, Tracker2=4444, Evader=5555). Then, within the setup function, the robot will connect to the server via wifi using the instantiated values. It will confirm this connection by sending a packet to the server. It then moves onto the loop function, in which it checks to see if a new packet has been sent by the server. If so, the robot will read the packet, flush the UDP object, and dissect the data from the packet, as it will contain the new cam_theta, target_theta, and target_v values.

We then set up the udp_server code to become Server_Cmd_Center. We did so by initializing the local IP and local port corresponding to the UDP address and port the mouse initializes. The server then initializes the UDP socket and binds the IP and port. With the socket set up, the server waits to receive a packet: the packet the robot creates during its setup process. When the server receives the packet, it will save the address from that robot. After that, the server code enters the while loop, where it constructs a packet containing 3 float values (cam_theta, target_theta, and target_v). It sends that packet to the robot using the address saved during the setup process. The sending of packets from server to robot will continue indefinitely.

From this point, testing began. We started by having the robot simply connect to the server, then receive a single packet giving it a direction and velocity to move in. During our testing, we noted that the server had to be up and waiting to receive the setup packet before turning on the robot and allowing it to send said initial packet. Once this worked consistently, we tested sending more than one packet by having the server wait 5 seconds before sending a different set of navigation values. This, too, worked well. The next step was to link multiple robots at once. This was simple, as it only required duplicating the current code (socket creation/binding, packet receiving, address saving, packet constructing/sending), then changing the local port to the the value for the second robot. It was even easier for the movement code, as it only required changing the UDP port value before uploading it to the robot. We were able to send each robot individual packets with different instructions. The basic testing was complete.

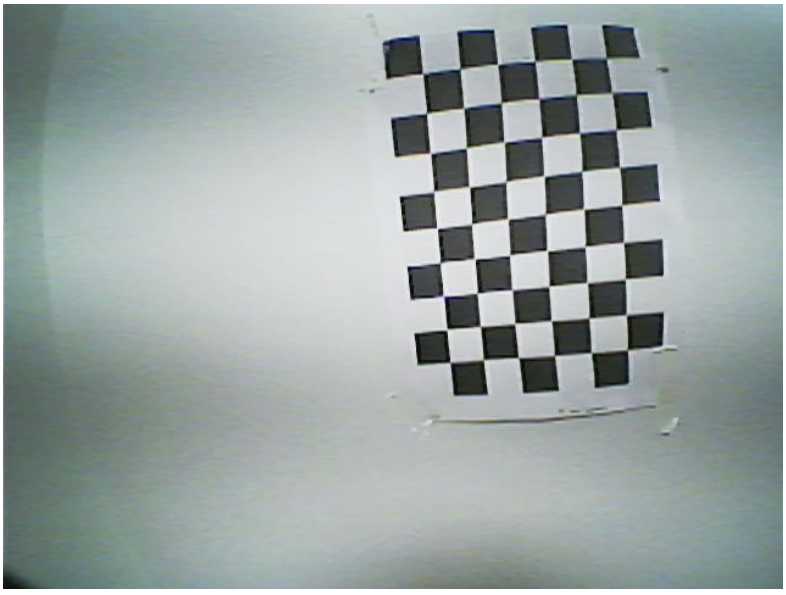
However, things took a slight turn for the worst when we began integrating our code. For some reason when we combined the server and camera detection code the server would not consistently receive the setup packet from the robot. What would continuously work with one packet was now failing 80% of the time. So, rather than sending one packet from robot to server, we created a while loop to send a limitless amount of packets. Once the server received one of these packets and saved the robot's address, it would send a single packet to the robot to confirm it received the setup packet. When the robot receives the confirmation, the while loop is broken and the rest of the mouse_movement code and Server_Cmd_Center code can run as intended. When the rest of the code was integrated, the server would only send a packet after the sound source's *beep* was heard, the time delay was processed, and the robot locations were recorded.

In addition to the necessary communication between the two robots and the laptop we also had to ensure that the microphones were properly connected to the laptop. This was relatively straightforward, but there were some issues that came up. The bluetooth microphones came with receivers which were connected to the laptop through USB-A ports. The receivers accepted the sound data recorded by the microphones and allowed the laptop to access that data. However, there were some issues we faced with respect to the bluetooth microphones. The main issue we faced was the faulty connection between the microphone and the receiver. We had to reset the connection multiple times by connecting the microphone to the receiver and connecting the receiver to the laptop. This was the reset process we followed every time we suspected a faulty microphone connection. Overall, the connection between the microphone and the laptop was crucial in allowing us to perform the necessary calculations for the time delay.

3.4. Software Testing Progress – Camera Detection

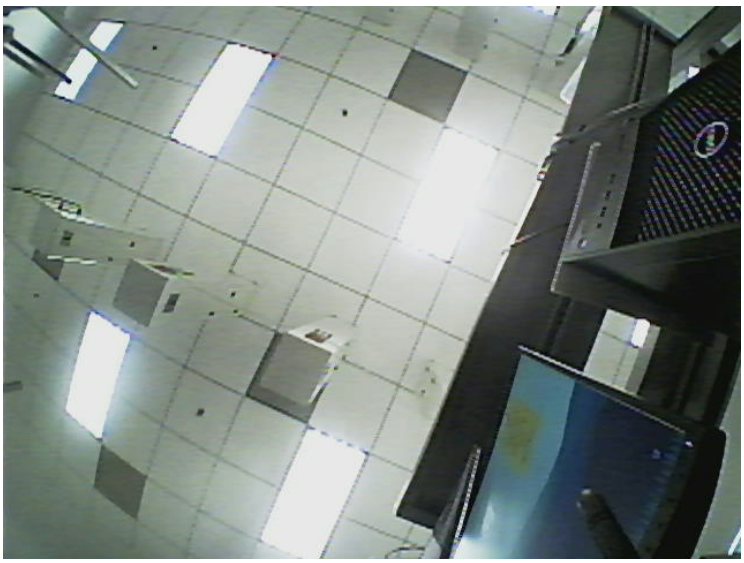
Our implementation used April tags in order to define the boundaries of the test area, as well as keeping track of the locations of the robots in the arena with relation to each other. The idea was to have a webcam suspended from above looking down at the floor. From there, there would be four tags placed in a square on the floor.

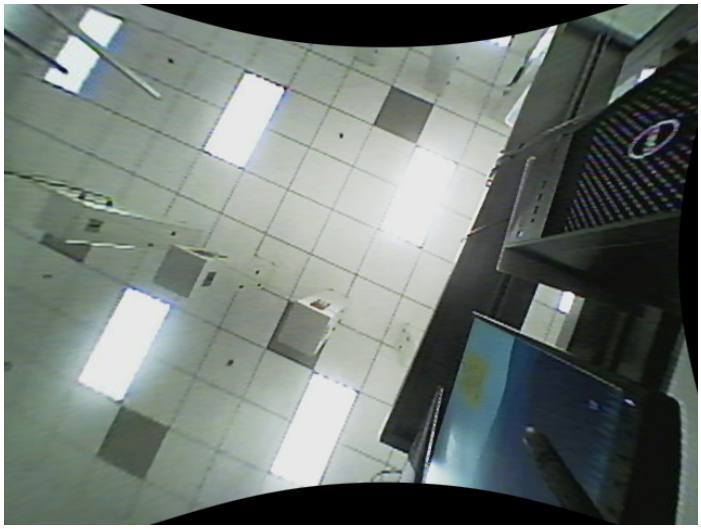
The first hurdle that we had to get through in order to use the cameras for our desired application was to calibrate them. Since cameras can sometimes contain some distortion based on the lens, we wanted to make sure we could mitigate that distortion as much as we could in order to ensure reliable detection of the April tags on the floor. At first, we used the wireless cameras that we were given in order to take pictures of checkerboards that would be used to calibrate the image that would be displayed.



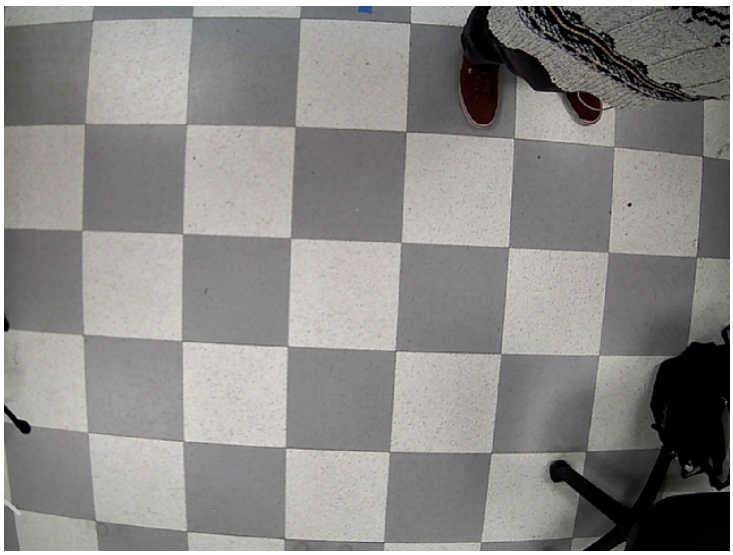
Once the raw calibration data was collected, we used a program named “get_params_fisheye.py” To create three output files, named K.npy, DIM.npy, and D.npy. These files were then used in the Python program “straighten_img_fisheye.py” In order to do the actual

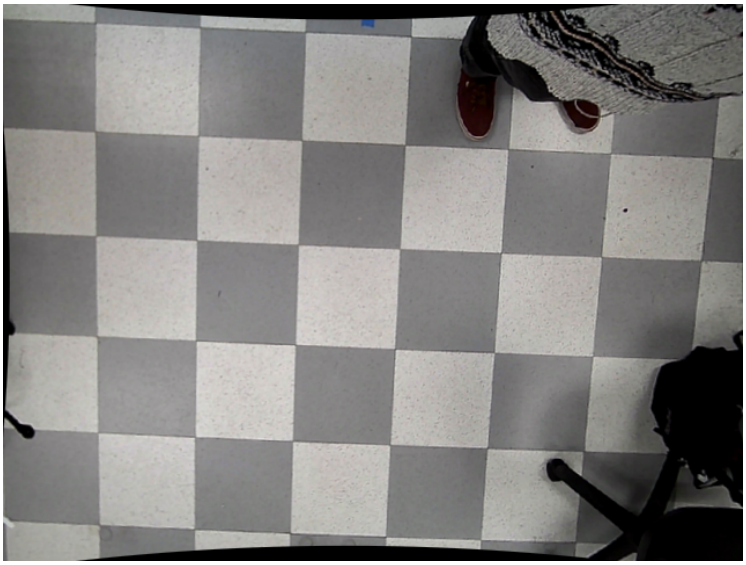
calibration of the camera output. We were then able to take images that were taken by the camera and straighten them so that they would not have the fisheye lens effect.



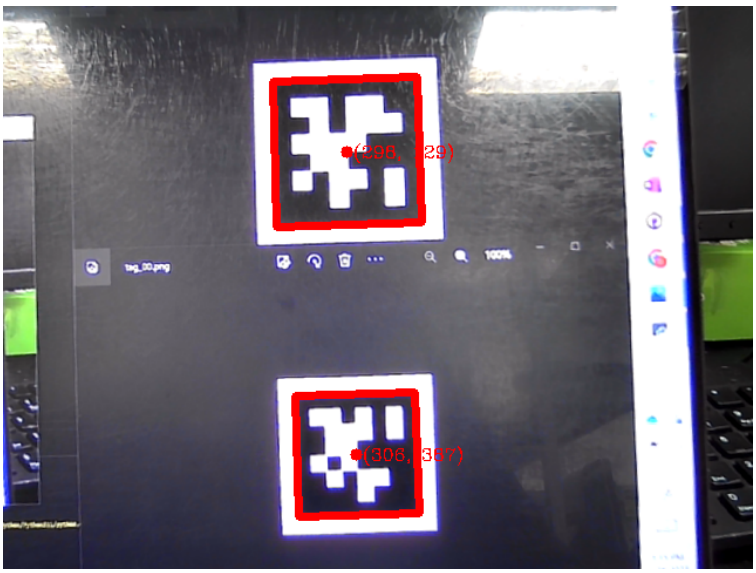


As we can see, the correction was made by “squeezing” the image made by the camera in order to counteract the effect of the lens. From here, we started work on the webcam calibration. We took some very rough measurements of the area covered by the webcam from the ceiling in order to get a sense of how much room we would have for the demo. We used the same calibration method for the webcam that we used for the smaller wireless cameras, resulting in similar results.

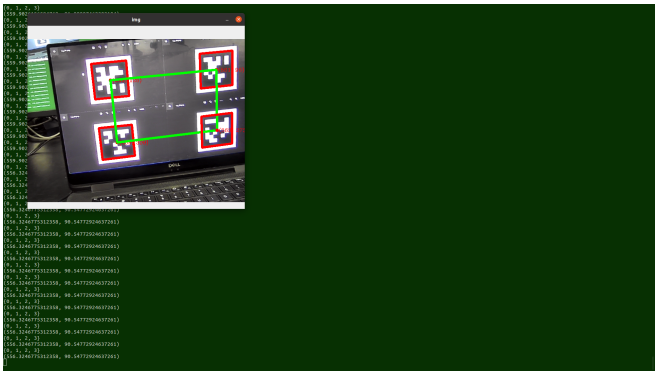




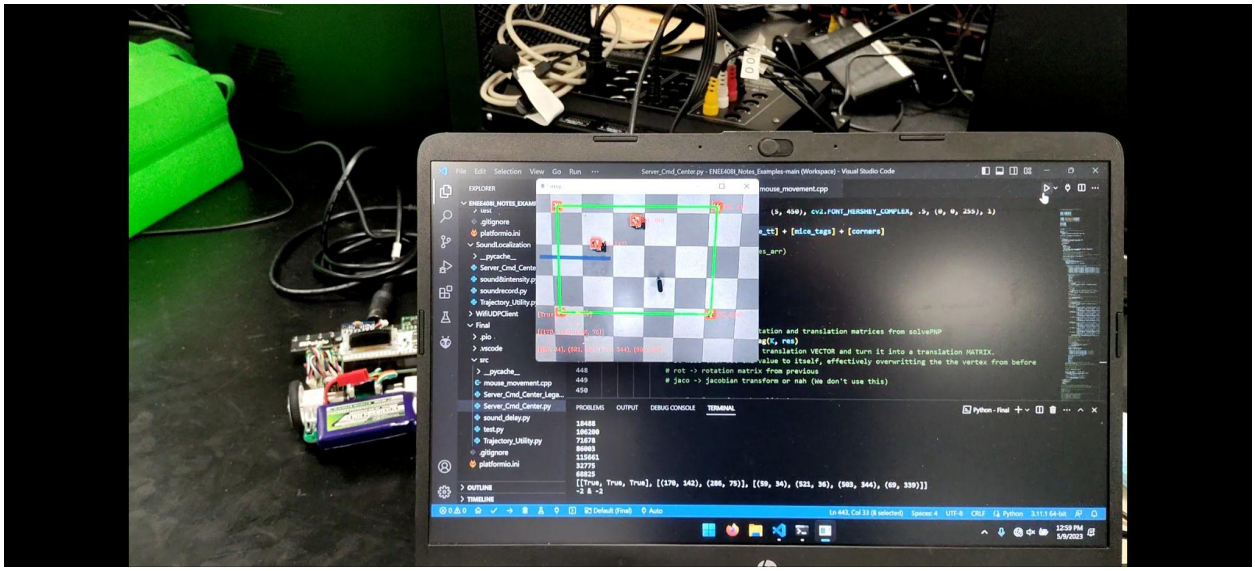
There was a slight issue with the data that was collected for the calibration that resulted in some unexpected outputs, however this was remedied by removing some of the sample data collected that may have come out too blurry or was not lit well enough. Once the webcam's still images were being rectified, we moved on to April_tag detection. As aforementioned, this was done mainly with the "pupil_apriltags" and "cv2" libraries. "cv2" was used for the lens correction as well as drawing the Apriltags on the screen, whereas "pupil_tags" was used for the detection and classification of the individual tags themselves.



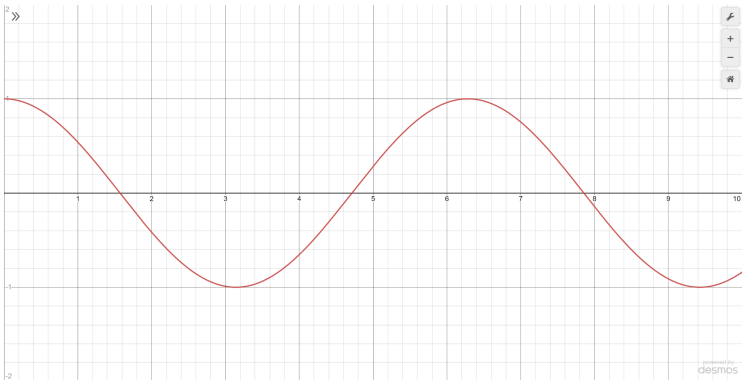
Once the apriltags were readily being picked up by the camera, we began working on drawing an arena on the computer screen in order to get a better understanding of any errors that could occur.



The overall boundary detection, which was the ultimate goal of the apriltags, was to be done by calculating the projections of the robot's apriltag positions onto the vectors that made up the arena area.



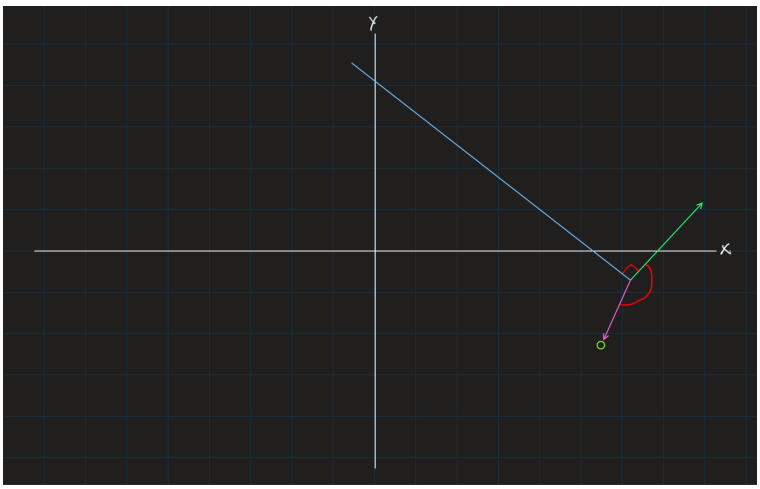
It all essentially boils down to the following equation: $a \cdot b = |a||b|cos(\theta)$. Where $|a||b|$ is essentially a scalar, which does not hold as much relevance for our purposes. What we really care is the $cos(\theta)$ part.



Essentially, the cosine graph remains positive while it is less than 90 degrees, negative otherwise, and this repeats infinitely. The nice part about this is that it will work no matter where in space the points are. A few things that need to be attended to are

1. The norm of the line that will be our boundary
2. The vector from a point on the line to our mouse

Knowing these things, we will be able to take the dot product and, based on how wide the angle is between the vectors, we will know where the mouse tag is.



Here, the green circle represents the position of the mouse in space, the blue line is the boundary we are taking into account, the lime arrow is the vector of the norm, and finally the pink arrow is the vector of the mouse from the line. As we can see, the norm is at a 90 degree angle with the line. If the angle from the norm (lime arrow) to the vector of the mouse (pink arrow) is greater than 90, we understand that it will be on the opposite side of the norm, whereas if the angle is less than 90, we can interpret that to mean that the mouse is on the same side as the norm. Depending on how we define the boundaries, we can simply look at the sign of the result and determine on which side we are on.

The fine details were a bit hard to follow, so let's break it down step by step: First, let's look back at the first equation: $a{\cdot}b = |a||b|cos(\theta)$. The point of having the $cos(\theta)$ in the equivalence is so that we understand where the negative is coming from, and how it allows us to make the distinction between "In" and "Out". "a " is the norm of the line, which may be found by the following equation: $a = [0\ -1;\ 1\ 0]{\cdot}(d_1 - d_0)$

Where $(d_1 - d_0)$ is nothing more than the difference between the two endpoints that make up the boundary line. This will yield a 2x1 array which we will need to transpose in order to dot it with the rotation matrix. The rotation matrix works by correlating the input y with the output x and the input x with the output y by their respective values. In this case, this will yield a 90 degree rotation counterclockwise. Depending on which line we are observing, this rotation matrix will need to be altered.

Finally, the “b” vector is nothing more than the vector from a point on the line to the mouse's position, and is given by $b = p_1 - p_0$.

Where p_0 is a point on the boundary line and p_1 is the mouse's coordinates. Note that it's crucial to subtract the "Arrowhead" by the "Tail". Additionally, note that p_0 could also very well be either d_0 or d_1. For our purposes, we have done this just to keep things simple. Finally, we dot both “a” and “b” as shown in the equation.

Another neat thing about this method is, due to the equivalence from the equation , we do not need to calculate $cos(\theta)$ every time. This will save us some computing power, though it is not as crucial in our case since our application is so small. In the end, the code ended up taking into account both the tags on the floor and the tags on the robots.

```
# If there is an additional tag in the array of detected tags, we want to perform boundary detection
# print("sorted_dict: ", sorted_dict)
if len(detect_keys) > 4:
    # Arrays to hold our stuff
    corners = []
    mice_tags = []
    b_arr = {}
    d_arr = []
    a_arr = []
    res_arr = []
    # Coordinates of center of tags
    # # corner_tags (List) -> [(x_coord, y_coord)]
    for x in corner_tags:
        corners.append(np.array([int(sorted_dict[x].center[0]), int(sorted_dict[x].center[1])]))
    print("corners: ", corners)
    # Find difference from each pair of tags
    # Arrowhead - nock
    #  d_arr (List) -> [()]
    for x in range(3):
        # print( corners[x+1][0] - corners[x][0],corners[x+1][1] - corners[x][1])
        d_arr.append(np.array((corners[x+1][0] - corners[x][0],corners[x+1][1] - corners[x][1])))
    d_arr.append(np.array((corners[0][0] - corners[3][0],corners[0][1] - corners[3][1])))
    print("d_arr: ", d_arr)
    # Rotation matrix
    rot = np.array([[0, -1], [1, 0]])
    # Take dot product of each difference and the rotation matrix to make the norm (?)
    for x in range(len(corner_tags)):
        a_arr.append(np.dot(rot, d_arr[x]))
    print("a_arr: " ,a_arr)
    # print("")
    # Coordinates of the mouse
    for x in range(4, len(detect_keys)):
        mice_tags.append(np.array((int(sorted_dict[x].center[0]), int(sorted_dict[x].center[1]))))
    print("mice_tags: ", mice_tags)
    # Hardcoding cases for how many mice are detected
    match len(mice_tags):
        case 1:
            b_arr.update({0:[np.array((mice_tags[0][0] - corners[0][0],mice_tags[0][1] - corners[0][1])),
                             np.array((mice_tags[0][0] - corners[1][0],mice_tags[0][1] - corners[1][1])),
                             np.array((mice_tags[0][0] - corners[2][0],mice_tags[0][1] - corners[2][1])),
                             np.array((mice_tags[0][0] - corners[3][0],mice_tags[0][1] - corners[3][1]))]})
        case 2:
            b_arr.update({0:[np.array((mice_tags[0][0] - corners[0][0],mice_tags[0][1] - corners[0][1])),
                             np.array((mice_tags[0][0] - corners[1][0],mice_tags[0][1] - corners[1][1])),
                             np.array((mice_tags[0][0] - corners[2][0],mice_tags[0][1] - corners[2][1])),
                             np.array((mice_tags[0][0] - corners[3][0],mice_tags[0][1] - corners[3][1]))]})
```

```
            b_arr.update({1:[np.array((mice_tags[1][0] - corners[0][0],mice_tags[1][1] - corners[0][1])),
                             np.array((mice_tags[1][0] - corners[1][0],mice_tags[1][1] - corners[1][1])),
                             np.array((mice_tags[1][0] - corners[2][0],mice_tags[1][1] - corners[2][1])),
                             np.array((mice_tags[1][0] - corners[3][0],mice_tags[1][1] - corners[3][1]))]})

        case 3:
            b_arr.update({0:[np.array((mice_tags[0][0] - corners[0][0],mice_tags[0][1] - corners[0][1])),
                             np.array((mice_tags[0][0] - corners[1][0],mice_tags[0][1] - corners[1][1])),
                             np.array((mice_tags[0][0] - corners[2][0],mice_tags[0][1] - corners[2][1])),
                             np.array((mice_tags[0][0] - corners[3][0],mice_tags[0][1] - corners[3][1]))]})

            b_arr.update({1:[np.array((mice_tags[1][0] - corners[0][0],mice_tags[1][1] - corners[0][1])),
                             np.array((mice_tags[1][0] - corners[1][0],mice_tags[1][1] - corners[1][1])),
                             np.array((mice_tags[1][0] - corners[2][0],mice_tags[1][1] - corners[2][1])),
                             np.array((mice_tags[1][0] - corners[3][0],mice_tags[1][1] - corners[3][1]))]})

            b_arr.update({2:[np.array((mice_tags[2][0] - corners[0][0],mice_tags[2][1] - corners[0][1])),
                             np.array((mice_tags[2][0] - corners[1][0],mice_tags[2][1] - corners[1][1])),
                             np.array((mice_tags[2][0] - corners[2][0],mice_tags[2][1] - corners[2][1])),
                             np.array((mice_tags[2][0] - corners[3][0],mice_tags[2][1] - corners[3][1]))]})
        case _:
            print("Mouse length array OOB")

        print("b_arr: ", b_arr)
        print("b_arr[0]: ", b_arr[0])
        # print("a_arr: ", a_arr)
        # Find the result from each dot product
        print("a_arr and b_arr (BEFORE): ", a_arr[0], b_arr[0][0])
        # a_arr = np.reshape(4,1)
        # b_arr = np.reshape(4,1)
        for x in range(len(a_arr)):
            for y in range(len(b_arr)):
                print("a_arr and b_arr(AFTER): ", a_arr[x], b_arr[y])
                res_arr.append(np.dot(a_arr[x], b_arr[y][x]))
        for x in range(0,len(res_arr)):
            print(res_arr[x])
```

We store the x and y coordinates of each corner tag's center into arrays, which are then stored into a list since lists are easier to iterate through:

```
for x in corner_tags:
    corners.append(np.array([int(sorted_dict[x].center[0]), int(sorted_dict[x].center[1])]))
```

The for loop iterates over the tag numbers specified in the array corner_tags. After that, we find the vectors between tags, and store then in a similar fashion; a List made of numpy arrays:

```
# Find difference from each pair of tags
# Arrowhead - nock
#  d_arr (List) -> [()]
for x in range(3):
    # print( corners[x+1][0] - corners[x][0],corners[x+1][1] - corners[x][1])
    d_arr.append(np.array((corners[x+1][0] - corners[x][0],corners[x+1][1] - corners[x][1])))
d_arr.append(np.array((corners[0][0] - corners[3][0],corners[0][1] - corners[3][1])))
```

The reason we only use the for loop for three vectors and not all 4 is that the last vector is the one from tag03 to tag00, which could not be cleanly implemented. Again, d_arr is a List of numpy arrays."

```
# Rotation matrix
rot = np.array([[0, -1], [1, 0]])
# Take dot product of each difference and the rotation matrix to make the norm
for x in range(len(corner_tags)):
    a_arr.append(np.dot(rot, d_arr[x]))
```

Above, we rotate the vector in order to get a vector that is normal to the original. This will be used to determine the projection of the mice onto the sides of the arena.

```
# Coordinates of the mouse
for x in range(4, len(detect_keys)):
    mice_tags.append(np.array((int(sorted_dict[x].center[0]), int(sorted_dict[x].center[1]))))
```

The array detect_keys contains the ID numbers of the tags that were detected. Since we know the first four tags are for the corners, we start from tag 4 and work our way till the end of the list, appending these tags to a new list that will store the coordinates as arrays.

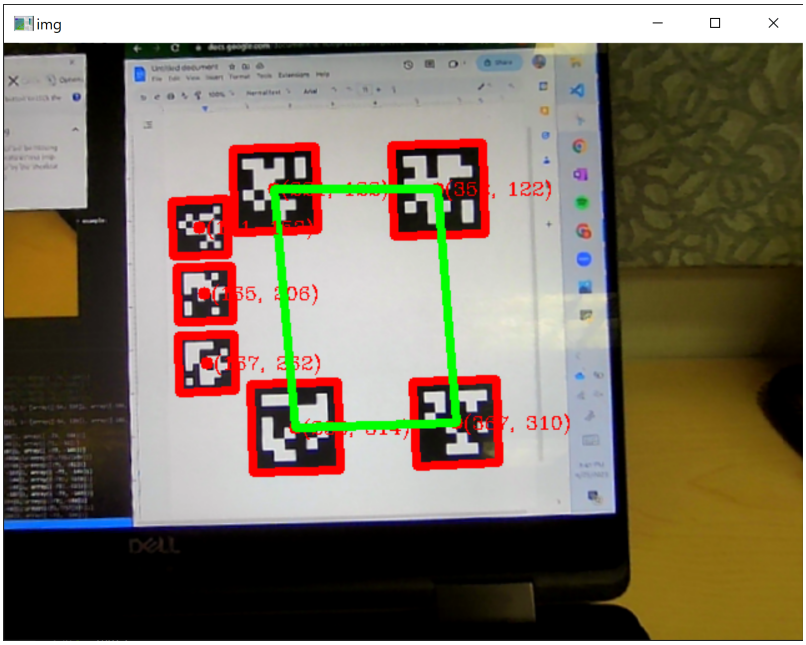
Within the context of this project, there can be from 1 to 3 mice in the arena. We need to find the difference between each mouse and each of the four corners in order to calculate the projections onto them. b_arr is a dictionary, meaning it can be indexed by something other than an integer. One change that could have been made was using an array, however we left it as a dictionary for organization:

```
match len(mice_tags):
    case 1:
        b_arr.update({0:[np.array((mice_tags[0][0] - corners[0][0],mice_tags[0][1] - corners[0][1])),
                         np.array((mice_tags[0][0] - corners[1][0],mice_tags[0][1] - corners[1][1])),
                         np.array((mice_tags[0][0] - corners[2][0],mice_tags[0][1] - corners[2][1])),
                         np.array((mice_tags[0][0] - corners[3][0],mice_tags[0][1] - corners[3][1]))]})
    case 2:
        b_arr.update({0:[np.array((mice_tags[0][0] - corners[0][0],mice_tags[0][1] - corners[0][1])),
                         np.array((mice_tags[0][0] - corners[1][0],mice_tags[0][1] - corners[1][1])),
                         np.array((mice_tags[0][0] - corners[2][0],mice_tags[0][1] - corners[2][1])),
                         np.array((mice_tags[0][0] - corners[3][0],mice_tags[0][1] - corners[3][1]))]})

        b_arr.update({1:[np.array((mice_tags[1][0] - corners[0][0],mice_tags[1][1] - corners[0][1])),
                         np.array((mice_tags[1][0] - corners[1][0],mice_tags[1][1] - corners[1][1])),
                         np.array((mice_tags[1][0] - corners[2][0],mice_tags[1][1] - corners[2][1])),
                         np.array((mice_tags[1][0] - corners[3][0],mice_tags[1][1] - corners[3][1]))]})
     case 3:
        b_arr.update({0:[np.array((mice_tags[0][0] - corners[0][0],mice_tags[0][1] - corners[0][1])),
                         np.array((mice_tags[0][0] - corners[1][0],mice_tags[0][1] - corners[1][1])),
                         np.array((mice_tags[0][0] - corners[2][0],mice_tags[0][1] - corners[2][1])),
                         np.array((mice_tags[0][0] - corners[3][0],mice_tags[0][1] - corners[3][1]))]})

        b_arr.update({1:[np.array((mice_tags[1][0] - corners[0][0],mice_tags[1][1] - corners[0][1])),
                         np.array((mice_tags[1][0] - corners[1][0],mice_tags[1][1] - corners[1][1])),
                         np.array((mice_tags[1][0] - corners[2][0],mice_tags[1][1] - corners[2][1])),
                         np.array((mice_tags[1][0] - corners[3][0],mice_tags[1][1] - corners[3][1]))]})

        b_arr.update({2:[np.array((mice_tags[2][0] - corners[0][0],mice_tags[2][1] - corners[0][1])),
                         np.array((mice_tags[2][0] - corners[1][0],mice_tags[2][1] - corners[1][1])),
                         np.array((mice_tags[2][0] - corners[2][0],mice_tags[2][1] - corners[2][1])),
                         np.array((mice_tags[2][0] - corners[3][0],mice_tags[2][1] - corners[3][1]))]})
    case _:
        print("Mouse length array OOB")
```

Finally, here we are finding the projection of the mice tags onto the vectors. If a value comes back positive, we know that the tag is within the vector space, i.e the arena. If a value comes back negative, we know that it is outside of the vector space:

```
for x in range(len(a_arr)):
    for y in range(len(b_arr)):
        print("a_arr and b_arr(AFTER): ", a_arr[x], b_arr[y])
        res_arr.append(np.dot(a_arr[x], b_arr[y][x]))
for x in range(0,len(res_arr)):
    print(res_arr[x])
```

We get back the projections of the mice tags on the vectors, but all we really care about is the sign of the numbers. For example, when all the mice are out of the arena:



We get the following output:

```
...

10828
18135
3938
35590
36201
35537
13948
6768
20732
-12017
-12755
-11858
```

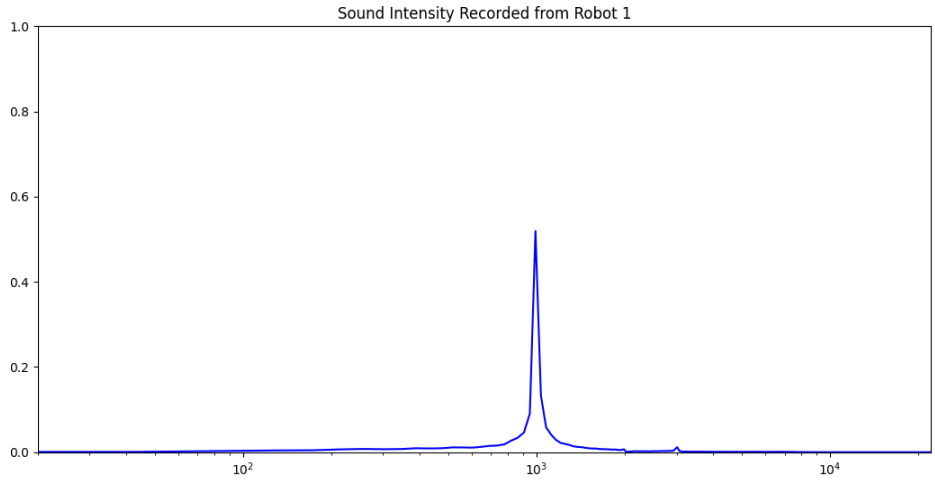
Once we had gotten to this point, we were able to serialize it into an array for ease of use:

```
[Loc. tag00, Loc. tag01, Loc. tag02, Loc. tag03, Loc. mouse00, Loc. mouse01, Loc. mouse02, ...
|   Tuple  |   Tuple   |   Tuple   |   Tuple   |    Tuple    |    Tuple    |    Tuple    |

... mouse00 OOB, mouse01 OOB, mouse01 OOB]
    |  Boolean  |   Boolean  |   Boolean  |
```

In the end, the array ended up being an array of arrays, where the first sub-array contained whether or not any one mouse was inside the arena or not, the second the locations of the mice, and the third the locations of the tags on the ground.

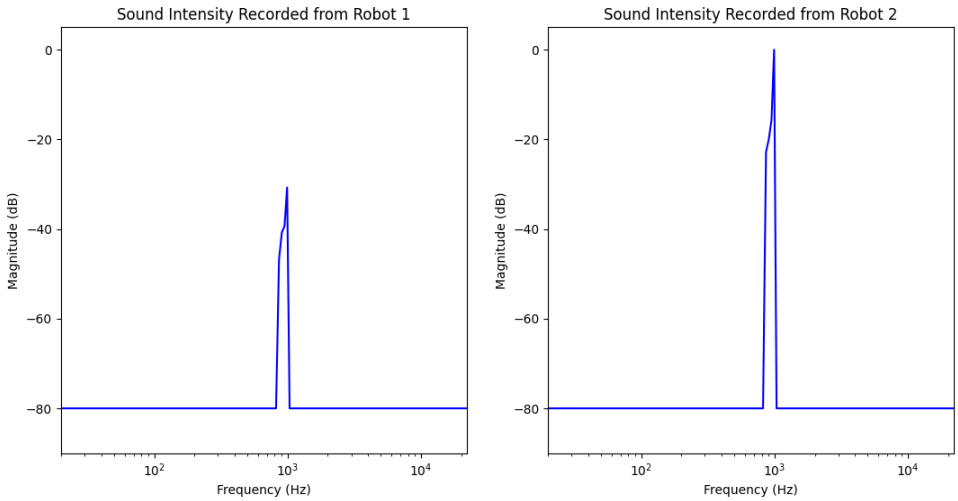
Software Testing Progress – Sound Data

Recording the sound data obtained from the microphones was crucial towards being able to control the robots so they'd localize a sound source. The first step towards this involved testing the microphones and creating a simple Python script which displayed the sound data recorded by the microphones. This was done through the use of the PyAudio module which we used to create a script which would continuously run to allow us to analyze sound data in real time. We then converted this sound data into the frequency domain by performing a Fourier Fast Transform to visualize the amplitudes at the different frequencies. This was necessary since our planned sound source was a 1kHz chirp signal. To further analyze the sound intensities near the 1kHz frequency we calculated the average intensity between intensities 937.5Hz and 1078.125Hz. We did this by calculating the "buckets" the microphone was registering different frequencies in. This was done by dividing the sample rate of the microphone which was 48000Hz by the chunk size which was 1024. Dividing these two numbers resulted in 46.875 which was the frequency range of one "bucket." We then multiplied this by 20 which resulted in 937.5Hz. In the same vein we multiply 46.875 by 23 to obtain 1078.125Hz. This meant to calculate the average intensity we summed four elements which can be denoted by sound_intensity(x) where x was 20, 21, 22, and 23. After summing these four "buckets" we divide by four to obtain the average. The plot of the average intensity in amplitude only (not decibels) is shown below.



The average intensity was then changed into a decibel scale in order for us to visualize and analyze the results better. This was calculated by taking the log base 10 of the average intensity and multiplying that result by 20. This allowed us to better visualize the intensities of the different frequencies. After this calculation we set any frequency which wasn't in between 937.5Hz and 1078.125Hz to zero. This was done to focus exclusively on frequencies near 1kHz

since our sound source was a 1kHz signal. It is important to note that for our time delay approach this process was done for two microphones at the same time. That is to say we were able to display a plot which contained two subplots with the sound data for two different microphones. That plot is displayed below.



The development of the sound localization algorithm was difficult to say the least. We tried three different approaches and only the last approach succeeded. The first approach involved using sound intensity to guide a robot towards the sound source. The robot would move in a sine-wave and would adjust the direction of the sine wave depending on the different sound intensities recorded. The basis for this approach was in a control system algorithm called extremum seeking control. This involved the use of the parameters velocity and phi which are originally from the Trajectory_Tracking code. Phi was declared as a sine wave of the form phi = A*sin(wt-b) - c*(correlation_variable). This is when correlation steps in and becomes crucial to our approach. Levi helped us in this aspect by publishing a sample script in Python which calculated the difference between two sine waves which were not in sync. This difference was crucial to our calculations since we needed to find the difference between the sound intensity recorded and our “sample” sinusoid declared for phi. This would then allow us to set the variable called correlation_variable to continuously calculate phi to adjust the robot direction and track the sound source. Unfortunately there were many difficulties with this approach. The main difficulty we faced was during the final testing of this algorithm and involved not having the robot change direction effectively. The robot would move like a sinusoid, but when we placed a sound source on its left or right the robot would keep moving forward. The robot would somewhat change direction, but nowhere near enough to reach the sound source. We decided we didn’t have enough time to keep troubleshooting this approach and moved on to our second approach.

Our second approach was relatively simple compared to our first approach. It consisted of having the robot rotate 360 degrees while recording the sound intensities. The robot would also record its position for every intensity recorded. This would allow the robot to “remember” at which position the sound intensity was highest. It would then rotate back to that position and

move forward for a certain amount of time before repeating the entire process again. This approach was quickly implemented and we were able to rotate the robot 360 degrees perfectly, but we encountered a huge obstacle with this approach. The entire approach relied on the assumption that the microphones we were using were at least somewhat directional. This meant that if the microphone wasn't facing the sound source the sound intensity recorded would be relatively small. We quickly found the opposite was true. The microphone would basically record the same sound intensity no matter the orientation of the microphone with respect to the sound source. We tried to solve this problem by creating a funnel which would theoretically filter out any sound that wasn't coming from the front of the microphone. We tried to create a makeshift funnel from paper by rolling up multiple papers to try and make the funnel thicker. The assumption was that if the funnel was thick enough then sound wouldn't be able to penetrate through the funnel. We found this to be relatively ineffective which left us with no choice but to proceed with a different approach.

Our third and final approach consisted of using two robots with one microphone on each to record the time delay from when the sound source was played. This approach used the correlation calculations previously stated. The main difference was that we were now comparing two sound intensity signals recorded by the two microphones. In theory the sound intensity signals were exactly the same since the microphones were recording the same sound source, but there was one important difference. This difference was caused by the fact that the robots would be at different distances from the sound source. Thus one of the sound intensity signals would be further along and the other signal would be delayed. The correlation calculations were needed for this exact reason since we needed the signals to "line up" in order to calculate the sound intensity difference between the two. Based on this calculation the farther robot would move towards the sound source for some time and then the robot which was closer would become the farther one. The calculations for where the robots had to move also needed the camera which recorded the arena and the position of the robots. The time delay calculated between the two robots/microphones was only one parameter needed to move the robots towards the sound source. In the end we were not able to fully implement all of the aspects discussed, but this will be discussed in the Demo Result section.

Overall, we tried three different sound localization methods of which two used sound intensity as their main way to direct the robots to the sound source. We found that using sound intensity was not possible since the microphones weren't directional. This presented a difficult obstacle since we were running out of time and we needed to implement a sound localization strategy as fast as possible. We were able to successfully calculate the time delay between two microphones which was then used to determine which microphone was closest to the sound source. This data was originally supposed to be used in conjunction with the camera data which would calculate the positions of the robots. The robot positions would then be used to determine the trajectory the robots needed to move in order to reach the sound source. However, we were not able to reach this stage, but instead we were able to have the robots move in response to the sound source. This will be further discussed in the Demo Result section.

4. EXECUTION/DEMO RESULT

The project demonstration took place on May 9, 2023 and we believe it was a success considering the challenges and obstacles we faced. The setup consisted of placing four different Apriltags in four corners of the room to create the Apriltag arena previously mentioned. In addition, a webcam was taped to one of the pillars extending from the ceiling. The purpose of this webcam was to calculate the different positions of the two robots and to create an imaginary boundary which would inform us if one of the robots was outside the boundary. The webcam was connected to an HP laptop to send off the information gathered for calculations. On the other hand, the arena had two robots placed inside of it and one stationary sound source which was a bluetooth speaker connected to a phone. The speaker was used to amplify the 1kHz signal we were playing intermittently. We'd play the signal for roughly two seconds and then play it again after around five to ten seconds of silence. These intervals were chosen after multiple rounds of testing which revealed that if we played the 1kHz signal too quickly it would crash the Python program since it was too quick for the computer to run calculations. Each robot was attached with one bluetooth microphone and the two bluetooth receivers were connected to the HP laptop. We now move on to the actual execution of the demonstration.

Our demonstration consisted of placing the two robots inside the Apriltag arena and placing a speaker to the right of both robots. The speaker would play the 1kHz chirp signal previously discussed and the sound data from the two microphones would be sent to the HP laptop for calculations. The program calculated which robot was closer to the sound source and told the farthest robot to move forward until it was close enough from the sound source that the other robot was now the farthest robot. At the same time the webcam set up on the ceiling was calculating the positions of the two robots and outputting whether or not the robots were inside the boundary. Unfortunately we were unable to implement a solution so the robots would be forced inside the boundary. We tried manipulating theta from the Trajectory_Tracking file, but we were out of time. In addition, we faced some serious issues related to the networking the day of the demonstration. Essentially, the problem involved the packets being sent from the robot to the HP laptop. The packets were only being sent once when the robots were turned on which was a problem since the packets were prone to getting "lost." We realized this problem during testing before the demonstration since the connection between the robots and the laptop was extremely erratic. This was a serious problem because for the demonstration to work the robots had to be connected to the laptop. We quickly solved this problem with the help of Levi the day of the demonstration. We edited the C++ code being uploaded to the robots so that the packets would be sent continuously until a connection was established. This ensured a stable connection between the two robots and the HP laptop. Overall, we showed two things the day of the demonstration: the first was the successful calculation of the time delay between the two robots and the second was the successful implementation of the webcam to record and track the positions of the robots inside the boundary.

A video of our robot demonstration can be found in the following YouTube link: https://youtu.be/XBJg9KBv6tc.

5. SUMMARY, FUTURE SUGGESTIONS, & FEEDBACK

Overall, our final product was very different from the initial ideas proposed at the beginning of the semester. We were very ambitious in our goals and originally envisioned two robots actively tracking one robot with a sound source attached to it. This evader robot would then use information from a camera, which recorded the positions of all three robots, to determine the best path of escape. In reality we weren't even able to test an evader robot due to the difficulty we encountered throughout our project. In addition, we weren't able to fully implement having the robots move towards a sound source since we were out of time. However, we were able to successfully calculate the time delay between two different microphones and set up a camera which recorded the positions of the tracker robots along with creating an imaginary Apriltag arena.

We encountered many difficulties in our project ranging from sound localization issues to integration problems towards the end of the course. We were able to overcome these issues by working together and brainstorming different ideas on how to tackle our issues. In addition, Levi was incredibly helpful in every problem we encountered and we wouldn't have been able to complete this project without his help.

While our final demonstration did differ significantly from what we expected originally, we were still able to accomplish the major aspects of the project we had originally planned. To anyone that might want to continue our project we would suggest trying to implement the proper tracking algorithms we developed for the tracking robots. The code for this aspect of the project is mostly done, but we weren't able to fully test it due to time constraints. If that is successful we would suggest moving on to programming the evader robot. The evader robot would be the last piece of the puzzle to complete what we originally set out to do.

Working on this project has taught us a multitude of things. First, we learned how to work as a group and divide our work to be more efficient with our time. However, at the same time we would work together if any of us got stuck on an obstacle. The teamwork we learned working on this project will be invaluable in our future career as engineers. Second, the sheer amount of coding done for this project taught a lot about software and how it interacts with hardware. This is because although all of the calculations were performed in Python we still had to integrate it with the C++ files we were uploading to the robots. This meant we needed to understand how these two different aspects of the project interacted with each other. In the end, this capstone course taught us how crucial teamwork is when working on engineering projects.

Concerning feedback for the course, there is not much we would change. We appreciated the freedom we had in choosing our project as well as not being bogged down by unnecessary assignments. However, we would suggest providing a challenge that is slightly easier or having more sample code to work with. This project was about 2-3 weeks out from being completed. We were close, just not quite at the finish line. Other than that, we really enjoyed the project.

6. REFERENCES

Realtime Spectrum Analyser using Python | Part-1: https://fazals.ddns.net/spectrum-analyser-part-1/

Realtime Spectrum Analyser using Python | Part-2: https://fazals.ddns.net/spectrum-analyser-part-2/

Scipy Correlation Function:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.correlate.html

ENEE 408I notes and examples: https://github.com/UMD-ENEE408I/ENEE408I_Notes_Examples

Desmos graphing calculator: https://www.desmos.com/calculator

ECE5760 Robot Navigation Using Sound Localization:
https://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/f2009/jmd242_aps243/576JonAllisite/site/highlevel.html