INDOOR LOCALIZATION WITH SMARTPHONES

Final-Project Report
Spring Semester 2019
- Full Report -

Members:

Mitch Avis, Computer Engineering
Lane Barnes, Computer Engineering
Ethan Courtney, Computer Engineering/Computer Science

Saideep Tiku, Graduate Student
Dr. Sudeep Pasricha, Supervising Professor

Department of Electrical and Computer Engineering
Colorado State University
Fort Collins, Colorado 80523

Report Approved By:
Dr. Sudeep Pasricha

ABSTRACT

GPS has provided accurate localization for consumers for years. The uses of GPS are numerous and varied, largely due to the importance of accurate localization. Although incredibly useful, GPS has a glaring weakness. Due to GPS signals being harshly attenuated by roofs and walls, the signal does not penetrate indoors. This leaves consumers with no access to accurate and easy-to-use localization indoors. This is the problem our project seeks to address. By creating an easy-to-use end-user app, and by streamlining the technical processes involved with setting up a building with indoor localization, our team seeks to create a cohesive process for indoor localization. Our project is especially relevant in large public spaces where navigation can be difficult.

Our team's goal was to take the technology of indoor localization using Wi-Fi fingerprinting and created a unified product that a user could download from an app store and use without technical knowledge. The technology of indoor localization using Wi-Fi fingerprinting relies on the collection of numerous fingerprints from locations all around a building and taken on several different, unique phones. A fingerprint is the pairing of two items: the location the fingerprint was taken at, and a collection of signal strength measurements for all the nearby Wi-Fi access points. The process of collecting these fingerprints can be incredibly tedious if undertaken manually. An important step for our project was the automation of fingerprint collection. Our team developed a rover that can safely navigate around obstacles and collect fingerprints automatically. This will significantly cut down on the time it takes an operator to set up indoor localization in a building. This rover can be placed on the ground, allowing to move independently and collect accurate fingerprints without the operator's input. These collected fingerprints can then be used by an app that allows the end-user to determine their location on a map based on the Wi-Fi signals in the area.

Our team's findings are largely in regards to the development of a rover that can collect Wi-Fi fingerprints autonomously. We have extended current developments by partially solving the problem of autonomous collection. Our rover can move safely, and given sufficient instructions, follow a path throughout a building. One of the most important components of a Wi-Fi fingerprint is the location that the fingerprint was taken at. This location needs to be as accurate as possible to ensure that when users request their location from the app, the result will also be accurate. The rover can provide this location, but as time progresses, the rover's location accuracy will accumulate error. Though this problem has not yet been completely resolved, we have made significant improvements to the drift-correction functionality of the rover's navigation code. Overall, our findings confirm the hypothesis that Wi-Fi fingerprint collection can be done autonomously.

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

CHAPTER 1: INTRODUCTION

With the race to space in the 20th century came the creation of an effective global navigation system using space-based satellites. The world now refers to this system as GPS, and it allows one to determine where on the Earth they are located. This system is widely used around the world, but it has its limitations. Because of how the satellites used for triangulation work, their signals are not always reliable indoors. Furthermore, the accuracy of GPS is generally only within eight meters, which is not optimal when trying to determine one's precise location within a building.

Due to the shortcomings of GPS, research into indoor localization has become quite popular over the last decade. Indoor localization makes use of Wi-Fi signals to determine one's location in a building. This can be done by looking at all wireless access points (WAPs) within range and recording the signal strength (RSSI) of each, measured in decibels. Because the relationship between the distances from a WAP and its RSSI is not linear or predictable, the same triangulation method that GPS uses cannot be applied to this method of indoor localization. To remedy this, a method known as fingerprinting can be used. One fingerprint is the coupling of a specific location's spatial (x, y, (z)) coordinates, relative to some known origin, with a list of WAPs and their associated RSSIs at that location. When mapping a new building or indoor area, fingerprints must be collected from individual points to improve the accuracy of the process, known as the "collection phase."

The collection phase can be tedious for a person to do manually, so it is ideal to automate this task. To accomplish this, an autonomous rover can be used. This rover, fitted with smartphones, can be instructed to move around an indoor area, while the smartphones collect fingerprints along the way. When the rover is done navigating its assigned path, the smartphones will offload their collected fingerprints to a central location for processing.

Processing the fingerprints for a given building or indoor area requires the application of machine learning techniques and the K-Nearest-Neighbors (KNN) algorithm. This enables the processing computer to take a lot of data and "learn" how to translate it to simple enough data for an end-user indoor navigation app to retrieve when needed.

When a user wants to use the indoor navigation app on their mobile device, their phone will scan the RSSI of each WAP in the area and compare it to the known, optimized fingerprints stored in the central location. Based on this information, the app can then accurately predict the location of the user on a pre-loaded map of the area.

This project's approach to indoor localization takes place in two steps. First, data needs to be collected about the strength of various Wi-Fi signals in a variety of specific locations in a building. Second, this data needs to be analyzed using machine learning so that the observed Wi-Fi signal strengths can be translated back into a location in the building. This needs to be placed into a smartphone app to be made available to the consumer. Our approach to this process was to use a rover equipped with a LiDAR sensor to navigate an indoor environment. This rover collects Wi-Fi signal data by incorporating smartphones mounted on the rover. By

correlating Wi-Fi data with the rover's location (which is continuously tracked using motor encoders and PID control), this data can then be analyzed using machine learning. The end-user app first trains the machine learning algorithm with the gathered fingerprints then scans the nearby wireless access points to create an individual fingerprint and finally compares this fingerprint with the database of analyzed fingerprints using the KNN algorithm to inform the user of his or her location on a given map.

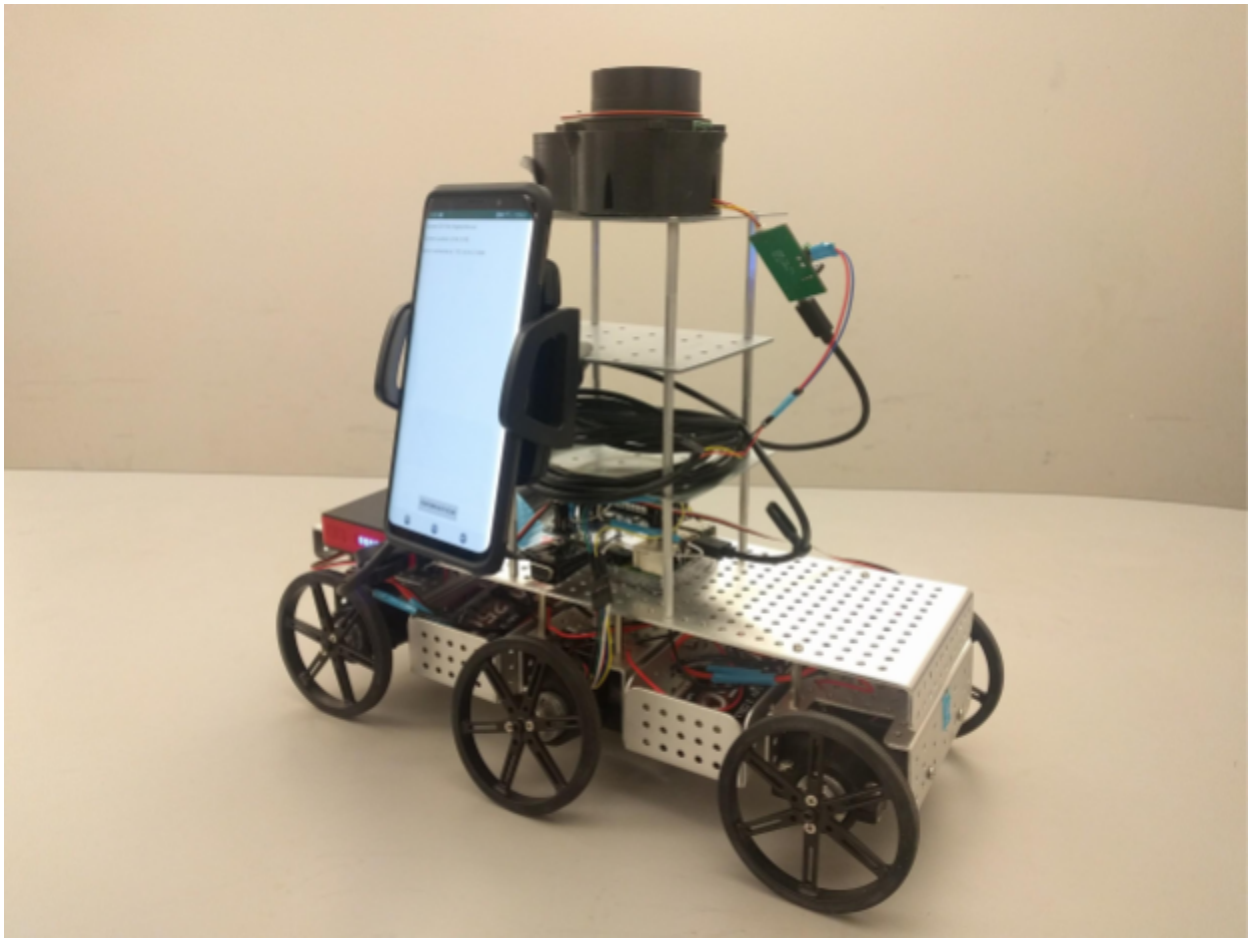

Figure 1: Autonomous Rover with Smartphone, LiDAR, and Raspberry Pi

Chapter 2 of this report summarizes the work of the previous year's team and where they left off, followed by what we started on and where we left off at the end of last semester. Chapter 3 goes into specifics about the rover, how it is controlled, and how it keeps track of its location. Chapter 4 discusses the details of safe and accurate rover navigation. Chapter 5 explains how the fingerprint-collecting app works and how data is communicated between it and the Pi. Finally, Chapter 6 talks about the end-user app and how it works to provide accurate indoor localization.

CHAPTER 2: SUMMARY OF PREVIOUS WORK

The last team on this project was given the same task we have been given: try to automate the fingerprinting process. Their original approach was to reprogram a Roomba cleaning robot to follow a rope and take a fingerprint at every blue tape marking on the rope. After deciding that the Roomba was not going to be viable, they ordered a six-wheeled off-road chassis with motors and ESCs to be used for our team. We were able to salvage the LiDAR from the Roomba and use it for our primary vision for the rover.

In the previous semester, much of our work was focused on the fundamentals of the automation process. This included upgrading the rover significantly, and getting it to move more precisely using encoders and higher gear ratio motors. We got the LiDAR working and built a code library for quickly retrieving and manipulating the LiDAR data. The fingerprinting app for collecting fingerprints on other phones was developed and tested, but not yet integrated with the Pi.

CHAPTER 3: ROVER MOVEMENT

3.1 Motors

At the beginning of the semester, the rover we have received from the old team was not in working condition. One of the ESCs (Electronic Speed Control) units was damaged and we had to buy a new one. In addition, the motors currently installed on the six-wheeled chassis had a gear ratio of 34:1. This means that for every 34 rotations of the internal motor, the wheel will turn one time. This gear ratio was problematic because it caused the motors to have a very high minimum speed. It is ideal to have the rover move very slowly so that it can navigate as accurately as possible. We have purchased motors with a 75:1 gear ratio to replace the old motors. With a higher gear ratio, the motors trade speed for torque. While having more torque is not a requirement of ours, it means that the motors will have a minimum speed half that of the 34:1 gear-ratio motors. After using the 75:1 for a while, it became apparent that their lowest operating was still too fast. This semester, we decided to go one step further and purchased 172:1 gear-ratio motors. This lowest operating speed was incredibly slow while the highest speed was just a normal speed, which is great for accuracy.

Each side of motors is provided voltage from an ESC. Each of the two ESCs receives a PWM (Pulse Width Modulation) signal from the Pi. The rate at which the motors spin depends on the frequency of the PWM signal. Typically, a PWM frequency at 370 is an idle state, ~460 is the maximum speed moving forward, ~280 is the maximum speed moving backward. Setting this PWM is our primary method of controlling the motor.

3.2 Encoders

Starting out, this rover has no means of tracking the rotation of the wheels. It was the assumption of the old team that the LiDAR would be the end all solution of identifying where the rover is located inside a building. Locating the rover with only the LiDAR would be an incredibly difficult task. To alleviate this difficulty, we have decided to order motors with encoders attached. The encoders will be the primary method to determine the location of the rover while the LiDAR is now primarily used for obstacle avoidance.

The encoders work by outputting two binary values on two wires, called A and B. As the motor turns, A and B iterate through a sequence of grey code. Each time A and B change, we call that a count. Each count is a 0.1-degree turn of the wheel. This can be seen below in Figure 2. We add up all the counts to keep track of the angular position of the wheel. When we reach 3600 counts, we know the wheel has rotated once.

A big problem of the encoders is that their frequency of change is greater than the frequency at which the Pi can sample. This causes aliasing, causing our count values to be incorrect. Our goal is to get the sampling frequency to be at least twice that of the encoder frequency, satisfying the Nyquist rate.

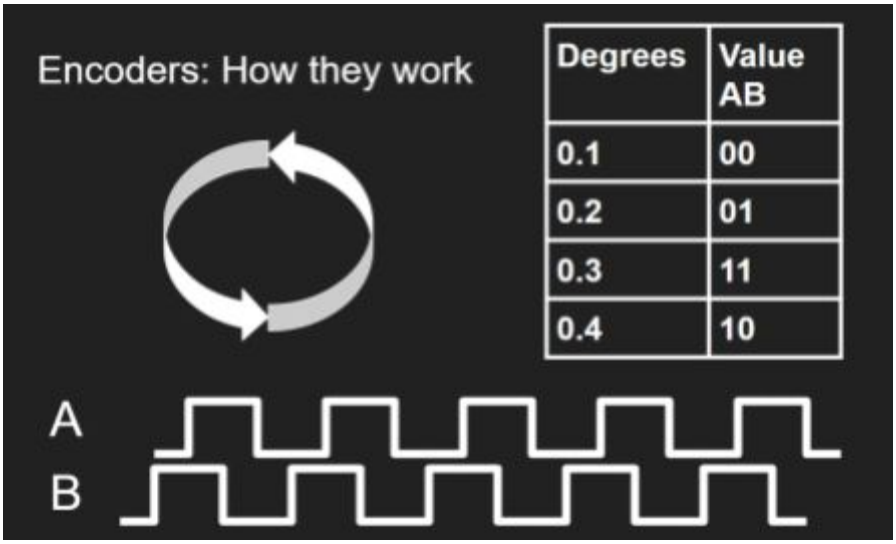

Figure 2: How Encoders Work

The first step to solve this aliasing issue is to completely ignore the B channel. When we do this, we decrease our encoder output from a 2-bit value down to a 1-bit value. The total number of outputs and likewise the resolution of our encoder reduces by a factor of four. The reduction in resolution, however, is not a big deal for us, as we still get a resolution of 900 counts per wheel rotation, which is enough for our purposes. The big advantage we get out of this is that the encoder frequency has also been reduced by a factor of four.

The second step to solve aliasing is to implement multithreading. To begin, we have had sequential code that first samples the motor encoders and then next set the PWM for the motors. Compared to the sampling speed, the speed at which the set PWM is much slower. While the PWM is being set, one or two counts could have passed and the progress of the wheel is untracked. To solve this, the sampling of the encoder gets its own dedicated thread while the PWM setting code remains on the original thread. With both functionality running concurrently, the PWM set code can no longer bottleneck the sampling code.

With these two solutions combined, we can get a sampling rate of about five times that of the encoder frequency, satisfying the Nyquist rate. Each side of wheels is now properly tracked at all times. Given the circumference of the wheels and the width of the rover, we can calculate how many counts go into one meter and how many counts can turn the rover in place by one degree.

3.3 PID Control

Now that we are able to track the rotation of the wheels, the next goal for the rover is to get it to drive in a straight line. To drive in a straight line, both sides of wheels need to, at least on average, spin at the same speed. The speed at which the motors spin is largely dependent on process variation. In addition, we have two separate LiPo batteries powering each side of the

wheel. One of these LiPo batteries may be at different battery life than the other, causing the depleted battery to spin its motors much slower.

The only way to get the wheels spinning at the same rate is through a software solution. With the help of the motor encoders, we can check the speed of the motors and calculate the error between the two sides. We can change the PWM of one of the ESCs to provide a larger RMS voltage to the slower side of motors. After correcting this error, both sides will drive equivalently and the rover will drive in a straight line.

To achieve this, we will use PID control [1]. We will have one side be the master side and the other be the slave side. It is the goal of the slave side to match the speed of the master side. We will calculate the error as:

$$CountError = CountMaster - CountSlave$$

We then calculate a correction value that can be called "PIDValue." PID stands for proportional, integral, and derivative. To calculate PIDValue, we do:

$$PIDValue = pK * CountError + iK * \int CountError + dK * d/dtCountError\ dt$$

where pK, iK, and dK are tuning constants. These values are tuned such that the error can reach zero as quickly as possible without any oscillation. Once PIDValue is calculated, we do:

$$PWMSlave = PWMMaster + PIDValue$$

where PWMMaster and PWMSlave are the PWM frequencies to set the master and slave ESC. When the PID is tuned correctly, the slave side will quickly match the speed of the master side, CountError will decrease and the rover will drive straight.

In addition to driving straight, we want to be able to turn the rover either in place or as it moves. To achieve this, we modify the error equation and the PWM equation as such:

$$CountError = TF * CountMaster - CountSlave$$
$$PWMSlave = TF * PWMMaster + PIDValue$$

where TF is the "turning factor." TF is a scaling factor that tells how much should the slave side match the speed of the master side. To make the rover turn slightly while it moves forward, TF could be a value of 1.1. The slave side will reach a speed that is 1.1 times the speed of the master, causing the rover to turn. Let us say we want to rotate the rover in place. We set TF to -1, and the slave will match the speed of the master, but in the opposite direction, causing the rover to spin in place.

3.4 Movement Functions

Now that we have a system of movement, we create a function:

Move(Master, TotalCounts, PWM, TF)

where Master is the side of wheels to be master, TotalCounts is the total amount of counts the master will move, PWM is the PWM frequency (or the speed) of the motor, and TF is the factor by which the slave will match the speed of the master. This Move function can tell the rover to move in any desirable way possible.

In preparation for getting the rover navigating, we want to create simple functions that can be called by all the other software components of the rover. These functions include:

MoveMeters(Meters)
RotateDegrees(Degrees)

which are both self-explanatory. Each of these functions calls the Move() function with predetermined Master, PWM, and TF values. MoveMeters converts Meters to counts and has a TF of 1. RotateDegrees converts degrees to counts and has a TF of -1.

To find these conversion factors, we estimate using the circumference of the wheel. We estimate 2387 counts is one meter. To test this, we made the rover travel between two markings on the ground in the hallway outside the lab. The distance between these markings was about 6.3m. The function MoveMeters(1) was called six times and then MoveMeters(.3) was called once. After calculating the error of where the rover thought it was to where it actually was, we scaled the number of counts per meter by 2213. After repeating this test, the rover is off by two or three inches, which compared to 6.3 m is about a 1% error.

To keep track of where the rover is, we will have three variables that are preserved between Move() calls. They are X, Y, and θ. When RotateDegrees() is called:

$\theta = \theta + \text{Degrees}$

when MoveMeters() is called:

$X = \cos(\theta) * \text{Meters}$
$Y = \sin(\theta) * \text{Meters}$

By doing this, we keep a running X and Y coordinate that we update every time we move. This method is not perfect, however, and error will accumulate as the rover moves along. It is our hope to use the LiDAR to recognize landmarks such as hallway corners as a checkpoint and allow the error of X and Y to reset itself.

3.5 Improving Accuracy

In an attempt to improve the accuracy of the rover's movement (where it is versus where it thinks it is), we replaced the motors, the wheels, and the batteries.

Last semester, we replaced the very fast 34:1 gear-ratio motors with slower 75:1 motors. However, the slowest operating speed of the motor was still too fast. This would be a problem when we tried to move in small increments. A good example of this was when we attempted to rotate the rover by ten degrees and the wheels would only rotate a small amount. The movement function program polls the encoder counts and checks to see if the required encoder count has been reached. The interval between polls was not quick enough to stop the motors in time, and they would end up overshooting by a large margin of error. We can reduce this by making the program anticipate if the encoder count will be reached on the next cycle, but this only helps so much. We cannot make the Pi go faster, so we needed to make the motors go slower. When the motors go slow, they 1) further reduce the risk of encoder aliasing and 2) make the encoder counts increment slower, giving the count check more time to stop the wheels closer to where they need to be. So instead of 75:1 motors, we now use 172:1 motors, which give us a much slower and more useful minimum operating speed.

The wheels of the rover were designed to perform in rugged outdoor environments such as dirt, sand, grass, etc. They were big, wide, and had rubber spikes, which caused problems with the rover's accuracy. The spikes of the wheel would cause the wheel to "jitter" up and down, which caused a lot of unnecessary vibration. The thickness and curvature of the wheels, along with the rover's suspension system, caused the wheel to have different diameters depending on the angle of the wheel relative to the ground. These wheels were replaced with what you see in Figure 1. They are thin, small, and smooth, which is ideal for the flat surfaces the rover will encounter indoors.

The two LiPo batteries that were left with us by the old team were of adequate quality at the beginning of last semester. However, late in this semester, the batteries degraded and would hold their charge for less time. The accuracy of the rover suffered when the batteries were no longer at full charge, as one battery would lose its charge faster than the other, creating an imbalance and causing one side of motors to spin much slower than the other side. This causes the PID to oscillate and make the rover wiggle as it moves forward. At one point, one battery completely died and was no longer rechargeable. After some research, we discovered that the chargers that we used for the batteries were of low quality and greatly decreased the lifespan of our batteries. To fix this issue, we bought two new batteries and two high-quality chargers that have built-in circuitry to prevent damage to the batteries. Now the batteries hold their charge for much longer and we should not have to worry about them degrading. The accuracy of the rover over the course of a day will benefit from a healthy power supply.

3.6 Testing Accuracy

To test the accuracy of the autonomous movement of the rover, we must compare where the rover thinks it is and where the rover actually is after it has moved. We will perform two types of tests: 1) driving forward in a straight line, and 2) rotation in place. These movement options are fundamental for the rover to navigate anywhere on a floor, so we must measure how much error each produces.

The first test involved asking the rover to continuously move forward 1m, 2m, 4m, and 6m, each with three trials. We also ask the rover to move 6m in 1m increments. In Figure 3, shown below, the y-axis represents the distance forward from the rover's starting position (0,0), while the x-axis represents the distance right or left from the rover's position. The triangle represents the ideal position of the rover while the dots represent the actual position of the rover. Some of the dots overlap each other and there are indeed three trials each. As you can see, the forward Y motion is very accurate and it appears the ideal and actual are on the same line. However, the rover likes to drift to the right and the left instead of going directly straight. The further the rover travels forward, the more the rover drifts in the left or right directions. Over the course of 6m, the worst the drift gets is 0.3m, which means 0.05m drift per 1m.

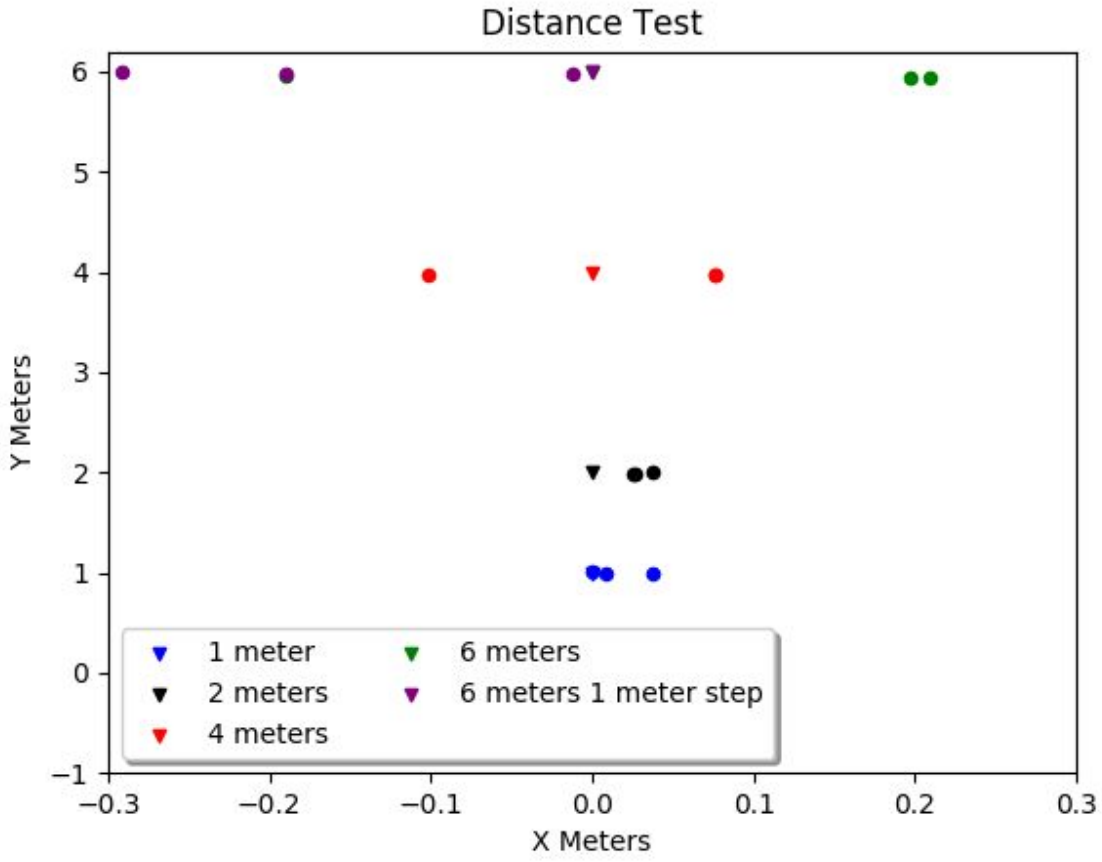

Figure 3: Distance Test

In Figure 4, shown below, the continuous 6m test and the 6m in 1m increments test are shown with the y-axis zoomed in. This shows that over six meters, the rover is at worst off by 7.5cm with an error of 1.25%.

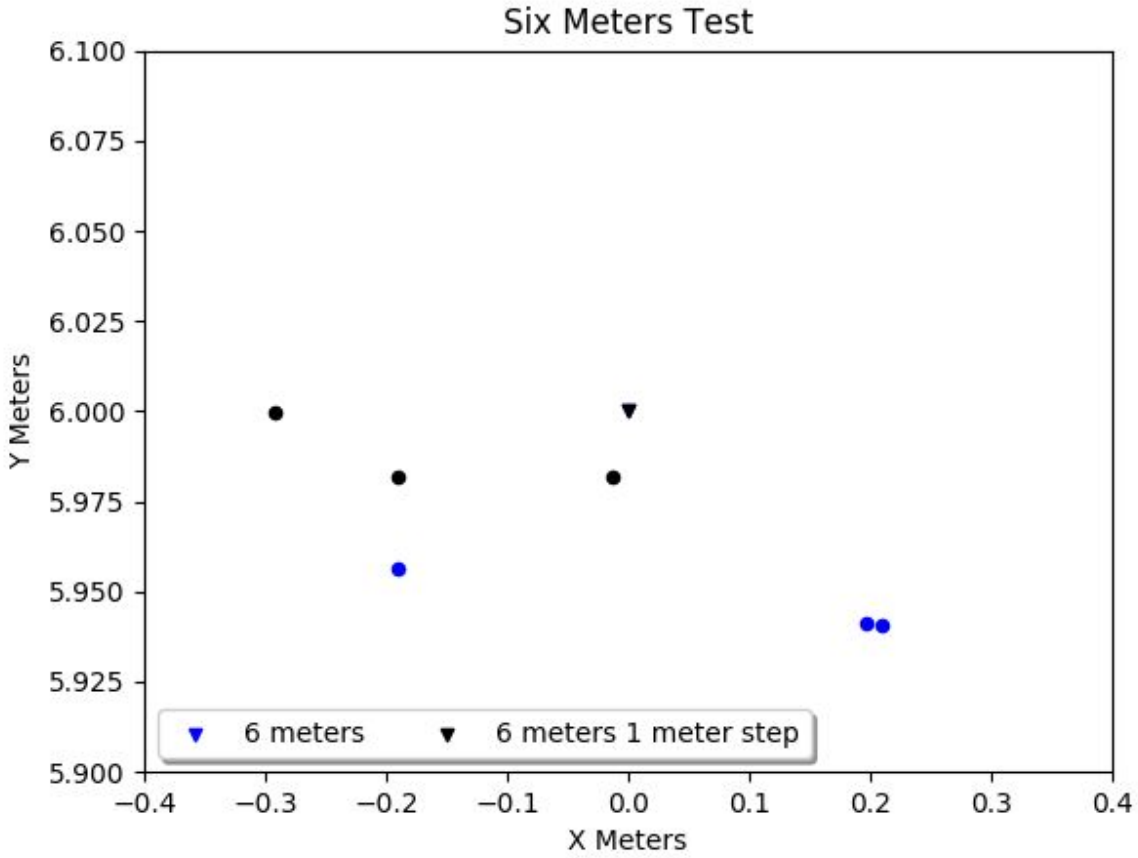

Figure 4: Six Meters Test

Next, we test for the accuracy of rotation in place, as shown below in Figure 5. We tell the rover to continuously rotate +-10°, +-30°, +-60°, and +-90° each with three trials. We represent the angle of the rover with a unit vector dot represented on this graph. The (0,1) position is the starting vector at 0°. Worst case error is about 4°. Each of these rotations is stepped at 10 degrees. For example, the ninety-degree turn is nine ten-degree turns. This is because we can tune the ten-degree turn to be as precise as possible, increasing our overall accuracy.

The triangles in the graph represent the actual rotation for a given degree. The circles represent how much the rover actually turned. Each of these tests is shown in a different color code. Three important things are of note. Firstly, the data is very tightly clustered around the actual rotations, meaning the triangles are mostly obscured. This is a good thing as it demonstrates that our data is fairly precise. Secondly, it should be noted that the data varies to both the left and right of the actual. This is also a good thing, as hopefully over multiple ten-degree turns the errors will average out. This is more noticeable in the zoomed graph, but the amount of error is

not going up with the number of ten-degree steps. For example, the ninety-degree stepped turn is not significantly less accurate than the ten-degree turn.

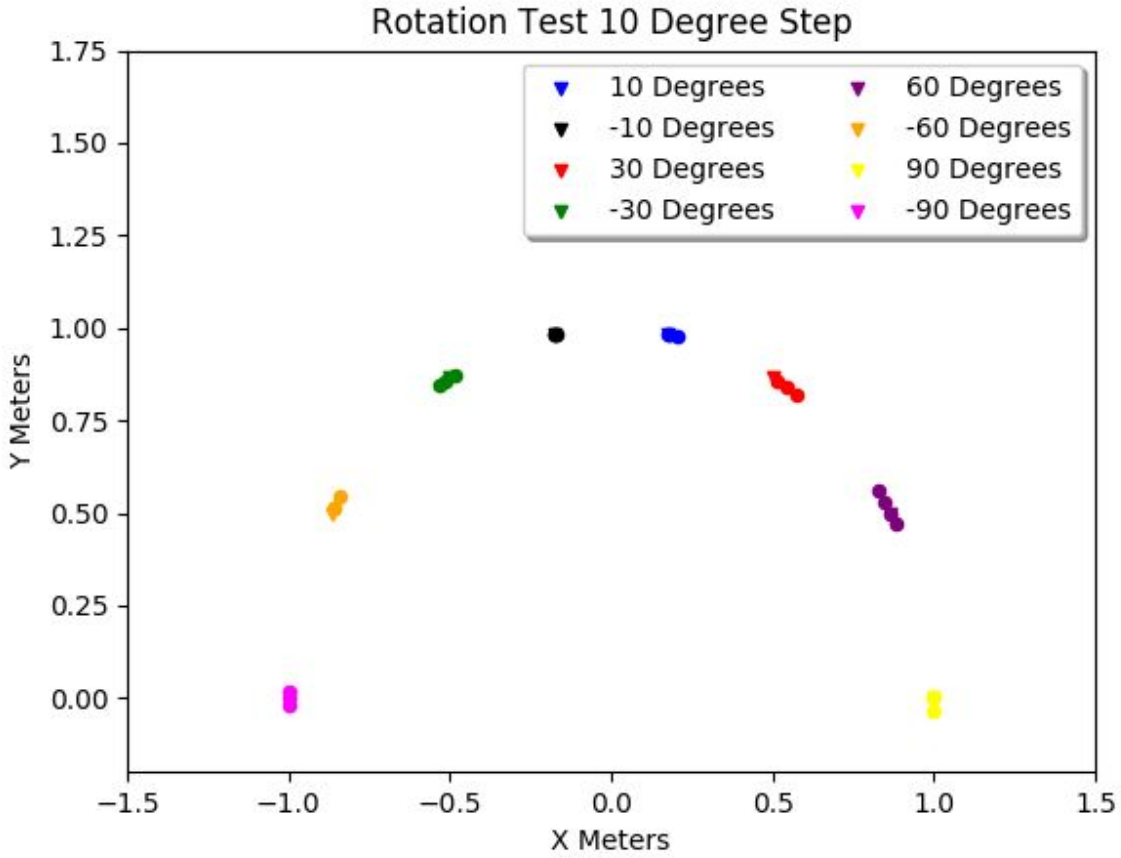

Figure 5: Rotation Test, Ten-Degree Step

Below in Figure 6 is a zoomed-in view of the positive side of the rotation graph. This more clearly illustrates the amount of variance that should be expected when turning. Getting the rotation to be as accurate as possible was one of our primary goals over this semester as it is crucial for the location accuracy of our rover. Section 4.10 goes into more detail on how to potentially counteract rotational accuracy. Additionally, better hardware more suitable for reliably turning might also be recommended.

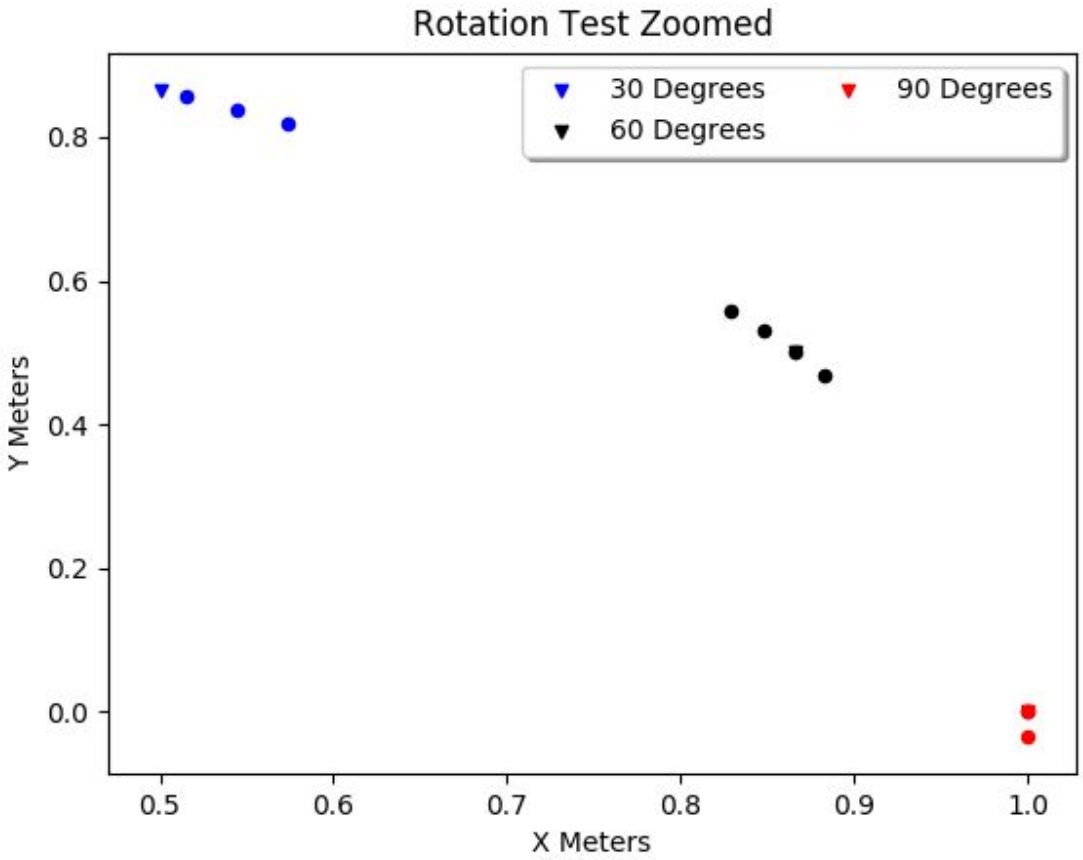

Figure 6: Rotation Test, Zoomed

CHAPTER 4: Navigation

4.1 Introduction

Navigation is one of the core challenges of our project, as our rover needs to be able to traverse indoor environments safely and reach its goal. Additionally, it needs to know its precise location at all times in order to take accurate Wi-Fi samples. In order to reach this goal, all of the rover's subsystems were used in conjunction.

4.2 Previous Work

Last semester, a reliable and useful code base was created for working with the rover's LiDAR. This code was packaged together in a single python file called lidar.py. This file uses a specific data structure and provides reliable access to LiDAR data via multiple read functions, as well as data correction and validation, measuring distances at specific angles, plotting functionality, and ensuring that a path is safe for the rover to travel. This mini-library of useful LiDAR functions was instrumental in building the main navigation code, as it provides vision which can be used to move safely and generally navigate the environment.

The other subsystem used from last semester was the code base for working with the rover's motors. All of the code for this module was packaged together in a python file called motors.py. This code base included functions that allow the rover to move a specific distance either forward or backward, and to rotate a specific number of degrees. It also reports how far the rover actually moved using its encoders. Furthermore, it provides access to where the rover believes it is currently located, relative to the start location. It should be noted that the accuracy of these functions was improved this semester, as detailed in Section 3.4.

Finally, the fingerprinting app was used to collect fingerprints every meter. This module was not fully developed by the start of this semester. This is discussed in further length in Chapter 5.

4.3 Overview

The navigation module of the rover is split into several distinct parts. Each part has its own function, purpose, and place in the control flow. A diagram depicting the control flow and module names is depicted below in Figure 7.

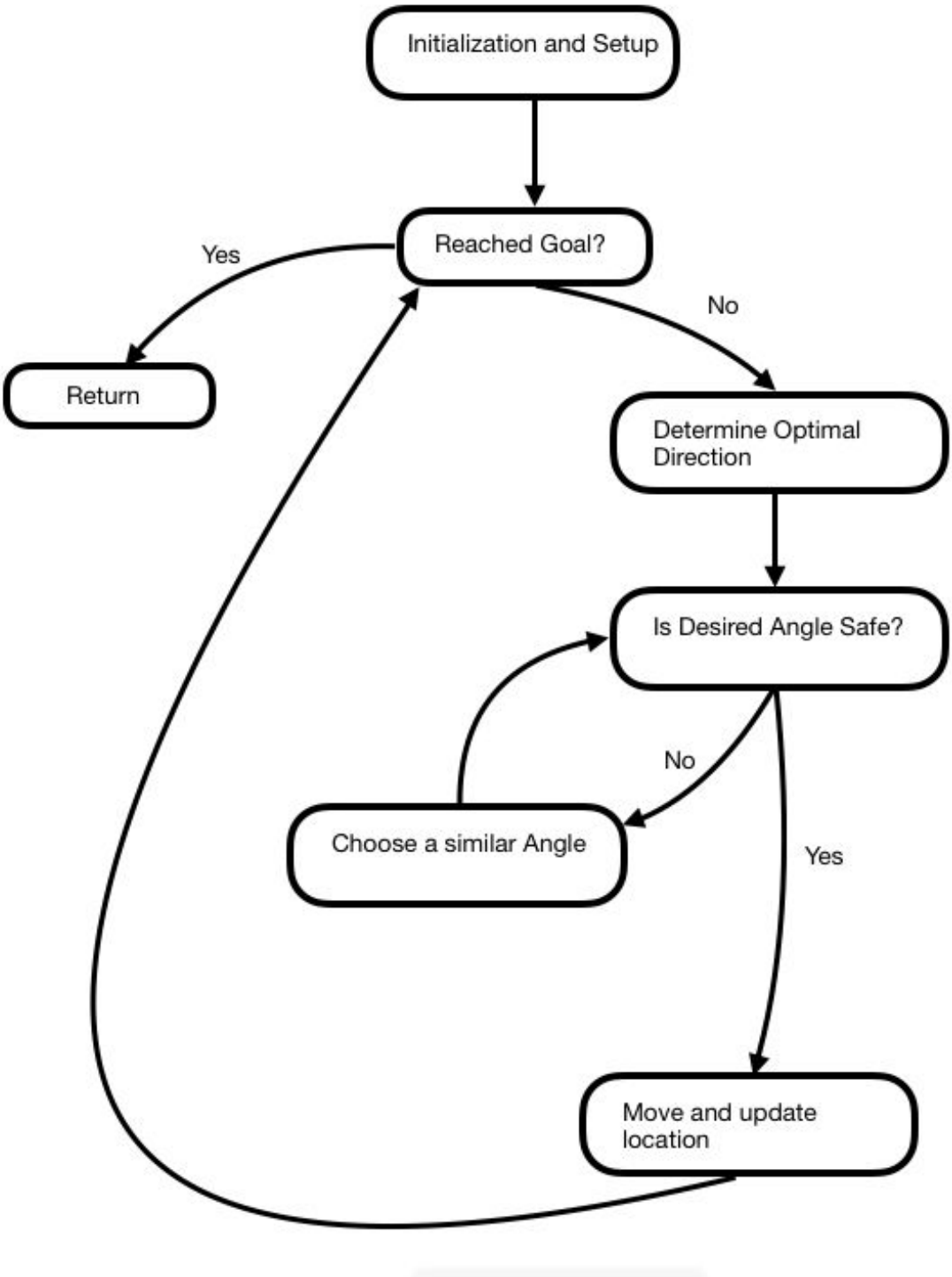

Figure 7: Navigation Control Flow Diagram

4.4 Initialization and Setup

The initialization and setup steps are fairly basic. First, it halts all rover motion. Then it simply ensures that it has all of the required data to establish its goal. This primarily includes its current location and orientation, along with the goal's distance and direction, as well as the range of acceptable distances. This data can all be provided automatically, but if it does not exist, it asks the user for it. Then it does some error-checking to ensure that all variables are within bounds, and establishes a range of acceptable location values that should be considered as the rover having reached its destination. As part of the setup, this step also creates a new instance of the Hallway class, as discussed in Section 4.9.

4.5 Determining if the Rover has Arrived at its Goal

This first portion of the loop takes two data points into location. The first datum is the location. If the rover is not within the minimum distance requirements for the goal, it returns and passes control onto the next item in the control flow. Alternatively, if the rover's location meets the minimum distance requirements and the navigation module knows what direction it is going to travel in next, then it will look for a hallway or open space in that direction. This is the second data point. If it does not find any, and the location is not too close to the maximum specified, it will move forward and look for open space on the next iteration. This process looks for open space by collecting data on the normal hall distances while it is traveling down the hallway, compares a current distance to the existing distance, and tries to determine if the new data should be considered as an outlier. If it is, then that direction likely has a lot of open space. This is discussed in more detail in Section 4.9.

4.6 Determining the Optimal Direction

To determine the optimal direction, the rover ignores the existence of obstacles. This simplifies the process of determining which direction to consider first, as it always considers the potential best case first. The rover determines the optimal direction by first measuring the middle of the hallway. This is done using the Find Middle function, which is described in Section 4.9. If the rover is near the middle of the hallway, it simply continues on towards the goal. If the rover has turned so that it is no longer facing the goal, it will return the desired angle that faces it back towards the goal. However, if the rover is outside of the middle of the hallway, its desired angle will point it back towards the middle.

The rover should stay in the middle of the hallway for three reasons. First, the ideal Wi-Fi fingerprints are taken near the middle of the hallway as that is a decent approximation of where most people stand or walk in hallways. Taking a Wi-Fi sample near the walls would not be representative of most use cases and would likely produce a significantly different Wi-Fi sample. Second, the rover tends to drift to the left or right and is incapable of measuring the drift. If the rover is in the middle of the hallway, it can reset the component of its direction that is orthogonal to the angle of the hallway to be in the center. This counters accumulated error. Third, in the event that the LiDAR misses an obstacle, the middle of the hallway is the safest place to be.

4.7 Determining a Safe Direction

The navigation software can use the LiDAR library to determine if a direction is safe to move in. It simply gives the function the angle, distance, LiDAR data, and width of the rover. The width of the rover is constant and set to half a meter. The distance is set to the sample distance which is usually set to one meter. Therefore, the only variable is the angle. The first angle this module tries is the desired angle returned from the optimal direction. If that angle is unsafe, it looks at angles to the left and right of the desired angle. The rover is tuned to turn in 10-degree increments with the most accuracy so that both the optimal direction and this software module will only consider multiples of ten. If this tuning is ever changed or updated, the code will not need to be modified. This is true for all tuning variables.

4.8 Movement

This function is extremely simple. Once the navigation software has determined what angle it should move in, it simply uses the motors module to rotate towards that direction. It then moves forward one meter. Finally, it updates all of the state variables that relate to location and orientation.

4.9 Find Middle Function and the Hallway Class

The Hallway class is used to contain distance data on a specific hallway. This allows for statistical analysis of new data. The primary use of this class if the Find Middle function. This function allows the rover to determine where the middle of the hallway is. It does this by measuring the distance from the rover to the wall (or closer obstacles) at several angles by using the LiDAR function Get Distance At Angle. These angles are determined by using offsets from the rover's angle relative to the hallway. These angles are designed to point towards the sides of the hallway. From there it uses trigonometry and the angle it was measured at to determine the component that is orthogonal to the angle of the hallway. These orthogonal distances usually represent the x-component of the distance vector. The Find Middle function uses all the data points collected in the hallways and the current measurements. Any of the current measurement pairs (left and right side of the rover) that it considers outliers, it ignores. The centers of the remaining measurement pairs are then calculated. The centers are then averaged, which becomes the calculated value for the middle of the hallway. Ignoring outliers helps the Find Middle function ignore data points from obstacles obstructing the LiDAR or data points from doorways or hallways. Either case would skew the calculation for the middle of the hallway and should be ignored.

As previously mentioned in Section 4.5, the other use for outlier detection is to detect hallways when the rover has reached its goal and knows that it needs to turn.

4.10 Improvements

The first way this module could be improved is by improving the Hallway class. This class works pretty well for excluding data points when it comes to detecting the middle of the hallway, but not well with detecting the presence of the hallway. This is likely due to the inherent limitations of the LiDAR. In places with large distances, the LiDAR tends to skip over angles with large distances. This results in fewer packets, which in turn results in each data point representing an average over a larger sector. This results in measuring what should be large distances not tripping the outlier threshold, while more medium distances are reported accurately, and therefore are considered outliers. Additionally, there could be other problems preventing Hallway from working as reliably as the other modules.

Secondly, only one form of drift is accounted for. Drift to the left or right is account for through the Desired Angle function and the Find Middle function. On the other hand, rotational drift is never counteracted, and could potentially accumulate. Drifting to the left or right is usually consistent over a single session of using the rover and therefore quickly accumulates error. This is why the Find Middle function was developed to counteract the drift. However, in the case of rotation, the errors should average out over time. As this is not guaranteed, it is recommended that a function to counteract this is developed. This could be done using something very similar to the Find Middle function, but instead of measuring the centers of pairs of measurements, lines could be formed from data of the walls on either side of the hallway to calculate the angle relative to the rover. The Hallway class should be used to drop outliers. This functionality was not developed, as it was a lower priority because the errors should cancel out instead of accumulating.

4.11 Testing and Analysis

The Find Middle function was tested by measuring the middle of the hallway manually and then comparing that to what the rover measures. The hallway walls were smooth and straight, and the Find Middle function was extremely accurate. This is shown in Figure 8**.** The triangles in the figure represent the actual location and the circles represent the measured locations. Through observing the behavior of the rover in hallways with extrusions and obstacles, the accuracy of the Find Middle function is significantly improved by using outlier removal through the Hallway class. However, this is difficult to test empirically, as the LiDAR does not produce consistent data over time, and data at different locations along the hallway is required for the Hallway class. Therefore, the results are not easily repeatable.

In Figure 9, the actual location is plotted versus the rover's reported location. The triangles are the actual location and the circles are the rover's reported location. Cardboard boxes were added as obstacles so that the rover is forced to turn and avoid them. The third stop is very accurate and the circle is beneath the triangle. In general, these results demonstrate that the rover's location is accurate enough for the measurements the team requires.

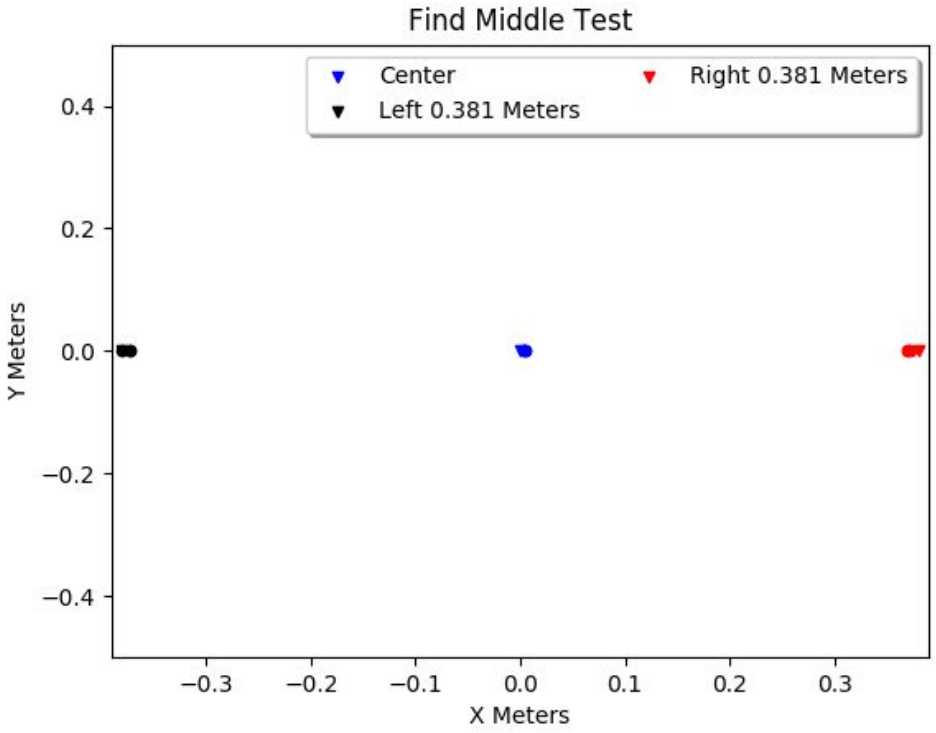

Figure 8: Find Middle Test

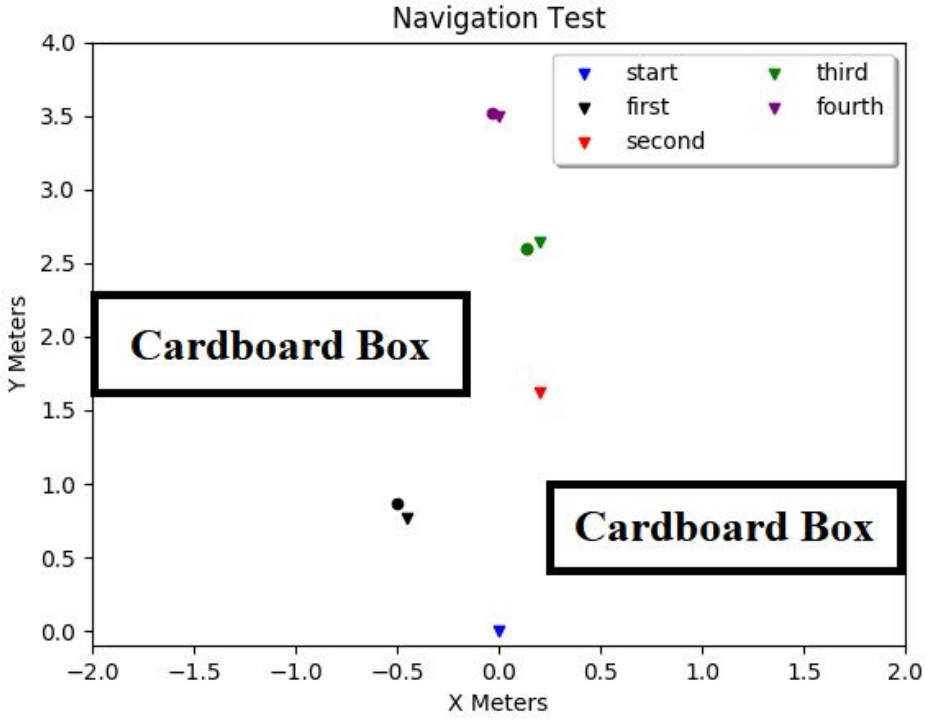

Figure 9: Navigation Test

CHAPTER 5: ANDROID APPLICATIONS

5.1 Overview

One of the main goals of this project was to allow for the autonomous collection of fingerprints using the rover we built. To actually perform the Wi-Fi scans, a smartphone application needed to be developed. This app needed to be small, reliable, and efficient at scanning wireless access points, connecting remotely to a predetermined server (in this case, the Raspberry Pi mounted on the rover), and reading from/writing to CSV files stored internally. The completed app can then be installed on multiple smartphones, up to three of which can simultaneously be mounted on the rover. Then, the Raspberry Pi can communicate directly with each phone over separate TCP connections, signaling each phone when the rover has stopped and then providing the current location to be used for each set of fingerprints.

5.2 Configuring the Raspberry Pi to Act as a Standalone Access Point

Before the smartphone app could even be written, the Raspberry Pi needed to be configured to act as a Wi-Fi access point running a standalone network. This way, each smartphone with the app that is mounted on the rover has a way to directly access the Pi, thereby allowing instructions and location signals to be communicated easily. This was accomplished by following a guide on the Raspberry Pi website [2].

To configure the Pi as an access point, a couple of utilities needed to be installed first, namely dnsmasq and hostapd. Then the Pi needed to be configured to have a static IP address assigned to its wireless network interface, which was done by editing the dhcpcd.conf file located in /etc/. Next, the DHCP server (provided by the dnsmasq software) was configured in such a way as to allow the Pi to statically assign up to nine unique IP addresses to connected devices (smartphones), with addresses between 192.168.4.2 and 192.168.4.10. This was done by editing the dnsmasq.conf file located in /etc/. Finally, the access point host software (hostapd) was configured to specify what type of network the Pi would broadcast, the name of the network, the password to access it, and various other standard settings, including which channel to broadcast on and the type of security/encryption scheme protecting it. To do this, the hostapd.conf file was created in /etc/hostapd/ and pointed to by the hostapd file in /etc/default/. Once all of that was set up, the service could be started, as well as be made to automatically start when the Pi was turned on.

Connecting a smartphone to the Raspberry Pi's new standalone network is then a simple matter of enabling Wi-Fi, scanning for available networks, and connecting to the correct one, using the SSID (RamLocPi) and password (RamLoc19) that were set when configuring the Pi.

5.3 TCP Server/Client Connections

For the Raspberry Pi to reliably and effectively communicate information between itself and any connected smartphones, we wrote a Python script for the Pi to act as a TCP client that can perform three different actions. First, when the main navigation script starts, signifying the start of a run by the rover, the client is called to ping all the IP addresses of its network, to which any connected phones respond with an acknowledgment. All confirmed IP addresses are then saved locally on the Pi to be used for future use. Second, every time the rover stops, the client is called again and supplied with the (x, y) coordinates of the rover. It sends these coordinates to each smartphone (using the locally stored IP addresses), which uses them to couple to MAC address signal strengths to create fingerprints (see next section). Finally, when the rover is done with its run, the client is called one final time to request the CSV files containing all the gathered fingerprints from each smartphone (again using the locally stored IP addresses). On the smartphone side, a TCP server is set up so that each request by the Pi client is received by each phone and handled appropriately.

5.4 Fingerprint Collection Application

Our fingerprint collecting app needed to be built to fulfill several purposes. First, it needed to be allowed to enable Wi-Fi on the phone and connect to the Raspberry Pi's network, which requires network permissions from the app to be allowed. Next, it needed to set up and configure a TCP server on the smartphone for sending and receiving data to/from the Pi (as stated in the previous section). The app also needed read/write access permissions to the phone's internal storage, so that it could read from and write to the fingerprints.csv file stored in the app's data directory (or create it if it does not already exist). Finally, the most important requirement of the app was to scan the surrounding area, match each signal strength to its unique wireless access point MAC address, and then pair that data with the rover's current location, which was sent by the Pi along with the request for a fingerprint scan. Each scan produces one fingerprint, which is then added to the CSV file as a new row.

Because this app was written for the Android API, storage, location, and network permissions need to be granted for it to work on any phone on which it is installed. Without these permissions, the app will open but do nothing. Most of the problems that were encountered early in the development stage were entirely to do with the format of the CSV file when new fingerprints were written to it. Because the CSV files generated by this app are used by the end user app for training a machine learning algorithm, these CSV files need to be in a very precise format. The first row acts as a header containing all the different MAC addresses seen along the rover's run, with the last two columns containing "x" and "y." The remaining rows contain the individual signal strengths of each MAC address, as detected from the passed in (x, y) location, with the last two columns containing these coordinates. However, due to the nature of performing Wi-Fi scans at various points resulting in seeing different numbers of access points, it makes more sense to append fingerprints to the unfinished CSV files with the (x, y) coordinates placed in the first two columns. Once the rover is done and the CSV files are sent to the Pi, the locations are moved to the final columns (see final paragraph of this section).

Initially, the app attempts to read from the fingerprints.csv file in the app's data directory, if it exists. If this file is not empty, the app extracts all existing MAC addresses and adds them to an array list. When the Pi client sends the app a request to perform a Wi-Fi scan, the provided (x, y) coordinates are stored and the app's WifiManager object is instructed to start a scan. The WifiManager then gets the scan results and calls the processWifiScanResults function. This function goes through the list of scan results and adds any new MAC addresses to the list of existing ones. Then it initializes all signal strengths to -100.00 dB, representing a signal that is too weak to reliably detect. Finally, the function steps through the scan's results and attempts to match each individual MAC address and signal strength with the index of that MAC address in the existing array list, thus updating the signal strengths for any access points that are in range. From this is created a new array list comprised of signal strengths, matched to indices of MAC address. This array list is then passed back to the calling function, which pairs the rover's current location with the signal strengths, creating a fingerprint which can be appended to the CSV file. Once that is done, the app sets the doneScanning Boolean variable to "true," prompting the server to send back a message to the Pi stating that the app has completed the requested scan.

5.5 CSV Finalization

Once the rover is done with its run and the client sends a request to each smartphone for their CSV files, the app on each phone sends its copy of fingerprints.csv to the Pi. These CSV files are each renamed according to the IP address of the smartphone which created them, as seen by the Pi. To finalize the CSV files, the client script on the Pi moves the (x, y) coordinates from the first two columns of the file to the last two columns, and then performs some adjustments to convert the location values from meters to pixels. These adjustments are map-specific and need to be determined manually by taking two known points that the rover has stopped at and locating these points on the map for that area, measured in (x, y) pixels from the top-left corner. Doing this will result in three figures: the x- and y-offsets (in pixels) and a value for pixels per meter. Once this is done, the CSV files are finalized, saved locally, and ready to be used for training the KNN machine learning algorithm that makes up the first part of the end user app (see next section).

5.6 RamLoc Application

Once the CSV files are finalized, they can be combined in one place and placed on a smartphone that has the RamLoc app installed on it. This app was written by our graduate student advisor, Saideep Tiku. The app requires two files in its own directory: a PNG map of the area which the user wishes to use to find their location, and a CSV file containing all the gathered fingerprints. The names of these files are referenced within the code of the RamLoc app. The app itself has two stages. First, it uses the CSV file to "train" the KNN algorithm, essentially building a database for the next stage to refer to. Then, with the end-user holding the phone, the second stage is essentially a loop that scans for Wi-Fi signals, compares the collected data to the trained data using the KNN algorithm, and returns an estimate of the user's location. This location is conveyed to the user in the form of a small baseball hat overlaid on the PNG map of the building. The accuracy of this location is greater with more gathered fingerprints.

5.7 Functional Testing

To test the fingerprint collection app, a basic user interface was also created. This interface displays the name of the CSV file (fingerprints.csv by default), the current (x, y) location, the status of the TCP server connection (IP address and port), and a button to manually perform a Wi-Fi scan. Each scan takes up to three seconds, and once complete, a listing of all nearby access points is displayed in order of decreasing signal strength, along with their MAC address and SSID. Below this list is a small window that displays any TCP requests from the client. Additionally, the phone's built-in file manager can be used to access the CSV file being used to manually verify that it has been updated after each scan.

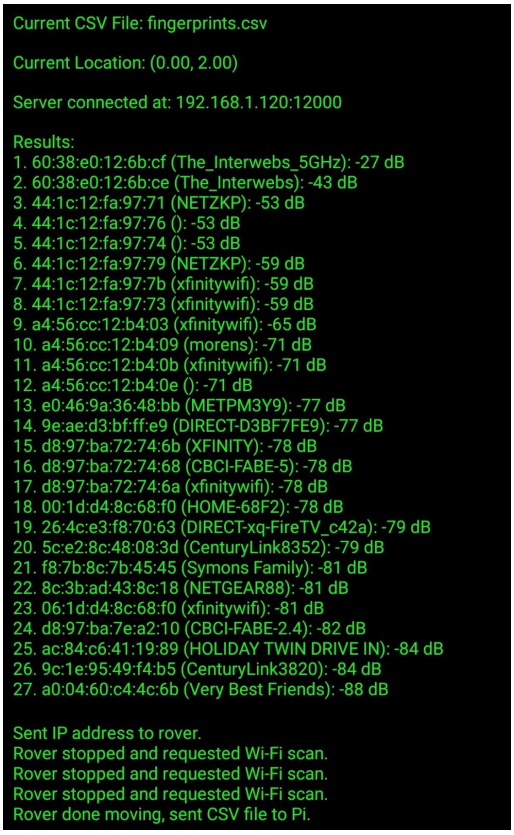

Figure 10: Fingerprint Collection Application

To test the functionality of both apps, we tested the entire system. We instructed the rover to navigate down a hallway for a specified distance and stop every meter. Before starting the run, the Raspberry Pi scanned all its assigned IP addresses and collected the list of smartphones mounted on the rover. Then, at each stopping point, the Pi sent the rover's location to the phones and requested a Wi-Fi scan. To improve the accuracy of the end-user app (RamLoc), we instructed this Wi-Fi-scanning step to be repeated three times at each stopping point. Finally, at the end of the run, the CSV files were retrieved from each phone and finalized on the Pi. After manually transferring the CSV files to a phone with the RamLoc app on it and renaming them according to the map being tested, the app could then be run. Our final trials showed that the process does work well, though it could use more strenuous stress-testing.

CHAPTER 6: STANDARDS

For the most part, this project did not require us to follow any set engineering standards or protocols, as we were creating a rover of our own design and writing all the code from the ground up. A good majority of this project required us to program in Java and Python, and to do so, we did follow common coding conventions for both. For the apps we wrote in Android Studio, we followed the Android Open Source Project (AOSP) coding style [3]. For the Python scripts, we followed the PEP 8 style, which is generally thought of as the standard for Python [4].

CHAPTER 7: CONCLUSIONS

Our team successfully achieved all of the goals we set out to accomplish a year ago. We have written a functioning app that can be used for the collection of Wi-Fi fingerprints. This app, which can be controlled by the Raspberry Pi, collects and saves the fingerprints it creates in the correct format within a CSV file. The movement of the rover has been refined and is now as accurate as currently possible. This, coupled with a robust LiDAR mini-library allowed us to achieve our goal of safe, reliable, autonomous navigation. Due to our work on the rover's tuning and error counteracting methods, our rover is also accurate enough to collect accurate fingerprints. We have demonstrated our rover's capability to generate usable data for the end-user app.

CHAPTER 8: FUTURE WORK

Another team is going to continue our work on this project. They will be focusing on the end-user app and developing additional functionality and reliability. Our portion of this will be providing extensive notes on all of our code and on setting up the rover. We will also be providing the new team with access to our Google Drive, GitHub, and Slack, as well as giving them our contact information for any questions they might have.

From our conversations with Professor Pasricha, it seems like the focus of next year's team will be on developing the end-user app, which was written by Saideep Tiku and has not been modified by us. Essentially, our team goals were focused on the autonomous collection of accurate fingerprinting data. The next team will be focused on using the collected data in new and interesting ways in the end-user app. These two goals have almost no overlap. The thing we can recommend is that the next team keep in regular contact with Saideep Tiku, the graduate student assisting our work, and Professor Pasricha, who both have a strong vision of the path ahead for continuing work.

One important recommendation that we have for the next team is to do some post-processing on the collected data and remove the Raspberry Pi's own access point from the data collected via the rover. In order to provide a static IP address that the connected phones can connect to, the Raspberry Pi needs its own access point. The fingerprinting process measures all of the available access points, so the Raspberry Pi's access point is always in the data and thus needs to be removed. Another recommendation is to look into raising the collected phones further off of the

ground. Currently, the phones are mounted to the rover at a level that is significantly below waist height, which may affect the collected data, as around waist height is generally where people tend to hold their phones when using them.

For next year's team, please note that, because of the nature of the LiDAR and its very unreliable data, placing obstacles or support structures close to the rover will result in a significant degradation of vital LiDAR data. Therefore, we have placed the LiDAR at the top of the rover to improve visibility. However, if we were to elevate it to waist height, the rover would no longer be able to avoid obstacles at ground level. This is an engineering challenge that the next team may or may not be required to solve, based on their research of whether this problem results in significant enough error in Wi-Fi fingerprints.

Next year's team will need to use our code base and rover to collect data. We do not expect that the next team will need to spend much time modifying our code, but if they have to for any reason, we have left extensive comments throughout all of our code. Each method has at least one comment explaining its purpose.

We are also in the process of placing notes on the rover's usage in the 'notes' directory of the team's GitHub and/or Google Drive. These notes will detail several things, including:

- How to correctly wire both the motors and motor encoders to the Raspberry Pi hat.
- How to correctly connect the primary batteries to the motors and electronic speed controllers.
- How to charge the primary batteries, safe operating procedures, and how to prevent the batteries from draining while not in use, without disassembling the rover completely to get access to the batteries.
- How to correctly connect the secondary phone battery to both the Raspberry Pi and the LiDAR.
- How to connect the USB cable from the LiDAR to the Raspberry Pi.
- How to switch the LiDAR from receiving power from the USB cable to receiving power from an external power source.
- How to plot LiDAR data continuously on an updating graph.
- How to drive the rover manually.
- How to set up the rover so that it can move down hallways without input.
- A list of known issues with the rover and its navigation software.
- How to use both the fingerprinting app and Saideep Tiku's RamLoc end-user app.
- Recommendations for further development of our code.
- How to disable the Raspberry Pi's access point so that it can connect to the csu-eid network normally.

REFERENCES

[1] "PID Controller," wikipedia.org. [Online]. Available: https://en.wikipedia.org/wiki/PID_controller.

[2] "Setting up a Raspberry Pi as an access point in a standalone network (NAT)," raspberrypi.org. [Online]. Available: https://www.raspberrypi.org/documentation/configuration/wireless/access-point.md.

[3] "AOSP Java Code Style for Contributors | Android Open Source Project," android.com. [Online]. Available: https://source.android.com/setup/contribute/code-style.

[4] "PEP 8 -- Style Guide for Python Code | Python.org," python.org. [Online]. Available: https://www.python.org/dev/peps/pep-0008/.

APPENDIX A: ABBREVIATIONS

App	Application
CSV	Comma Separated Values
DHCP	Dynamic Host Configuration Protocol
ESC	Electronic Speed Control
GPS	Global Positioning System
IP	Internet Protocol
LiDAR	Light Detection and Ranging
LiPo	Lithium Polymer
MAC Address	Media Address Control Address
PID	Proportional/Integral/Derivative
PWM	Pulse Width Modulation
RMS	Root-Mean-Square
RSSI	Received Signal Strength Indicator
SSID	Service Set Identifier
USB	Universal Standard Bus
WAP	Wireless Access Point

Table 1: List of Abbreviations

APPENDIX B: BUDGET

Item	Transaction	Running Total
Initial Funding		
CoE funding ($200 x 3 team members)	+$600	$600
Leftover funds from previous team	+$106	$706
Project Spending		
75:1 Motors with Encoders ($37 x 2 + $10 shipping/tax)	-$84	$622
75:1 Motors ($22 x 4 + $10 shipping/tax)	-$98	$524
Replacement ESCs ($11 x 2 + $8 shipping/tax)	-$30	$494
172:1 Motors with Encoders ($35 x 2 + $18 shipping/tax)	-$88	$406
172:1 Motors ($20 x 4 + $22 shipping/tax)	-$102	$304
Robotic Wheels ($17 x 3 pairs + $16 shipping/tax)	-$67	$237
Batteries ($56 x 2)	-$112	$125
Battery Chargers ($28 x 2)	-$56	$69
Battery Status Checker	-$20	$49
All Battery transaction tax	-$14	$35
Technical Spending		
None	-$0	$35
Administrative Spending		
None	-$0	$35
Total Budget Left		**$35**

Table 2: Project Budget

APPENDIX C: PROJECT PLAN EVOLUTION

Deliverables/Objective	**Date**	**Assigned Roles**
Fix broken wiring on EDC	9/15/18	Lane, Mitch
Order motor with encoders and step down for Pi power	9/16/18	Lane
Fix broken servo hat connection	9/18/18	Team
LiDAR operational and reading data correctly	9/28/18	Ethan
Fully functional driving	10/1/18	Lane, Mitch
Pi communicates with smartphone to take fingerprint	10/5/18	Mitch
Rover can navigate simple hallways	10/12/18	Lane, Mitch
Android app can take fingerprints	10/26/18	Team
Rover can take fingerprints with Android devices	11/16/18	Team
Testing collection phase with the rover is fully complete	12/7/18	Team

Table 3: Fall 2018 Project Timeline

Deliverables/Objective	**Date**	**Assigned Roles**
Replace motors	2/5/19	Lane
Fix Raspberry Pi access point	2/23/19	Mitch
Rover drives autonomously	3/1/19	Ethan
Replace wheels	3/4/19	Lane, Mitch
Finish rover movement accuracy tuning	3/26/19	Lane
Replace batteries and chargers	4/12/19	Lane, Mitch
Implement TCP server/client connection between Pi/app	4/12/19	Ethan, Mitch
Testing rover accuracy	4/15/19	Ethan, Lane
Fingerprint collection integration with autonomous navigation	4/17/19	Ethan, Mitch
E-Days poster	4/18/19	Team
E-Days	4/19/19	Team

Table 4: Spring 2019 Project Timeline

ACKNOWLEDGMENTS

We would like to thank Saideep Tiku for providing us with patient guidance toward accomplishing our goals this semester. We would also like to thank both Professor Sudeep Pasricha and Saideep Tiku for providing us with the opportunity to contribute to the continuing development of indoor localization.