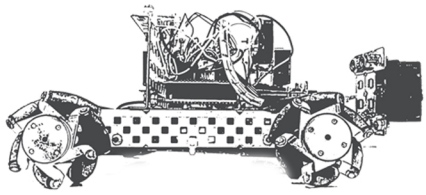
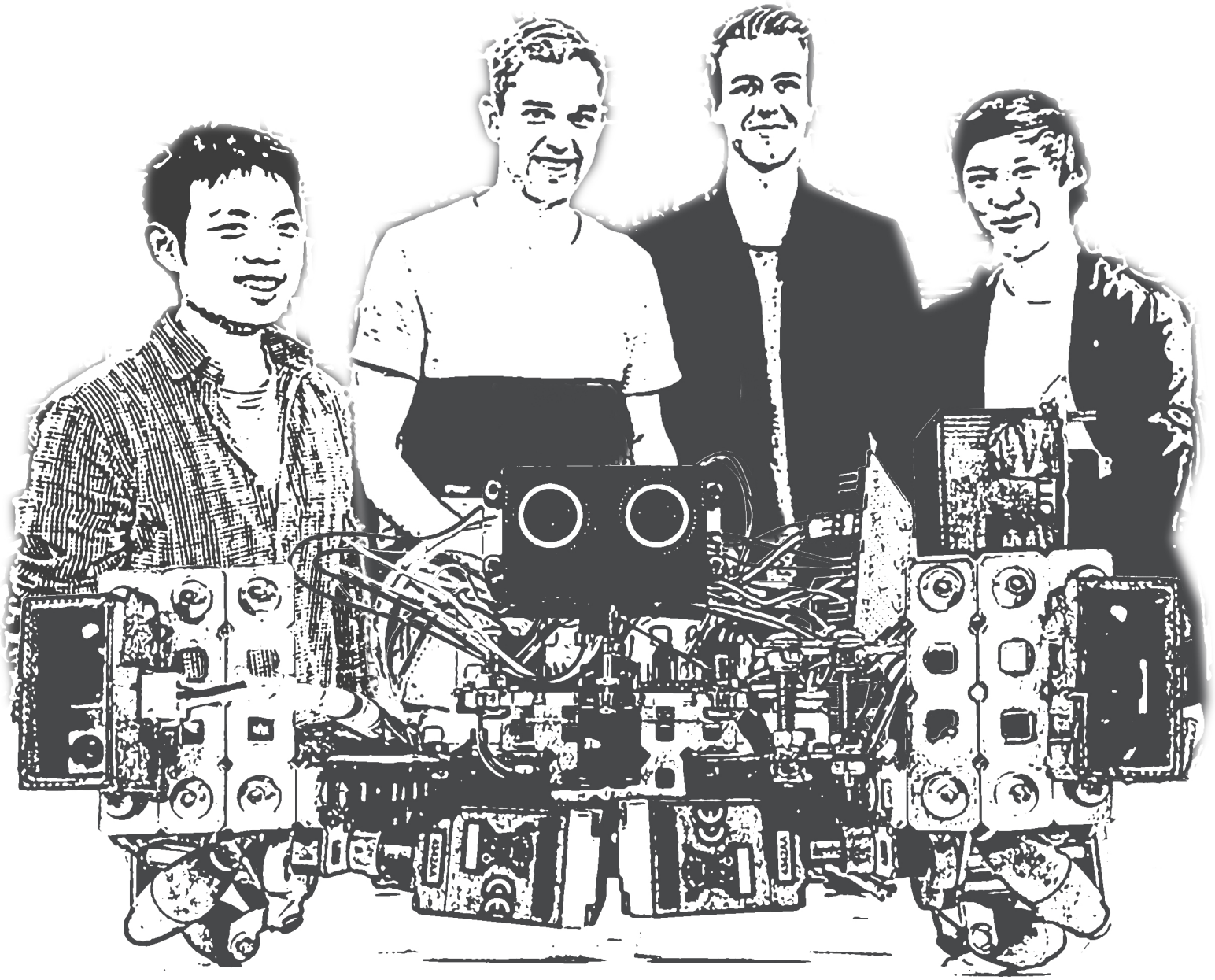


AUTONOMOUS SLAM ROBOT

Blair Duncan
Craig Tims
Sam Wrait
Andy Yu



MECHENG 706 **Group 5**

Executive Summary

Simultaneous Localisation and Mapping (SLAM) is a difficult problem to solve. This is because of the cross dependency of both aspects of the problem: mapping and localisation. The aim of the project is to design autonomous robot system that incorporates the SLAM algorithm to cover a room for the purpose of vacuum cleaning. This project involved many aspects of the Mechatronics discipline; Mechanical design, hardware calibration and software intelligence. These aspects were simultaneously integrated throughout the whole project.

The hardware included four infrared(IR) ranging sensors, one ultrasonic ranging module, one motion processing unit (MPU), a servomotor, a bluetooth module and a partially built robot with 4 Mecanum wheels. These are used in combination to create sensory perception for decision making processes and the resulting motion generated by the autonomous robot.

This report contains the methodologies and ideas behind finding a solution for the assigned problem, including the thinking behind the path taken by the robot, obstacle detection and avoidance and mapping method.

Ultimately the demonstration run achieved perfect obstacle avoidance with a relatively quick run. The coverage of the room was an issue however, as the robot lost its reference after a misalignment, resulting in only about 50% coverage. As such, there are aspects for improvement. The utilisation of the provided servomotor would help improve the robots ability to distinguish between obstacle and walls. More accurate and tolerant sensor signal processing could help with reliability and reduce misalignment. Finally the application of full SLAM, where the created map if used to avoid with navigation and path optimisation would greatly improve the performance of the robot in a variety of courses.

Contents

List of Figures

1 Introduction

Simultaneous Localisation and Mapping (SLAM) involves finding the location of a moving object in an unknown environment while at the same time creating a map of it. The aim of this project is to create an autonomous robot capable of performing SLAM to 'vacuum clean' a small, controlled environment. An Arduino controlled, omnidirectional robot and a small range of sensors are provided to achieve this aim. In order to function as a vacuum cleaner, the robot has three main goals. It must cover as much of the room as possible, while avoiding any obstacles. This coverage should be completed in as short a time as possible.

This is often considered a difficult problem as neither the position of the robot, nor the layout or size of the room is known. As a result, it is important to accurately collect and interpret data about the movement of the robot and its surroundings. As every positional estimate is based on a multitude of factors, as shown in Figure 1, incorrect associations and errors can quickly accumulate, causing the robot to lose its reference leading to catastrophic consequences.

Furthermore, to carry out these complex tasks, it is necessary to integrate a range of sub-systems to successfully work together. To do this, processing power, time allocation and program memory must be shared between tasks in order for the robot to carry out measurements and respond to its environment in a timely manner. These sub-systems include, but are not limited to, visual sensors, a blue-tooth data transmitter, servomotor actuators and signal processing modules.

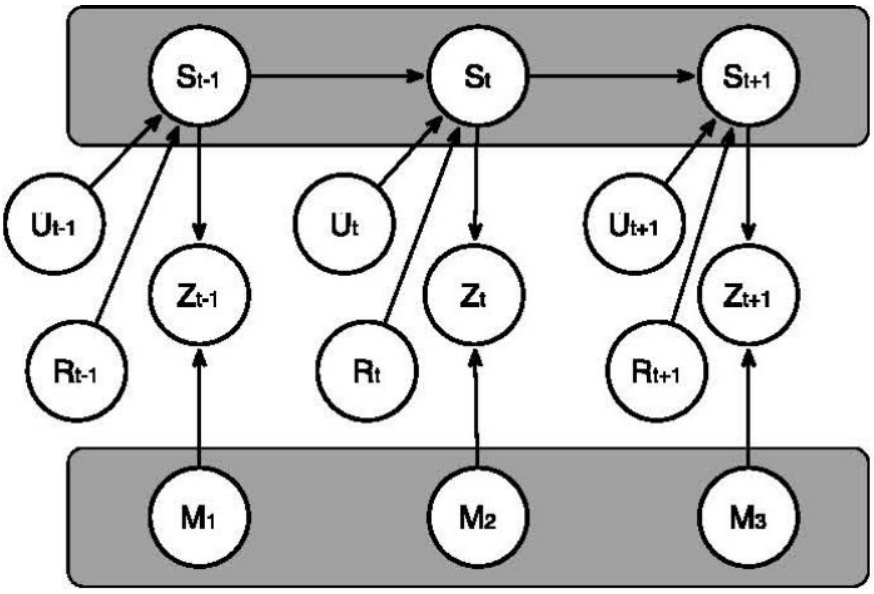


Figure 1: Conceptual Diagram of the SLAM Problem - involving information about the robot state (s), odometry (u), sensor measurements (z), landmark positions (r) and environment map (m) [1]

2 Problem Description

The main requirement of this project is to design and build an autonomous robot capable of Simultaneous Localisation and Mapping (SLAM) to vacuum clean a rectangular room. The robot must avoid making contact with any obstacles or walls it comes across. A partially completed mechanical design comprising of a chassis, four motors drivers attached to Mecanum wheels, a power supply and an Arduino Mega. To complete this project a series of sensors were mounted and software was written to carry out the required task. Limitations on hardware included no more than four infrared (IR) sensors, one ultrasonic sonar, a servomotor, a Motion Processing Unit (MPU) and a blue-tooth module for communication with a computer or phone.

3 System Overview

3.1 Robot Path

Three different pathing strategies were considered under which the robot could traverse the room; an inward spiral, outward spiral or a zig-zag. Difficulties arise in finding a reference point from which to start when using an the outward spiral due to the presence of obstacles, leading to its dismissal. An inward spiral allows one side of the robot to always face the walls of the room, eliminating the need for sensors to face both sides which would be required in the case of a zig-zag path. An inward spiral was determined to be the simplest and most efficient path to implement and is illustrated in Figure 2.

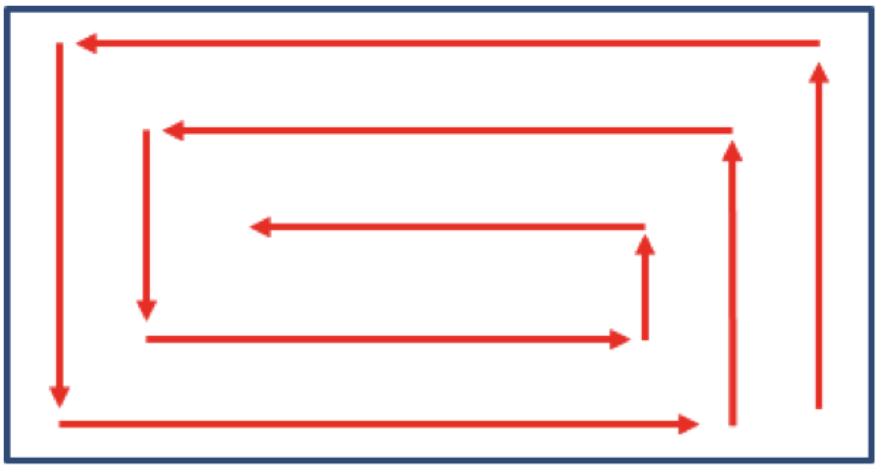


Figure 2: Predetermined Inwards Spiral Path

3.2 Sensor Placement

Since the path of the robot is a spiral, the sensors are placed in a suitable manner. Two long range infrared (IR) sensors are mounted facing the sides of the robot, allowing functionality for alignment purposes and more reliable information on the orientation and location of the robot relative to the boundaries of the room. Since the long range IR sensors have a longer

sensing range, theoretically, they will still be accurate as the robot moves further into the centre of the room. To eliminate problems that arise from objects or walls entering the sensor dead bands, a short distance in front of the long range IR sensors (approximately 20 cm), they are positioned so that these dead bands span across the width of the robot, and measurements can be obtained from as close to the robot as possible.

Two short range IR sensors are mounted at the front edges of the robot for the purposes of obstacle and wall detection. Initially, the short range IR sensors were placed too close to the ground. This caused some of the emitted IR light to bounce off of the ground and produce inaccurate measurements. A solution was implemented by mounting the sensors at a higher position.

The sonar has the greatest accuracy and range of all the provided sensors. It is used to determine how far forward the robot moves by measuring its front distance relative to the wall ahead, whether it is far away or close to the robot. The sonar is centred at the front of the robot, which allows the detection of obstacles directly in front of it that the short range IR sensors would not be able to see. The sonar is mounted on top of the servomotor, widening its potential to a range of other uses, such as obtaining data about the blind side of the robot by rotating the sonar 90° counter-clockwise for a wider field of view.

Since the sensors are pointing at different directions across the robot, they are mounted in a manner as to not interfere with each other's field of view. This is a problem mainly for the long range IR sensor and the sonar that are located closer to the front of the robot. The positioning of the sensors on the robot can be seen in Figure 3. The MPU is mounted near the centre of the robot at a reasonable height, in order to mitigate the noise, particularly electromagnetic, that the chip may experience during robot movement.

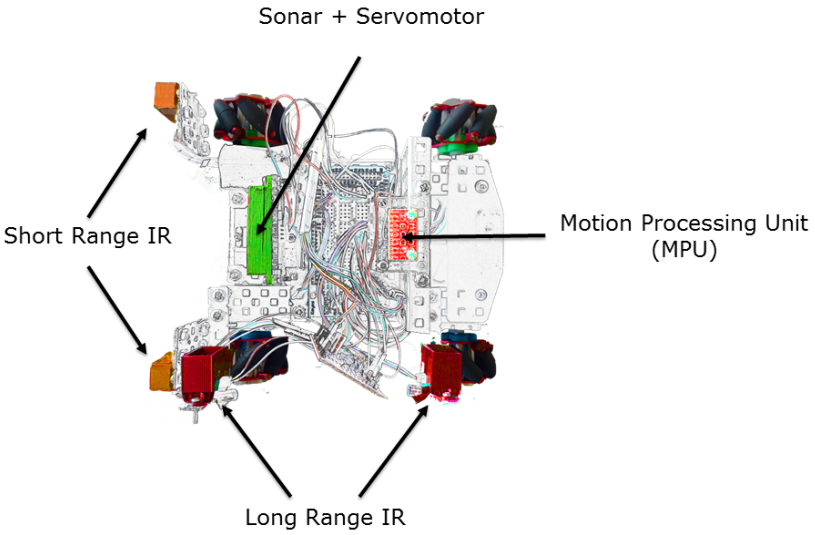


Figure 3: Sensor Placement

3.3 Software & Hardware Integration

The software for the processing of the sensor signals and the actuation of the robot should accommodate for how the sensors are mounted. This is done by adding offsets, obtaining best fit equations and limiting the allowable ranges of the sensed signals using saturation levels.

4 Hardware

4.1 Short Range IR Sensors

The short range IR sensors have a useful sensing range of 4 cm to 30 cm. Electrolytic 100 μF bypass capacitors are soldered between the ground and source pins of the sensors to improve the signals received by eliminating spikes. Rectangular covers shown in Figure 5 are also placed around the sensors in an attempt to provide more consistent readings in different environments with different lighting. Multiple calibrations were done in order to find a best fit model for the short range IR sensors. Figure 4 shows the plot of the relationship obtained between the measured distance using the manufacturer's recommended equation and the actual distance. It can be seen in the plot that the relationship is not directly proportional so a new equation of appropriate fit was obtained and added onto the recommended equation to obtain the final model for each sensors. A linear relationship was determined with the front left sensor shown by the blue line and the right front sensor shown by the orange.

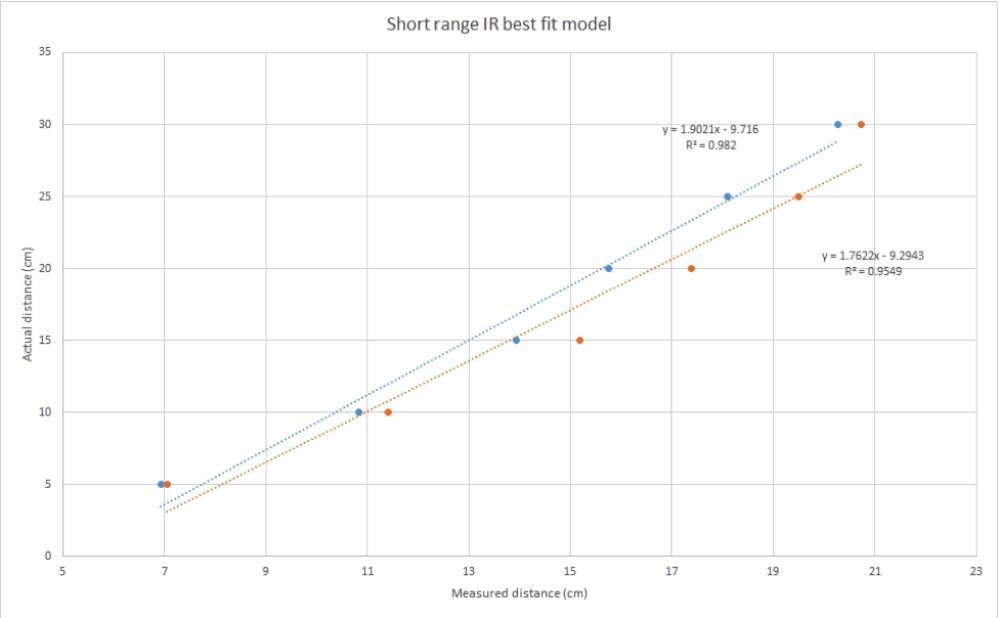


Figure 4: Short Range IR Sensor Calibration

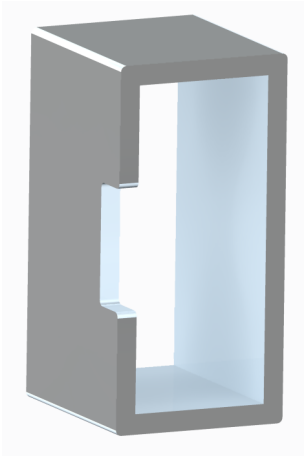


Figure 5: Rectangular Casing for IR Sensors

4.2 Long Range IR Sensors

The long range IR sensors have a useful sensing range of 20 cm to 150 cm according to the specifications. Through experimental results, the accuracy of the sensors was seen to decrease significantly beyond the range of 80 cm. The long range IR sensors were also calibrated multiple times, upon mounting onto the robot. An example plot of the relationship found between the actual distance and the measured distance is shown in Figure 6, where it is seen to be a 2nd order polynomial equation. The same cover shown in Figure 5 was 3D printed and added for the long range IR sensors to improve its performance by mitigating inconsistencies from variations in lighting. A bypass capacitor is also wired to each sensor to decrease signal spikes.

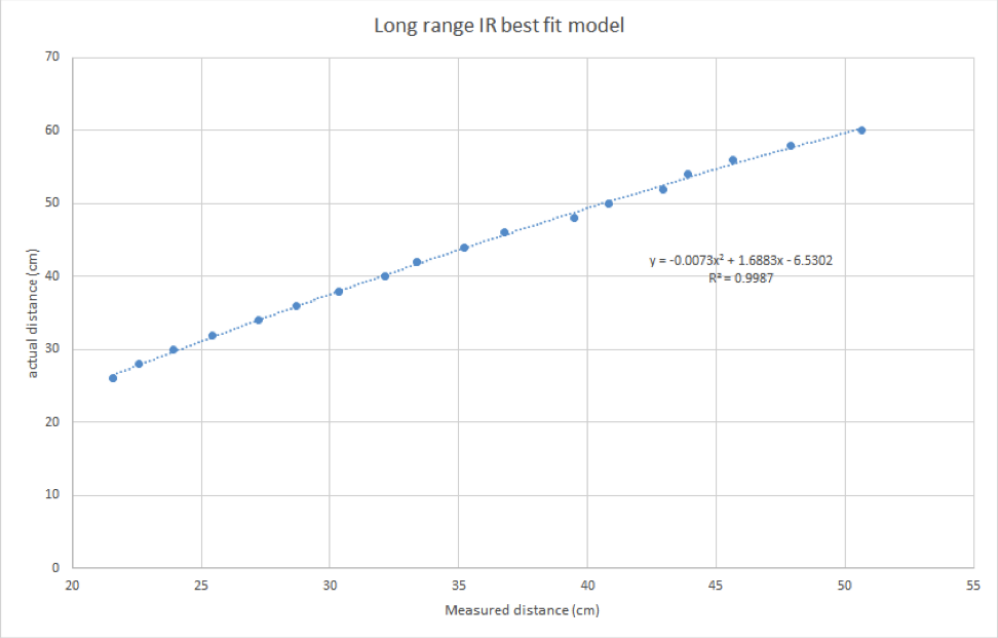


Figure 6: Long Range IR Sensor Calibration

4.3 Analogue Filter for IR Sensors

Due to a significant amount of noise present in the IR sensor signals, a simple passive low pass analogue filter was proposed. The circuit was implemented on veroboard as shown in Figure 7. However after implementing this design it was found that there was a signal attenuation. Therefore instead of connecting the IR sensors to the filter, bypass capacitors were added to the IR sensors instead.

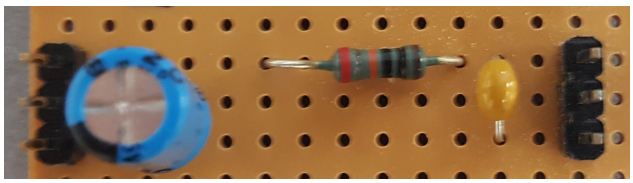


Figure 7: A Low Pass Filter Implemented on Veroboard

4.4 Sonar

The HC-SR04 ultrasonic sonar sensor (Figure 8) has a useful sensing range of 2 cm to 400 cm. This proved to be the most reliable and versatile of the ranging sensors available. Measurement readings from the sonar sensor involved generating a short signal of approximately 10 microseconds and then reading the resulting signal from the receiving echo pin. Prior to each sensor reading the trigger pin is held low for 2 microseconds to ensure a clean trigger pulse.

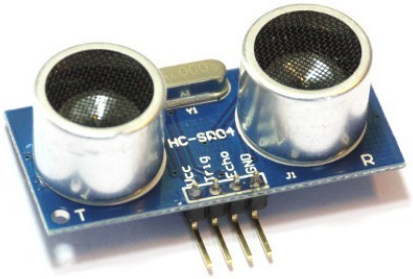


Figure 8: HC-SR04 Ultrasonic Sonar Sensor

As specified by the information found on the data sheet, the sonar sensor should not have a measurement cycle less than 60 milliseconds to ensure a reliable measurement. Originally this was implemented using the delay command to allow the project to progress. However as timing became more of an issue and the fact that the sonar is continuously read, lead to the conclusion that this was an insufficient method as it meant there was a large amount of downtime in the program. To fix this a simple conditional statement makes use of the Arduino's 'millis' counter to ensure the sonar module is not read too often while also ensuring that the reading frequency is not so low as to force the robot to rely on outdated data.

In order to place the sonar sensor on the robot, an external casing had to be made. The case designed and produced consists of two parts shown in Figure 9 and a full detailed drawings can be found in the appendices. It is able to enclose the sonar as well as provide a connection to the servomotor, while allowing space for the wires to reach their pins.

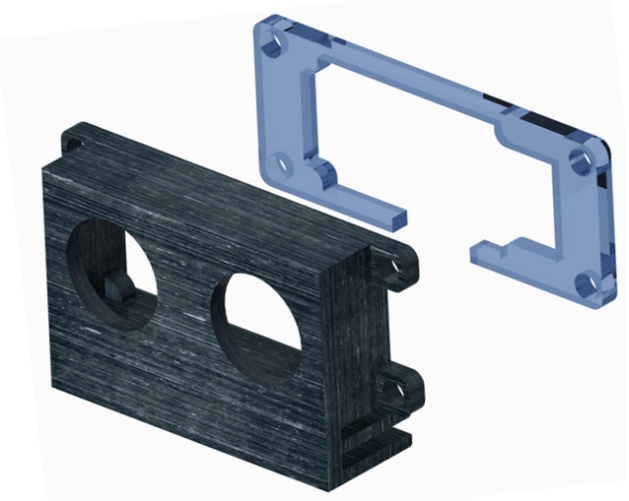


Figure 9: Sonar Sensor Casing

4.5 Motion Processing Unit

The InvenSense MPU-9150 sensor (Figure 10) is used to track the orientation of the robot. The MPU-9150 has nine-degrees of freedom consisting of readings from a magnetometer, accelerometer and gyroscope. The MPU communicates with a main processor using the I2C protocol, consisting of two signal lines (clock and data).

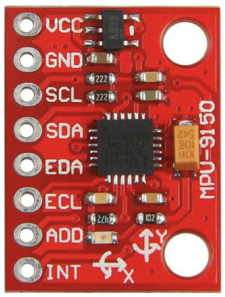


Figure 10: MPU-9150 Breakout Board

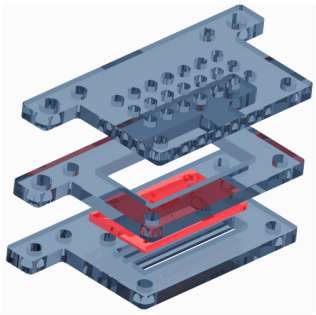


Figure 11: MPU Case

Research into past SLAM projects highlighted the unreliability of using the magnetometer readings to determine the heading of the robot. The unpredictability and inaccuracy of the magnetometer readings arise from electromagnetic interference from the various mounted electronic components (motors and other connected sensors). An attempt was made to minimise this interference by elevating and isolating the MPU above the other components of

the robot. To achieve this a case was made as seen in Figure 11 that effectively sandwiches the MPU to hold it in a constant position and also make it mountable to the robot. However, readings still remained unreliable, with both drift and fluctuations observed, when trying to fuse all the sensor readings. For these reasons, the magnetometer readings were not used in the data fusion process to determine the heading of the robot.

The MPU-6050 open source library that is used is capable of calculating pitch, roll and yaw values (rotations about the x,y and z axis respectively) from signals obtained from the three-axis gyroscope and accelerometer within the MPU. The absence of the magnetometer signals in its calculation for direction heading made the measured values more stable. The only reading used by the robot is the yaw values as operation of the robot occurs only in the x-y plane with rotations about the z-axis. Upon initialisation of the robot, the gyroscope is calibrated, which must be done at rest. Normalised values are read in and the yaw is determined. The code also has a 'setThreshold' method that determines the sensitivity of the readings where movements above the set threshold trigger a change in the yaw readings. A threshold of zero is susceptible to drift while a threshold of three is unable to track very slight changes in yaw from operations such as going forward. A threshold of one was found to be robust against drift while sensitive enough to detect small changes in yaw from slight misalignment in the robot going forward. The yaw drift is compensated for by the on-board Digital Motion Processor (DMP) algorithm which is thought to classify the changes in heading values while the robot remains stationary as error and compensate the outputted values appropriately.

4.6 Servomotor

A servomotor (S05NF) with a rotation range of 180° is used to rotate the sonar. The servomotor can take up to 6 volts and can be used on either the analogue or digital pins found on the Arduino. However after trying many different pins and checking the motor code was correct, the servomotor could not be successfully integrated to work with the rest of the robot. To mount the servomotor onto the robot a platform needed to be made. This required a simple design to be laser cut on a 3mm board as shown below in Figure 12.

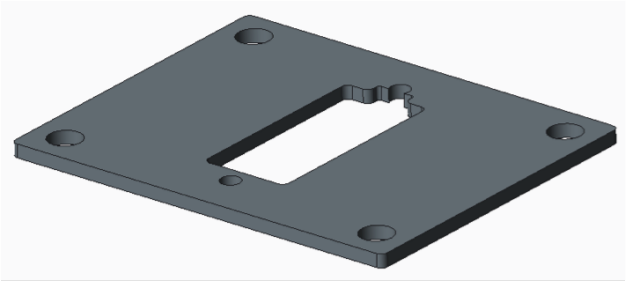


Figure 12: Servomotor Mount

5 Software

5.1 Sub-Functions

Sub-functions were written for robot to produce simple motions that would allow it to navigate its way through the room. This includes going forward, rotating, strafing and aligning against the wall. These functions utilise simple closed loop control with proportional gains to prevent drift due to unequal friction in the wheels and inconsistencies of the servomotors. All code mentioned in this section can be found in the appendices.

Forward Straight

The robot had difficulty maintaining a straight line when moving forward due to the variable friction between the Mecanum wheels and the room floor as well within the motors themselves. A forward straight function then had to be determined that would allow the robot to maintain a straight path. The function utilises a proportional controller to maintain the yaw value of the robot within a certain range.

Yaw Aligning
Utilisation of the MPU's yaw values meant that continuous tracking of the robot direction could be achieved and any deviations from the set direction could be adjusted for by a controller that would determine an appropriate response. A proportional controller was developed that measured the error as the difference between the measured yaw value and reference yaw value given. The reference yaw value is placed in a dead band to prevent unstable oscillations from continuous minor changes around the reference value. The dead band is defined as the reference yaw value +/- 2° and the error is measured from the edges of this band. The controller response involves a rotation of the robot proportional to the error in order to correct the heading of the robot. Use of the yaw for a straight line however is not always reliable when incorporated with the other functions as the reference yaw given is not always at a parallel heading to the wall.

IR Aligning
IR readings are used to continuously align the robot to the wall to ensure it is parallel. This is achieved by taking in the long IR sensor values that face the wall and comparing them to check whether there is a difference. If there is a difference in values, the robot will rotate to maintain its parallel path. A rotation correction was chosen over a strafing correction as although the strafing would maintain the robot at a consistent distance from the wall it could not ensure a straight path for the robot where a path similar to a zig-zag would result. A limitation in implementing this method meant that should there be an obstacle located between the robot and the wall, the robot would then try to align itself to the obstacle, stuttering around it in a circular manner.

Limitations of both these functions were identified leading to their implementation in certain situations, where the 'Yaw Aligning' forward motion is used for the inner loops of the robots spiralling path and avoiding obstacles. While the 'IR Aligning' forward motion is used for the outer loops of the robots path closest to the walls.

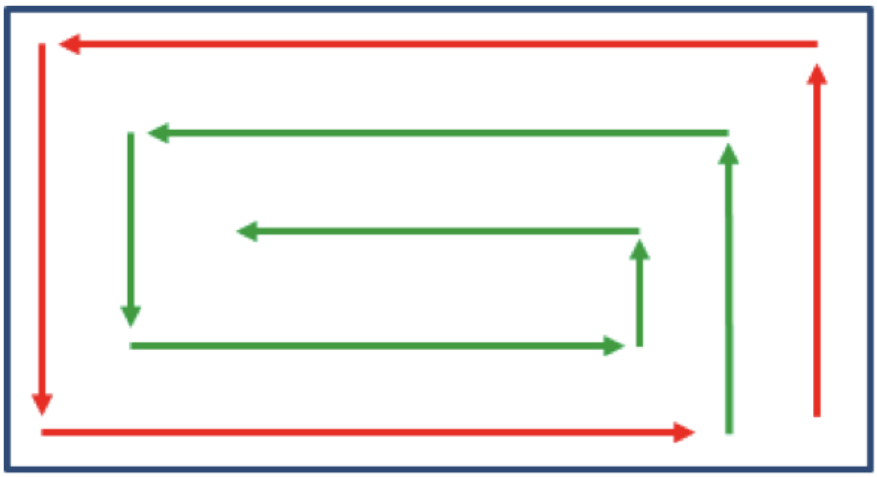


Figure 13: IR Aligning (Red) vs. Yaw Aligning (Green)

Strafe Straight

Similar to the problems faced in maintaining a straight line moving forward, the robot also struggled to strafe in a consistent direction. The inaccuracy of the strafe movement impacted on the performance of the forward straight function mentioned earlier as the reference yaw passed to it was inaccurate. A function is implemented that takes in a reference yaw and a strafe direction to maintain a constant heading of the car parallel to the wall. Although the strafe can still result in a diagonal since the robot is still facing forward the impact on the proceeding forward straights with yaw will be less impacted and achieve a better path. Yaw is used instead of IR readings to keep the robot parallel to the wall during a strafe due to it being a simpler calculation where only deviations from the reference yaw need to be accounted for as opposed to measuring a consistent rate of change between the two IR sensors.

Rotate

The rotate function utilises a proportional control and sensor feedback from the MPU to produce a controlled turn of the robot. This is done by continuously calculating the difference between the robot's current yaw value and the yaw value it is meant to be moving too. This difference is then multiplied by a proportional gain. As this function is solely used to rotate through 90° turns on corners, it was tuned for this use. It was found that a small gain of 6.5 was ideal for a 90° turn. Since the rotate function can not handle overshoot, a secondary check was added after each time the rotate function was called to ensure that rotate function hadn't over or under turned. As the MPU isn't perfectly accurate, the robot needs to align to the wall after calling the rotate function.

This method above is our final and most successful rotation method however it is not the first method attempted. The first method involved rotating the robot for a set amount of time (1.5 seconds) before aligning it to the wall. This method proved to be relatively successful, however other methods were explored that tried to improve upon this.

The second method that was attempted was unsuccessful. This method used the forward_straight function to turn 90° and then in the same function move forward. However due to inaccuracies with the MPU, there was far too much error even when the robot aligned itself before turning. This method could have potentially worked if the program incorporated a strafe function to move either closer or further away from the wall when needed.

Wall Alignment

Unreliability of the rotate function due to the insensitivity of the MPU meant that after the rotate function has been run the robot is not always reliably aligned with the wall. This misalignment causes problems for the forward straight yaw aligning function where it will maintain the misalignment during the whole length of its straight path and never correct to align with the wall. The wall alignment function is implemented to remove this problem, aligning the robot against the wall after a rotation using both wall facing long range IR sensors.

The readings from these sensors are passed into the alignment function, and the difference between them designated as the error. Proportional control is then implemented to turn the robot clockwise or counterclockwise, until the sensor readings are within 0.5 cm. The gain of the proportional control was tuned to minimise overshoot, and an upper limit is also placed on the control value sent to the servos. Once this 0.5 cm condition is reached, the function outputs a true boolean indicating it has aligned.

Testing indicated that as the readings had a tendency to fluctuate, upon occasion the sensor values would briefly come between 0.5 cm , even when the robot didn't align. As a result, a 100 ms delay was implemented to check, after which the readings were checked for alignment again. If they were no longer within 0.5 cm, the alignment code was run again. The whole process has a timeout of 1 second to stop the robot oscillating endlessly in an effort to align. This function can be seen as 'align_wall' in the appendices.

5.2 Main States

The overall control of the robot was carried out using a series of states in a finite state machine shown in Figure 14.

Initialising

This is the initial state, triggered when the robot is turned on, or the reset button is hit. During this state the servo objects are connected to the correct pins on the Arduino and a short delay is then implemented to give the MPU time to calibrate.

Cornering

Once initialised, the robot moves to the cornering state. In this state the robot will find the corner that is in front, and to the left of its current facing. It then uses this a reference/start point. The state continuously calls a function called 'find_corner'. This function returns a boolean that is used to check whether a corner has been found. If a corner has been found

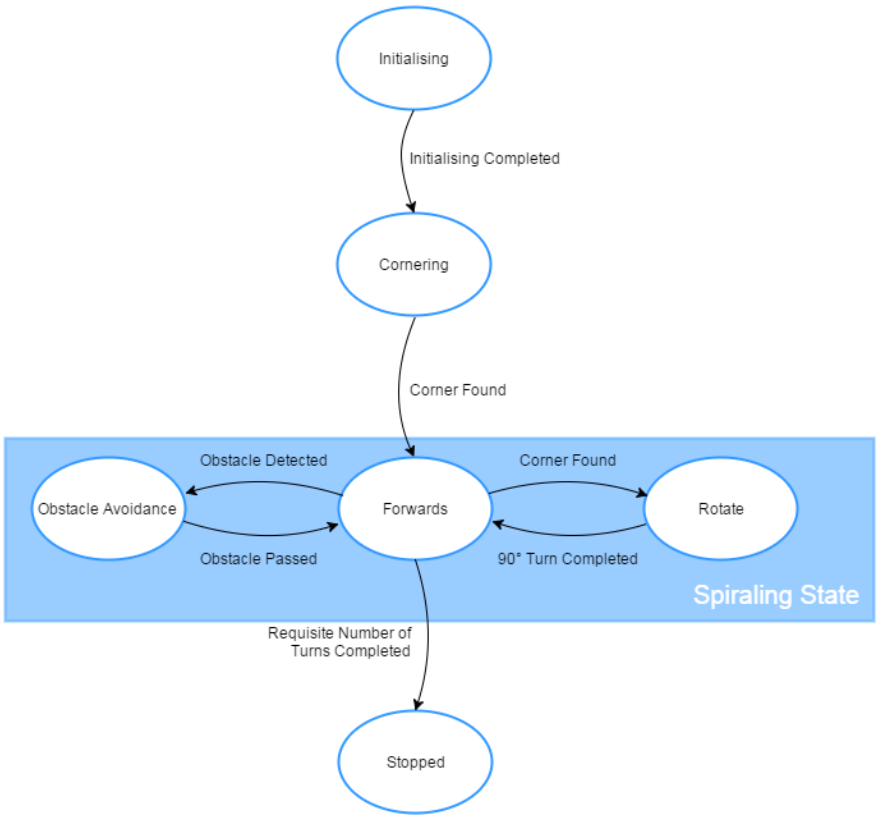


Figure 14: State Diagram

the program exits the 'cornering' state and enters 'spiraling'.

Find_corner works by first looking for a wall. To do this it will move forward while continuously monitoring the two from IR (short range) sensors. Once one of the front IR sensors reads a value below a threshold value (10 cm) the robot will align itself to the wall using the from IR sensors.

To align itself to the wall, the robot continues to read the front IR sensors. It compares these values using the align_wall function mentioned in section 5.1. However after testing this code multiple times it was found the robot could not sufficiently align to the wall leading to a secondary alignment check being implemented This check compares the two front IR sensors and ensures the difference between them is no more than 0.5 cm. If the difference is not below 0.5 cm the program will call align_wall again. Due to the reasonably low proportional gain used in align_wall, the robot struggles to make adjustments when the difference between the two IR's are low. This means the robot would sometimes get stuck trying to align to a wall because it couldn't supply a large enough voltage to the motors to overcome their starting torque. To counter this, Arduino's 'millis()' counter was used to ensure the robot did not spent more than 0.75 seconds to align to the wall. If the robot took too long or was sufficiently aligned, it would break out of this alignment loop.

To find a corner, the robot needs to find a second wall. To achieve this it first rotates 90° by

Figure 15: Wall Alignment

passing a value of 90 into the 'rotate' function discussed in section 5.1. The next wall is then found in a similar way as the first wall is found, checking when the front IR sensor readings go below a threshold value. However this time the robot needs to be travelling straight. This is achieved by calling the 'check_straight' function which uses the long range sensors mount on the side of the robot as discussed in section 5.1.

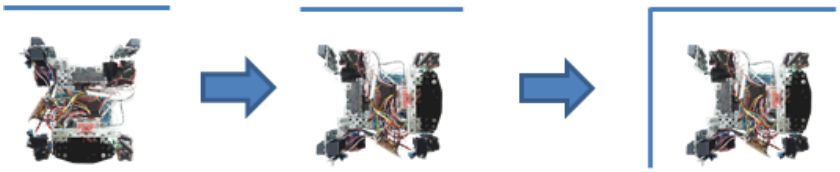


Figure 16: Finding the Second Wall

After completing the previous steps, the robot will have found a corner. However it will first realign itself to the wall it is facing. This is needed as it is difficult for the robot to perfectly travel straight due to inaccuracies in the IR sensors. This alignment works in the exact same way as the previous alignment except this time the conditional only ensures the difference between the IR sensors is less than 1 centimetre. Once the alignment is complete or if 0.75 seconds pass during alignment, the program will return a boolean value of 1 and spiralling will occur.

Different methods were tested to improve the way in which the robot finds a corner. One of these methods was to rotate 360° on start up to find the closest wall. The theory behind this was that the robot could potentially travel an entire length of the track to find the first wall which would waste a lot of time. However after implementing this early on, it was discovered that this was counter productive as it took too long for the robot to rotate a full 360° and then rotate back to face the closest wall. It would also have had to take into account obstacles which would have overcomplicated the program for very little or no benefit.

Another method trialled for finding the corner used strafing. This involved first moving

forward until a wall is found, aligning to this wall and then strafing to the right until the second wall is found. The theory behind this was the robot would require one less turn and alignment, saving a small amount of time. However strafing is slower than moving forward which counteracted the time saved. It also had the problem of colliding with obstacles placed on walls as it had no way of detecting obstacles with the two side IR sensors.

Spiralling

To map the room the robot moved in an inwards spiral direction and requires the robot to be in corner before moving to the spiral state. The spiral state is continuously called and is controlled using a flag called 'semaphore'. Semaphore is either 0 or 1 and is used to control whether the robot needs to turn (0) or move straight (1).

When semaphore is 0 the car needs to turn. This is done by passing a value of 90 into the 'rotating' function as discussed in section 5.1. Prior to calling this function yaw is reset to zero to ensure a correct reference before trying to turn 90°. Due to the various inaccuracies in using MPU further alignment was required after turning 90°. This was done using the 'align_wall' function discussed in section 5.1 with an extra condition to make sure the robot has correctly aligned to the wall.

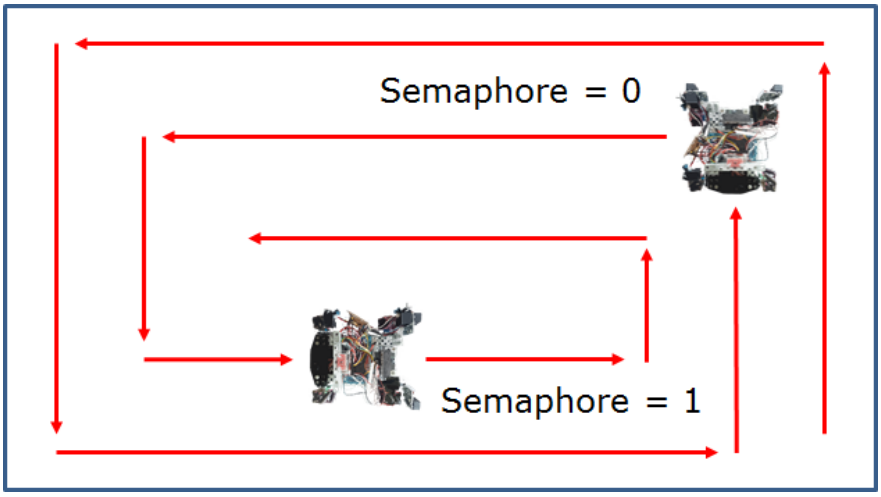


Figure 17: Ideal Spiral Path with No Obstacles

When semaphore is 1 the car needs to move forwards ensuring it stays as straight as possible. For the first 4 lengths this is done by calling the check_straight function which uses the long range IR sensors mounted on the side of the robot to control the straightness of the robot as discussed in section 5.1. For the remaining lengths the robot uses the MPU to control its straightness. While moving forward the robot also monitors all front sensors (sonar and two short IR sensors) to look for obstacles (as discussed in section 5.2 and to determine when it needs to turn. The sonar sensor is used to keep count of how many lengths have already been completed. When the sonar sensor reads a value less than $10 + 22 lengthCount/3$ semaphore will be set to 0.

Obstacle Detection

Obstacles placed around the room are cylindrical in shape with a diameter of about half of the robot's width. An obstacle detection function is used to both assist the robot in distinguishing between walls and obstacles and also to determine whether the robot has passed an obstacle. Fuzzy logic was considered for differentiating between obstacles and walls however it was determined to be much simpler to implement a basic comparative case between sensors where the provided sensors provided sufficient information to make this distinction.

Obstacle in front
With three front sensors facing forward (two short IR & one sonar) obstacles can be successfully identified in front of the robot provided the path is relatively parallel to the wall at the time. The obstacle detection function begins comparing the three front sensor readings when at least one of them reads a value below 15 cm, well in range of all the sensors and close enough for the robot to consider whether to move to avoidance or continue to spiral. Once in range of all the sensors, a comparison is made between all three of them to determine whether there is a large difference in readings that satisfies one of the various scenarios illustrated in Figure 18. Upon fulfillment of these conditions an obstacle is successfully detected in front of the robot and it moves into its avoidance state.

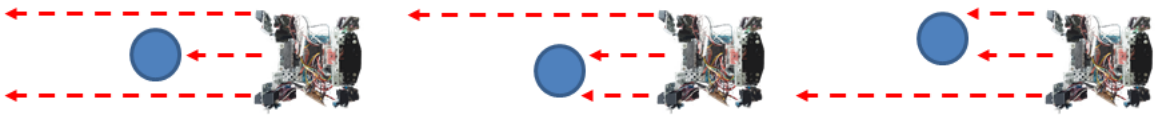


Figure 18: Different Cases for Obstacle Detection

Obstacle passed
Two wall facing sensors (two long IR) are used to determine whether the robot has passed an obstacle. The two sensors take in a reference distance at the start of each straight path and then continuously monitor any large changes in the readings of the sensors from the reference distance that was set. Upon a large change the reference distance sets itself to the current reading to prevent constant triggering of large changes as it is in the process of passing the obstacle. These changes are counted by the function until it determines four large changes have been made, indicating that both IR sensors have passed the obstacle and the robot can return to its original distance to the wall at the beginning of its straight path.

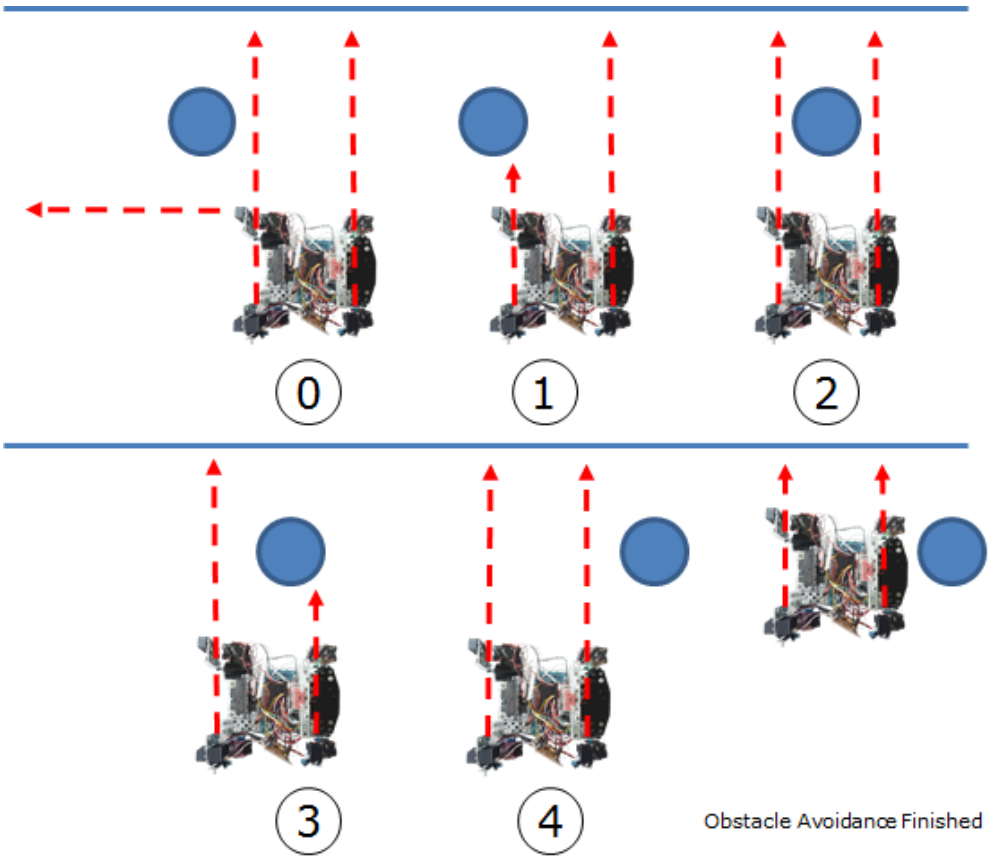


Figure 19: Robot Counting Four Large Changes in Long Range IR Sensor Readings

Obstacle Avoidance

After either the obstacle in front or obstacle passed condition is satisfied, the obstacle avoidance state is entered and determines how the robot responds depending on which condition is satisfied. Originally the obstacle avoidance state entered a fixed sequence of strafing to avoid, moving forward until passed, and then returning to its original position. This implementation limited flexibility of the robot where for the duration of that sequence it would be unable to detect further obstacles. The current implementation enables the robot to be more flexible where it will strafe to avoid and then return to what it was doing (spiral state), able to then strafe to avoid another obstacle or strafe back to its original path.

Robot avoid
An obstacle detected in front of the robot sets a boolean called 'strafe' true. Within this state the robot then begins to strafe toward the centre until it is clear of the obstacle. Upon termination of the robot strafing, the avoidance state returns to the spiral state and sets forward straight to the yaw alignment method. Yaw aligning straight is set over IR aligning straight in this circumstance due to the obstacle interference with the IR aligning.

Robot return
An obstacle detected that the robot has passed sets a boolean called 'needReturn' true.

Within this state the robot then begins to strafe toward the wall until the wall facing sensors read values around the range of the starting reference. Upon satisfying this condition the robot has returned to its original distance for the spiral and returns to its spiral state.

Stopped

During spiraling, the length of the arena is measured, and the number of lengths required to cover the arena is calculated. Once this amount is reached, the robot stops - its run is completed.

5.3 Debugging Methods

For a project of this scale, testing is very important. As the Arduino environment doesn't have very advanced debugging capabilities, the serial monitor and bluetooth were utilised. However the initial bluetooth receiver supplied was found to be faulty. This lead to reliance on bluetooth applications found on the Android app store. A large number of these apps were downloaded and tested, however the preferred application is 'TerminalBT', the user interface for the app is shown in Figure 20. This meant that sensor readings and state information of interest could be read out, allowing for easier identification of the source of a problem.

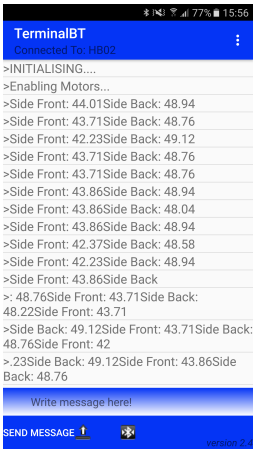


Figure 20: Example Output being Read with TerminalBT

In the program written for this project there is a testing state where sections of code could be tested without running the entire program. This proved to be very useful for sensor calibration/testing. Testing logic was a little more difficult and usually required two stages. The first stage involves finding out where in the program an error is occurring. This is done by printing lines that correspond to different stages in the code and sending them to the bluetooth module. The second stage involves working out why that particular section of code isn't working by sending all related sensor data to the bluetooth device, providing enough information to work out what was going wrong.

6 Localisation & Mapping

6.1 LabVIEW

After consideration of different softwares for use in programming and displaying the mapping functions of the robot, it was decided to implement this function in LabVIEW, mostly due to members existing familiarity with the program as well as its ability to easily create an aesthetically pleasing user interface to display the map.

The first challenge with this was developing a module with which to communicate with the robot. This was achieved through the use of a LabVIEW's Virtual Instrumentation Software Architecture (VISA) VIs. The serial port connection was configured by feeding in the port number and details such as baud rate, data bits and parity. The first buffer usually contained incomplete data, and was flushed using the 'flush I/O buffer.' A timed while loop then periodically read the port and stored the data from the buffer, with a frequency of 200 Hz. This was chosen to be of a much higher frequency than data was output, so that no data was missed. At the end of the program, the port connection was then closed. Figure 21 shows this module.

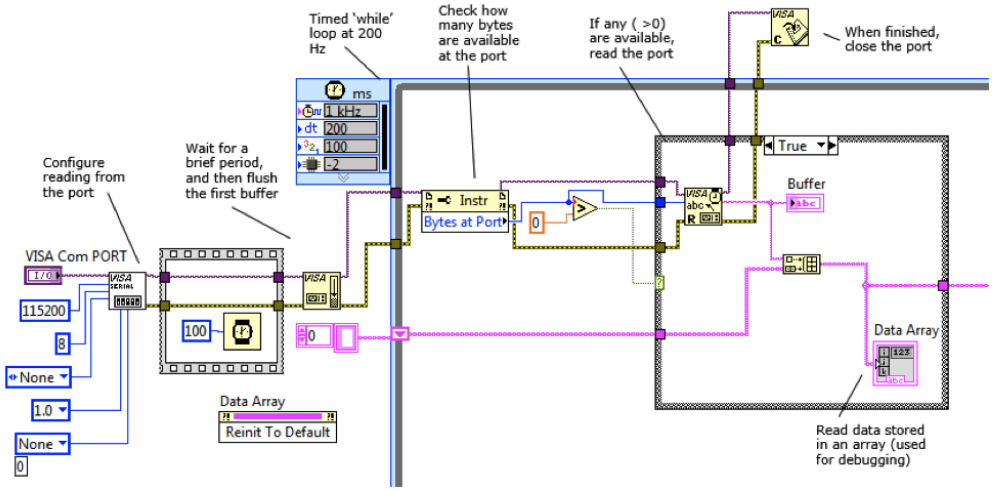


Figure 21: Reading Data from the Bluetooth Receiver in LabVIEW

A problem that was encountered was when the program tried to read the buffer when no data was available. This would sometime crash the program, and later on could interfere with the mapping algorithms. As a result, a vi was used to check how many bytes were available, and the port was only read if this was greater than zero.

Data Collection

In order to obtain the data for these case, the long range IR and sonar readings were outputted to LabVIEW at specific times. This data was sent using the 'Serial1.print' command as well as string formatting operators. Shown in the figure below is the an example of the code used for outputting data. In this case, it shows the data used for tracking the position

of the robot.

```
if (millis() - previous_millis > 50) {
  previous_millis = millis();
  sonar_sensor = sonar();
}

side_front = read_sharp_IR_sensor(SHARP_YA, side_front_IR_pin);
side_back = read_sharp_IR_sensor(SHARP_YA, side_back_IR_pin);


 if ((millis() - bluetooth_millis) > 250) {
  bluetooth_millis = millis();
  bluetooth_string = "1, ";
  bluetooth_string = bluetooth_string + length_Count + ", " + sonar_sensor + ", " + side_front + ", " + side_back + ", 0, " + millis();
  Serial1.println (bluetooth_string);
}
```

Figure 22: Section of Code Sending Sensor Data to the LabVIEW Program

First the sonar and IR readings were obtained. Then every 250 milliseconds this data was outputted to LabVIEW. This value was chosen to be as small as possible, while still allowing time for the sonar to have time to update its reading. The outputted format was as per Figure 23.

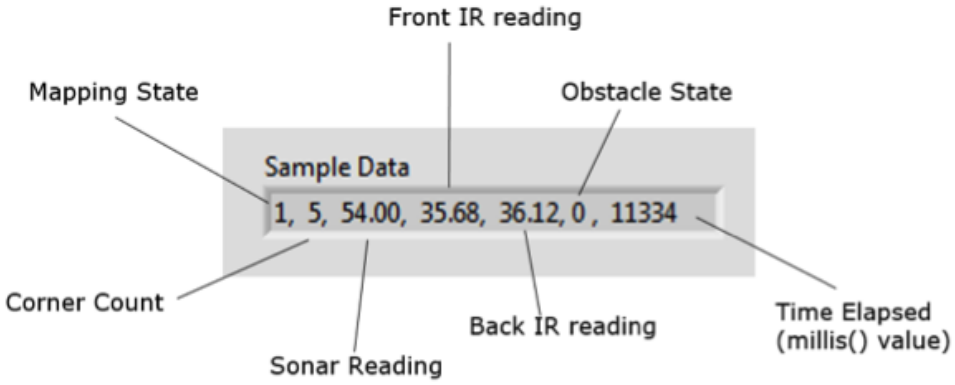


Figure 23: Layout of Sensor Readings in LabVIEW

The next module created split the data obtained via bluetooth into separate numbers. This could then be interpreted by algorithms, based on what mapping state was indicated at the start of the string. The obstacle state was necessary to tell whether the robot was strafing or moving forward, and the time elapsed could be compared to the previous value for determining how much time had passed since the last reading.

6.2 Localisation

A simplified version of the Kalman filter is used for the robot localisation. The algorithm involves combining sensor data with information about the position of the robot, both with uncertainty to produce a more accurate estimate of where the robot is. As it requires very little memory and if very fast to compute, it is ideal to implement in this situation - where sensor data and position contain inaccuracies.

Prediction

The phyiscal model of the robots movement needed to be define. As friction and other resistive forces we ignored, and the assumption the all movement was either straight ahead or straight to either side, this was a very simple model:

$$x_i = x_{i-1} - v\Delta t \tag{1}$$

Where, x_i is the new position, x_{i-1} is the position in the previous state, v is the velocity of the robot in that direction and Δt is the time since the last calculation. This equation could be applied separately to each direction. To apply this equation however, the odometry of the robot needed to be considered to determine the velocity of the robot.

The first method tried using the MPUs accelerometer readings and integrate to find the velocity of the robot at a given time. While this was able to be calculated on the Arduino, the values were not deemed accurate enough. This was due to both the small errors in the readings being magnified by acceleration and the constant realigning of the robot means that its own axes were not always aligned with the global axes. It was attempted to create a model that read the microsecond values submitted to each servo, however the constant realigning of the robot meant that these values changed were quickly and frequently, rendering the model developed ineffective and very complex. Ultimately, all movement in a certain direction was assumed to be at one speed. This meant that the movement speed in each direction of movement was able to be experimentally calculated. The robot was set up in the arena, and timed to determine how long with took to move 100 centimetres using each type of movement (forward and strafing), with 20 values for each direction measured and an average calculated. This resulting in an average forwards speed of 16.1 cm/s forwards and 10.7 cm/s when strafing.

Sensor Data

The collected data from the IRs and sensor needed to be converted from a reading to an estimate of the position of the robot. In order to do so the following, the sensor reading was combined with the coordinates of the corner the robot was facing. The formula of this combination was depending on the direction the robot was facing, and these formulae can be found in appendix XX.

Kalman Filter Fusion

The Kalman filter algorithm has two steps. Firstly a predictive step is carried out to determine the position of the robot based on the previous position and the model of the system. Then this value is fused with the sensor data to give a more accurate prediction of the position. As the x and y coordinates are unrelated values, they can be passed through the filter independently. The formula for the predictive step is as follows:

$$\hat{y}_{t|t-1} = F_t \hat{y}_{t-1|t-1} + B_t u_t \tag{2}$$

Where $\hat{y}$ is the predicted y coordinate of position, t is the current time, $t-1$ is the previous time, F_t is the physical model to map the previous position to the new position, u_t is the control input and B_t is the conversion to map how this input affects position. In this case, the filter is interested only in position, so all values are scalar. Taking the physical model calculated in equation 1, F_t is one, u_t is $v\Delta t$ and B_t is also one, resulting in:

$$\hat{y}_{t|t-1} = \hat{y}_{t-1|t-1} + v\Delta t \tag{3}$$

The Kalman filter also estimates the uncertainty in the position estimate using a state variance variable, P, which is also updated in the predictive state. The initial value of P depends on the accuracy of the system and was tuned to find an optimal value, in this case determined to be 0.5. The formula for calculating subsequent values is:

$$P_{t|t-1} = F_t P_{t-1|t-1} F_t^T + Q_t \tag{4}$$

Q_t represents the uncertainty in the system, due to uncontrolled external factors such as wind. In this case, it was assumed these factors had no effect, as the car was inside and sheltered, allowing the formula to simplify to:

$$P_{t|t-1} = P_{t-1|t-1} \tag{5}$$

The next step is to fuse in the values from the sensor using the following formula:

$$\hat{y}_{t|t} = \hat{y}_{t|t-1} + K_t(Z_t - H_t \hat{y}_{t|t-1}) \tag{6}$$

In this case K_t is the Kalman gain. This can be calculated using further formula, shown in equation 8. Z_t is the sensor measurement, H_t is a value to map the predicted value to the same scale as the measurements, in this case, one as both values are in centimetres and $\hat{y}$ is the value taken from the predicted step. The state variance is also updated from its predicted value using the formula:

$$\hat{P}_{t|t} = P_{t|t-1} - K_t H_t P_{t|t-1} \tag{7}$$

Finally the Kalman gain is calculated:

$$\hat{K}_t = P_{t|t-1} H_t^T (H_t P_{t|t-1} H_t^T + R_t)^{-1} \tag{8}$$

R_t is the uncertainty associated with the measurement readings. This was also tuned experimentally, and found to be zero for sonar readings and 0.4 for the long range IR sensors. These five formulae are continuously updated for both x and y at a frequency of 200 Hz to track and calculate the robots path.

The filter was implemented inside LabVIEW using a formula node structure. In order for the correct velocity, sensor reading and orientation of the robot to be used, this formula node was placed inside a case structure as shown below:

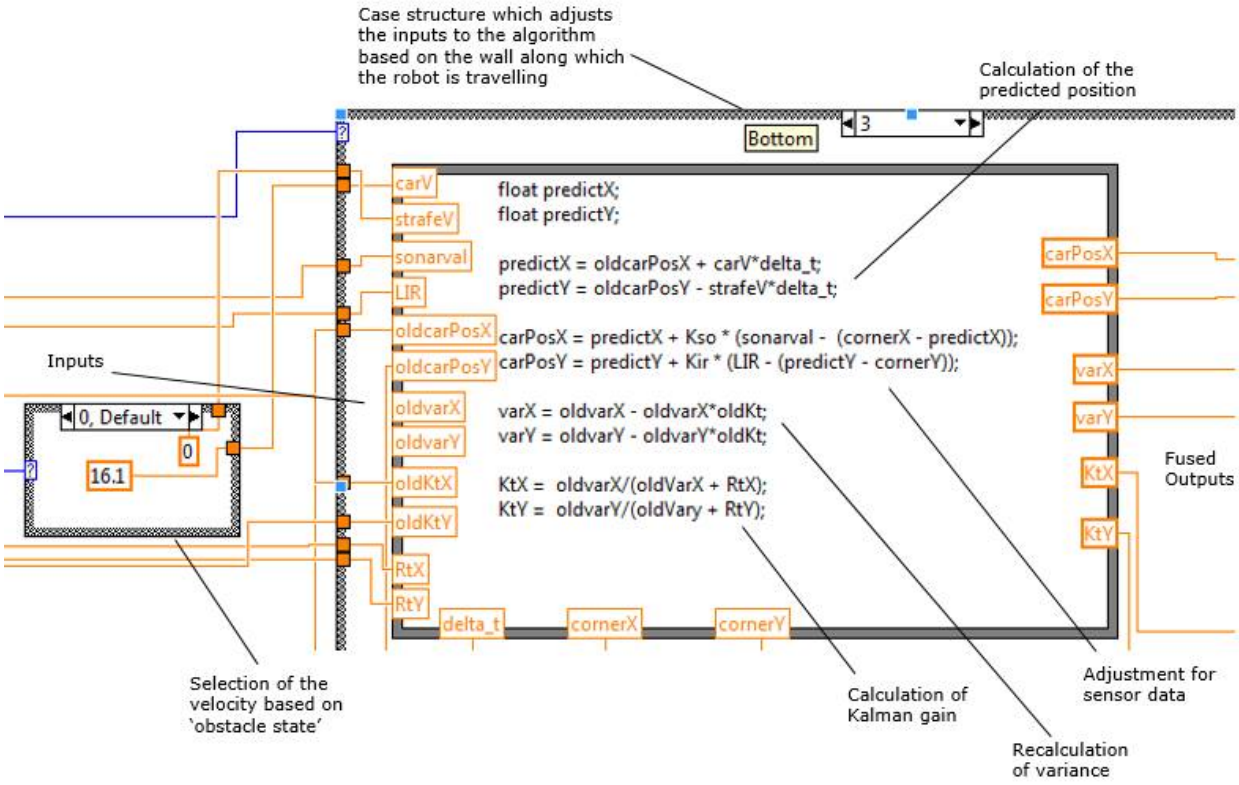


Figure 24: Formula Node Implementing the Kalman Filter

6.3 Mapping

Mapping States

The first state is the determination of the room boundaries. Each time the robot faces a corner it has not encountered before, it pauses and takes sonar and long range IR readings, before outputting these along with a mapping state of 0, the corner count and the time elapsed. The sonar and average IR readings are added to the robots current position to determine the corner coordinate. This addition can be seen in the Figure 26.

Case	Case Number	Description
Path Mapping	1	Plots a dotted line showing the current position of the robot, as well as the path travelled
Room Boundaries	0	Plots a thick white line showing the boundaries of the room

Figure 25: Mapping cases

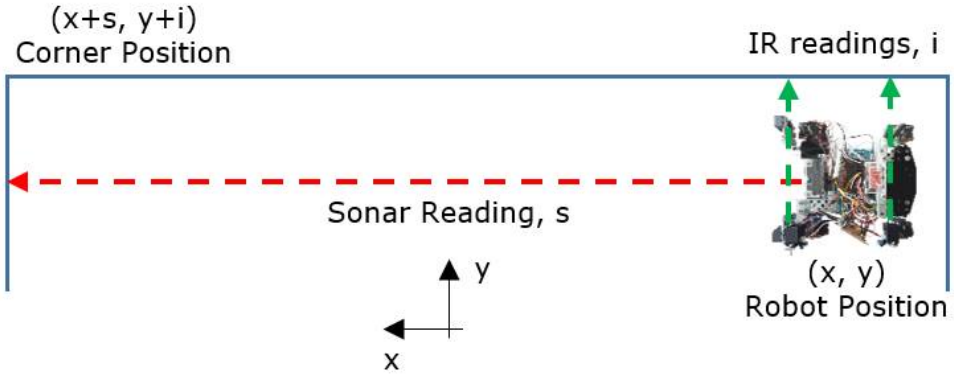


Figure 26: Corner Determination

The most complex of these case is the path mapping, for which the previously mentioned Kalman filter is used to fuse sensor data with the robot odometry. In order to do this, data from the long range IR sensors and the sonar is used to determine the x and y coordinate of the centre of the car at any point in time, with the car's reference point (0,0) being the position of the car when it first enters the 'spiraling' state.

Producing the Map

The outputs from these algorithms are continually appended to the ends of separate arrays for x position, y position, x corner coordinates and y corner co-ordinates. In order to output this data for a user, the x and y data is combined through a cluster and plotted on an XY graph as shown in Figure 27.

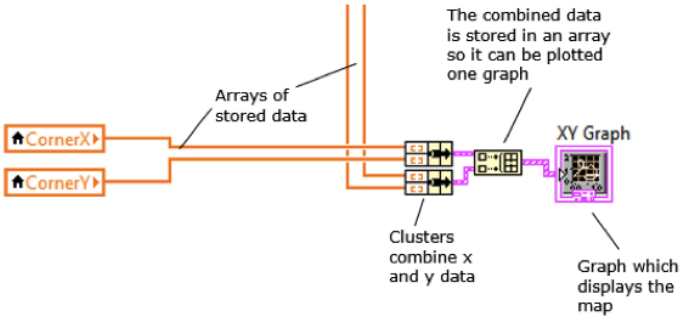


Figure 27: Combining Arrays to Produce the Map

While the mapping function does not currently have the ability to account for the location of obstacle, (In Figure 28, these were added in later) the path of the robot does make it obvious where they are positioned. The run shown in Figure 28 is taken from an earlier test run, using two obstacles. The spiraling method was successful, but it can be seen that the further from the edge, the more shaky the movement of the robot is.

Additionally, the simplification of the physical model of the robot, particularly the assumption that it was purely moving in either the x or y direction lends the robot path map more linearity than was actually the case.

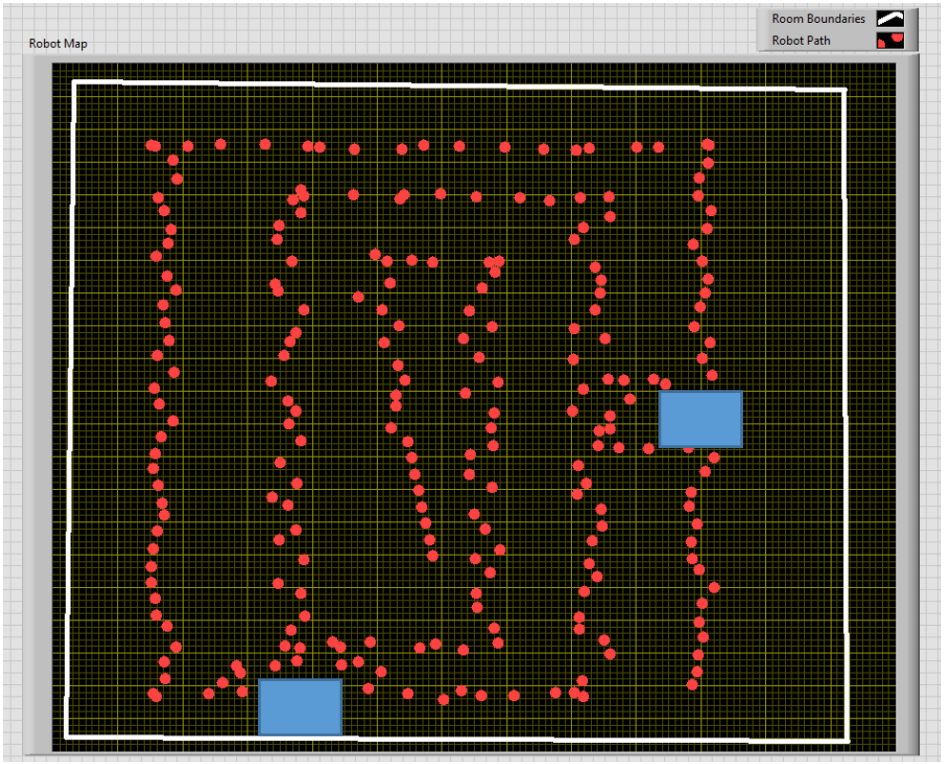


Figure 28: An example map of a successful run

Overall the map produced is successful at defining the boundaries of the room, as well as providing a reasonably accurate live portrayal of the path it is taking. Additionally the live plotting nature of the map meant that with some further work, interpretation of the map could have been used to send information back to the robot, allowing for full SLAM to be performed.

7 Results

During the final run the robot managed to cover about half of the room in a short amount of time without colliding with any obstacles or walls. This was a lower amount of coverage than expected. During the run the wall alignment module was overly sensitive causing the robot to oscillate frequently as it moved. While it was still able to avoid the first obstacle, it never realigned properly and lost its reference to its surroundings. The code was robust enough for it to continue to avoid obstacles, however it no longer followed the intended spiral coverage pattern and ended up finishing towards the upper right corner of the room. The actual map of this run is shown in Figure 29.

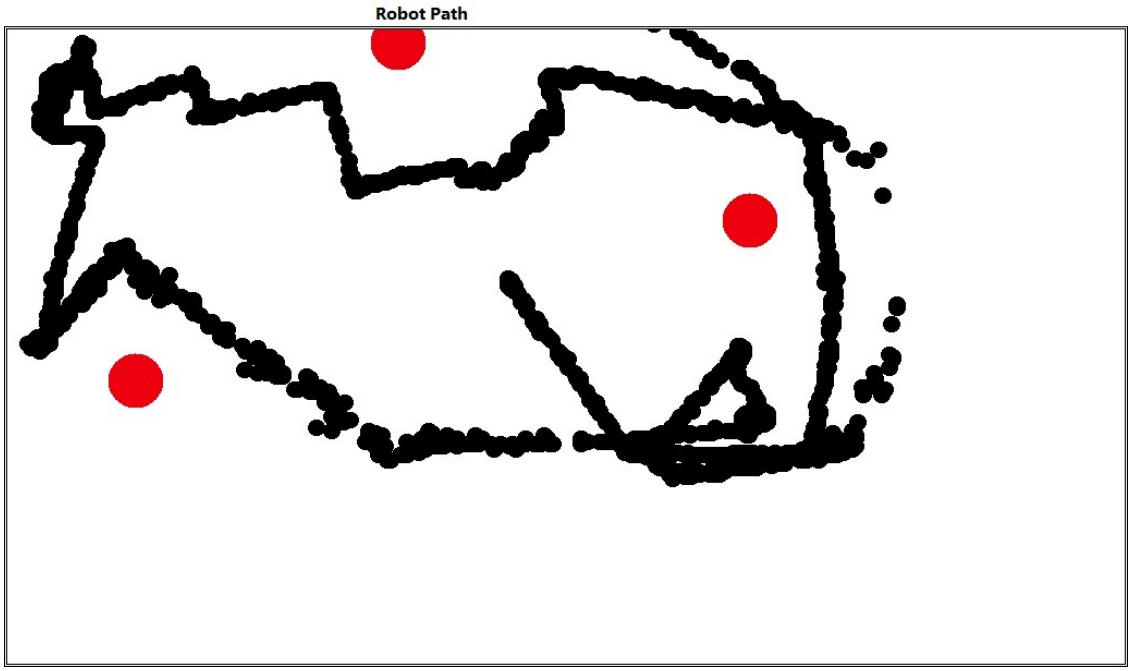


Figure 29: Demonstration Run Result

However this demonstration run is not the best run that was achieved by the robot. During previous runs on the day before the demo, the robot had successfully completed a full spiral while also avoiding obstacles, proving that all sub-system had been integrated to create a robot capable of following SLAM principles.

A video of one of these runs can be found on YouTube at:

https://www.youtube.com/watch?v=bXcSZD9pTAk

8 Discussion

The robots ability to map and navigate the course is highly dependent on the accuracy and performance of the sensors. Sensor readings were thought to possibly fluctuate with varying battery life. This was particularly apparent just before the demo where the robot battery was charged to ensure it would perform for the demo. It was hypothesized that the battery life affected the sensor readings, where it could be seen in the final demo that the robot violently rotated back and forth in an attempt to align itself to the wall, eventually timing out and begin moving forward in a misaligned direction. This misalignment caused further problems where walls approached at an angle were thought to be obstacles which would then cause the robot to strafe continuously. Sensor readings were earlier seen to cause problems for the program, as test runs showed that the robot struggled to align to a brightly lit wall (exposed to sunlight) as opposed to a darker wall incorporated into a mock test course that had been set up on the ground. Despite the variable sensor readings the basic functionality given to the robot was successful enough to navigate a reasonable amount of the room while avoiding contact with obstacles and walls.

8.1 Improvements

Obstacle Detection

Alternative Sensor
A problem was found in the front obstacle detection where due to the limitations of the short range IR sensors, the obstacle detection will only occur when an obstacle or wall enters its range. For the spiral state, a sequentially increased distance from the wall is sought to determine the next turn for the next stage of the spiral but towards the centre the only sensor that can measure this distance effectively is the sonar. The use of only the sonar to check the distance from the wall means that if there is an obstacle in front of the sonar in one of the inner loops, it will think that it has reached the required distance from the wall and thus wrongly classify the obstacle as a wall and rotate to begin the next inner loop. Medium range sensors would be better able to quantify the distance between the robot and wall inside the inner loops to increase the detectable range of an obstacle. However this shortcoming was realised too late into the project to make the change. Another alternative to distinguish walls from obstacles would have been possible if full SLAM had been implemented in this project. The robot would then not turn as it would have known where it should be in the room as opposed to having to interpret the sensor readings to show where it was. The robot would correctly identify the obstacle and avoid it until it reached its desired destination on the map where it would then know to turn.

Servomotor Integration
An idea was pursued that utilised the sonar mounted on the servomotor. The servo would rotate the sonar from facing forward to a known angle and compare the two readings. Through using trigonometry, the second measurement could then be accurately compared to the first where if the robot were to be facing a flat wall these values would be the same as seen in Figure 29 but if an obstacle were in front they would be different. Successful implementation of this obstacle detection method was achieved in isolation but when combined with the rest

of the system, the servo caused a lot of interference with the other functions of the robot. This problem lead to the servo being removed from the Arduino, as the servo could not be integrated into the system reliably.

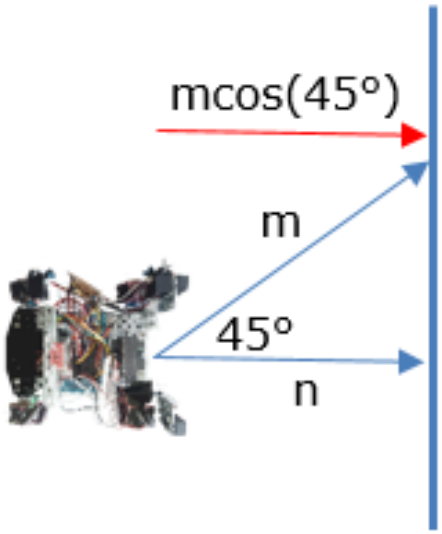


Figure 30: State Diagram

Robot Rotation
The inability to integrate the servo lead to an alternative idea to rotate the whole robot to achieve the same effect. This idea however would rely significantly on the accuracy of the MPU in determining the whole robot's orientation to calculate the equivalent distance of the second reading. Additional rotations of the robot however would lead to further problems with the robot's alignment to the wall where it would become more susceptible to becoming misaligned, especially towards the centre of the course. Furthermore these actions take the robot more time to perform, impacting on the total time taken to map the course which is an important factor in assessing the robot's performance. For these reasons this idea was not implemented where a reduced time with less susceptibility to misalignment was preferred over the improved wall/obstacle detection.

Signal Processing

The robot's performance depends significantly on the accuracy of the readings provided by the sensors. Although the sensors were calibrated multiple times and provided near consistent measurements, on different days the readings would fluctuate, degrading the robots performance. A further improvement would be implementing a simple digital filter such as a median filter. The readings from the sensors were usually constant however it could sometimes be seen that one reading would differ greatly from the other values. This spike would causes problems if the robot used this wall to align. Furthermore the magnitude of this spike would bias the readings from an average filter. The alternative (a median filter)

would be far more efficient in reducing this problem and the implementation of a digital filter would have produced a more consistent performance.

9 Conclusion

The solution presented in this report is an effective implementation of SLAM principles which would allow the system to be used for an autonomous vacuum cleaner. The final result integrates four calibrated infra-sensors, two short range and two long range, an ultrasonic ranging module and an MPU-9150, of which 6 degrees of freedom were used. This design enables proportional control of alignment with the wall regardless of facing, as well as precise rotation control. These are combined to enable the robot to spiral inwards while covering the room. The short ranged IRs and the sonar also combined for complete coverage of the front of the robot, resulting in a obstacle collision rate of zero. Accurate live mapping is also a feature, with a LabVIEW implemented Kalman filter using sonar and long range IR readings to plot the robots path.

Improvements are still possible however, particularly with respect to the differentiation of obstacles and wall and a more comprehensive tuning method for the control system used to align with walls. Further work on improving the reliability of the sensors for differing amounts of ambient lights and battery levels would be particularly beneficial to improving the resilience of the design.

Ultimately, an autonomous robot capable of navigating the room and producing a satisfactory map has been created. During demonstration all obstacle were successfully avoided, with an approximate time of one and a half minutes. The coverage achieved was disappointing however, mainly as a result of the robots inability to properly align to the wall and thus maintain a consistent heading.

References

[1] Lemus R., Diaz S., Gutierrez C., Rodriguez D., Escobar F. (2014) SLAM-R Algorithm of Simultaneous Localization and Mapping Using RFID for Obstacle Location and Recognition. Journal of Applied Research and Technology, Vol. 12 (3), p. 551 - 559

Appendices

Software Code

```
int sonar() {
  long echoDuration, sonarDistance;
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);
  echoDuration = pulseIn(echoPin, HIGH);
  sonarDistance = ((echoDuration / 2) / 29.1) - 6.5;
  return sonarDistance;
}
```

Figure 31: Code to Run the Ultrasonic Sonar Sensor

```
void rotate(int reference) {
  double P = 7.5;
  double control, error = (reference - yaw);
  double errors[10];
  int settle = 0;

  while (error > 6) {
    find_yaw();
    error = (reference - yaw);
    control = P * error;
    constrain(control, -350, 350);

    left_font_motor.writeMicroseconds(1500 - control);
    left_rear_motor.writeMicroseconds(1500 - control);
    right_rear_motor.writeMicroseconds(1500 - control);
    right_font_motor.writeMicroseconds(1500 - control);
  }
}
```

Figure 32: The 'Rotate' Function. Used to Rotate 90°

```
void forward_straight(int reference_Yaw) {
  float kp = 150;
  float control;

  find_yaw();

  if (yaw > (2 + reference_Yaw)) {
    control = (yaw - reference_Yaw - 2) * kp;
    if (control > 800) {
      control = 800;
    }
    else if (control < -800) {
      control = -800;
    }
    left_font_motor.writeMicroseconds(1500 + control);
    left_rear_motor.writeMicroseconds(1500 + control);
    right_rear_motor.writeMicroseconds(1500 + control);
    right_font_motor.writeMicroseconds(1500 + control);
  }
  else if (yaw < (reference_Yaw - 2)) {
    control = (yaw + 2 - reference_Yaw) * kp;
    if (control > 800) {
      control = 800;
    }
    else if (control < -800) {
      control = -800;
    }
    left_font_motor.writeMicroseconds(1500 + control);
    left_rear_motor.writeMicroseconds(1500 + control);
    right_rear_motor.writeMicroseconds(1500 + control);
    right_font_motor.writeMicroseconds(1500 + control);
  }
  else if (yaw > (reference_Yaw - 2) && (yaw < 2 + reference_Yaw)) {
    control = 0;
    forward();
  }
}
```

Figure 33: The 'forward_straight' Function Uses the MPU

```
void strafe_straight(int reference_Yaw, bool strafeDirection) {
  float kp = 150;
  float control;

  find_yaw();

  if (yaw > (2 + reference_Yaw)) {
    control = (yaw - reference_Yaw - 2) * kp;
    if (control > 800) {
      control = 800;
    }
    else if (control < -800) {
      control = -800;
    }
    left_font_motor.writeMicroseconds(1500 + control);
    left_rear_motor.writeMicroseconds(1500 + control);
    right_rear_motor.writeMicroseconds(1500 + control);
    right_font_motor.writeMicroseconds(1500 + control);
  }
  else if (yaw < (reference_Yaw - 2)) {
    control = (yaw + 2 - reference_Yaw) * kp;
    if (control > 800) {
      control = 800;
    }
    else if (control < -800) {
      control = -800;
    }
    left_font_motor.writeMicroseconds(1500 + control);
    left_rear_motor.writeMicroseconds(1500 + control);
    right_rear_motor.writeMicroseconds(1500 + control);
    right_font_motor.writeMicroseconds(1500 + control);
  }
  else if (yaw > (reference_Yaw - 2) && (yaw < 2 + reference_Yaw)) {
    control = 0;
    if (strafeDirection) {
      strafe_left();
    }
    else {
      strafe_right();
    }
  }
}
```

Figure 34: The 'strafe_straight' Function

```
void check_straight() {
  int allowance_left_right = 0.5;
  side_front = read_sharp_IR_sensor(SHARP_YA, side_front_IR_pin);
  side_back = read_sharp_IR_sensor(SHARP_YA, side_back_IR_pin);

  if (abs(side_front - side_back) > allowance_left_right) {
    if (side_front > side_back) {
      cw();
    }
    else if (side_front < side_back) {
      ccw();
    }
  }
  else {
    forward();
  }
}
```

Figure 35: The 'check_straight' Function. Uses the Long Range IR Sensors

```
STATE avoid_left() {
  int left_IR, sonar_sensor, right_IR, side_front, side_back;
  yaw = 0;

  //Strafe while there is an obstacle in front of car
  while (strafe == true) {
    //Read in all sensor values to update them
    left_IR = read_sharp_IR_sensor(SHARP_DX, left_IR_pin);
    sonar_sensor = sonar();
    right_IR = read_sharp_IR_sensor(SHARP_DX, right_IR_pin);
    side_front = read_sharp_IR_sensor(SHARP_YA, side_front_IR_pin);
    side_back = read_sharp_IR_sensor(SHARP_YA, side_back_IR_pin);

    //If all sensors clear of an obstacle, stop strafing and return to spiraling
    if ((left_IR > 15 && right_IR > 15 && sonar_sensor > 15)) {
      prev_side_back = side_back;
      prev_side_front = side_front;
      strafe = false;
      obstacle = true;
      return SPIRALING;
    }
    else {
      //Strafe and maintain yaw at 0, strafe left
      strafe_straight(0, true);
    }
  }

  //Return only if triggered from passing an object
  while (needReturn == true) {
    side_front = read_sharp_IR_sensor(SHARP_YA, side_front_IR_pin);
    side_back = read_sharp_IR_sensor(SHARP_YA, side_back_IR_pin);

    if (side_front <= initial_side_front && side_back <= initial_side_back) {
      pastFront = 0;
      pastBack = 0;
      prev_side_front = side_front;
      prev_side_back = side_back;
      //Serial1.println("RETURNED");
      obstacle = false;
      needReturn = false;
    }
    else {
      //Strafe and maintain yaw at 0, strafe right.
      strafe_straight(0, false);
      //Serial1.println("Strafing back");
    }
  }

  return SPIRALING;
}
```

Figure 36: The 'avoid_left' Function

```
bool align_wall (double error) {
  double kp = 40;
  double wheel_speed = kp * error;

  constrain (wheel_speed, -80, 80);

  left_font_motor.writeMicroseconds(1500 - wheel_speed);
  left_rear_motor.writeMicroseconds(1500 - wheel_speed);
  right_rear_motor.writeMicroseconds(1500 - wheel_speed);
  right_font_motor.writeMicroseconds(1500 - wheel_speed);

  if (abs(error) <= 1) {
    return true;
  } else {
    return false;
  }
}
```

Figure 37: The 'align_wall' Function

```
void find_yaw() {
  timer = millis();

  // Read normalized values
  Vector norm = mpu.readNormalizeGyro();

  // Calculate Yaw
  yaw = yaw + norm.ZAxis * timeStep;

  // Wait to full timeStep period
  delay((timeStep * 1000) - (millis() - timer));
}
```

Figure 38: The 'find_yaw' Function

```
bool find_corner() {
  bool first, aligned = false;
  boolean findingCorner = true;
  static unsigned long previous_millis, sprevious_millis;
  //Begin by taking in sensor readings from front
  bool found;
  left_IR = read_sharp_IR_sensor(SHARP_DX, left_IR_pin);
  sonar_sensor = sonar();
  right_IR = read_sharp_IR_sensor(SHARP_DX, right_IR_pin);

  //Readings from side sensors
  side_front = read_sharp_IR_sensor(SHARP_YA, side_front_IR_pin);
  side_back = read_sharp_IR_sensor(SHARP_YA, side_back_IR_pin);

  //Definitions of limits
  int closest = min(left_IR, right_IR);
  int allowable_Clearance_Front = 10;
  int allowance_left_right = 2;

  //Move forward until close to a wall
  while (closest > allowable_Clearance_Front) //Change to closest once both front sensors are being used
  {


    right_IR = read_sharp_IR_sensor(SHARP_DX, right_IR_pin);
    left_IR = read_sharp_IR_sensor(SHARP_DX, left_IR_pin);
    closest = min(left_IR, right_IR);

    forward();
  }
  stop();

  //Rotate until front sensors give a relatively similar reading (from allowance)
  sprevious_millis = millis();
  while (!aligned)
  {
    right_IR = read_sharp_IR_sensor(SHARP_DX, right_IR_pin);
    left_IR = read_sharp_IR_sensor(SHARP_DX, left_IR_pin);

    aligned = align_wall(right_IR - left_IR);

    delay(100);
    if (abs(left_IR - right_IR) >= 0.5) {
      aligned = false;
    }

    if (millis() - sprevious_millis > 2000) {
      sprevious_millis = millis();
      break;
    }
  }
  rotate(90);
```

Figure 39: The First Half of 'find_corner' Function

```
  yaw = 0;
  right_IR = read_sharp_IR_sensor(SHARP_DX, right_IR_pin);
  left_IR = read_sharp_IR_sensor(SHARP_DX, left_IR_pin);
  closest = min(left_IR, right_IR);

  while (closest > allowable_Clearance_Front) {
    check_straight();
    right_IR = read_sharp_IR_sensor(SHARP_DX, right_IR_pin);
    left_IR = read_sharp_IR_sensor(SHARP_DX, left_IR_pin);
    closest = min(left_IR, right_IR);
  }
  stop();

  aligned = false;
  previous_millis = millis();
  while (!aligned)
  {
    if (millis() - previous_millis > 750) {
      previous_millis = millis();
      return true;
    }

    right_IR = read_sharp_IR_sensor(SHARP_DX, right_IR_pin);
    left_IR = read_sharp_IR_sensor(SHARP_DX, left_IR_pin);

    aligned = align_wall(right_IR - left_IR);

    delay(100);
    if (abs(left_IR - right_IR) >= 1) {
      aligned = false;
    }
  }

  //delay(2000);
  return true;
}
```

Figure 40: The Second Half of 'find_corner' Function

```
STATE spiraling() {
  yaw = 0;
  float side_Length, max_Corner, arena_Width = 0.0; // Variables to determine stopping point
  static unsigned long previous_millis, sprevious_millis; // Variables for timing
  //Readings from front sensors
  static int semaphore = 0;
  static int longest_Side = 0;
  int servo_Position; //Will need to check the servos position before using data from the sonar.

  int allowable_Clearance_Front = 10;
  int allowance_left_right = 1;
  int closest, andy;
  bool aligned = false;
  static bool first_Run = true;
  static bool second_Run = false;

  //Determine the distance from the wall which the robot should turn
  andy = (length_Count / 3) * 22;

  if (millis() - previous_millis > 50) {
    previous_millis = millis();
    sonar_sensor = sonar();
  }
  /************************* Get Sensor Data *********************************/
  right_IR = read_sharp_IR_sensor(SHARP_DX, right_IR_pin);
  left_IR = read_sharp_IR_sensor(SHARP_DX, left_IR_pin);
  closest = min(left_IR, right_IR);

  side_front = read_sharp_IR_sensor(SHARP_YA, side_front_IR_pin);
  side_back = read_sharp_IR_sensor(SHARP_YA, side_back_IR_pin);

  if (semaphore == 0) {

    yaw = 0;
    rotate(90);
    yaw = 0;
    semaphore = 1;

    if (length_Count >= 4) {
      obstacle = true;
    }
    else {
      obstacle = false;
    }
```

Figure 41: The First Part of the 'spiraling' Function

```
  while (!aligned) {

  side_front = read_sharp_IR_sensor(SHARP_YA, side_front_IR_pin);
  side_back = read_sharp_IR_sensor(SHARP_YA, side_back_IR_pin);

  aligned = align_wall((1.5 * (side_back - side_front))); ///((length_Count%4)-(length_Count-1)));

  delay(100);
  if (abs((2 * (side_back - side_front))) >= 0.5) { ///((length_Count%4)-(length_Count-1))) >= 1){
    aligned = false;
  }
  if (millis() - sprevious_millis > 1500) {
    sprevious_millis = millis();
    break;
  }

}

  initial_side_front = read_sharp_IR_sensor(SHARP_YA, side_front_IR_pin);
  initial_side_back = read_sharp_IR_sensor(SHARP_YA, side_back_IR_pin);
  prev_side_front = initial_side_front;
  prev_side_back = initial_side_back;
}
if (length_Count >= 4) {
  //After 4 lengths use yaw for straightness
  obstacle = true;
}
//After obstacle reaches here.
aligned = false;
sprevious_millis = millis();

while (!aligned) {

  side_front = read_sharp_IR_sensor(SHARP_YA, side_front_IR_pin);
  side_back = read_sharp_IR_sensor(SHARP_YA, side_back_IR_pin);

  Serial1.print("Re-aligning   ");
  Serial1.print(side_front);
  Serial1.print("    ");
  Serial1.println(side_back);
  aligned = align_wall((1.5 * (side_back - side_front))); ///((length_Count%4)-(length_Count-1)));

  delay(100);
  if (abs((2 * (side_back - side_front))) >= 0.5) { ///((length_Count%4)-(length_Count-1))) >= 1){
    aligned = false;
  }
```

Figure 42: The Second Part of the 'spiraling' Function

```
  if (millis() - sprevious_millis > 1500) {
    sprevious_millis = millis();
    break;
  }

}
side_front = read_sharp_IR_sensor(SHARP_YA, side_front_IR_pin);
side_back = read_sharp_IR_sensor(SHARP_YA, side_back_IR_pin);


/************ Determine arena size, and thus number of corners to turn ***********/
if (length_Count < 2) {
  side_Length = sonar();
  if (side_Length > arena_Width) {
    arena_Width = side_Length;
  }
  // 25.0 = car width
  max_Corner = (arena_Width / (25.0 / 2.0)) + 2.0;
}


while (semaphore == 1) {
  if (obstacle) {
    //Serial1.println("Obstacle so normal Straight");
    forward_straight(0);
  }
  else {
    //Serial1.println("No obstacle");
    check_straight();
  }

  if (millis() - previous_millis > 50) {
    previous_millis = millis();
    sonar_sensor = sonar();
  }
  left_IR = read_sharp_IR_sensor(SHARP_DX, left_IR_pin);
  right_IR = read_sharp_IR_sensor(SHARP_DX, right_IR_pin);
  side_front = read_sharp_IR_sensor(SHARP_YA, side_front_IR_pin);
  side_back = read_sharp_IR_sensor(SHARP_YA, side_back_IR_pin);
  needReturn = false;
```

Figure 43: The Third Part of the 'spiraling' Function

```
    //If the distance ahead is within the readable IR range start comparing sensors.
    if ((  (left_IR < 15) || (right_IR < 15) || (sonar_sensor < 15)) && ((abs(left_IR - right_IR) > 6) \
    || (abs(sonar_sensor - right_IR) > 6) || (abs(left_IR - sonar_sensor) > 6))) {

      strafe = true;
      change = 0;
      return AVOID_LEFT;
    } else if ((sonar_sensor < 10 + andy)) {
      semaphore = 0;
      length_Count++;
      stop();
      return SPIRALING;
    }

    if (abs(side_front - prev_side_front) > 10 && pastFront != 2) {
      Serial1.print("CHANGE ");
      change = change + 1;
      pastFront = pastFront + 1;
      prev_side_front = side_front;
      Serial1.println(change);
    }
    //Repeat obstacle pass detection using back IR
    else if (abs(side_back - prev_side_back) > 10 && pastBack != 2) {
      change = change + 1;
      pastBack = pastBack + 1;
      prev_side_back = side_back;

      Serial1.print("CHANGE ");
      Serial1.println(change);
    }
    //Four changes represents complete passing of obstacle
    if (change == 4) {
        delay(200);
        needReturn = true;
        change = 0;
        return AVOID_LEFT;
    }
    if (millis() - previous_millis > 50) {
      previous_millis = millis();
      sonar_sensor = sonar();
    }
  }
  return SPIRALING;
}
```

Figure 44: The Fourth Part of the 'spiraling' Function

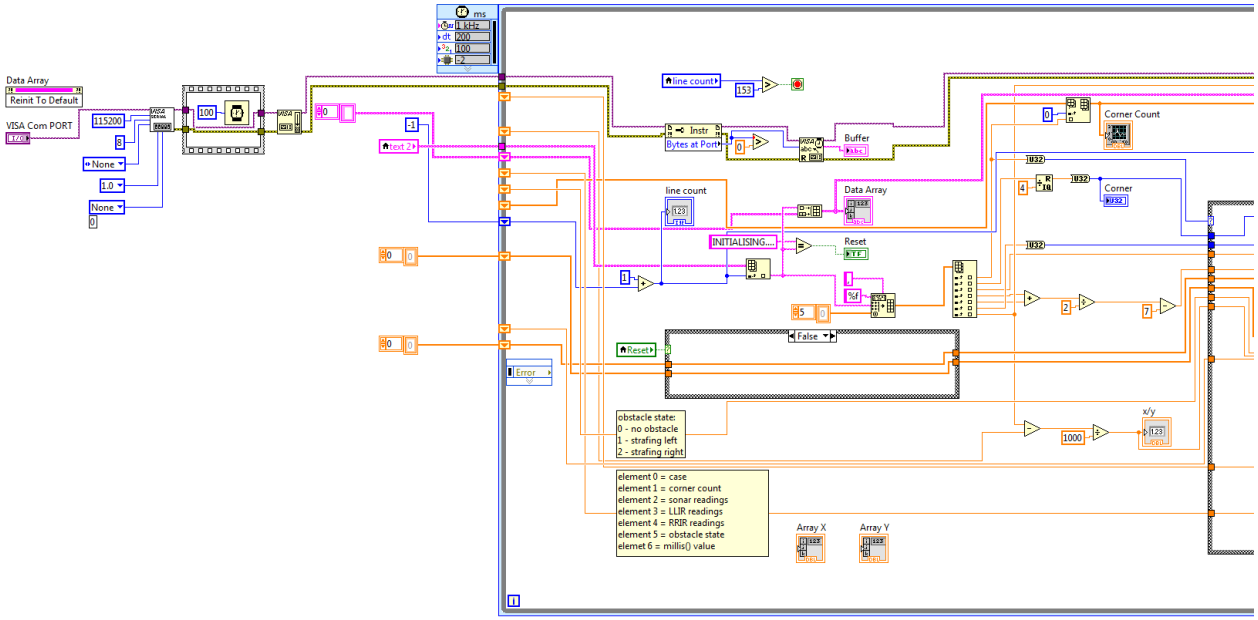


Figure 45: LabView Code - I/O and String Splitting

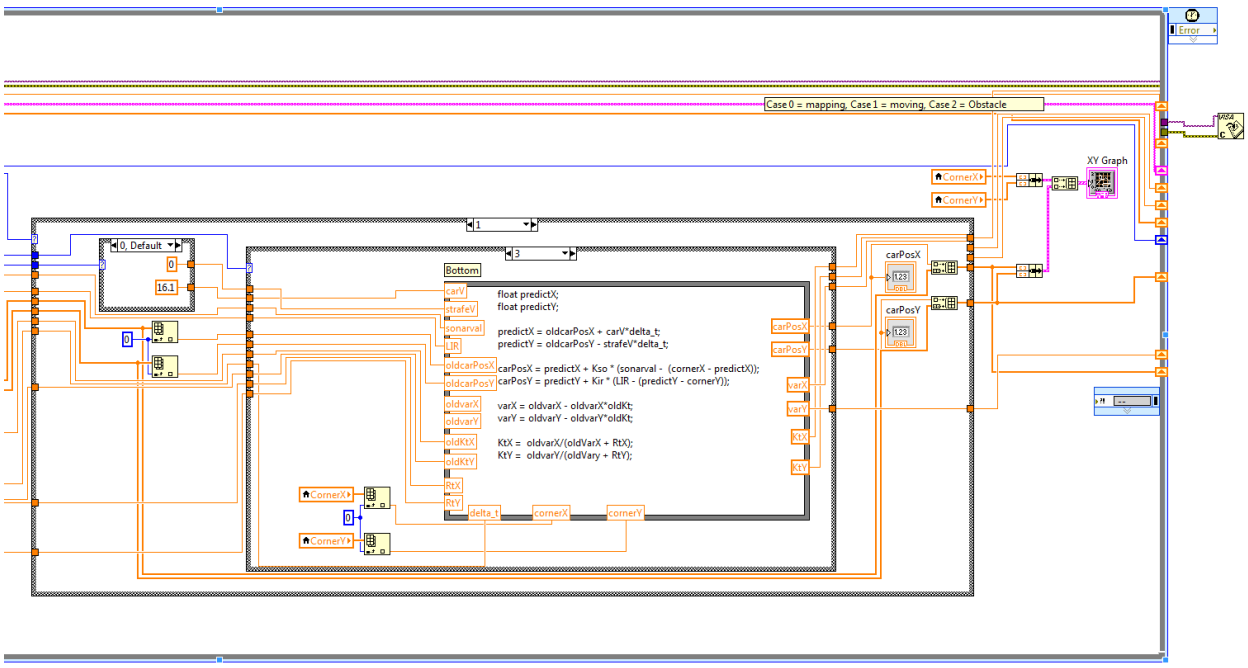
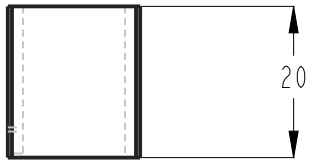
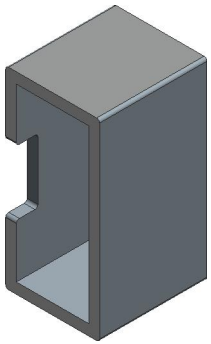
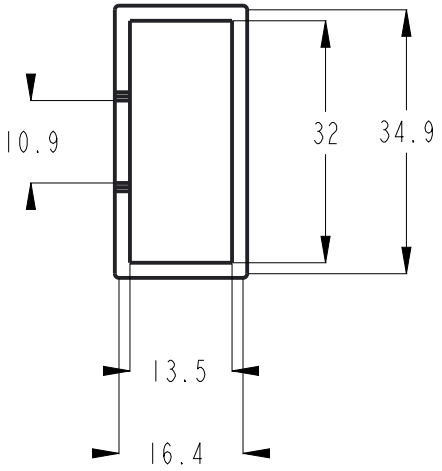


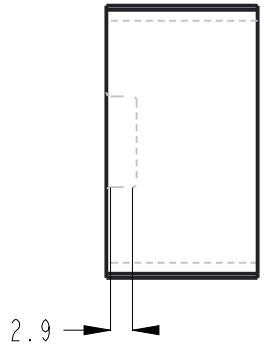
Figure 46: LabView Code - Kalman Filter and Map Plotting

Drawings

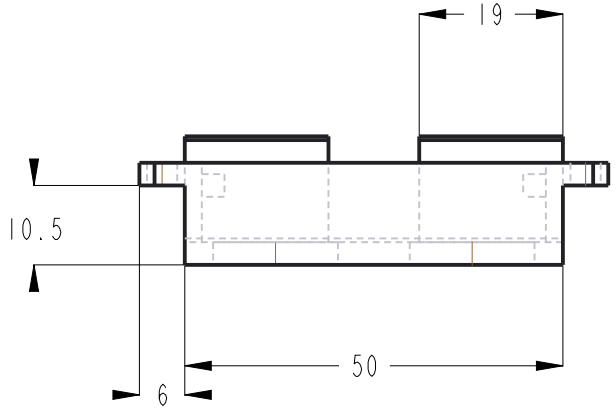


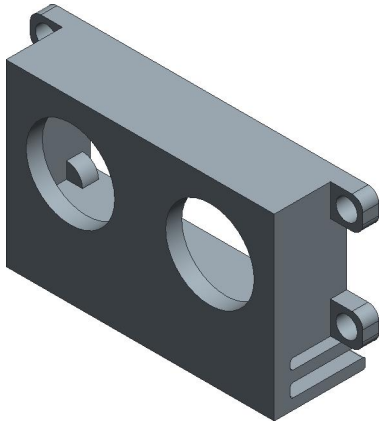


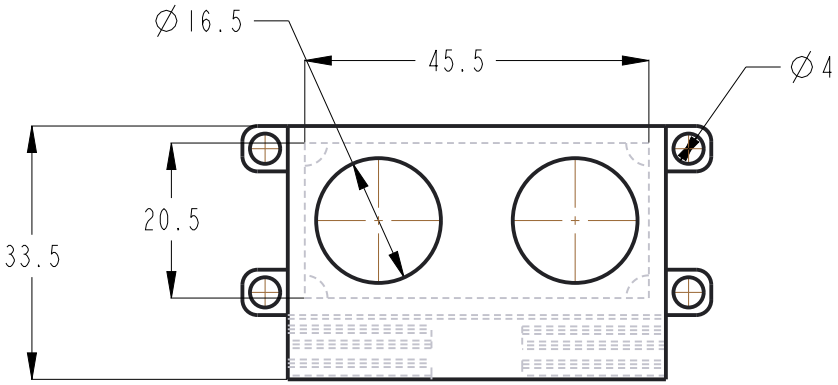


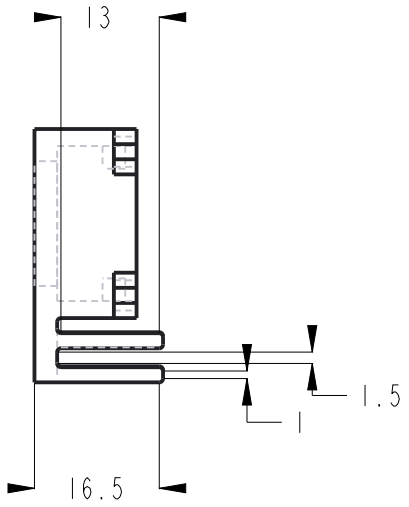


	THE UNIVERSITY OF AUCKLAND NEW ZEALAND	DRAWN BY ANDY YU	TITLE INFRA-RED SENSOR CASE				
		TUTOR ALLAN VEALE	DEPARTMENT MECHANICAL ENGINEERING				
		GROUP No. 5					
		FILENAME	FILE LOCATION				
ALL TOLERANCES ARE ±0.1mm UNLESS OTHERWISE SPECIFIED	ALL DIMENSIONS IN MILLIMETRES	DATE 05/06/2016	MATERIAL ABS PLASTIC	Qty. 4	SCALE 1:1	SHEET	A4

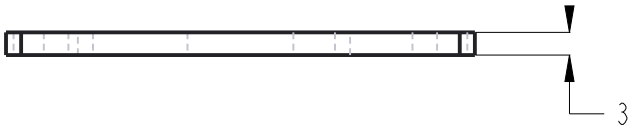


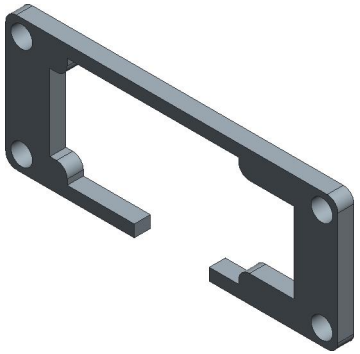


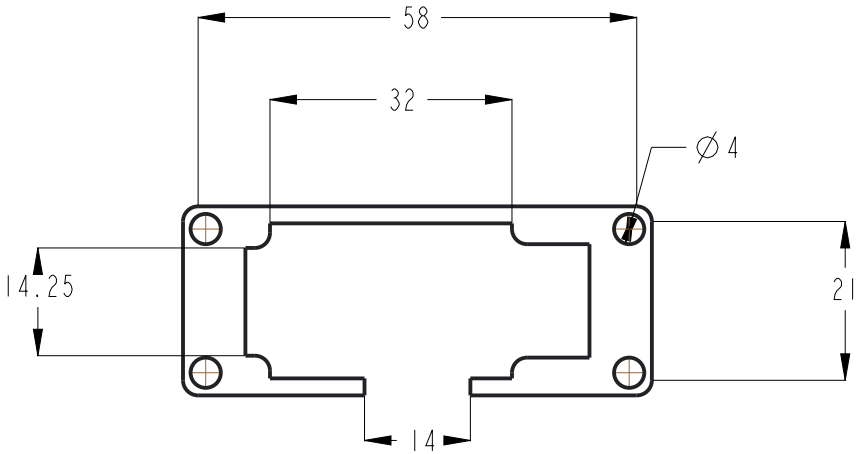




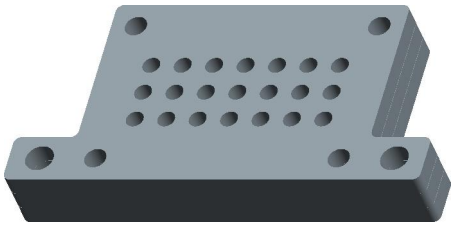
	THE UNIVERSITY OF AUCKLAND NEW ZEALAND	DRAWN BY ANDY YU	TITLE SONAR CASE				
		TUTOR ALLAN VEALE	DEPARTMENT MECHANICAL ENGINEERING				
		GROUP No. 5					
		FILENAME	FILE LOCATION				
ALL TOLERANCES ARE ±0.1mm UNLESS OTHERWISE SPECIFIED	ALL DIMENSIONS IN MILLIMETRES	DATE 05/06/2016	MATERIAL ABS PLASTIC	Qty. 1	SCALE 1:1	SHEET	A4

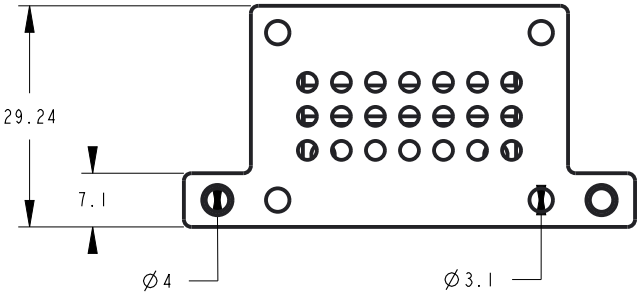






	THE UNIVERSITY OF AUCKLAND NEW ZEALAND	DRAWN BY ANDY YU	TITLE SONAR CASE BACKING				
		TUTOR ALLAN VEALE	DEPARTMENT MECHANICAL ENGINEERING				
		GROUP No. 5					
		FILENAME	FILE LOCATION				
ALL TOLERANCES ARE ±0.1mm UNLESS OTHERWISE SPECIFIED	ALL DIMENSIONS IN MILLIMETRES	DATE 05/06/2016	MATERIAL ABS PLASTIC	Qty. 1	SCALE 1:1	SHEET	A4





Change No.	Name of Item	Changes Made	OK	Date

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

DRAWN BY	CRAIG TIMS	TITLE	MPU CASING
TUTOR	ALLAN VEALE	DEPARTMENT	Mechanical Engineering
GROUP No.	5		
FILENAME		FILE LOCATION	
DATE	05/06/2016	QUANTITY 1	SCALE 1 : 1

SHEET 1 of 1 A4

ALL TOLERANCES ARE ±0.1mm UNLESS OTHERWISE SPECIFIED

ALL DIMENSIONS IN MILLIMETRES