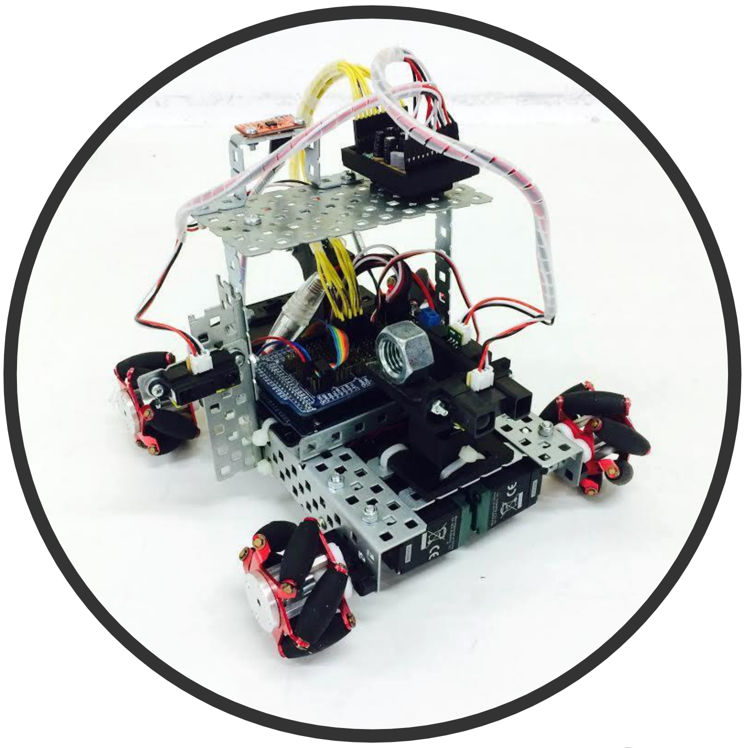
THE UNIVERSITY OF AUCKLAND | DEPARTMENT OF MECHANICAL ENGINEERING



Autonomous SLAM Robot

Project Report

MECHENG 706 Mechatronics Design

Group 4:

Vicky Gao

Wenbo Ge

Aaron Yong

Arlo D'Cruz

Report Date: 7th June 2015

Executive Summary

An autonomous SLAM robot was designed for the MECHENG 706 Mechatronics Design Project. This involved hardware and software design to build an autonomous vacuum cleaning robot capable of performing vacuum cleaning tasks in a room.

Objectives of the project were to develop an autonomous robot that utilised SLAM (Simultaneous Localisation and Mapping) to navigate a room, given a limited number of prescribed sensors. The robot was required to decide on a path and traverse the room for complete coverage, ensuring that any obstacles or walls in the room were avoided. Additionally, a map of the room displaying the robot's movement was to be built.

The robot was controlled using an Arduino microcontroller on a pre-built Mecanum robot. It used four infrared sensors and one MPU (Motion Processing Unit) for detecting walls and obstacles, sensing position and orientation, and subsequently generating a map of the room.

As a result, an intelligent and versatile vacuum cleaning robot was designed with fully integrated systems for its control and navigation. It was able to navigate a platform and avoid collision with walls and obstacles, reaching 85% coverage of the room in 4 minutes. The robot thus met all of its main objectives and succeeded in performing as an autonomous vacuum cleaning robot.

Table of Contents

Table of Figures

1.0 Introduction

This project was designed to employ mechanical, electrical and computer systems knowledge to design and build an autonomous vacuum cleaner robot. A vacuum cleaner is a device that cleans floors and surfaces by sucking up dirt, collecting it in a bag for later disposal. Vacuum cleaners are typically used in domestic household applications, but can also be used for industrial purposes.

With the progression in technology, autonomous vacuum cleaners have become more popular, with robotic vacuum cleaners being able to perform the task of cleaning floors and carpets without human intervention. This is carried out using sensors, algorithms, tuning, and intelligent programming.

The most popular form of autonomous robotic vacuum cleaners is the Roomba, shown in Figure 1, which uses basic sensors to allow it to change directions, detect dirty areas, and sense stairs. While easy to operate and able to clean a room thoroughly, the Roomba has been seen to have a number of limitations. A bumper around the robot is used to bump into objects and walls, protecting the robot's interior from collision damage. However, this still introduces some force of impact on the robot. The pattern of cleaning carried out by the Roomba relies on simple algorithms, such as spiralling out while cleaning, or by changing its direction to continue cleaning after bumping into an object. While simple, this proves to be a time inefficient method of cleaning a room.

Figure 1: Roomba - An autonomous vacuum cleaning robot

This project aimed to improve these limitations by introducing path logic of the robot's movement, along with collision detection and avoidance, resulting in the entire room being cleaned in the shortest amount of time possible. This was to be done using SLAM – Simultaneous Localisation and Mapping, the concept of using a mobile robot to build a map of the unknown environment, while using the generated map to navigate the environment at the same time.

1.1 Problem Description

To perform as an autonomous vacuum cleaner, the robot was to be capable of navigating to cover the entire room, while being able to detect randomly placed obstacles and navigate itself around them without collision. The testing platform or "room" that the robot performed in has dimensions of 2 m x 1.2 m, with its operation started in a random location. Cylindrical objects acting as obstacles were also placed randomly throughout the room.

Based on its sensors, the robot had to decide on its path of movement, with the aim of achieving complete coverage of the room in the shortest amount of time. It was required to build a map of the room, complete with walls and objects, showing the path of the robot's movement, and also be able to carry these functions out without any form of human intervention, hence making it autonomous.

1.2 Project Specifications

A kit was provided for this project, including a pre-built Mecanum robot with an Arduino Mega Board and shield, four infrared sensors, one MPU sensor, one sonar sensor with a servo motor, and one Bluetooth module. The robot's motion was carried out using Mecanum wheels, which allowed for more manoeuvrability, as the robot was able to move in any direction.

With the project being open-ended, a number of additional features were included in the robot for improved operation and design. These included features such as PID controllers, hardware and software filters, a real time map interface, and wall alignment, all of which are explained in detail throughout the report.

2.0 Sensor Integration

Five sensors have been used to control the robot; four infrared sensors and one MPU (Motion Processing Unit) sensor. The four infrared sensors were comprised of three medium range sensors, sensing distances from 10 cm to 80 cm, and one long range sensor, sensing distances from 20 cm to 150 cm, as shown in Figure 2. These sensors were used for scanning and mapping the environment, as well as obstacle detection. One of the long and medium range sensors were used in conjunction with a servo motor to aid in scanning the room, as described in later sections.

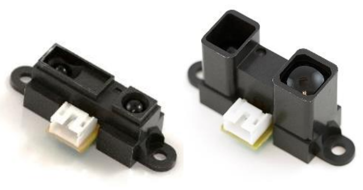


Figure 2: Infrared Proximity Sensors - Medium Range (left) and Long Range (right)

The MPU sensor provided was the MPU-9150 from SparkFun, a 9 degree of freedom sensor consisting of a tri-axial accelerometer, gyroscope, and magnetometer, capable of measuring the acceleration, angular velocity, and magnetic heading respectively. It has a Digital Motion Processor (DMP) that can fuse the three sensors together to make Euler angles, which was used measure the heading and orientation of the robot as it was operating.

Figure 3: MPU-9150 Sensor

An ultrasonic sonar sensor was also provided with the kit. Although initial testing was carried out on this sensor, the finalised design of the robot's operation in motion and mapping was able to be carried out with only the infrared sensors and the MPU; hence the sonar sensor was deemed unnecessary. It was also seen as advantageous for the robot as this reduces its overall cost and complexity while still being able to carry out its function successfully.

2.1 Placement and Mounting

2.1.1 Sensor Layout

The placement of the sensors on the robot was such that all directions which the robot moves in are covered, so that walls and obstacles can be detected before collision. The layout can be seen in Figure 4 below, and may be used for referral in later sections. The servo motor is placed at the front of the robot, with a long range infrared sensor (pin A2) facing the front, and a medium range infrared sensor (pin A4) facing the left. The long range sensor placed at the front allows the robot to make scans of the area in front of it.

Another medium range sensor (pin A1) was placed towards the back of the robot facing the left, which in conjunction with the medium sensor on the servo motor, allowed the robot to align to the wall on its left. The last medium range sensor (pin A3) was placed on the right side of the robot facing the front, to cover the area in the front-right of the robot. The infrared sensors were mounted horizontally on the robot, as vertical mounting resulted in useless readings. The MPU was placed on a platform in the middle of the robot, and was elevated to mitigate the interference effects of metal in the robot on the magnetometer.

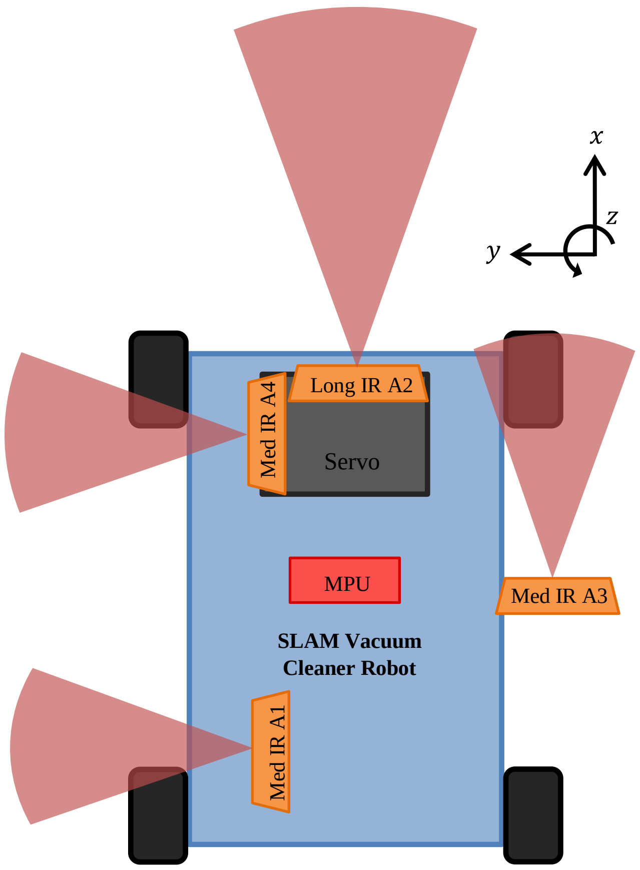


Figure 4: Sensor Placement on the Robot

2.1.2 Mounts

Mounts were made for the servo motor, the left facing sensor, and the hardware low pass filter to secure them in place on the robot, as shown in Figure 5. These were designed in CREO and 3D printed with slots on the bottom that are used as tight fits into the holes on the robot surface.

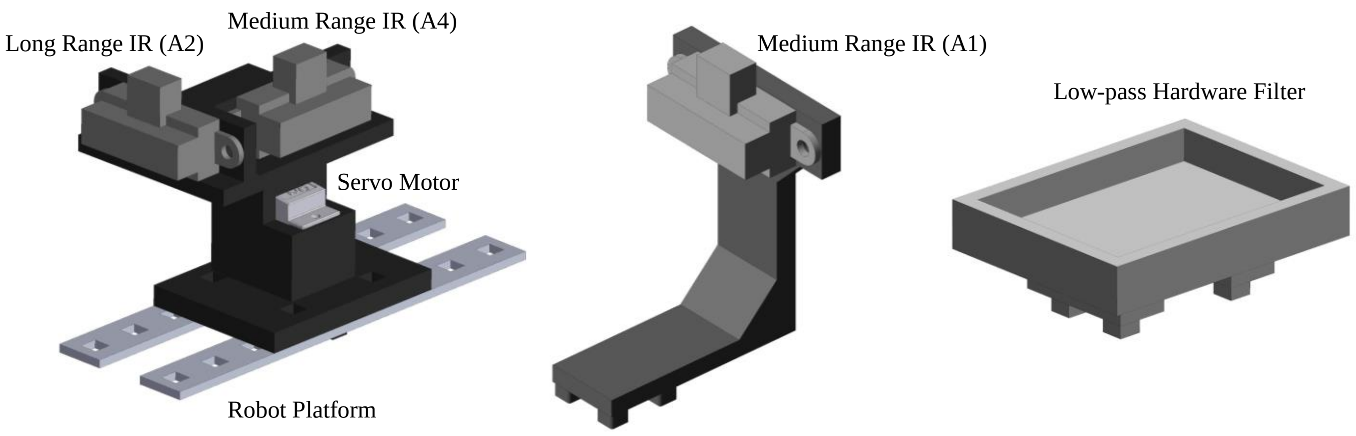


Figure 5: Mounts for the Servo Motor (Left), Side Sensor (Middle), and Hardware Filter (Right)

The MPU was secured using a small bracket with screws, as this was more stable than using slots to fit into the robot surface. This was particularly important for the MPU, as any small movements such as vibrations from travelling would affect the output values.

2.2 Infrared Sensor Filtering and Calibration

Two cases for the infrared sensors were identified, when the robot was stationary and when the robot was moving. When the robot was stationary, it was important that the IR sensor read a consistent result for several key decision making and state changing conditions. Consistency in readings was also vital for an accurate calibration of the sensors. When the robot was moving, it was important for the sensor to be responsive and not fluctuate significantly, as this could trigger false conditions. All these specifications were achieved with a variety of filters listed below, along with experimentation and results.

2.2.1 Low Pass Hardware Filter

Initial testing of the infrared sensors revealed a significant amount of high frequency signal noise. To mitigate this noise, filtering was carried out on each of the sensors. A low pass hardware filter was designed and connected to each of the four infrared sensors to filter out sharp voltage spikes, as shown in the circuit diagram in Figure 66.

The low pass hardware filter was in the form of a simple RC passive filter, using a 1 μF capacitor and a 10 Ω resister to pass V_{signal} to the V_{input} on the Arduino. This gave a cut-off frequency of:

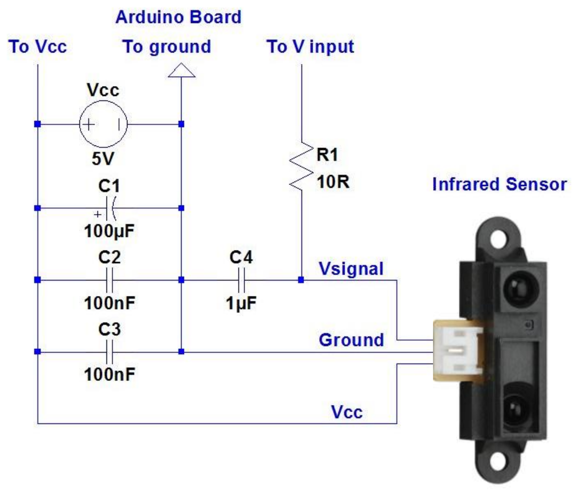
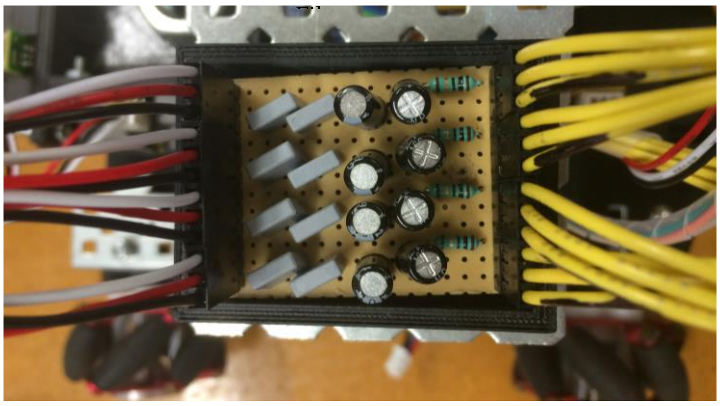


Figure 6: Infrared Sensor Hardware Low Pass Filter

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi \times 0.0001\ F\ \times 10\ \Omega} = 16\ kHz$$

Additional smoothing of the voltage signal was implemented via a 100 μF capacitor placed close between V_{cc} and ground pins to limit voltage peaks, and two 100 nF capacitors were used to smooth the noise from V_{cc}. This was overall found to be more efficient than using a low-pass filter algorithm, as each input reading was filtered, rather than a group of samples being filtered. The final hardware filter built for the robot can be seen below in Figure 7.


Low-pass
Unfiltered data from infrared sensors
Filtered data to Arduino pins
Figure 7: Low Pass Hardware Filter on Robot

2.2.1.1 Testing

Testing was carried out on the filtered sensors using an oscilloscope, which showed a significant decrease in high pass frequencies, as shown in Figure 8. These readings were taken with a wall that was 40 cm away from the long range A2 (referring to Figure 4) infrared sensor. The scale intervals for both graphs were 100 mV on the y axis, and 250 μs on the x axis.

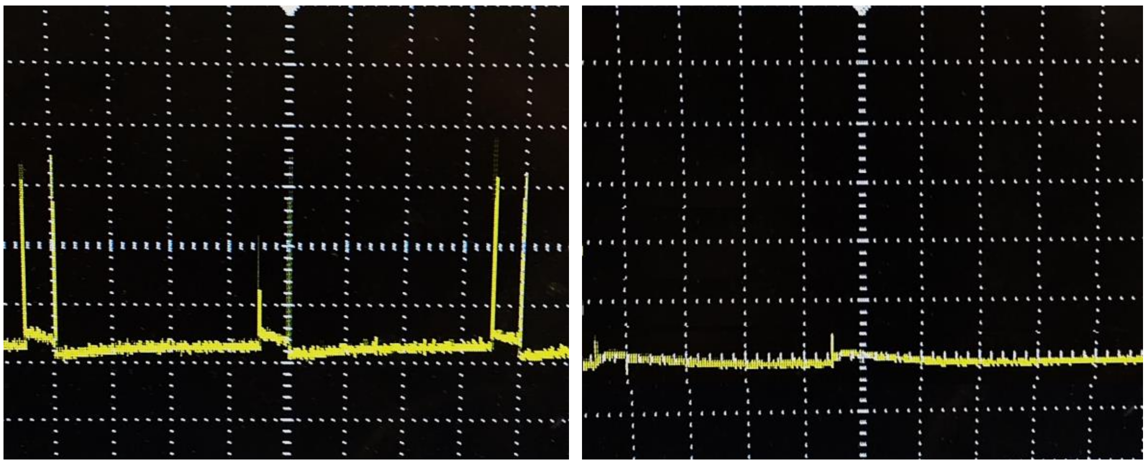


Figure 8: Oscilloscope Readings for Non-Filtered (left) and Low-pass Filtered (right)

It can be seen that the high frequency voltage noise shown on the left oscilloscope reading is clearly filtered out in the right, leaving more accurate readings.

The variance of distance readings from the long A2 infrared sensor can be seen in the graph below. The variance of readings for the non-filtered sensor were compared with the low-pass filtered sensor and found to be significantly worse, particularly at longer distances from 80 to 100 cm.

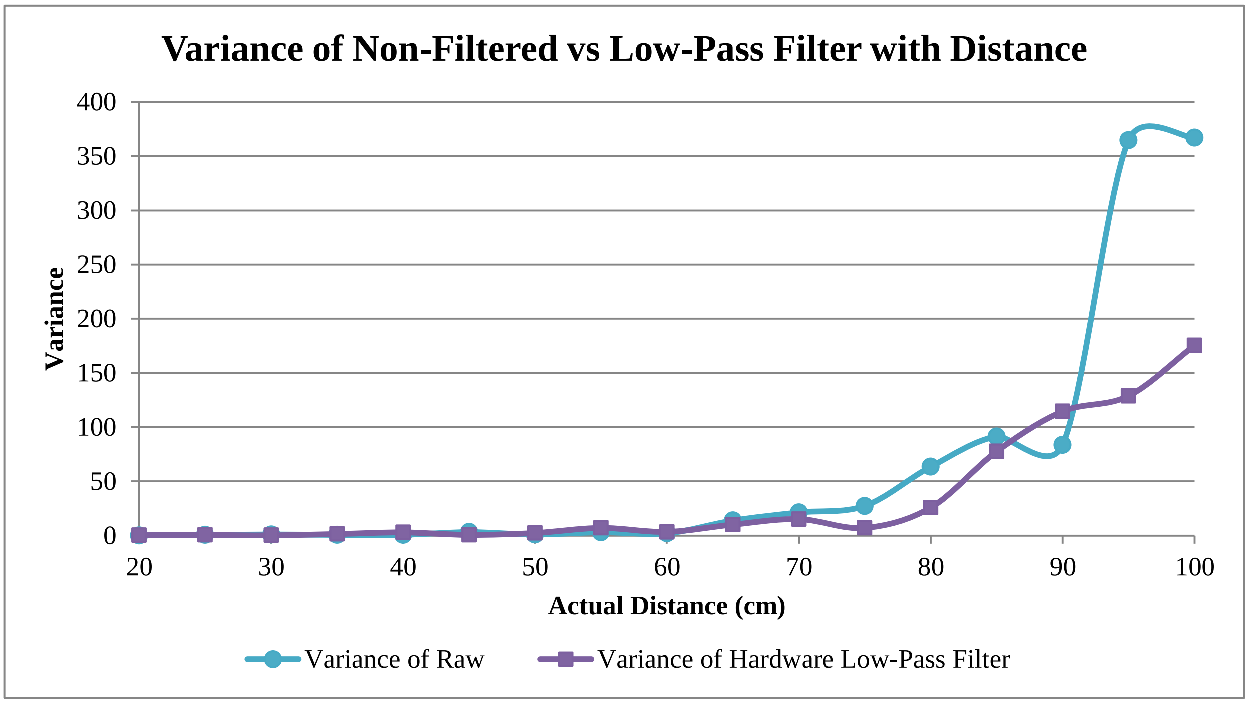


Figure 9: Variance of Non-Filtered vs. Low-pass Filtered with Distance

2.2.2 Stationary Filters

As mentioned previously, the main judgement of performance of the infrared sensors while stationary was the consistency of output readings. These are vital for calibration, robot logic, and decision making. It was also very important for initial mapping during the scan.

2.2.2.1 Mode Filter

As the infrared sensor values still tended to fluctuate even with the addition of a hardware low pass filter, software filters were also applied. A mode filter was used to take sensor readings when the robot was stationary, taking 25 samples to determine the mode, which is the reading that appears the most in the samples. This was implemented using an Average Library from Arduino Playground. If no mode was found, -1 will be returned, where it would take the mean of the samples. If several modes were found, it would average the multiple mode values. This was found to have a decent consistency, although it too often found no mode and returned the average reading instead, which made it more susceptible to outliers.

2.2.2.2 Enhanced Mode Filter

Due to the aforementioned mode filter often finding the average values instead of the mode, an enhanced mode filter was made that gave each sample a tolerance range of ± 1. For example, this meant that a reading of 30 cm would be registered as both 29 and 31 cm. If there were multiple modes, they were averaged to get consistent readings. However, fluctuations caused by low frequency noise occasionally resulted in false readings for an extended period of time, which caused the mode value to drop.

This extended mode filter was implemented within the same Average Library mentioned above, but the comparison of samples to find the mode was changed to include a comparison of the samples ± 1 cm. With every mode value found, a counter is incremented to find the value with the highest counter at the end of the 25 samples. If multiple had the highest counter, the modes were averaged, and if no mode was found, -1 is returned and all of the samples are averaged, similar to the normal mode filter.

2.2.2.3 Kalman Filter

To further improve on the previous filters, a simple one-dimensional Kalman filter was applied to the readings. This reduced the effects of the low frequency fluctuations wand resulted in more stable values by limiting the slew rate of the noise. This meant that any spikes due to noise had a reduced effect based on the limited slew rate, which gate a much steadier sample for the enhanced mode filter to work on.

2.2.2.4 Average Filter

The output readings of the infrared sensors had a resolution of 1 cm, which was found to be too large for when the robot was aligning to the wall. To solve this, an average filter was used to increase the reading resolution to 1 mm, which gave a more accurate value that was consistent enough to be used for the alignment of the robot to the walls. This was carried out by reading 60 values and calculating the average, which ensured absolute dependability of the refined value.

2.2.3 Stationary Testing and Results of Filters

Testing of the individual filters and combinations of the filters was carried out using the long range A2 infrared sensor. As these were solely for the robot at stationary state, and consistency was the most important aspect, performance of the filters was measured by taking the variance of six samples at 5 cm intervals, starting from 20 cm up to 100 cm. The filters tested were the Mode filter only, the Enhanced Mode filter only, and a combination of the Enhanced Mode filter and the Kalman filter, shown below in Figure 10. The same test was carried out on the non-filtered sensor readings and the low-pass hardware filter, as shown previously in Figure 9.

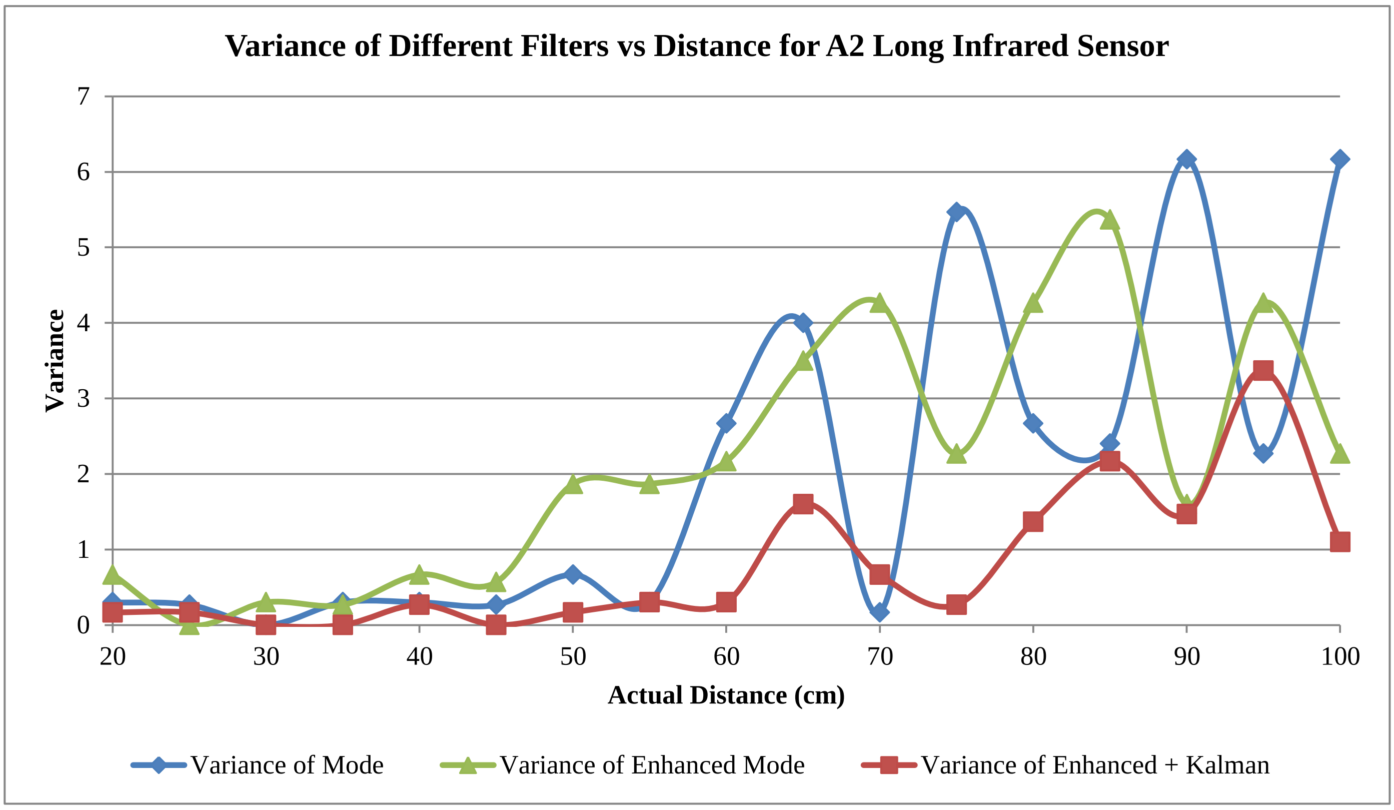


Figure 10: Variance of Different Filter with Distance

It is clear that the combination of the Enhanced Mode filter and Kalman filter resulted in the best response, with the lowest variance at all distance values. This was thus the combination of software filters used for the stationary infrared measurement reading. With these filters employed on all of the infrared sensors, a large portion of ripples and high frequency components that existed in the voltage outputs were removed.

2.2.4 Moving Filters

During phases where the robot was moving, the judgement of performance was responsiveness and size of fluctuations. These were vital as they would be used to detect obstacle collision and as inputs to the PID. The same method for stationary filters could not be used as when the robot is constantly moving, the samples will never have the same values and any variation of the mode filter would be rendered useless.

2.2.4.1 Kalman Filter

The initial method for obtaining the best readings while the robot was moving was a one-dimensional Kalman filter, the same as the one described for the stationary filter. This data refinement method provided a very noise-resilient signal, but at the expense of the responsiveness of the system due to the lag in slew rate. This was not acceptable and was thus rejected.

2.2.4.2 Rolling Average Filter

Instead of a Kalman filter, which added lag to the system, a moving average filter of 20 readings was applied to the infrared sensor readings. This filter was largely successful, both in reducing the noise levels and maintaining fast response times.

2.2.5 Calibration

After filtering, the distance readings from the infrared sensors required calibrating in order to make the measured distance seen by the sensors equal to the actual distance. To calibrate, the robot was placed at regular intervals, starting at the sensor's minimum range and increasing the distance by 5 cm until the limits of the sensor were reached.

Six sensor distance measurements were taken at each interval, which were then averaged and plotted against the actual distances. A trendline was fitted to these data points using polynomial regression, with the equation obtained being used to calibrate the infrared sensors in the code. This was carried out separately for all four infrared sensors, as shown below in Figure 11, as each was different in accuracy.

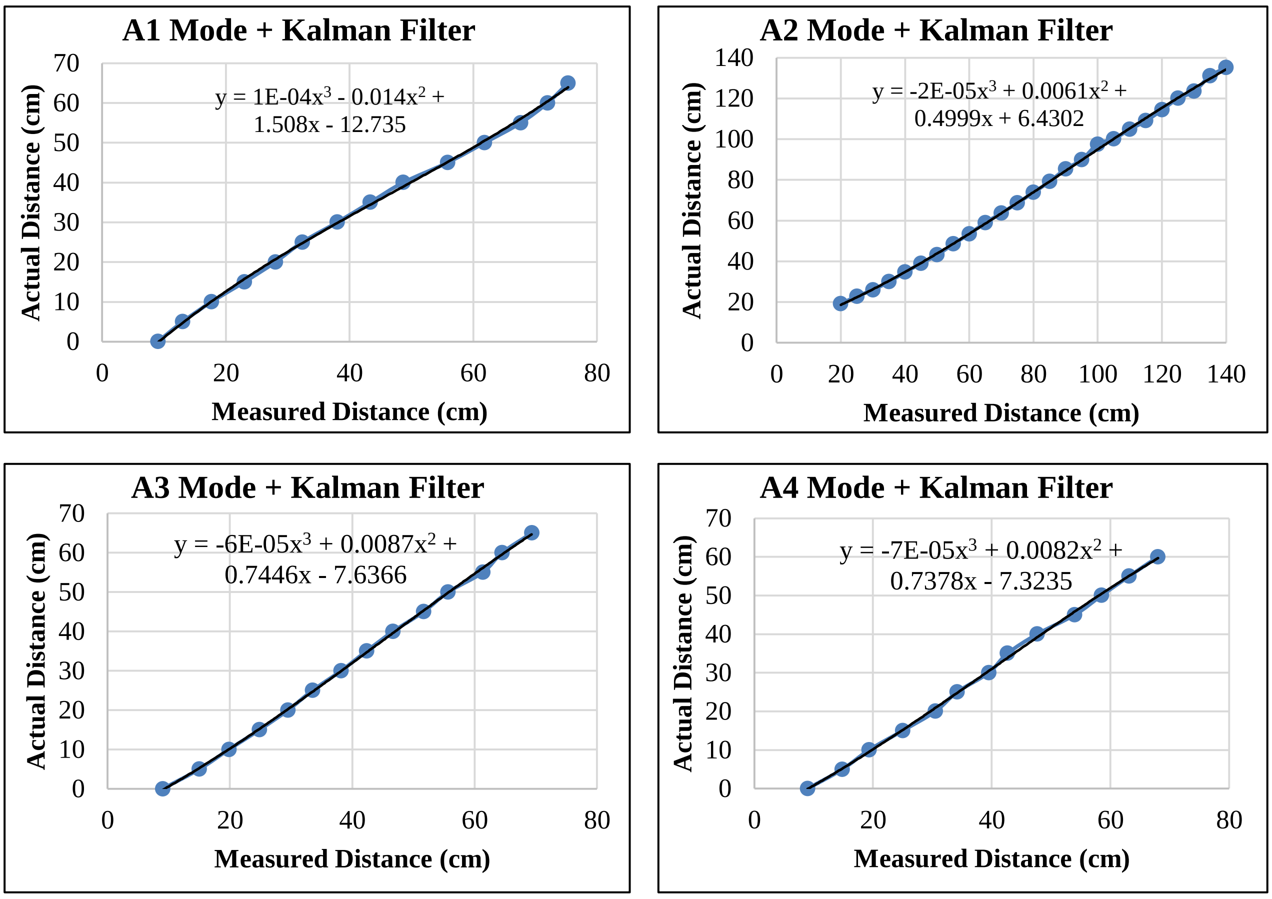


Figure 11: Calibration Testing for the Infrared Sensors

2.3 Motion Processing Unit

Data from the MPU was obtained using an external library which contained several sketches for calculating the fused Euler angles (roll, pitch, and yaw, which are the rotations about the x, y, and z axes on the robot respectively as shown in Figure 4 and 12), as well as for calibrating the accelerometer and magnetometer.

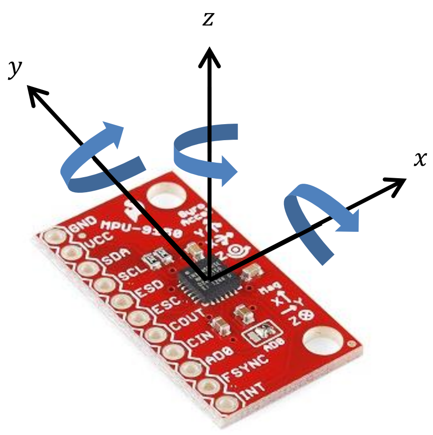


Figure 12: MPU-9150 Sensor Axes

Using the library, the yaw (θ) angle was found to determine the heading of the robot. This was made to be a combination of the gyrometer and the magnetometer. The gyrometer gave the angular velocity about each axis, which was integrated to find the roll, pitch, and yaw of the robot. Since it was assumed that the testing platform and environment that the vacuum cleaner robot would be working in was 2D, only the yaw angle was required to be integrated. The magnetometer, which had a constant reference to magnetic north, was combined with this which compensated for the drift of the gyrometer.

The accelerometer was originally used to find the position of the robot by integrating twice, however the accumulation of error was found to be too large to get an accurate estimation of the position. Because of this, position tracking was done using the dead reckoning technique, which is further explained in the Localisation section (4.2).

2.3.1 Calibration

During initial testing, the MPU was found to experience noise and drift in the accelerometer and magnetometer respectively, which gave errors in the output data. To remove these errors, calibrating was carried out using sketches provided by the MPU external library.

Calibration of the magnetometer was carried out with the MPU sensor already mounted on the robot to compensate for any magnetic fields that were formed by metal. Hard-iron errors caused by metal tended to cause the magnetometer measurements to be off-centre, and were calibrated by moving the magnetometer around to get data in all orientations, which found the minimum and maximum measurements output values for each axis. From this data, the amount of offset was found and then subtracted from each axis to centre the magnetometer readings. This was found to reduce the effect of metal from the robot on the data received from the magnetometer.

Calibration of the accelerometer was carried out in much the same way as the magnetometer calibration, with the minimum and maximum output acceleration values found by moving the sensor around in all orientations.

3.0 Motion Integration

3.1 Movement Method

Two methods of traversing the platform were considered for the robot's movement; a square zigzag across the platform, or a spiral inwards to the centre, as shown in Figure 13. Both methods were to conduct an initial scan to identify the walls of the room and areas in which obstacles may be in. From this scan, the robot finds the closest wall detected and aligns itself to this wall. It then moves along the wall until a corner is found, with this corner becoming the starting position (0, 0), and the vacuum cleaning operation commences.

For the square zigzag method, the robot moves forwards across the platform, strafes to the right, and moves backwards across the platform again. It repeats this until it has reached the opposite side of the platform. For the inwards spiral method, the robot follows a cascading spiral movement, moving inwards until its stop point at the centre of the room.

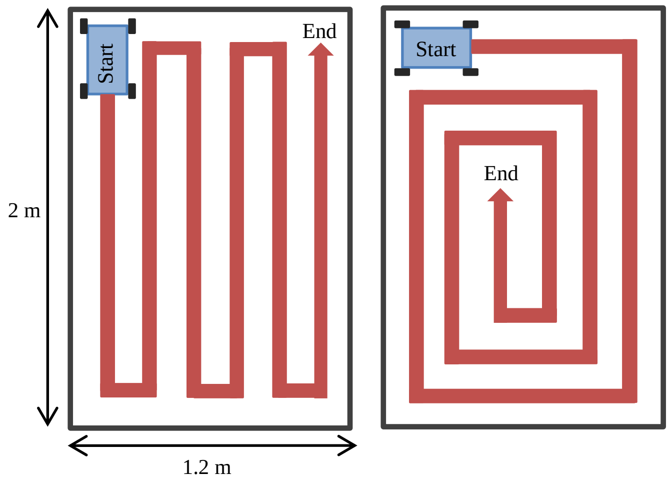


Figure 13: Movement Methods of the Robot, Square Zigzag (left) and Inwards Spiral (right)

The effectiveness of each method was based on the amount of coverage that could be achieved in the least amount of time, the complexity of the movements required, and the amount of difficulty in which SLAM can be integrated. Because of this, the inwards spiral method of movement was chosen. While both methods were seen to have the same amount of coverage potential, the square zigzag method was seen to have less potential for SLAM and mapping integration. Due to the limited range of the sensor, each zigzag movement will only extend the map's range by the short distance that it has strafed, whereas an inward spiral can potentially map the entire platform within the first loop, and then confirm this map on subsequent loops. This produces a fuzzy map in a shorter amount of time.

The inwards spiral movement over the platform was able to be carried out in two different ways; either with the robot rotating 90° at every turn, or with the robot not rotating at all and moving in all four directions to cover the map. It was decided that the robot will rotate, as it will only require infrared sensors to face the front and the left side which faces the wall, as opposed to sensors needing to be on all sides of the robot.

3.2 Motion Controller

The main movements of the robot for its operation are rotation, forward, and strafe. All motion control on the robot is achieved through a PID (Proportional Integral Derivative) controller, used in conjunction with the inverse kinematic equations of motion for the Mecanum wheels. The output wheel drive speed is then converted to a pulse width period to be sent to the motors of the wheels.

A different PID is used for each of the main movements of the robot, as an all-encompassing PID controller did not work due to the nature of the sensors. The infrared sensors return the position of the robot relative to what it is currently pointed at, meaning that if the robot was to rotate, the sensor will return different x and y values as its position has changed. Because of this, the motion controller has been split into separate PID controllers that are customised for each robot movement. These are classified by different states in the finite state machine, with separate gains and inputs. The PID controllers for rotation, forward, and strafe are described below.

3.2.1 Rotation

A function for the rotation of the robot about the z axis is done using an Arduino PID function in addition to the inverse kinematic equations of motion. The controller made for this can be seen below in Figure 14.

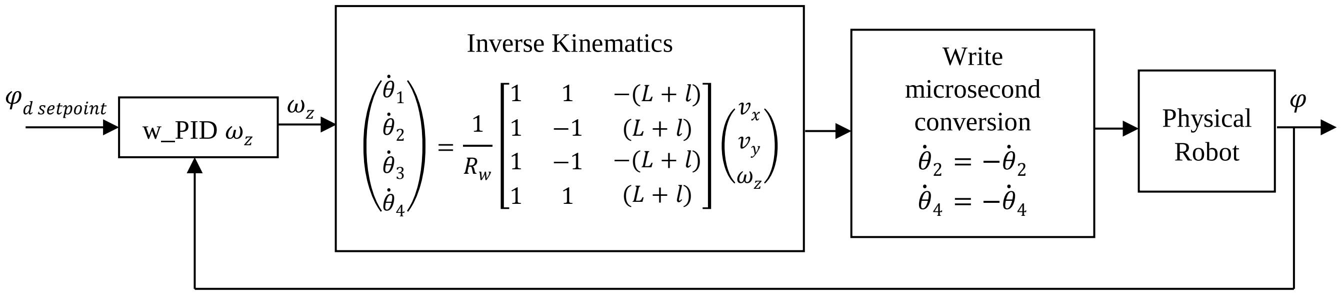

$\varphi_{d\ setpoint}$
w_PID ω_z
ω_z
Inverse Kinematics
$\begin{pmatrix}\dot\theta_1\\\dot\theta_2\\\dot\theta_3\\\dot\theta_4\end{pmatrix} = \frac{1}{R_w}\begin{bmatrix}1 & 1 & -(L+l)\\1 & -1 & (L+l)\\1 & -1 & -(L+l)\\1 & 1 & (L+l)\end{bmatrix}\begin{pmatrix}v_x\\v_y\\\omega_z\end{pmatrix}$
Write microsecond conversion
$\dot\theta_2 = -\dot\theta_2$
$\dot\theta_4 = -\dot\theta_4$
Physical Robot
φ

Figure 14: Rotation Motion Controller

An angle set point $\varphi_{d\ setpoint}$ was sent to the PID function along with the current angle determined from the MPU, the PID gains, output limits, direction, and sampling time. This gave an output rotational velocity ω_z with arbitrary units, which was then interpreted using the inverse kinematic motion equation to produce the drive speed for each of the four wheel motors.

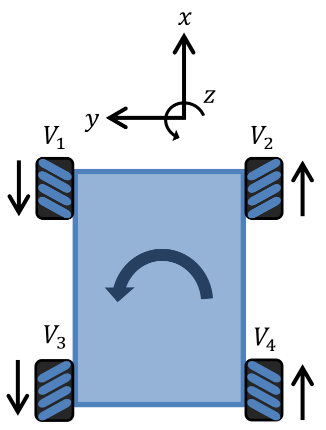


Figure 15: Mecanum Wheel Directions for Rotation

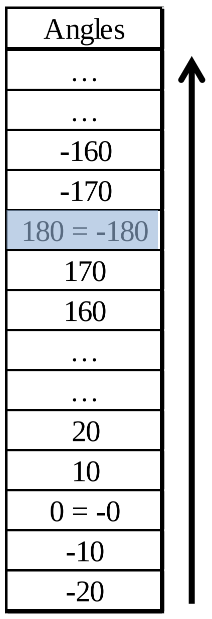
The raw output from the inverse kinematic equation gave two positive values and two negative values of the motor drive speeds. In the case of rotation, all wheels must be driven in the same direction, so the Arduino writeMicroseconds function was used to reverse the sign of outputs $\dot\theta_2$ and $\dot\theta_4$. This allowed a positive rotation input to the wheels to make the robot rotate in an anti-clockwise direction about the z axis, as shown in Figure 15.

3.2.1.1 Angle Nonlinearity

A significant issue that occurred during rotation was due to the nonlinearity of the MPU values, as the magnetometer ranged from -180º to +180º. When the angle crossed the outer limits of this boundary, it started again from the other end of the scale, as shown in Figure 16.

One workaround with this issue was adding or subtracting 360º if the set point was outside of the scale, which means if it was to run a full loop around the platform, three of the corners would be 90º turns, and the last one would be 270 º in the reverse direction. This had a problem that if the set point was too close to the boundaries, such as 178º, any overshoot would cause it to jump to the other end of the scale and start spinning again.

This problem was eliminated using a code defined cumulative angle. At every time interval, the angle heading was taken from the MPU, and the difference between this current angle and the previous angle was added to a cumulative variable. If the difference was too large, such as a difference of 356º caused by a set point of 178º which has overshot to -178º, the difference became ± 360º to make the robot only see an increase in angle of 4º. This ensures that the robot will not continue rotating on the spot due to overshoot. However, this had some minor timing issues leading to slight inaccuracies, so this method was only used when the set point was close to the outer limits or when it would cross the boundary. In each case, the normal MPU magnetometer reading was used at the input to the w_PID, and would switch over to the corrected difference method when approaching the extreme boundaries.



Angles
…
…
-160
-170
180 = -180
170
160
…
…
20
10
0 = -0
-10
-20

Figure 16: Angle Scale

3.2.2 Forward

The forward motion of the robot, shown in Figure 18, is achieved through a combination of an *x* axis position PID controller and *z* axis rotational PID controller, which can be seen below in Figure 17.

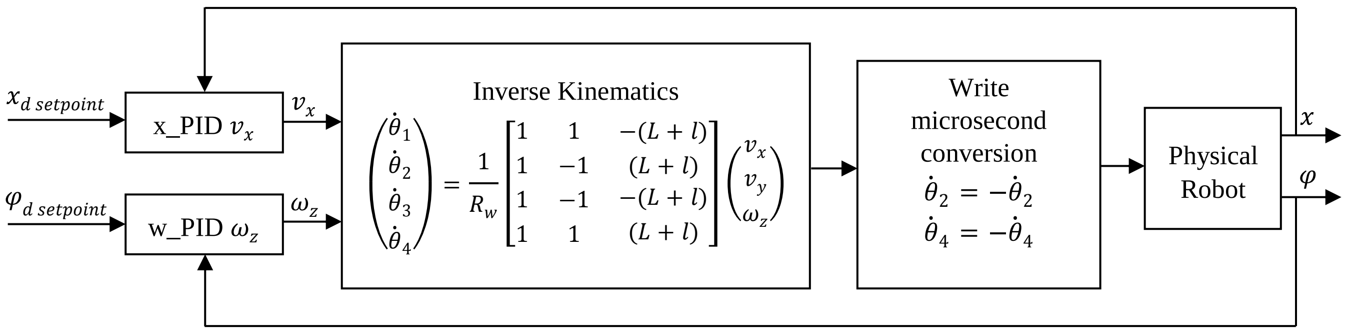

$x_{d\ setpoint}$
$\varphi_{d\ setpoint}$
x_PID v_x
w_PID ω_z
v_x
ω_z
Inverse Kinematics
$\begin{pmatrix}\dot\theta_1\\ \dot\theta_2\\ \dot\theta_3\\ \dot\theta_4\end{pmatrix} = \frac{1}{R_w}\begin{bmatrix}1 & 1 & -(L+l)\\ 1 & -1 & (L+l)\\ 1 & -1 & -(L+l)\\ 1 & 1 & (L+l)\end{bmatrix}\begin{pmatrix}v_x\\ v_y\\ \omega_z\end{pmatrix}$
Write microsecond conversion
$\dot\theta_2 = -\dot\theta_2$
$\dot\theta_4 = -\dot\theta_4$
Physical Robot
x
φ

Figure 17: Forward Motion Controller

The set point $x_{d\,setpoint}$ is an input to the forward PID controller as a distance travelled, which is calculated from the initial and current distance reading from the front facing infrared sensor. This gave an output forward velocity v_x. The set point $\varphi_{d\,setpoint}$ is the input to the rotational PID controller as the initial angular orientation which the robot will maintain whilst moving forwards, giving an output rotational velocity ω_z. This effectively drives the robot a set distance away from what it is currently pointed at, while keeping a constant angular orientation.

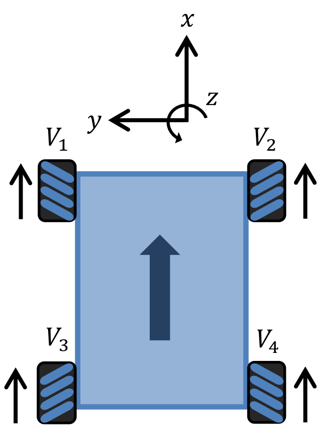


Figure 18: Mecanum Wheel Directions for Forwards Motion

Likewise with the rotational PID controller described above, the outputs of the PID are combined to find the driving speed of each motor using the inverse kinematic equation. This is then converted to the microsecond domain, with the $\dot{\theta}_2$ and $\dot{\theta}_4$ outputs again having their signs inversed to result in a forwards motion. Backwards motion uses the same PID controller as forwards, with the inputs entered as negative values.

3.2.2.1 Implementation

The Forward function takes in nine inputs, as shown in Figure 19, all of which are described below.

```
void  forward1(bool  XdTravelledMode,  float
XdTarget, bool GivenTime, int EndTime, bool
GivenMaintainAngle,  float  AngleToMaintain,
bool  ParallelMode,  bool  FirstLoop,  bool
ObjectAvoid)
```

Figure 19: Inputs of Forward function

TravelledMode determines how *XdTarget* was interpreted into the set point. It wa true if *XdTarget* was to be interpreted as the distance away from the wall, and false if it was to be the distance moved by the robot instead. When the front facing long range A2 sensor on the robot came within 35 cm of a wall or object, the servo motor moved the medium range A4 sensor to the front and used this to determine the input of the PID. This was because the long range sensor had a minimum reading distance of 20 cm, and thus cannot register obstacles at a close distance.

GivenTime is set to true if it is desired to move forward for a set amount of time *EndTime.* If this is set to true, the x_PID controller is disabled and Vx is set to a constant 1000. If false, it will record the *EndTime* in a global variable.

GivenMaintainAngle is set to true if it is desired to set angle W to a certain input angle *AngleToMaintain*, which makes the robot maintain this angle as it is travelling forwards. If it is set to false, it will maintain its current angle.

ParallelMode determines if the forward function will break if one of the side infrared sensors drifts too far from its initial side distance measured to the wall, thus putting the robot out of alignment. This is only reliable at close distances, as the infrared sensors at further ranges tended to fluctuate drastically.

FirstLoop is an input to determine if the robot is currently moving forward on its first loop. If this is set to true, the servo motor will rotate the infrared sensors to make the medium infrared sensor face the front, and will change between this angle and positive 20º relative to this. This is used to detect any objects in front or to the left of the robot. The PID controller in the x direction used the A3 (referring to the pins on Figure 4) medium front sensor to determine the error and the distance it needs to move to the wall.

ObjectAvoid is set to true if the robot had just strafed away from an object and needed to go forward to pass the object. It used the side medium sensor A1 to take an initial value, and drove the robot forward at a constant Vx of 1000. Once the reading from this sensor changed to be significantly closer, and then changed back to reach its initial value, this signified that the robot has passed the obstacle and movement can stop. If the sensor did not register that the object had passed, the robot will stop after 2 seconds of movement. As moving past an object takes 1 second, 2 seconds was an appropriate tolerance for completely avoiding obstacle collision.

3.2.3 Strafe

The function for strafing to the side using a similar PID controller to the forwards motion, but uses the infrared sensors on the side of the robot instead of the front to provide information in the y direction. The controller can be seen below in Figure 20. To maintain the desired y axis convention, $\dot{\theta}_1$ and $\dot{\theta}_3$ were required to be inversed.

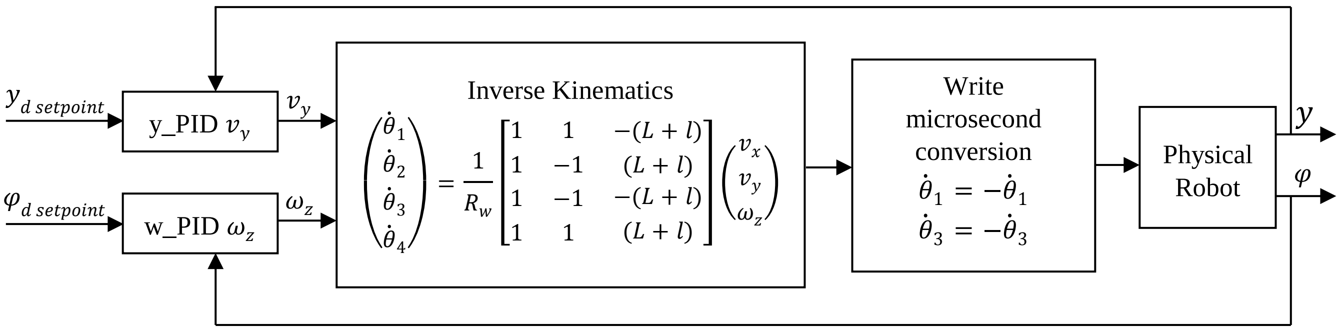

$y_{d\ setpoint}$
$\varphi_{d\ setpoint}$
y_PID v_y
w_PID ω_z
v_y
ω_z
Inverse Kinematics
$\begin{pmatrix}\dot{\theta}_1\\ \dot{\theta}_2\\ \dot{\theta}_3\\ \dot{\theta}_4\end{pmatrix} = \frac{1}{R_w}\begin{bmatrix}1 & 1 & -(L+l)\\ 1 & -1 & (L+l)\\ 1 & -1 & -(L+l)\\ 1 & 1 & (L+l)\end{bmatrix}\begin{pmatrix}v_x\\ v_y\\ \omega_z\end{pmatrix}$
Write microsecond conversion
$\dot{\theta}_1 = -\dot{\theta}_1$
$\dot{\theta}_3 = -\dot{\theta}_3$
Physical Robot
y
φ

Figure 20: Strafe Motion Controller

3.2.3.1 Implementation

The Strafe function takes in 6 inputs as shown in Figure 21. These are described below.

```
void  strafe1(float  YdTarget,  bool  FastStrafe,
bool  GivenMaintainAngle,  float  AngleToMaintain,
bool ObjectAvoidGoLeft, bool ObjectAvoidGoRight)
```

Figure 21: Inputs of Strafe function

YdTarget is amount of movement desired while strafing, and is interpreted to the YdSet point based on the initial A1 side sensor value.

FastStrafe determines which set of gains to use. True results in a lower gain with no Vy speed limiting, whilst false results in a much higher gain with Vy being limited to half that of the true alternative. True would ideally be used for larger distances that wants to be traversed faster, whilst false is used for small distances with precision.

GivenMaintainAngle and *AngleToMaintain* work in the same way as the aforementioned method in the forward section. If *GivenMaintainAngle* was true, strafe will maintain the *AngleToMaintain*, otherwise it will maintain its current angle.

ObjectAvoidGoLeft and *ObjectAvoidGoRight* strafe the robot until the object has passed one of the front sensors, either the long A2 sensor or the medium A3 sensor. If the object has passed the A2 sensor, it will stop immediately, with the remaining momentum causing the robot to finish strafing past the object. If the A3 sensor, the robot will delay stopping by half a second to finish strafing past the object.

3.2.4 Wall Alignment

The robot was required to be aligned to the wall at all times in order to have full coverage and a straight path. The wall alignment function worked off the same basis as the rotation function, with a difference in gains and how the set point was determined. The gain was much higher as this function was only used after rotation had occurred and had lost some alignment.

The set point was determined by reading the A4 side sensor and A1 side sensor, as shown in Figure 22. The difference Y_{diff} between the two readings was input into the equation shown below to determine the difference in angle (shown as θ) between the robot and the wall, where 10.73 cm was the distance between the two side sensors. This angle difference was added to the current robot angle $W_d Initial$ to determine the new set point, $W_d Set$, needed to align the robot to the wall.

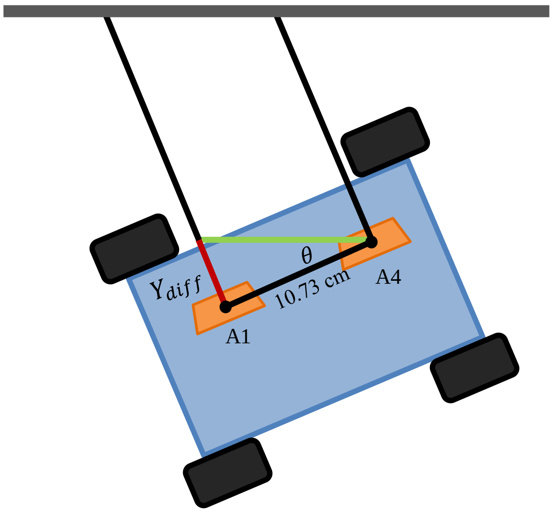


Figure 22: Wall Alignment Angle Calculation

$$W_d Set = W_d Initial + tan^{-1}\left(\frac{Y_{Diff}}{10.73}\right)$$

There was also a separate wall alignment function that corrected the very small differences with very high gains. This was occasionally called after the robot had been aligned once, and saved the set point of the previous align to move at very small rotational displacements.

3.2.5 Stop Conditions

Each of the movement functions were ran inside an infinite while (true) loop to execute PID computation and movement execution, and thus required relevant stop and break conditions. All motion functions: forward, strafe, rotate, and wall alignment, had a stop condition that looked at the corresponding sensors. If the sensor values did not fluctuate outside a tolerance range for a set amount of time, it was considered to have stopped.

However, with the forward function, the front sensor was seen to have larger fluctuations that were not within this tolerance range, so this range was increased with an additional stop condition if the wheels did not drive for a set amount of time. Each of these conditions individually caused false stops and different ends of the sensor range when too far or too close, but a combination of these resulted in a sufficient stop condition.

3.2.5.1 Detect Collision

Each sensor has its own collision detection as functions. This function is called within the while (true) loop of forward and strafe and determines if any of the sensors read anything below a certain threshold, meaning it is too close. It uses a running average filter as this gave the best response while the robot was moving. If one of the sensors determines something is too close, the robot will stop its motion and re-evaluate its surroundings to determine its next action.

3.2.5.2 Challenges

Due to the use of the PID library in from arduino, several issues occurred. One major issue was the integral portion of the output is never reset. Since the rotation function has an integral portion, any movement afterwards that involved a rotational PID, such as forward or strafe, would cause it to always veer to the rotation it was last in. A solution to this was to edit the PID library to reset the integral whenever the mode of the PID was changed, which is done before every motion function.

Another issue was that the outputs V_x, V_y, and V_w, would not reset after the PID had finished executing. As a result, the function that applies the inverse kinematic equation always caused it to move in a combination of the previous motion and the current motion. This was remedied using a function that reset the PIDs output before every motion function was called.

4.0 SLAM Integration

4.1 Initial Scan

Before the robot was able to commence its cleaning operation, a scan of the environment was taken using the long A2 infrared sensor mounted on the servo motor as shown in Figure 23. The servo motor was programmed to move in increments of 3° in a range of 9° to 167°, with the starting default position at 73°, which is 0° in the xyz coordinate system of the robot. This allows the robot to detect walls and objects, and also gives a sufficient amount of time for the magnetometer in the MPU to stabilise.

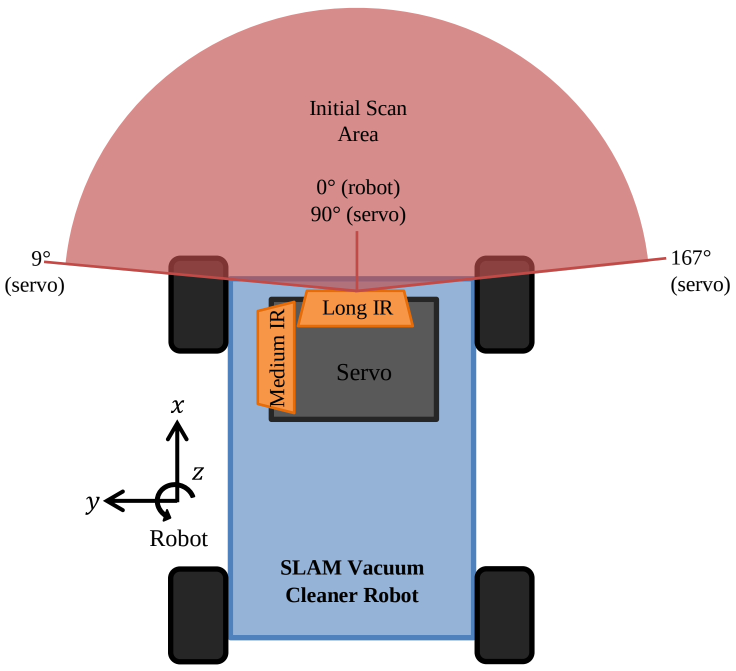


Figure 23: Area Scanned During Initial Scan State

From this, walls picked up by the scan are identified, allowing the robot to move to the closest wall and begin its operation.

Fuzzy objects are also picked up in the initial scan, and these points of interest are recorded. When the robot gets closer to these points of interest, it will conduct an additional smaller scan to confirm if these fuzzy objects are real or not. This is described further in the Mapping section.

4.1.1 Experimentation and Results

Testing of the sweep scan consisted of plotting x and y values returned from the function. The variable changed was the delay time from initiating a change in servo motor position to when samples were taken, as this was the time needed for the servo motor to move into position. It was desired for this delay time to be as small as possible to minimise the time needed for the robot to clean the room.

Below are the results of the long A2 sensor sweeping an empty platform, shown in Figure 24. It can be seen that a 0 ms and 100ms delay time caused jagged lines for the walls, and are thus were not suitable. When the delay was not long enough, the sensor recorded values as the servo was still moving, creating a wrong mode value generated. 200ms and 250ms both have the same response, meaning it takes about 200ms for the servo to lock into position. Therefore a delay of 250ms was used, the extra 50ms being a safety factor.

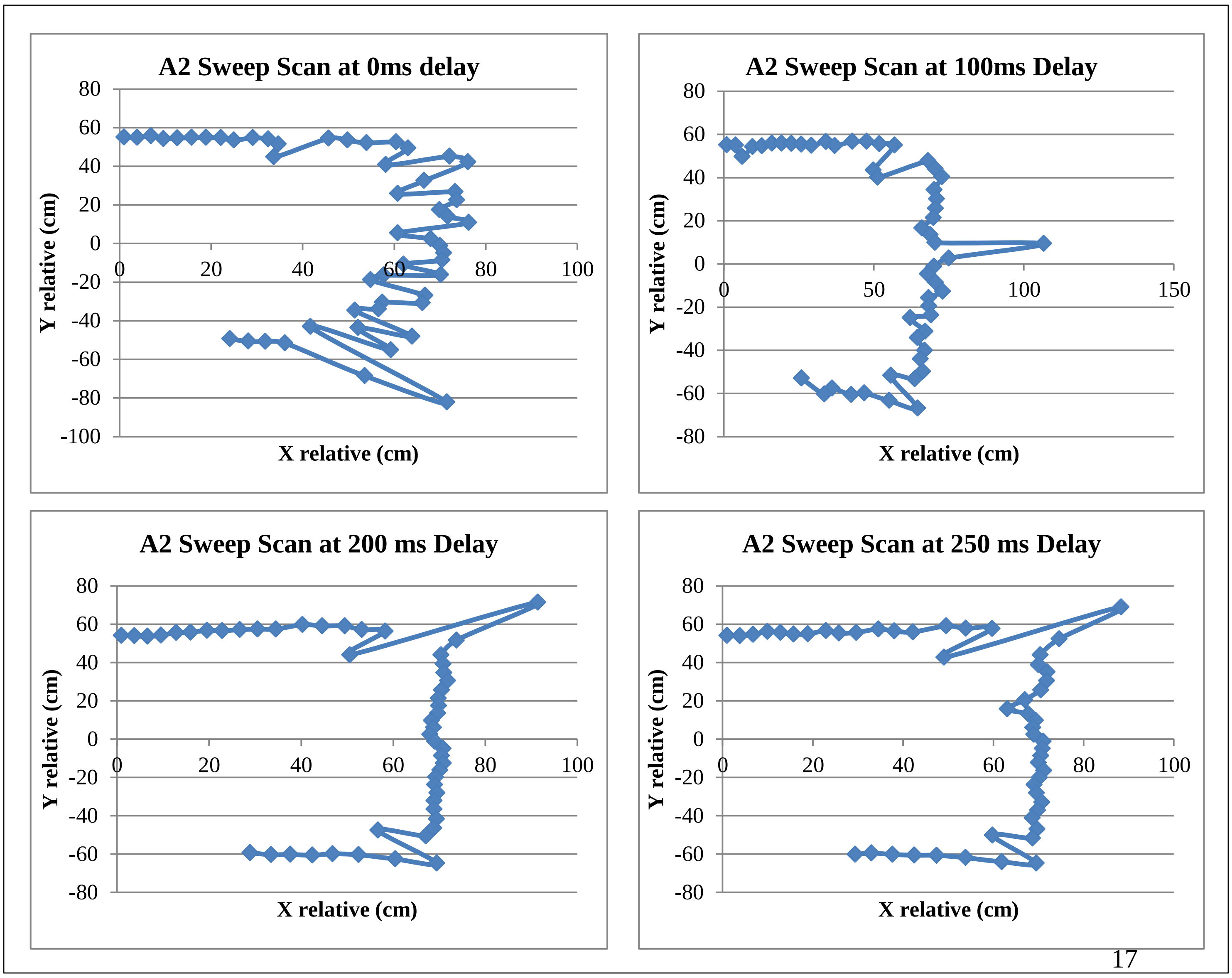


Figure 24: Initial Sweep Scan Testing

4.2 Localisation

For integrating SLAM, localisation of the robot in the xy plane was required to map its position as it is moving through the platform. However, when the robot is moving forwards, the long range infrared sensor facing the front cannot always necessarily find a wall, as its range is within 20 – 120 cm. Because of this, localisation of the robot while moving forward could not be accurately carried out with only the infrared sensors.

Localisation was hence accomplished through a form of dead reckoning, which was the process of calculating the robot's position from a previously known position. This was carried out by integrating the velocity of the robot with respect to time to get the distance travelled, and combining this with the heading to find the position of the robot.

The value of the robot's velocity, v_x, did not have a linear relationship with speed, so the distance calculated had to be calibrated to be equal with the actual distance travelled by the robot. Multiple points were taken to graph the relationship between the distance travelled and the velocity, and a trendline was fitted to these points. The equation of this trendline was used to calibrate the distance travelled by the robot to ultimately find the position. The relationship between the distance travelled and v_x can be seen below in Figure 25.

$$Distance = 3.901 \times 10^{-8} \times {v_x}^3 - 1.108 \times 10^{-4} \times {v_x}^2 + 0.1143 \times v_x - 13.94$$

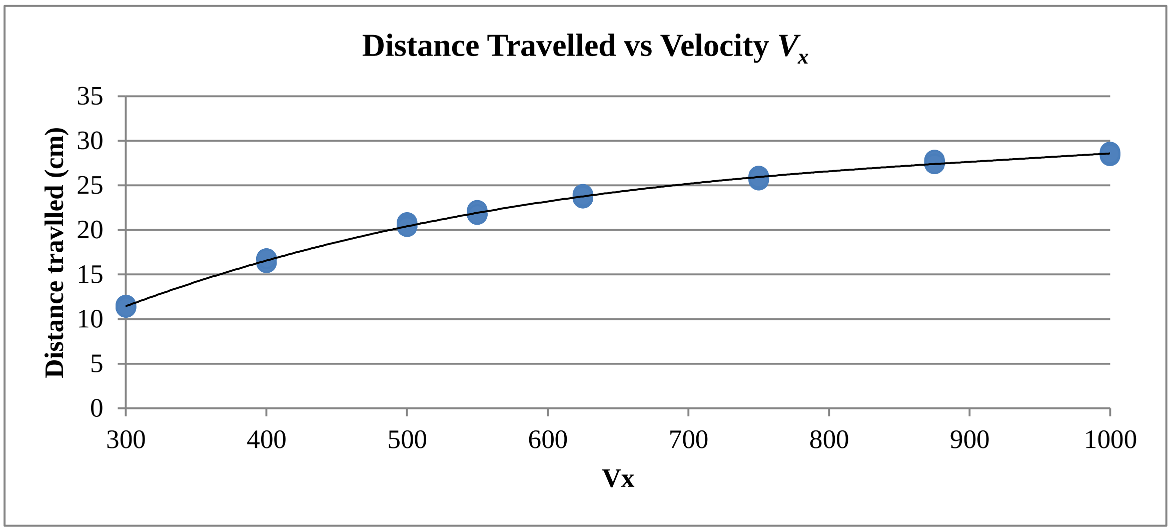


Figure 25: Graph of Distance Travelled vs Velocity v_x

Unlike for forward movement, when the robot was strafing to the side, the medium range infrared sensors facing the left side were able to be used for localisation. This is because the medium sensors were always able to see the wall to the left of the robot. The distance travelled was thus determined by finding the difference between the sensor readings at the start of the strafe and at the end of the strafe, with this difference being added to the previous position of the robot to find the current position. The current position was stored on the Arduino as a pair of xy coordinates which can be accessed at any time. This position was accurate to a centimetre resolution, so all small movements of the robot were able to be recorded.

4.3 Mapping

A map of the robot's path and the platform was created by storing points of interest in program memory. Three different types of points of interest were stored: edges, obstacles, and the position of the robot. These values were stored in lists which were able to be expanded or contracted when necessary. This system of storing points of interest as coordinates, as opposed to classifying and storing every position, minimised memory usage by eliminating the need to store empty space, while keeping all of the functionalities of mapping. Walls were assumed to be perpendicular to each other and connect the corners in a straight line, and hence were not stored as points of interest to conserve memory.

Using localisation, the robot knows its own position relative to the zero position, shown in Figure 15, which was the corner it first started its operation at. When a point of interest was added, its position was stored relative to the robot, which is relative to the zero position.

4.3.1 Room Mapping

When the robot reached a corner, the coordinates were added to a list of corners using the void AddCorner function, as shown in Figure 26. As only the current position of the robot is known, the true coordinates of the corners were dependent on the heading of the robot to compensate for the offset of the sensors that detected the corners. Case statements (one case shown in Figure 26) for each heading were used to find the true position of the corners. The corner coordinates were then updated onto the map that was stored in the global memory.

```
void AddCorner(int x, int y)  { // xy coordinates
switch (Heading)
  {
    case 0: //forwards
      {
        mapCorners[0][mapIndex] = x - 15;
        mapCorners[1][mapIndex] = y - 15;
        break;
      }
```

Figure 26: Corner Mapping Implementation in AddCorner Function

When a corner was added to the list of corners, void UpdateMapC() was called to print the coordinates of this to the map interface, as seen in Figure 26. Using this, corners were connected on the map to form the walls, with the assumption that the robot sequentially visits each corner following the adjacent walls. Walls were not saved in program memory in order to conserve memory space, but were displayed on the generated map interface.

```
void UpdateMapC()  {
  Serial1.println("Starting corner update...");
  String updatedCorners;
  for (int i = 0; i < mapIndex; i++)  {
    updatedCorners = updatedCorners + "x:" +
String(mapCorners[0][i]) + "y:" + String(mapCorners[1][i]) + "#";
  }
  Serial1.println(updatedCorners);
  Serial1.println("Corner update complete");
}
```

Figure 27: Updating Corner Coordinates to the Map Interface

4.3.2 Obstacle Mapping

When the front facing sensor detected an obstacle, this position was added to a list of obstacles using the void AddObject function, shown in Figure 28. The 'type' input into this function defined whether the object was fuzzy or non-fuzzy. Similar to adding corners, the true coordinates of the centre of the obstacle were displayed by offsetting the current position of the robot by a certain amount depending on its heading. This can be seen in the first case statement shown as an example in Figure 26.

```
void AddObject(int x, int y, int type)  { // xy coordinates,
type: object = 0, fuzzy object = 1
  switch (Heading)
  {
    case 0: //forwards
      {
      mapObjects[0][objectIndex] = x;
      mapObjects[1][objectIndex] = y + 5;
      mapObjects[2][objectIndex] = type;
        break;
      }
```

Figure 28: Object Mapping Implementation in AddObject Function

The function void UpdateMapO() was called in much the same was as UpdateMapC(), with object coordinates printed to the interface. During its operation, the robot was able to access these coordinates and check whether it was close to an object that it had already scanned. This allowed the robot to more effectively avoid collisions on later loops of its travel.

4.3.3 Path Mapping

The position of the robot was determined by integrating the velocity of the robot to find the distance travelled, as described in the Localisation section. Using this distance and the previous position, the current position of the robot was found. This current position was used to find the positions of corners and objects relative to zero. The position of the robot was updated by calling void UpdatePosition, as shown in Figure 29, where Elapsedtime was the amount of time passed since UpdatePosition had last been called, and Speed is the speed that the robot is travelling at. This function used the heading of the robot to determine the direction that the robot is moving in.

```
void UpdatePosition(float Elapsedtime, float Speed)
{
  //elapsed time is in milliseconds, speed is cm/s, position is
an integer
  switch (Heading)
  {
    case 0: //forwards
      {
        PosX = PosX;
        PosY += (float)(Elapsedtime / 1000 * Speed);
        break;
      }
```

Figure 29: Updating Position Coordinates to the Map Interface

4.3.4 Map Interface

The map of the room and the robot's path was generated dynamically as an image, and once the robot had finished moving, was saved as a text file. The map interface was generated using a Java based software called Processing, which read from the serial port and received information that the robot stored about its surroundings. These values were stored in the PC memory and displayed in a visually appealing way.

The map image updated dynamically as the robot moved through the room and found new points of interest. Corners were stored on the robot so that walls could be drawn between each corner. The path of the robot was also stored on Processing and shown on the map as a collection of arrows pointing in the direction of the robot's movements. An example of the map generated by Processing can be seen below, when the robot had completed one circuit of the room. This can also be viewed in a video format with the link https://youtu.be/Ut6CJ3il8Bo, which shows the dynamically updating map and the operation of the robot in real time.

The map image was saved once the robot completed its operation. An example of a generated map by Processing when the robot had completes one circuit of the room is shown below in Figure 30.

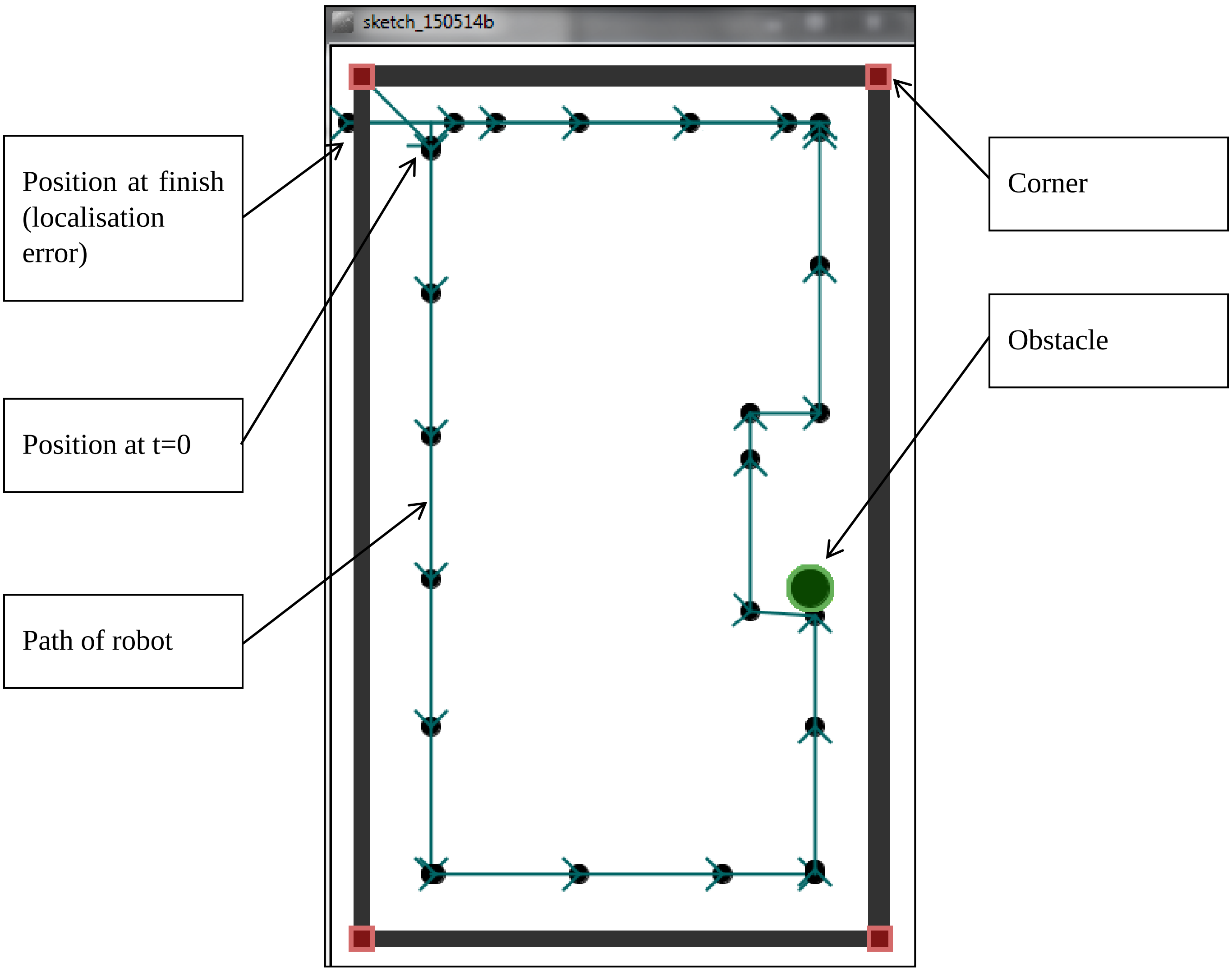


Figure 30: Map of Room after One Loop

The image and a text file which contained the coordinates of the corners and the obstacles were saved when the program exited. These files were generated using a naming convention which gave each file a timestamp so that multiple runs can be compared easily.

The map was communicated from the robot to Processing by leading a data transfer with a start recording flag. The recording is handled by a simple finite state machine in Processing, as shown below in Figure 31. A total of 5 states were used to check for start flags, reading the corners, objects, and position coordinates, and writing to a file.

```
switch (State)  {
     case 0:  {
        UpdateMap();
       if (val.contains("Starting corner"))  {
         State = 1;
       }
       else if (val.contains("Starting object"))  {
         State = 2;
       }
       else if (val.contains("Starting position"))  {
         State = 3;
       }
       else if(val.contains("Start writing"))  {
         State = 4;
       }

       break;
     }
```

Figure 31: Processing Finite State Machine

Coordinate values were read from the serial port as a string in the following format:

$$"x{:}XXXy{:}YYY\#"$$

Where “XXX” and “YYY” can be anywhere between 1 and 3 characters for the coordinates, and ‘#’ denotes the end of the transmission. The strings are parsed to extract the coordinate values of x and y which are appended to a stored list.

5.0 Finite State Machine

5.1 States

A total of 6 states were used in the finite state machine for autonomous coverage of the room. These were INITIAL_MAP, FIND_WALL, FOLLOW_WALL, TURN_CORNER, AVOID_OBJECT, and FINISHED. Autonomous movement of the robot was activated when the appropriate serial input was read from the user. Once read, the variable *AutoActive* was set to true and the state initialised to INITIAL_MAP.

5.1.1 INITIAL_MAP

To begin with the robot performed an initial scan of the room using the function FullSweepScan(). This was to provide an approximate layout of the room which could then be used as a starting point for the robot’s movement. Additionally, this state allowed sufficient time for the magnetometer to stabilise, ensuring correct performance from the start of each run. This initial scan is depicted in Figure 23.

5.1.2 FIND_WALL

The FIND_WALL state utilised the DetectWall() function to interpret the coordinates provided from the initial scan. DetectWall() receives a set of scan points from the initial scan. The data points are scanned for values that are grouped in a long straight line. Lines longer than 20cm long, with more than 16 points are considered to be part of a wall. Once all the walls have been found the closest wall can be found. The DetectWall() calls CalculateWallAngle() with the index value of the most visible wall, where the index value corresponds to the detected walls in the order they are detected. The CalculateWallAngle function returns the angle of the robot to the most visible wall as this will likely be the most accurate. The robot turns to face the wall using the rotate() function with the input of the angle returned from CalculateWallAngle. Once the robot is perpendicular to the wall, it will move to the wall using foward1() until it is approximately 5 cm away. Finally, the robot was made parallel with the wall by first rotating 90º, then AlignToWall() and AlignToWall2() commands were called. These two commands corrected any possible misalignment during its motion, and differ by their PID gains. The former used lower gains for rough alignment, and the latter utilised higher gains for finer alignment.

5.1.3 FOLLOW WALL

FOLLOW_WALL was the primary state of the state machine and therefore was the most complex. While 'following the wall' (through the use of the forward1() function) the robot was able to maintain the direction specified by the variable *MaintainAngle*, it was able to check its blind spots while moving, and it was able to stop a specified distance from the next fuzzy object it came into contact with – the next 'collision' that occurred.

When confronted with an unknown fuzzy object or wall, the robot was stopped and the fuzzy object was scanned and interpreted. The scan consisted of three key areas in front of the robot – to the left, straight ahead, and to the right, depicted in Figure 32. The left scan was then transformed into an x direction for more accurate comparison using trigonometry:

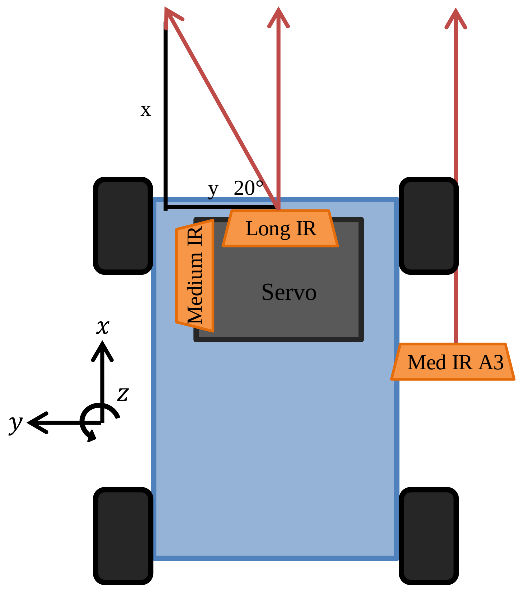


Figure 32: Fuzzy Object Detection

$$x_{dist} = measured \cdot cos^{-1}(20°)$$

Where 20 degrees is the angle difference from straight ahead scan and left scan.

The three distances from these positions were then compared to each other to determine if they were all the same, within some tolerance. Three equivalent distances signified the presence of a wall as the objects could not cover the entire front; the TURN_CORNER state would then be initiated. If the distances were not the same, fuzzy logic was applied instead. The distances were interpreted as close or NOT close, corresponding to the presence or the absence of an object respectively. These fuzzy sets were then fuzzified using the membership function shown in Figure 33 and subsequently defuzzified using the rules table shown below. In this way the appropriate *StrafeAmount* was able to be selected, along with the next state, being AVOID_OBJECT, TURN_CORNER, or FOLLOW_WALL. The magnitude of the variable, *StrafeAmount*, was extrapolated based on the dimensions of the robot and other experimentation and analysis.

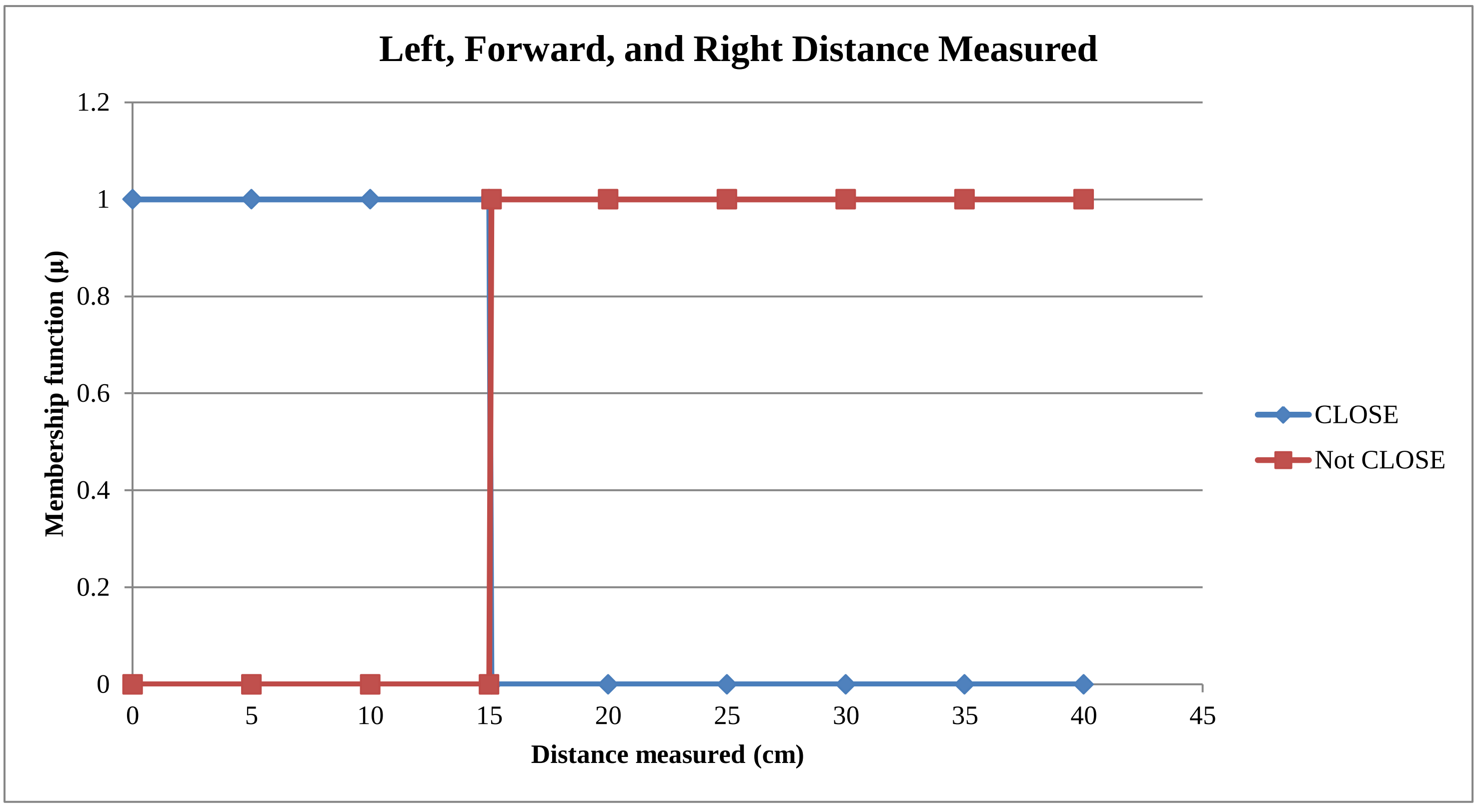


Figure 33: Membership Function showing Fuzzy Logic for Distance

Left	Straight	Right	OUTPUT
CLOSE	CLOSE	CLOSE	TURN_CORNER state
CLOSE	CLOSE	Not CLOSE	AVOID_OBJECT state, StrafeAmount = 15
CLOSE	Not CLOSE	Not CLOSE	AVOID_OBJECT state, StrafeAmount = 10
Not CLOSE	CLOSE	CLOSE	AVOID_OBJECT state, StrafeAmount = 25
Not CLOSE	CLOSE	Not CLOSE	AVOID_OBJECT state, StrafeAmount = 20
Not CLOSE	Not CLOSE	CLOSE	AVOID_OBJECT state, StrafeAmount = 30
Not CLOSE	Not CLOSE	Not CLOSE	FOLLOW_WALL state

5.1.4 AVOID_OBJECT

When faced with an object the AddObject() and UpdateMapO() functions were called to add the object to the map. AddObject() adds the position of the robot to an internal list, while UpdateMapO writes the full set of obstacle coordinates to the serial port. The robot then strafed around the object, moving to the right at first and then left again once it had passed the object. This meant that objects that were further to the left of the robot required smaller strafing distances than objects that were further to the right. Based on the *StrafeAmount* calculated from the FOLLOW_WALL state, the robot was able to strafe the appropriate distance to avoid any object.

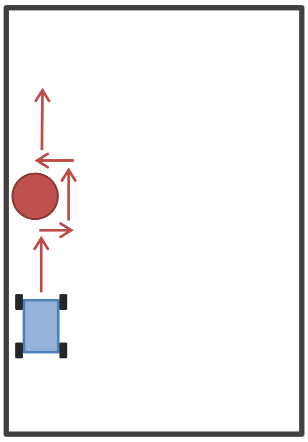


Figure 34: Collision Detection and Avoidance of Obstacles

To achieve this behaviour, the robot was required to remember the calculated *StrafeAmount* so that it could strafe back accordingly. It was also required to remember the distance to the left wall before moving past the object (readings taken from sensor A1). This distance was used to determine whether the robot had effectively passed the object. During the avoidance manoeuvre, the current A1 measurements were compared to the initial distance. If these were found suddenly decrease, then move back to be equal to the initial, the robot has cleared the object and was able to strafe back.

In the case when this distance check was not met – if the robot failed to recognise that it had passed the object, for example – a secondary, time-out condition of 2 seconds was utilised. This condition ensured that robot did not continue moving forward indefinitely after having passed an object.

Another feature within the AVOID_OBJECT state was position updating via the infrared sensors. The position was calculated as a function of time and updated using UpdateMapP().

5.1.5 TURN_CORNER

There are two ways to move into the TURN_CORNER state. When a ‘collision’ occurred, the fuzzy object was scanned and interpreted. If the fuzzy object was found to occupy the entire space in front of the robot at a consistent distance it was resolved to be a wall. Alternatively, if the first condition was indecisive, the robot was made to strafe. Comparisons between the pre-strafe position and the post-strafe position then ensued and if they were found to be identical within a given tolerance, the fuzzy object was interpreted as a wall. The functions AddCorner() and UpdateMapC() were called when any wall was found during the first loop of the robot’s movement, adding that corner to the map. After a wall was successfully identified, a rotation of 90° commenced and a new *MaintainAngle* (offset from the old one by 90°) was set.

5.1.6 FINISHED

Once the room was cleaned the robot ceased all activity until further input from the user. A counter, *TurnCornerCounter* was incremented each time the robot turned a corner, and when 12 turns was accomplished the room was defined to be clean. The writeMicroseconds() command was used to write to all the wheel motors, setting them all to 1500 (for a stationary response).

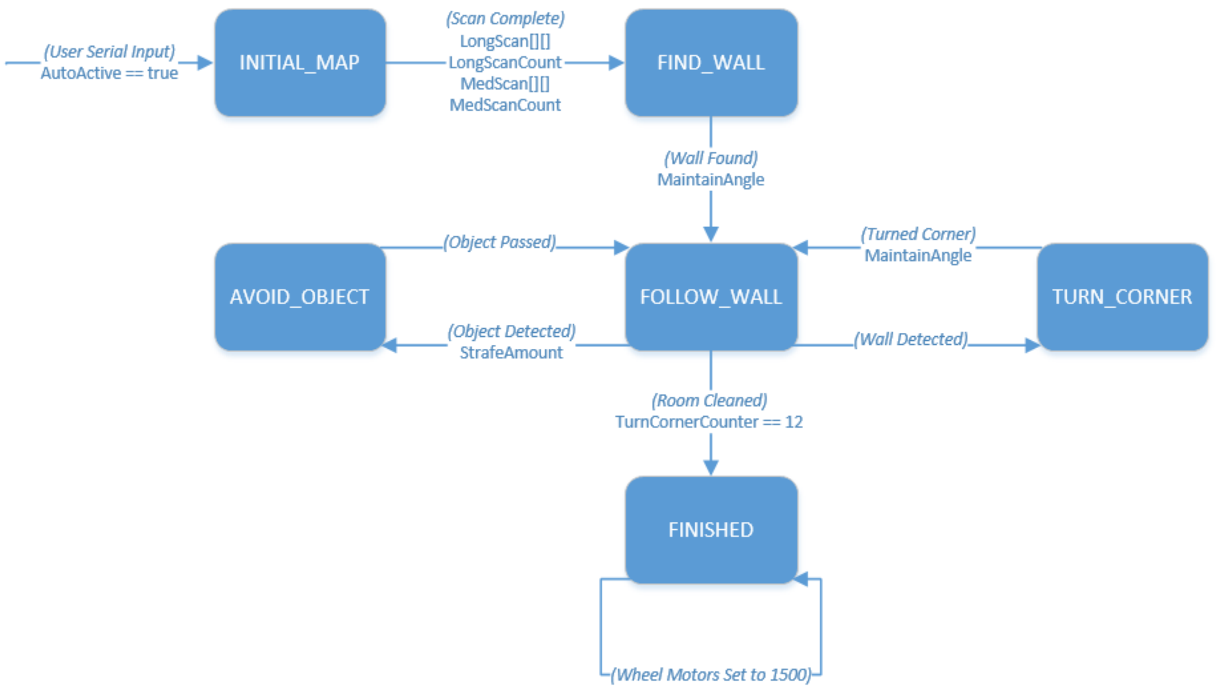


Figure 35: Finite State Machine

6.0 Results

The performance of the final implemented SLAM robot was reasonably successful, obtaining moderate coverage of the room (approximately 85%) with only 2 collisions, in approximately 4 minutes. The robot tracked the walls very closely and also left minimal area around objects when avoiding them. The robot failed to achieve 100% coverage due to an issue with the sensing accuracy.

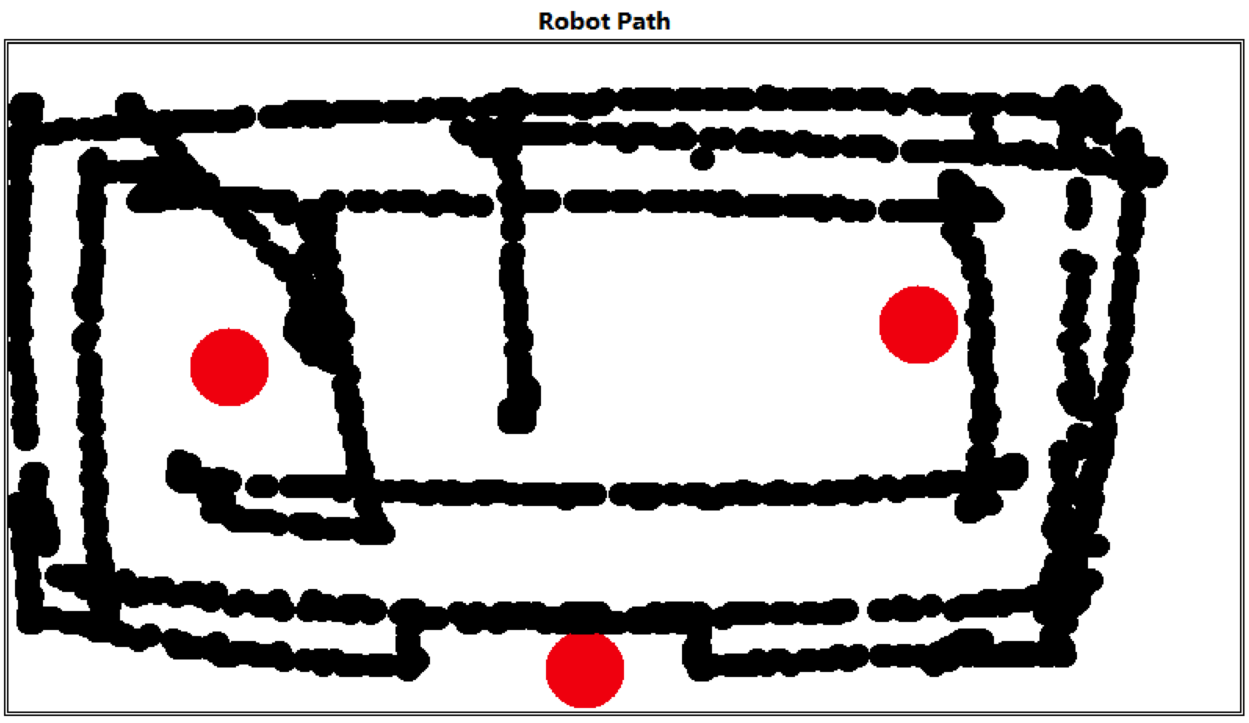


Figure 36: Coverage of the Room from Demonstration

7.0 Discussion

7.1 Attempted Ideas

7.1.1 Position Tracking

7.1.1.1 Through Accelerometer

The accelerometer gave acceleration along each axis, and was initially tested with the intention of using the data to infer the velocity and position of the robot. Since the platform is 2D, only the acceleration along the x and y axes were considered. The accelerometer data was first converted into readable values, and then integrated to find the velocity and position of the robot respectively. However, integrating these values gave an error which accumulated with each integrated sample. As the robot also tended to accelerate very fast, this causes more error to build up. The error was seen to be too large to gain a good estimate of the robot's position, and hence was not used.

7.1.1.2 Through Infrared Sensors

Another method to track the position of the robot was to solely use the infrared sensors. However, there were many situations where the infrared sensors would not be able to detect the walls to find its position, either due to its outer range of inner distance range. This means that any movement would be read with no change in position, resulting in a non-useable map.

7.1.1.3 Through Sonar Sensor

One idea to make localization easier was to exploit the long range of the sonar sensor, which has a maximum scope of 6.45 m. In using the sonar sensor, the robot would always be able to see the walls of the room and therefore it would always know its position, unlike with the infrared sensors. There were, however, several issues with regard to the sonar sensor.

Firstly the resolution of the sonar was extremely low, at 2.54 cm (1 inch). This meant that any distances or positions obtained from the sonar sensor would only serve as approximations. Furthermore, the sonar has a very wide beam. This feature, although having potential to be highly useful, also caused complications in which the beam of the sensor was reflected back by something other than the desired object or wall, such as the floor or a side wall. An attempt was made to channel the sonar better, using a tube lined with foam, but this proved to make negligible difference.

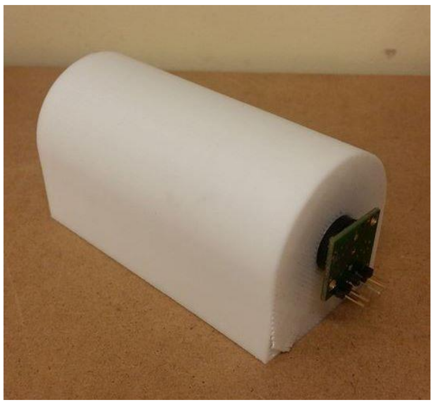


Figure 37: Tube used to narrow the Sonar Beam

7.1.2 Vertical Orientation of IR Sensors

The datasheet suggests placing the sensors in a vertical orientation, with the emitter and receiver in the same vertical plane. This orientation was experimented with using fake objects to detect how well it could sense edges and corners. The overall response was significantly better than its horizontal orientation. However, when it was tested in the platform for calibration, the long range IR sensors failed to read values after 70 cm. An actual distance change from 70 cm to

100 cm resulted in a measured change of around 4 cm. This was unacceptable, thus the infrared sensors were mounted the horizontal orientation.

7.1.3 All-Encompassing PID

An all-encompassing PID such as the one referenced from (visual dead reckoning for motion control of a mecanum wheeled mobile robot reference) that includes an x and y distance, and a z rotation, would not work due to the nature of the sensors and how x and y distances are determined. These distances are measured with the IR sensors relative to the wall, and are prone to noise. On top of this, when rotation occurs, the sensors no longer point to the same spot, possibly hitting a different wall or an object, which will give a hugely different reading. Therefore, accurately measuring distance travelled is impossible due to the rotation causing any relative measurement to be false.

7.2 Further Improvements

Although the SLAM robot can move around the room quickly, avoid obstacles, and create a map, it currently has limited success. Listed below are discussed improvements to be implemented in the future in order to create a robust fully functional SLAM robot.

7.2.1 Fine Tuning

The stop conditions inside the forward1() function (responsible for making the robot move forward) do not cause the function to exit quickly once it reaches its destination. This results in the robot reaching a wall, and stopping for long periods, sometimes up to a minute, before turning. This is caused by the robot trying to evaluate the stop condition, and then being invalidated by fluctuating readings. The stop condition needs to be fine-tuned to prevent this from happening. This could be done by increasing the size of the stop condition dead zone; so that when the robot stops moving the fluctuation of the sensors is well within the band of the stop condition.

The PID gains of all motion controllers should also be fine-tuned, for the best response.

7.2.2 Layout of Sensors

The sensors mounted on the servo consistently fell off, due to a loose fit between servo and its connection. This could be remedied by supports on the servo mount, so the servo only rotates the mounted sensors, instead of additionally supporting its weight.

The MPU also needs an improved mounting, as currently it is mounted at a slight angle, which may cause malfunctions or false readings. The data sheet mentions that the best readings are obtained when it is mounted on a flat surface parallel to the floor.

When the IR sensors were mounted vertically, the long range infrared sensor was rendered useless after 70 cm. This was solved by mounting them horizontally; however this reduced the quality of the readings. The option of having infrared sensor cones to support the performance was not explored at the time. This could be an option in the future to enhance performance in detecting corners and edges.

7.2.3 Magnetometer MPU Post-Processing

Using the magnetometer to find heading introduces problems due the non-linearity of angles from the MPU. It returns a range of values between -180 and 180. An already implemented solution would be to calculate the difference in angle dynamically and accumulate the corrected values. This, however, caused the robot to react much slower, reducing performance of the rotational PID and caused error in the final heading. This method will need to be revisited and improved upon in future. Another solution would be to explore the gyrometer.

7.2.4 Accuracy of Localisation

The current method of localisation is dead reckoning. The value Vx, used to control the motors speed in the x direction, is also used to calculate speed. The relationship between Vx and speed is nonlinear and relies on battery level and the friction of the surface. Vx is set as the value that the motors try to achieve, which means that each motor receives a different pulse width, and the desired Vx may or may not be achieved, meaing Vx is not a good indication of how the robot is moving. On a higher level, dead reckoning is inherently flawed, as it creates an open ended system, where there is no feedback from the sensors regarding the position. An infrared mouse sensor could be used to implement a dead reckoning system more accurately. In order to improve localisation, a form of feedback should be introduced to correct the error of dead reckoning, or replace it. This could involve GPS or tracking system that exists outside of the robot, such as a camera that tracks the robot. An external sensor relaying position will eliminate error due to slip and the compounding error in the dead reckoning position calculation.

7.2.5 Waiting Time

To combat drift in the Magnetometer, every time the robot turns on the first loop, it realigns to the wall using the Infrared sensors facing the wall. This realignment results in a long waiting time as the integral gain needs time to take effect. Before the introduction of *MaintainAngle*, this was the only way to maintain squareness effectively and thus had to be extremely accurate. However, after the introduction of *MaintainAngle*, the forward function can align itself as it moves without any issues, and is calculated in the alignment function. To improve response time, the alignment function does not need to be extremely accurate as the forward function will correct any misalignment, meaning the integral part of the gain can be removed. As mentioned in tuning, the stop conditions of all motion functions could be improved to reduce wait time.

7.2.6 Mapping Limitations

The map currently is limited to a 240x240cm area on the interface. This means that the interface is not scalable to large areas. A system to make the map dynamically expand as the robot explores new territory could be introduced.

The interface does not show the area covered well graphically or numerically. Only positions which the robot has visited are shown with no indication of the size of the robot or the area covered. New functionality could be built in to show area covered more obviously, and perhaps show a numerical value of the area covered.

7.2.7 Fuzzy Objects

The current build of the robot avoids obstacles as it finds them. This is not ideal as time is wasted determining whether an obstruction is an object, and can sometimes be incorrect. A better solution is to do an initial scan of the area, to find the general location of surrounding objects and save their general locations to the map. When the robot approaches the fuzzy object it does a quick scan to locate it properly before moving on. Fuzzy scans will help the robot move around the area much more efficiently, because it can know where it can move safely without hitting anything.

8.0 Conclusions

An autonomous SLAM robot has been designed and tested to perform as a vacuum cleaner. It was able to clear a room with reasonable success, achieving approximately 85% coverage in 4 minutes with a total of 2 collisions. The robot thus completed all of its main objectives and succeeded in performing as an autonomous vacuum cleaning robot.

The robot was fully mobile and autonomous, using a total of five sensors – one long range infrared sensor, three medium range infrared sensors, and one MPU. A variety of different filters, both hardware and software, were used to ensure accurate readings and optimal performance of the robot.

The map was successfully generated, displaying all physical attributes (walls, and objects) as well as the coverage of the robot.

In conclusion, all aspects of the project brief for an autonomous SLAM robot have been achieved, with several features added for improved functionalities.

9.0 References

[1] Aarenstrup, R. (2013, November 1). *Infrared Proximity Sensors*. Retrieved May 5, 2015, from Makerzone: http://makerzone.mathworks.com/resources/using-infrared-proximity-sensors-with-simulink-and-arduino-part-1/

[2] Barnett, R. (n.d.). *MPU-9150 Library*. Retrieved April 22, 2015, from GitHub: https://github.com/richards-tech/MPU9150Lib

[3] Bouchier, P. (2014, October 24). *Building an AHRS Tutorial*. Retrieved May 10, 2015, from GitHub: https://github.com/ptrbrtz/razor-9dof-ahrs/wiki/Tutorial

[4] *Filtering Sensor Data with a Kalman Filter*. (2009, December 18). Retrieved April 28, 2015, from Interactive Matter Lab: http://interactive-matter.eu/blog/2009/12/18/filtering-sensor-data-with-a-kalman-filter/

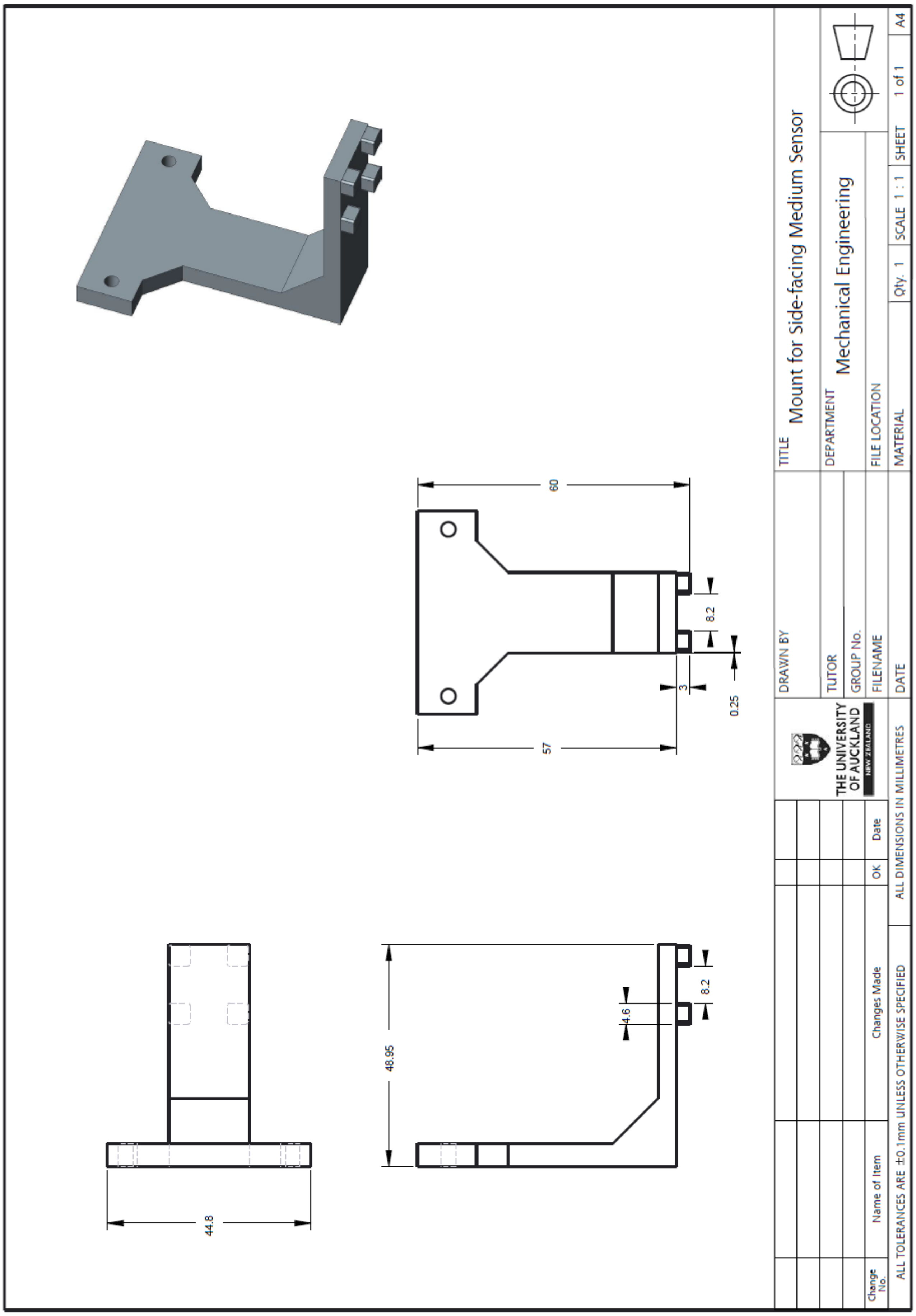
[5] *Infrared Proximity Sensor*. (n.d.). Retrieved May 20, 2015, from SparkFun: https://www.sparkfun.com/products/242

[6] *iRobot Roomba Vacuum Cleaning Robot*. (n.d.). Retrieved May 20, 2015, from iRobot: http://www.irobot.com/For-the-Home/Vacuum-Cleaning/Roomba.aspx

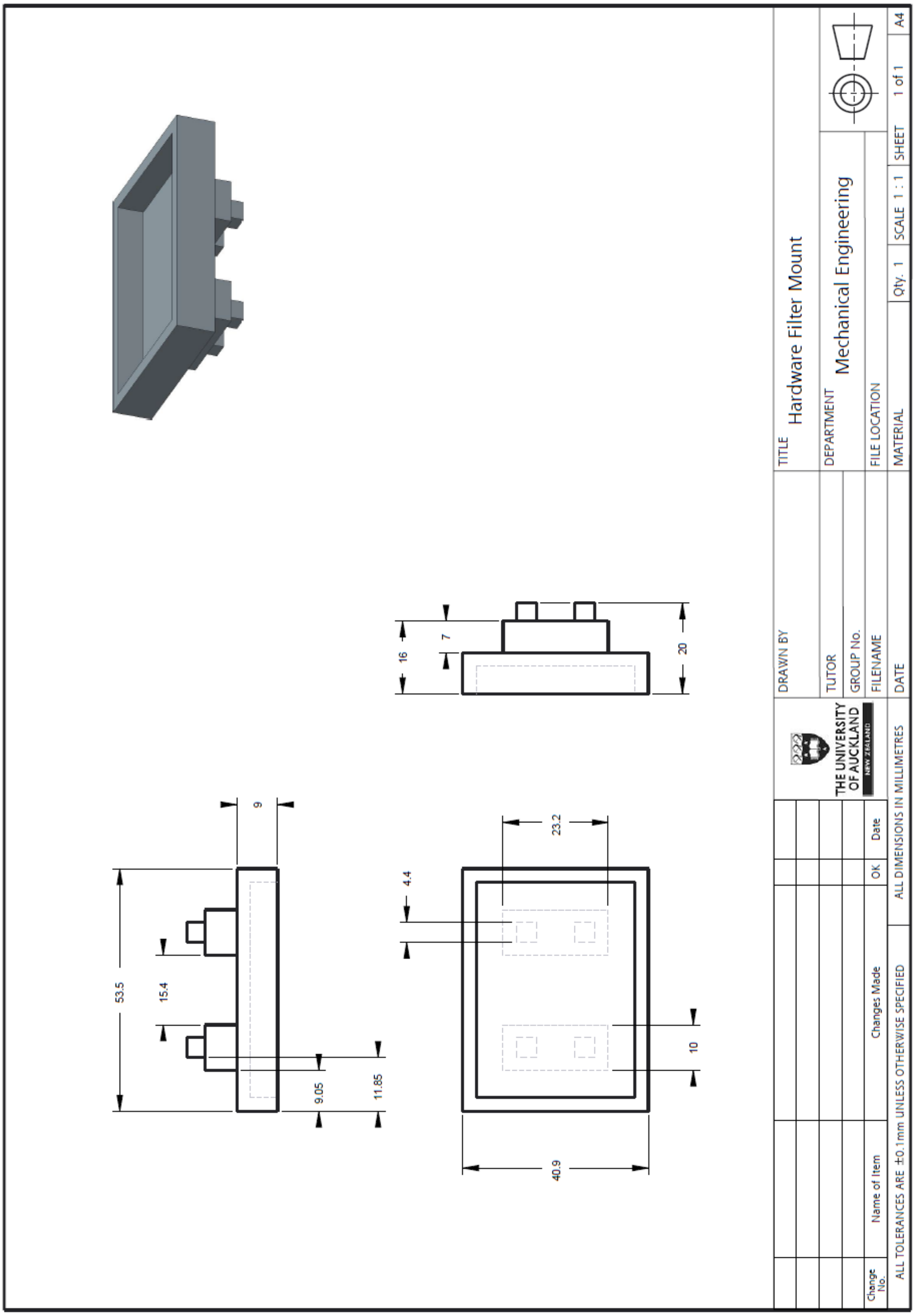
[7] *SparkFun 9 Degrees of Freedom MPU-9150*. (n.d.). Retrieved May 20, 2015, from SparkFun: https://www.sparkfun.com/products/11486

10.0 Appendices

10.1 Side Infrared Sensor Mount Technical Drawing



10.2 Hardware Filter Mount Technical Drawing


16
7
20
9
23.2
4.4
53.5
15.4
9.05
11.85
10
40.9
TITLE
Hardware Filter Mount
DEPARTMENT
Mechanical Engineering
FILE LOCATION
MATERIAL
DRAWN BY
TUTOR
GROUP No.
FILENAME
DATE
Qty. 1
SCALE 1 : 1
SHEET 1 of 1
A4
THE UNIVERSITY OF AUCKLAND NEW ZEALAND
Change No.
Name of Item
Changes Made
OK
Date
ALL TOLERANCES ARE ±0.1mm UNLESS OTHERWISE SPECIFIED
ALL DIMENSIONS IN MILLIMETRES