MECHENG 706: SLAM

GROUP 10

Theo Drissner-Devine
Nitish Lobo
Logan Porter
Justus Katzur
THE UNIVERSITY OF AUCKLAND | 5/6/16

1. Executive Summary

The aim of the project was to design a SLAM (Simultaneous Localisation and Mapping) system that a robot could use to navigate an unknown environment. The three main criteria for this project are the robot's ability to avoid obstacles, get the maximum area coverage and complete the course in the fastest time possible. The robot is equipped with an Arudino Atmega8 processor, omnidirectional wheels, and a number of sensors. The sensors that were available for use were: MPU, sonar and short, medium and long range infrared (IR) sensors. Moreover, the project allowed for the use of four IR sensors in any combination that was desirable. Three main software functionalities that this project can be divided into are wall following, corner and obstacle avoidance and all of these are dependent on the type of navigational logic (spiral, vertical/horizontal zig-zag, random memorization-type approach). A major challenge in this project was to get accurate distance readings from the sensor and to be able to figure out when the sensors where outputting large consecutive sets of outliers. However, all challenges faced were overcome and an efficient solution which covered all the area with minimum path overlap, avoided all obstacles and completed the course with a time of around 1 minute and 30 seconds.

Contents

Table of Figures

2. Introduction

Simultaneous Location and Mapping (SLAM) is a computational problem in which an agent has to create a map of its surroundings whilst navigating an unknown environment. Whilst initially this creates a paradox of needing your location to create a map, and a map to get your location, there are several algorithms which overcome this problem. As data input is usually limited, these algorithms are usually designed for operational compliance, rather than perfection. SLAM is used in multiple applications from autonomous vacuum cleaners to self-driving cars and UAVs.

The aim of the project was to create an autonomous robot for "vacuum cleaning" which would use SLAM to navigate and map an enclosed area whilst avoiding obstacles. The 3 main criteria were; area coverage, obstacle avoidance and time taken. A map also had to be displayed to show walls, obstacles and area covered by the robot.

To achieve this 4 infra-red sensors, a sonar and a MPU were given as sensors, along with an Arduino Mega 256 board to process the data and control the Mecanum wheels.

3. Project Description and Specifications

The project required an autonomous robot to navigate an unknown environment using SLAM with the following constraints:

The following equipment was available:

- 2 Long Range Infrared
- 2 Medium Range Infrared
- 1 Sonar
- 1 MPU 9150
- 1 Small Servo
- 4 Servo with Meccano wheels attached
- 1 Bluetooth module
- 1 Arduino Mega board

The following tasks needed to be completed:

- Navigate an unknown environment autonomously without colliding into obstacles
- Create a map of the environment and the robot's path
- Cover maximum area while taking the minimum time possible.

4. System Design and Integration

The design of the robot, in particular the layout of the sensor hardware on it, was a critical phase of the project as it determined what kind of path the robot would take in order to meet the task specifications. There were two main path types that were examined to determine the layout of the sensors. These were zigzag and spiral.

4.1. Strategy

4.1.1. Zigzag path

Zigzag path would mean that the robot initializes in the corner of the arena and then sweep back and forth between the two shorter walls, so to reduce the amount of corners the robot would have to turn. Two long range IR along with data fused from the MPU data were considered to make the robot travel in a constant straight line along the wall. The medium range IR's would be used for wall and obstacle detection together with the sonar which would be attached to the servo which could be swept in order to obtain data for mapping.

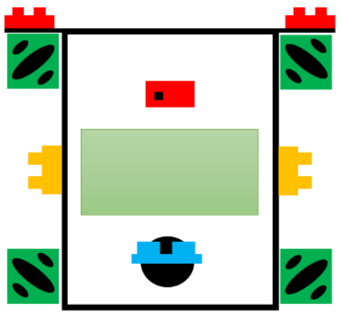


Figure 1: Possible sensor layout for Zigzag path

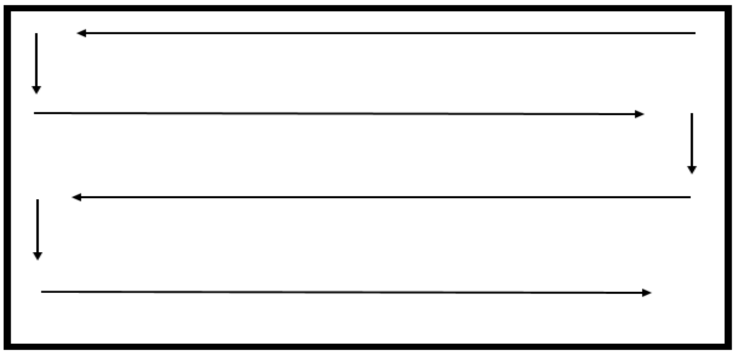


Figure 2: Zigzag path

4.1.2. Spiral path

The path would initialize by a wall and follow it around. The robot path would be defined by the wall structure of the arena. Each time the robot would pass the first corner, the distance to the wall would increment until it reached the middle of the map, where it could be programmed to stop. The two long range IRs would measure distance to the wall and be fused with the Motion Processing Unit (MPU) data to keep the robot straight, whilst the medium range IRs and sonar would be used for wall and obstacle detection/avoidance. The sonar would also be mounted on the servo to sweep around and gather data for the map.

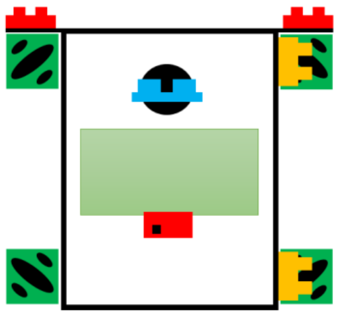


Figure 3: Possible sensor layout for a spiral path

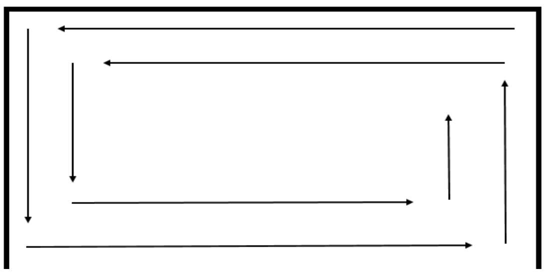


Figure 4: Spiral path

4.1.3. Final sensor layout

After testing and calibrating the sensors, the spiral path was chosen as there are at least two sensors pointing in each direction to compensate for any reading fluctuations. Further testing of the servo and sonar combination showed that the speed at which the servo would have to turn to obtain useful readings from the sonar was too slow for it to have any practical mapping function. The medium range IRs were also moved back as the dead zone proved to be too big to accurately detect the distance of a wall or object when it was too close to the robot. After further live testing, the medium range IRs were turned to be oriented in a vertical manner (as opposed to its original horizontal configuration) as this resulted in more accurate readings in the corner due to the way in which light would reflect back. The long range IR was also moved backward due to the larger than expected dead zone. In order to increase the accuracy of the robot's alignment with the wall and for the control to be stable, an extension rod was created on which the side-front sensor could be placed. .

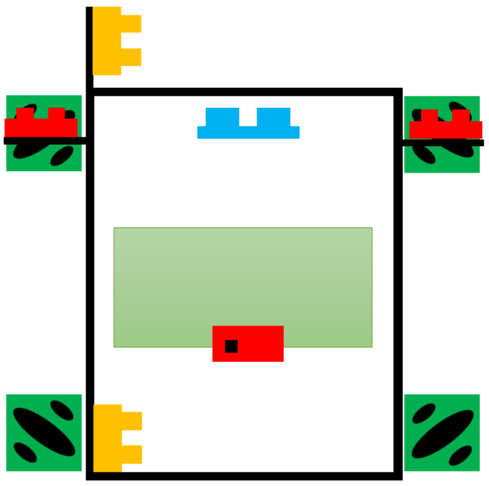


Figure 5: Final sensor layout design

4.2.Sensor Testing

4.2.1. Sonar Testing & Calibration

Sonars work by sending out high frequency sound and measuring the time it takes for the signal to return. Initially the sonar and servo were going to be used for mapping by rotating the sonar around to get distances to walls and objects independently from the rest of the robot. After trying the sonar on the robot, the data showed that to get accurate data the servo would have to turn too slow to be of any use whilst the robot drove around. The sonar was instead placed in fixed mount facing forwards to help with wall and obstacle detection. The sonar works in conjunction with the two medium range IRs to calculate distance to walls or objects, due to the fact that even when measuring the same distance, the sonar can give fluctuating readings. More on this can be seen in section 5.3 on filtering.

4.2.2. MPU Testing & Calibration

The 9 DOF inertial measurement unit (IMU) gives readings of linear and angular acceleration and magnetometer readings. The initial approach for calculating the linear motion of the robot was to use a low pass filter and then double integrate the output of the accelerometer to obtain information about the position. This was implemented on the robot but the data from the accelerometer proved too noisy to be useful. The reason for the excessive noise was due to the wheels of the robot not being circular causing the robot to rock back and forth as it drives. These large swings cause large spikes in the accelerometer data caused very large errors when integrated that make the resulting position data unusable.

To track heading of the robot the initial approach was to integrate the gyroscope readings of angular acceleration twice. This method was thought to be more reliable as it would not suffer from low frequency drift. However, when this was implemented the angle readings had similar errors to the accelerometer data. Instead the magnetometer readings were used to measure the heading of the angle. Magnetometers typically suffer from low frequency drift, to account for this a high pass filter with a cut off frequency of 80 Hz was used. The rest of the error in the magnetometer readings was dealt with in post processing of the sensor data. The magnetometer data is crucial in localisation of the robot in the mapping software.

The heading angle was determined by combining the magnetometer readings in the plane of motion of the robot. The plots in Figure 6 from the outputs of the two magnetometers of interest show that the magnetometers have intrinsic offsets from true values. These are accounted for by adding a constant to each magnetometer reading. The magnetometer heading reading can then be calculated by treating the two readings as Cartesian coordinates and translating the coordinates to polar form.

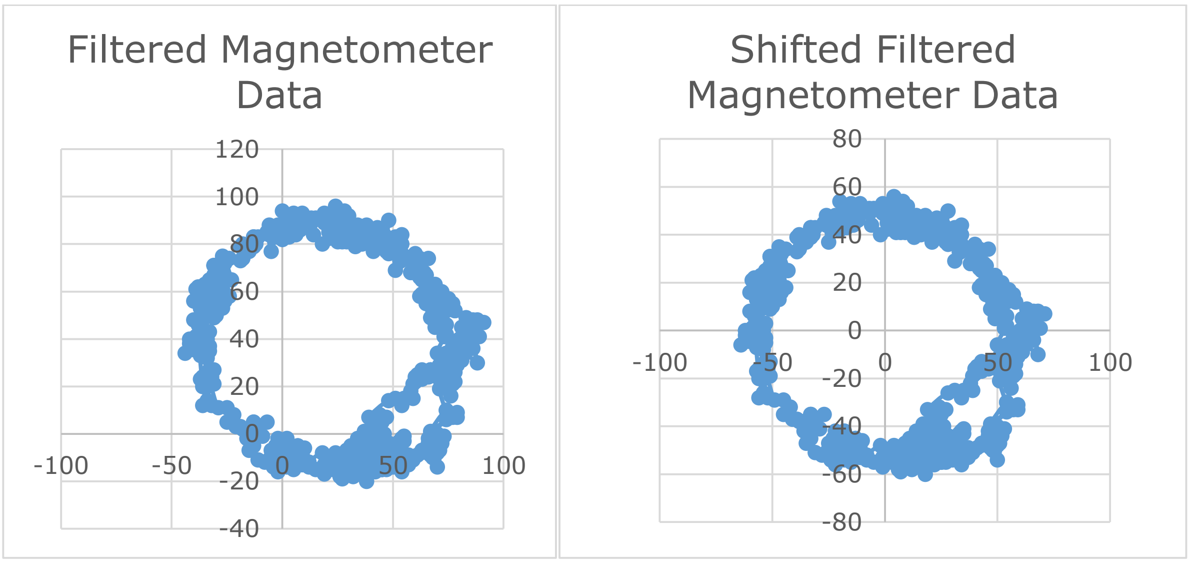


Figure 6: Graph of magnetometer data

4.2.3. IR Testing & Calibration

The first step of using the Infrared sensors was to characterise their response to different distances. The sensors were tested from the range of 25mm until their response became asymptotic. This was because it was decided that after this point no more data could be reliably gathered. The output from this testing can be seen in the graph below. This testing was also useful in determining the dead band of the sensor, where the response would give increasing readings for increasing distance, which was different to the rest of the response. This can be seen in the peak of responses in the long and short range sensors in **Error! Reference source not found.** below. The short range sensor peaked at 50mm and the long range sensor at 125mm. The result of this was that the sensors were mounted in such a way that they would never be able to get into this region.

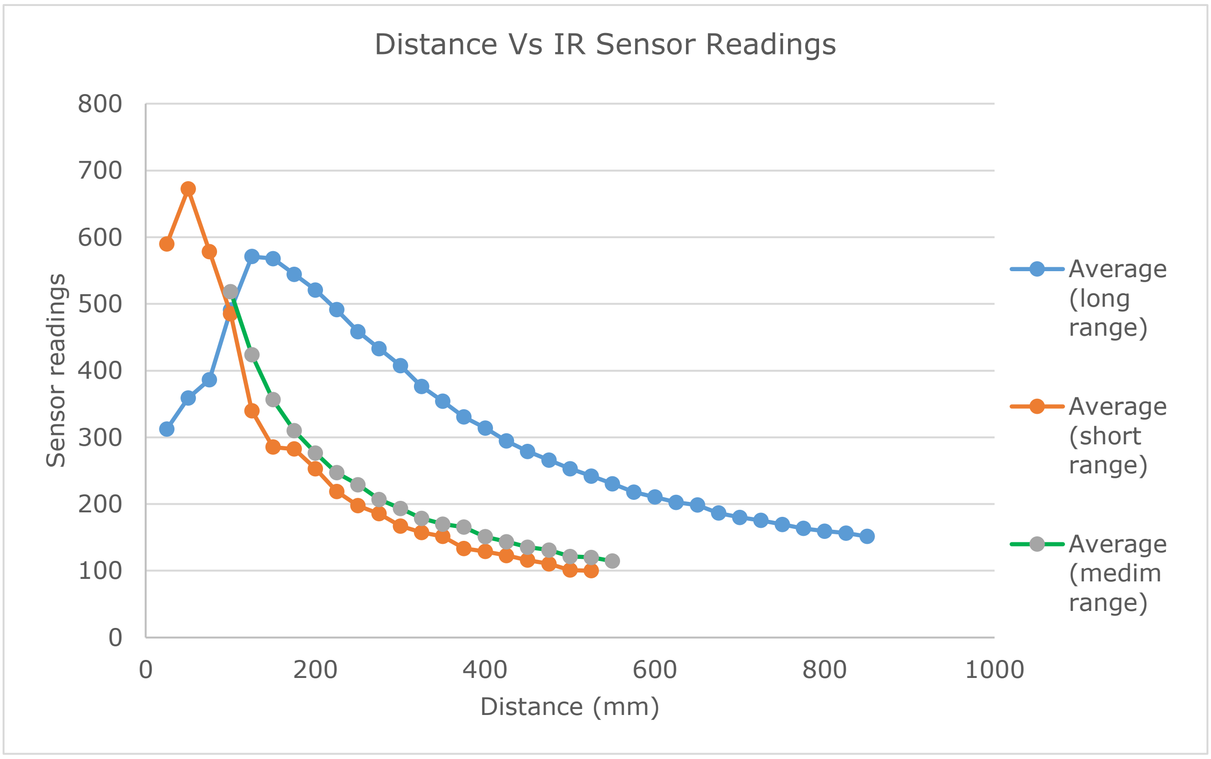


Figure 7: Chart of sensor response Vs. Distance

The readings for each of the sensors were looked at and a trend line was fitted to determine a model which could be used to give real world readings of the response. Only data from beyond the dead band was included as this was the range where the sensors would be used.

Initially short range sensors were used for obstacle and front wall detection but their readings proved too variable at the same distance to be useful. There would be large jumps in the readings which would make it problematic to use these readings. For this reason, they were exchanged for medium range sensors. The following models are the ones which were eventually used.

The model for the sensor response was found by plotting a trend line to the data in excel. The plot of the long range response is seen in blue in Figure 8: Prediction of distance from sensor data below with the trend line equation also shown. This is the result of inverting the graph above and using the distance as the response variable. The best fitting equation was found to be a power curve.

The same process was followed for the medium range sensor. The resulting model is seen below on the same figure, in green. This power model is a better fit than the long range sensor, probably because it has fewer data points.

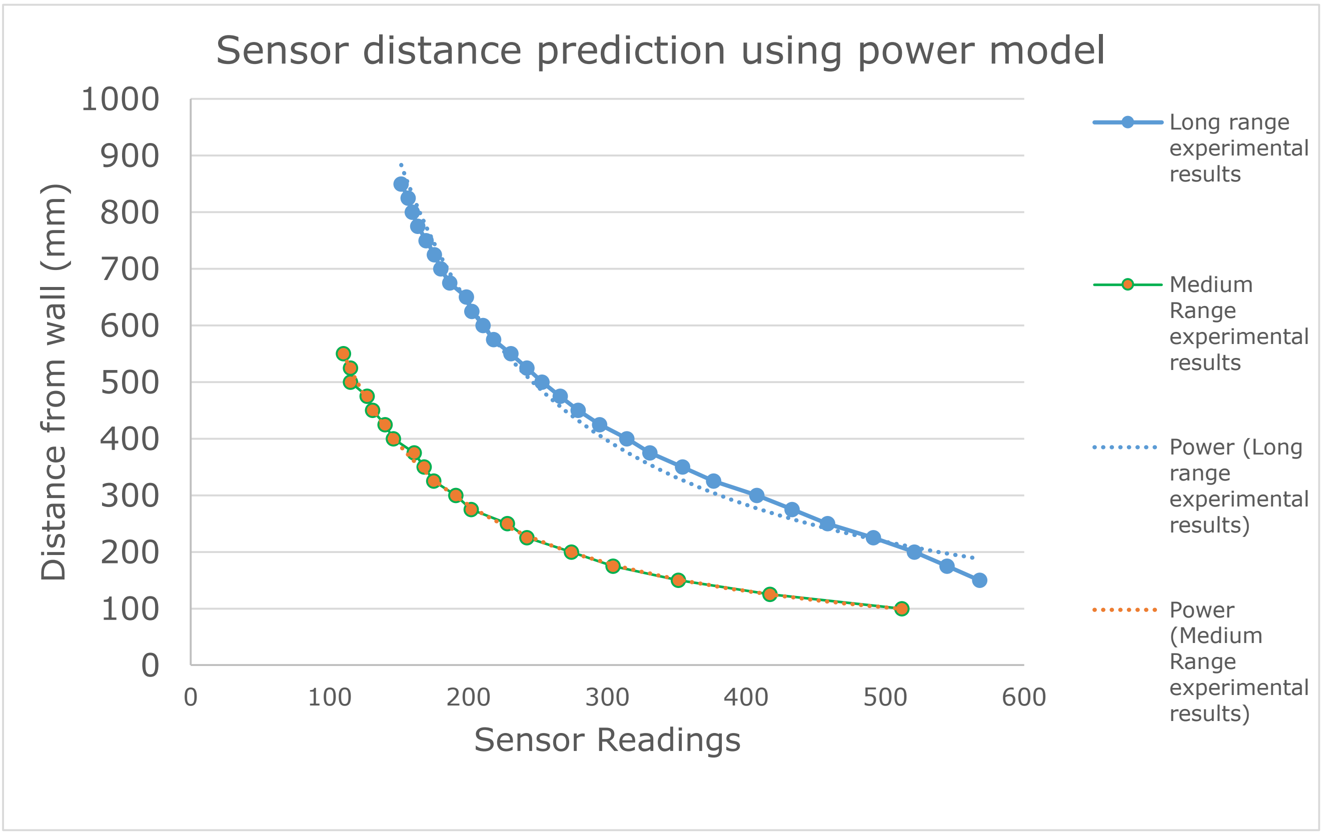


Figure 8: Prediction of distance from sensor data

4.3.Finite State Machine

The following diagram shows the possible states that can occur in the finite state machine and what their precursor and successor states are.

*All starred states can go to stopped state (due to battery loss or end of course).

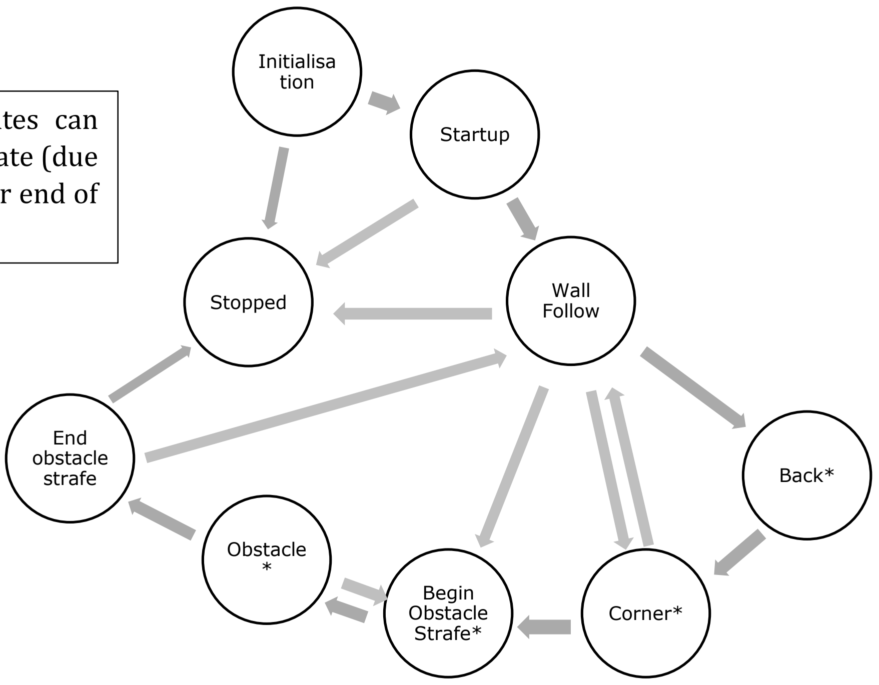


Figure 9: Finite State Machine diagram

4.3.1. Initialisation

When the robot is first switched on, the robot will always go into the initialisation state in which it enables the motors and waits for 1s to initialise before outputting the sensor data via the Bluetooth module.

4.3.2. Startup

Startup's main functionality is to figure out the shortest distance to the wall from its starting position by checking the difference in the distances from the side long IR sensor readings. If the difference between the two sensors is beyond a certain threshold, then the robot is able to know that there is an obstacle at its side and that it should not strafe in towards that direction. Whereas, if the two side sensors are within an acceptable range of each other, then it means that the robot has found a straight wall that it is able to align to.

There are three main possibilities for startup state:

1. Robot starts with the sides facing to a straight wall

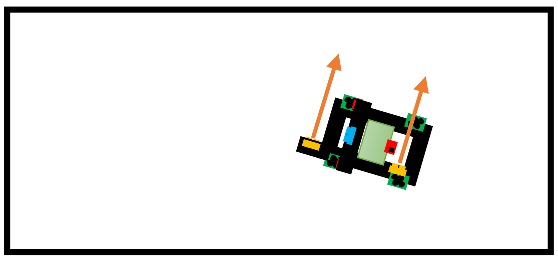


Figure 10: Robot starting nearly aligned to wall

This is a straightforward case where the robot first aligns itself to the wall until it is collinear and then strafes in towards the wall until it has reached the pre-defined distance from the sensor to the wall. Once it is at this distance, the robot will once again execute the align code to ensure that it is perfectly aligned to the wall.

2. Robot starts with sides facing to an edge

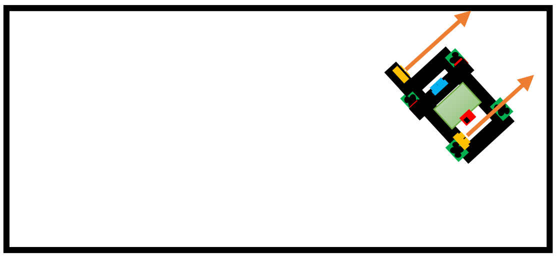


Figure 11: Robot starting in the corner

A large majority of the time, the robot's algorithm will be able to detect this case because of the large difference between the two side IR sensors. However, if the robot is perfectly positioned in such a way (robot's heading is offset 45 degrees to the corner) that both sensors give equal readings (or readings within an acceptable range of each other), then the robot will still strafe into the corner but the corner state will be executed as the front sensors will see the wall. This means that the robot will never collide with the corner.

3. Robot starts with obstacle at its side

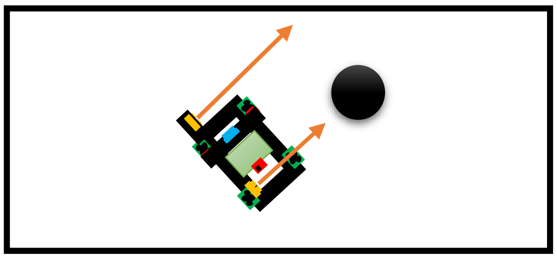


Figure 12: Robot starting close to obstacle

In this situation, the algorithm is able to detect whether the obstacle is at the front-side or the rear-side of the robot because the IR readings for where the obstacle is will be substantially lower. The robot will rotate clockwise or anti-clockwise to position itself away from the obstacle.

Once the robot rotates on its spot for a few seconds it will check the side sensor distance readings again for any further obstacles. The code repeats itself until no more obstacles can be detected and the robot can safely strafe in towards a wall.

4.3.3. Wall Follow State

Code in the previous states ensure that the robot is always at the required distance from the wall before it enters into the state. The purpose of the wall follow state is to keep the robot parallel to the wall by using the proportional controller seen in section 5.2. It will continue in this state until it sees an obstacle or a wall in which case it will go into the appropriate Corner or Begin Obstacle Strafe state.

In order to prevent obstacles from being mistakenly seen as walls from far away, the robot has to go to the range where the obstacle detection is reliable. This was set as 265mm. Because the robot spirals inwards at increasing distances from the wall, in later loops it will have to come closer than the set point distance to understand whether it is an obstacle or wall. As a result, it reverses outwards once it has come to the required reliable detection distance. It will then go into Back state. As a check to stop this moving into obstacles, a flag is set if an obstacle was detected in the most recent length and if so, it should not go back, but sacrifice coverage for not reversing into an obstacle.

4.3.4. Back State

Because the robot had to get into a certain range to detect if an obstacle is present, it sometimes has to move back out of this distance when it a wall rather than an obstacle. Inside the reverse state it will only move backwards until it gets far enough from the wall.

4.3.5. Corner

In this state, the robot turns counter-clockwise for 1000 milliseconds to find the next wall and the robot then aligns itself to this new wall using the long range IR sensors situated on the side of the robot. After finishing this sequence, the robot then strafes into or away from the wall depending on which track the robot needs to be on in the spiral algorithm.

4.3.6. Obstacle States

There are three main states for the obstacle avoidance:

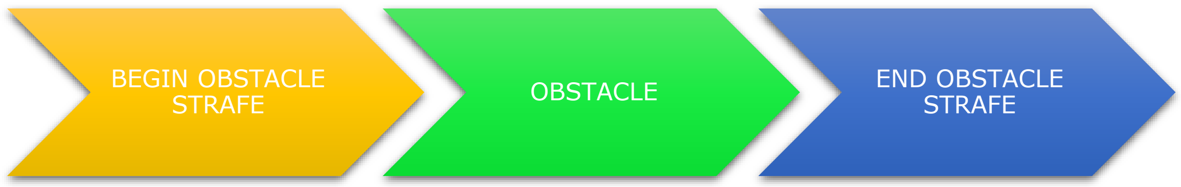


Figure 13: Obstacle state process

4.3.6.1. Begin obstacle strafe

The robot enters into the begin obstacle strafe when an obstacle has been detected in front of the robot. The obstacle strafe in towards the centre of the arena rather than towards the wall because if the obstacle is on the outer track of the arena, then the robot will strafe into the wall or otherwise the robot path will have double-ups thereby increasing the time to complete the course.

This can be visualised by the diagrams below:

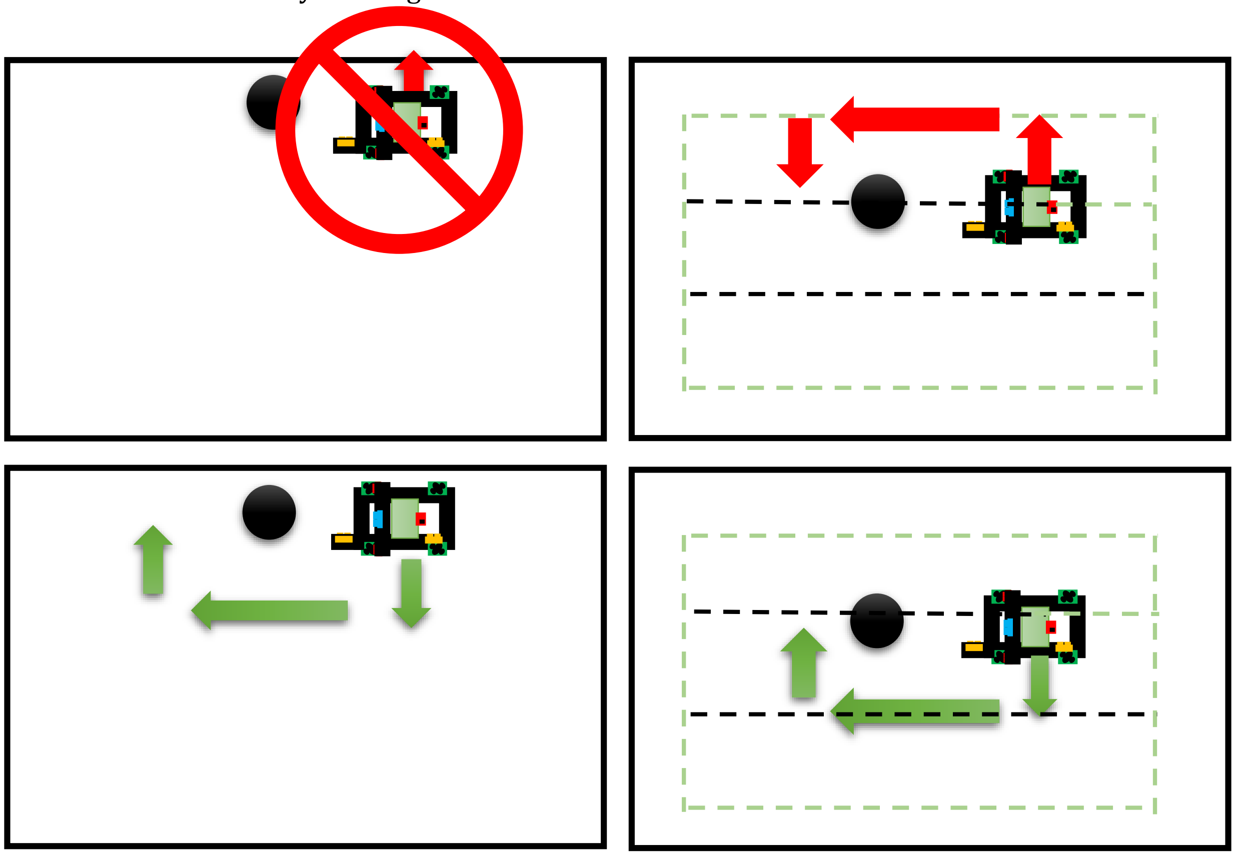


Figure 14: Obstacle avoidance process

The top left hand side diagram shows the robot not being able to strafe to the right of the obstacle because of the wall. The top right hand side diagram shows an unnecessary overlap of the path travelled by the robot. The green dotted lines show the area/path the robot has already covered, the black shows the path needed to be travelled over. A comparison of the two diagrams on the right shows that the robot has better area coverage when it strafes into the centre of the arena rather than strafing towards the wall. Hence, the final logic always made the robot strafe towards the centre of the arena when an obstacle was detected.

4.3.6.2. Obstacle

The robot keeps going forward until the side-rear sensor sees the obstacle block, in which case the robot will jump to the End Obstacle Strafe state. Once this happens, the robot will switch into the End Obstacle Strafe. The exception to this is if the robot sees the wall or an obstacle and thus cannot keep moving forward. If it senses a wall, the robot will jump to the Corner state and make a 90 degree turn. Whereas, if it senses another obstacle, then the robot will jump to the Begin Obstacle Strafe mode again and thus will strafe left towards the centre of the arena. It distinguishes between these two cases by checking the difference between the front sensor readings. First, a comparison is made between the front-left IR sensor and the sonar and a separate comparison is made with the front-right IR sensor and the sonar. If the difference between either of these is outside the tolerable range, then the logic indicates that an obstacle is present. However, if both comparisons indicate that they are within range of each other, then this logic indicates that a wall is present.

A fail safe measure for the Obstacle state is also implemented, in which the Obstacle state will timeout and the End Obstacle Strafe will be executed after a certain period of time.

4.3.6.3. End Obstacle Strafe

The primary purpose of this state is to strafe right towards the arena wall and get back to the robot's original path before it met the obstacle. The End Obstacle Strafe state will keep executing continuously until this condition has been met and once the robot is back on its original path, the robot will switch back to the Wall Follow state. The robot is able to decide when to stop strafing right (i.e.: End Obstacle Strafe) by using the long range IR sensors that point from the side of the robot towards to the side wall.

4.3.7. Stop

The stop state is entered into if the Lipo battery becomes too low or the robot completes the spiral. In this state the motors are disabled to prevent draining of the battery. This state is the last state in the Finite state machine and cannot lead go into another state.

4.4.SLAM

4.4.1. Mapping

By its nature SLAM is a complex problem as it requires multiple calculations dealing with multiple uncertain variables to calculate both a map and the location, whilst not having the other to rely on.

Due to the Arduino's limited processing power it was decided that the best approach would be to create a map of the environment off board to allow for accurate object detection and localisation. LabVIEW was chosen to create the map as it provided an existing an easy to use GUI. The main challenge for mapping is to fuse the data of all sensors to minimise the large amount of error present. This required an iterative approach to develop the mapping algorithm as no single sensor could be relied on to give accurate data at all times. Thus a mix of sensor readings and position inputs based on the controller output are used to determine the robot's location and generate the map.

4.4.2. Data Input

The filtered data and state information is passed to LabVIEW via the Bluetooth module or by inputting a text file to the program. The data passed to the mapping program is the timestamp, MPU heading reading, all distance sensor readings, and the controller state. The Arduino code is setup to only send data in the format required for mapping and does not send other strings so very little error checking is required from the mapping program. For the Bluetooth interface the data is read into the device via a serial port. Data is grouped by each timestamp and is read into the program and split into individual values to be processed. Grouping data is very important as measurement readings must take the position of the robot into account before they are included into the map.

In the current implementation data is gathered via Bluetooth and once it has all been collected it is passed to the mapping algorithm. This means the map is generated after the run has completed. An improvement on this would be to have the map being generated in real time as the robot is moving. Because the Bluetooth connection is only from the robot to the mapping program it was decided that real time mapping is not a priority for this application.

4.4.3. Algorithm

Looping through each row of given data, it gets broken down into the individual measurements from the sensors, and filtered once more to obtain usable distance and heading values. Extra filtering is required because the sensors occasionally output extreme values that are not accounted for by the on board digital filters. Sensors are also compared to past values to smooth out any random variation. When the robot first aligns itself with a wall, this becomes the origin of the map - (0,0). The MPU heading reading has a tendency to drift over the course of the run. While the impact of this is limited by the high pass digital filter implemented in the Arduino code it is still apparent in the output data. To account for this the mapping program takes an average of the first 10 MPU heading readings and uses this data to set

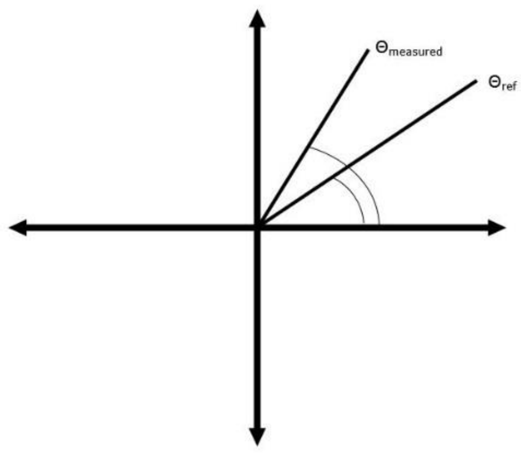


Figure 15: Calibration using the reference angle for Yaw

a coordinate system for the rest of the run. Because the robot only translates in the direction of, or perpendicular to the heading of the robot (as shown in the figure above), the effect of MPU drift is removed by assuming the robot will only travel perpendicular to or in line with the original heading. For example, for from the figure below Θ_{ref} is the initial heading reading and $\Theta_{measured}$ is the angle at the current time, the angle used in the mapping program will be Θ_{ref} as the measured angle is within ± 45 degrees of the reference.

To calculate the change in position of the robot the Kalman filter was investigated to track position change. A true Kalman filter was deemed unnecessary for the application of this project and a simplified version of this approach is implemented. The key difference from a classic Kalman filter implementation is that either the sensor readings or the estimated position, rather than a fusion of the two. The amount the robot has translated can be calculated in one of two ways dependant on the availability of reliable sensor data. The direction that the robot has moved is determined by the control output to the robot. When the robot is driving straight forwards or backwards the mapping algorithm assumes that it does not drift side to side as it is following the adjacent wall. The mapping algorithm checks if the front sensors are within range of the wall or obstacle in front. If the sensor readings are valid the distance the robot has travelled is the difference between the current sensor readings and the previous readings. By using the difference in current and past sensor readings rather than averaging the actual values it does not matter if there is an obstacle in front of the robot. When the forward sensor readings are not valid the algorithm estimates the speed of the robot based in the controller output and uses this estimation for distance travelled. The LabVIEW implementation of the position change estimation is given in appendix 9.4. Long range IR sensors were used for the side facing sensors, therefore they were always in range and the velocity estimate was never required.

The location of the robot in the map is then adjusted through the distance the robot has moved, either by using an estimate based on the controller output, or calculating the difference in the sensor readings if these are within range. Displacement and heading data is combined to calculate the change in X and Y distances of the robot, giving the new location. Path of the robot is stored in an array of coordinates is stored as an array that is then outputted to the user.

Obstacles and wall locations are calculated based on distance from the current robot position. As the front right IR sensor often measures the distance to the wall to the side instead of the wall in front, it is not used to plot wall and obstacle positions. All other range sensors are used to plot obstacles and walls, as long as they are within their valid range. To calculate where the measured point is relative to the current robot position, the measurement is transformed into a Cartesian coordinate in the robot's coordinate frame. The point relative to the robot is then transformed in the world frame. The LabVIEW algorithm used to transform sensor data readings is shown appendix 9.5: Yaw estimation: LabVIEW implementation. Obstacle and wall locations are stored in as coordinates in an array that is then displayed on the front panel of the program.

4.4.4. Interface

The two arrays representing robot path and wall/obstacle position are fed into a XY graph which plots each dataset in different colours. The front panel of the mapping program has been kept simple to ensure only the required information is displayed to the user. As shown in section 6.1, the user can select between inputting data directly from the robot via Bluetooth, or import data from a text file. The front panel has an option for selecting the serial port number. The map is displayed on a 2D plane with red dots to show the robot's path and white dots to show the detected walls and obstacles.

Due to the current communication setup on board the robot it is not possible to establish a two-way connection. This means that the robot cannot use the information generated by the map to optimise path planning and obstacle avoidance. One possible solution that was explored, was to run a simplified version of the slam algorithm on board the Arduino. This option was not pursued because the computational speed and memory size on board the Arduino was deemed insufficient to produce a usable map. Another potential solution would be to communicate map information back to the Arduino by some form of wireless communication. This solution was not implemented because of the extra hardware that is required for this approach.

5. Features

5.1. Obstacle Detection

From testing the front mounted sensors, it was determined that they could only be relied on to detect obstacles consistently within 30mm. This was because the sensor readings would diverge after this point and not be comparable. The algorithm worked by assigning three Boolean variables to one if their respective sensor was greater than one or zero if less than one. With three sensors this lead to eight possibilities of sensor combinations. These combinations are seen below.

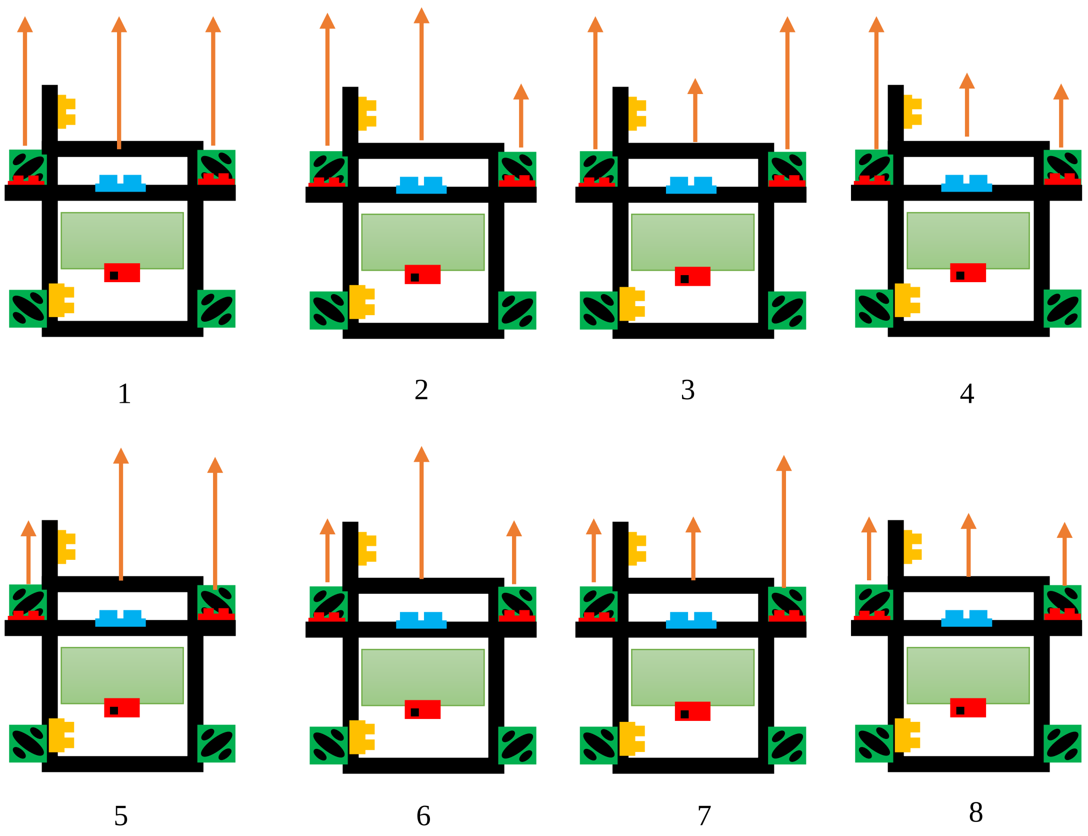


Figure 16: Possible scenarios of the sensor views

The above possible scenarios where the sensors can see longer or shorter than 250mm are shown. Cases 1 and 8 show a wall and so the output value should be zero. Case 6 should not happen in the expected scenario as there should only be one obstacle visible. All other cases indicate obstacles in front of the robot. These cases are seen in the table below. A is zero if the left sensor can see longer than 250mm, and one if it can see less. B represents the same for the mid sensor and C the same for the right sensor. Obstacle indicates if each scenario could indicate the presence of an obstacle.

Case	A	B	C	Obstacle
1	0	0	0	0
2	0	0	1	1
3	0	1	0	1
4	0	1	1	1
5	1	0	0	1
6	1	0	1	0
7	1	1	0	1
8	1	1	1	0

Table 1: Truth table for obstacle avoidance function.

The output of the above table was synthesised into the logic function below.

$$\overline{AB}C + \overline{A}B\overline{C} + \overline{A}BC + A\overline{BC} + AB\overline{C} = 1$$

$$\overline{A}C + \overline{A}B + A\overline{C} = 1$$

This code implementation can be seen in appendix 9.1

5.2.Going straight: Controller design

In order for the robot to go straight a function was created for the robot to go parallel to the wall. A proportional controller was created based on taking the difference in sensor readings from the side facing IR sensors (the heading value). The difference was multiplied by a suitable gain (selected for responsiveness at different speeds) and applied to the left and right side motors, in different directions to keep the robot straight while moving forward. This adjusted the yaw of the robot while keeping it moving forward.

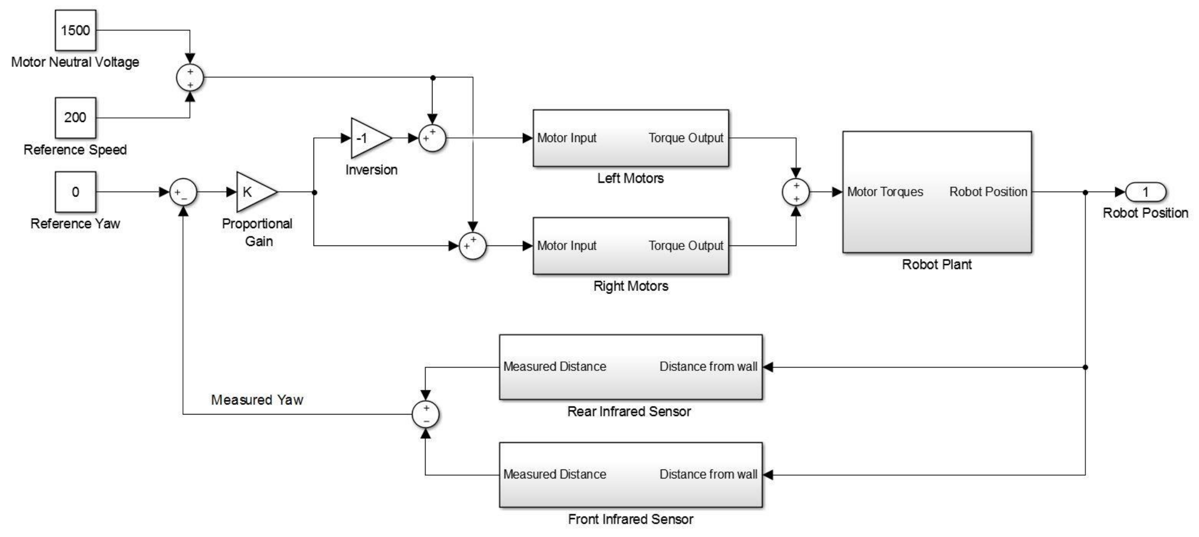


Figure17: Control diagram of parallel wall follower

The effective controller diagram can be seen below. The code implementation of the control can be seen in appendix 9.2.

5.3.Filtering

Because the IR sensor responses were affected by environmental conditions spikes in the readings were observed. Seen below is a sample of an output from the IR sensor while it was held at a constant distance from a wall. Noisy spikes can be observed and the readings fluctuate ±50mm around the mean value.

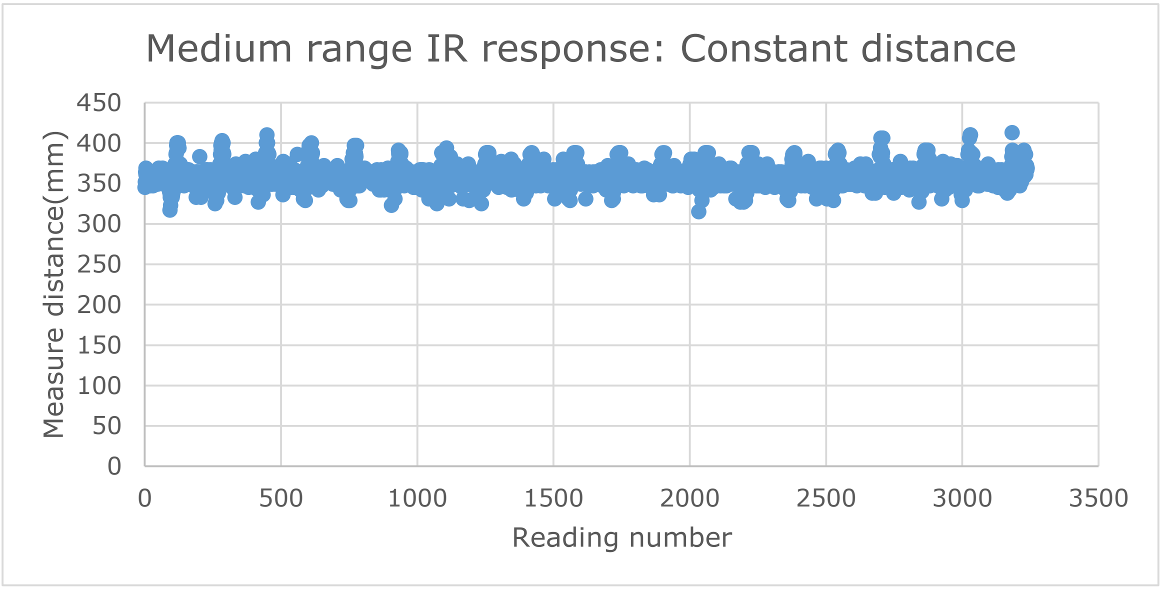


Figure 18: IR sensor response at constant distance

This noise had a significant impact on the robot performance and so some filtering was needed. Because some values had been observed to go negative and larger spikes than those in Figure were observed a moving average filter was deemed unsuitable. Any time of average filter would be able to be strongly influenced by large outlier values and would have proved detrimental to the performance. For this reason, a median filter with a width of 5 was implemented (see appendix section 9.3). This involved reading in 5 values for each sensor, ordering them and taking the middle value. This ensured that outlier values would not be affect the result, and the final result would be the most accurate. The value of 5 was determined experimentally and reflected the fact that when testing there would be at most two outliers on each reading.

The case of the sonar filtering was slightly different. It was found to be accurate up to 800mm but would experience serious fluctuations beyond this point. This can be seen in the figure below where the robot is first placed at 750mm from a wall and then moved beyond this threshold to 950mm. As can be seen in the figure below, there are major fluctuations at around the 5000th and 7000th readings. Some of these were filtered out by the above median filter and some were filtered out by error checks which rejected zero values or negative values, some of which were observed between the 7000th and 8000th reading.

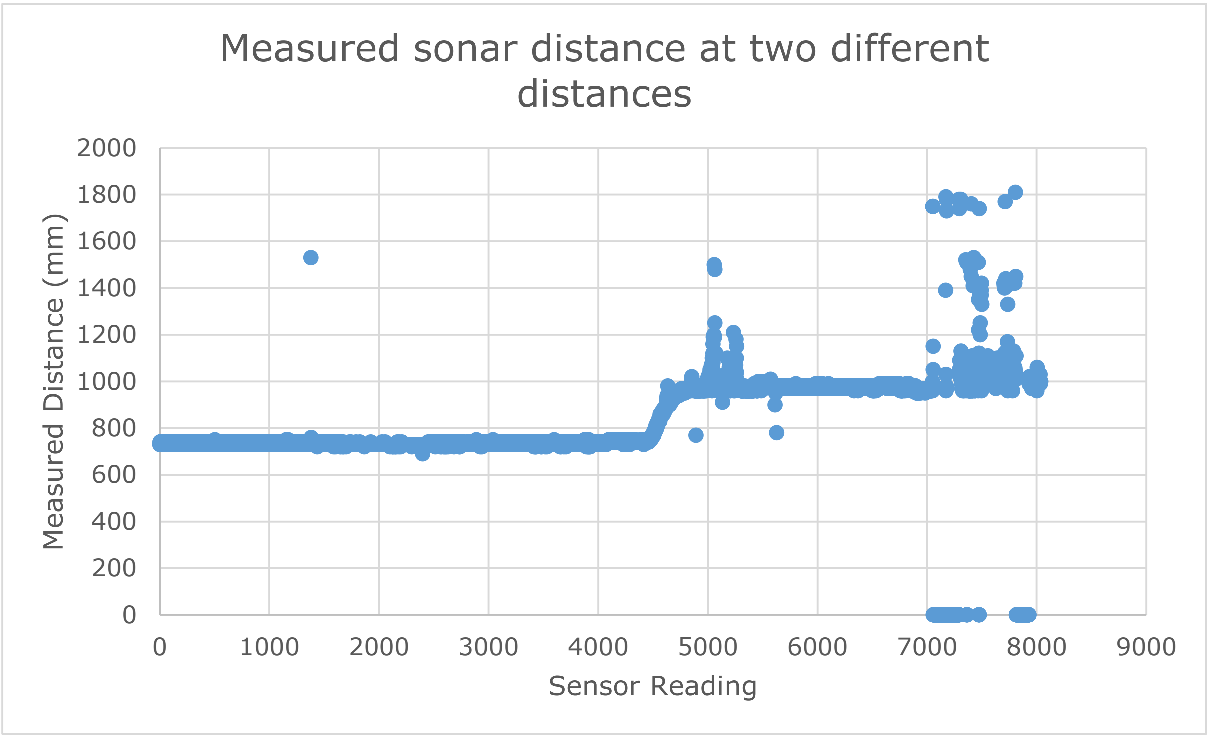


Figure19: Sonar response at different distances

While some of this noise was likely to be a function of the sampling frequency, there were other reasons which could not be identified, and so it was decided to just use the sonar for obstacle detection at short range and acknowledge that the data which was observed beyond 800mm for mapping would be less reliable.

5.1.Code Optimisation

Out of the overall code, the longest segment of code in terms of execution time would most likely be reading the sensor data because the robot has to wait for the readings to be relayed to the Atmega8 processor. Initially the code was reading sensor data and then sending it via Bluetooth in the main execution loop and the same readings were then collected again (re-read again) in the state functions (for obstacle avoidance, wall follow and corner states). This meant that the code was very inefficient and was consuming precious run time execution which may have also been the cause of the robot occasionally not detecting the obstacle and therefore going right into it. In order to solve this problem, the sensor data that was read for the purpose of sending it via Bluetooth module for mapping was also used for the on-board decisions of path planning. This meant that per execution of the main Arduino loop, rather than reading a sensor 10 times or even 20 times, the sensors were only read 5 times. This improvement dramatically improved the consistency of the obstacle detection and meant that the robot had more data to send to LabVIEW for the mapping of the arena. The total execution time of the main Arduino loop was reduced from 100ms to 50ms, which is a considerable performance improvement is given the velocity of the robot and the size of the obstacle course.

Another improvement that was made was the elimination of the obstacle buffer idea and using a counter instead. The obstacle buffer had to be constantly down shifted to update and reflect the latest sensor reading. Since this was done in the wall follow state, implementing the counter instead of a buffer substantially improved the run time of the code.

6. Testing and Results

6.1. Mapping Results

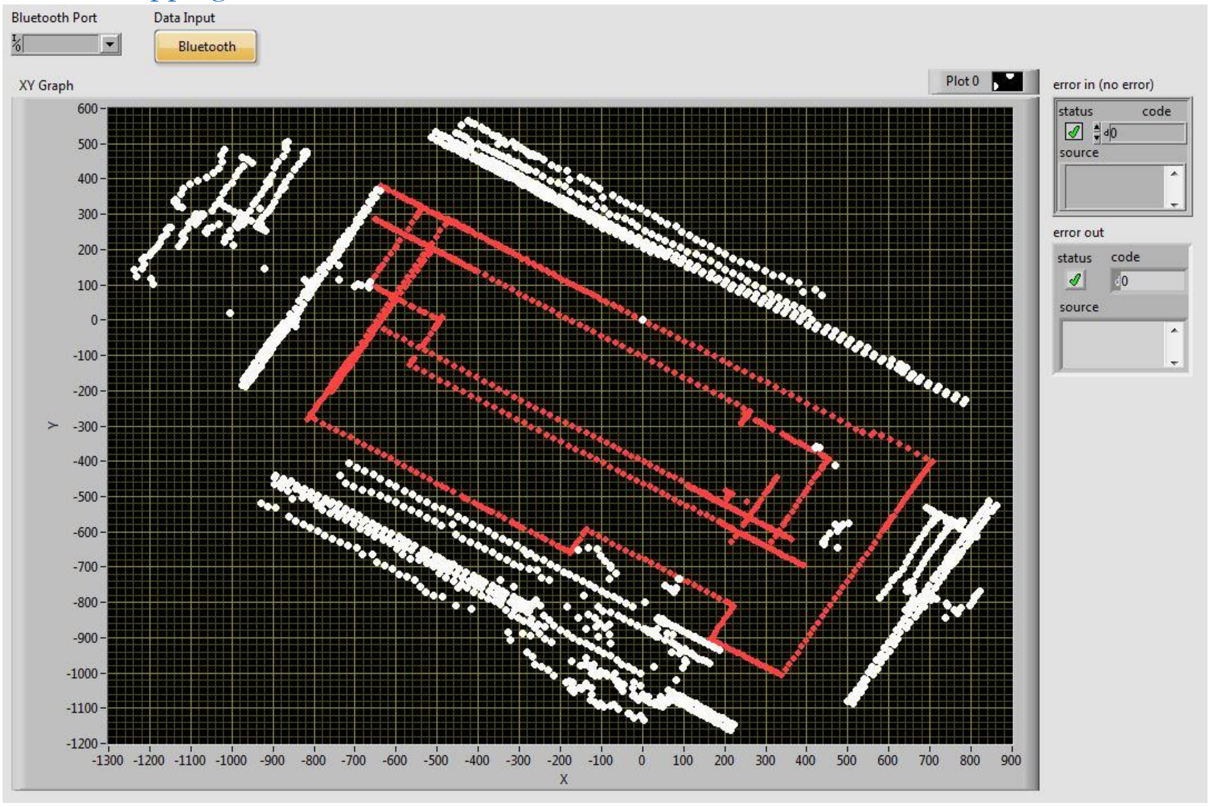


Figure20: Map of robot path from LabVIEW

The image below shows the true path taken by the robot in the test trial. Compared to the figure above it is clear that the mapping algorithm produces an accurate approximation of both the environment and the path the robot took. Error in the map robot position increases over the course of the run due to the large errors produced by the sensors used and because the SLAM algorithm makes no assumptions about the mapped environment. One improvement would be to implement a probabilistic map that estimates the probability of an obstacle being at a given location based on the number of sensor readings that have measured an obstacle in that location. Thus removing outliers as well as the effect of measurements with very large error at the end of the mapping run.

All three obstacles in the environment are visible in the generated map. One issue in obstacle identification is the presence of some outliers within the walled area. This is not a major issue as the outliers only contain a small number of data points which are caused by the cumulative error across the run. The walls of the arena are also discernible. The top wall is not obscured by obstacles for the majority of the run and is therefore the most clearly defined. Due to the evasive manoeuvres required to avoid the obstacles the other walls show some variability in the map estimation of their position.

The map below demonstrates that the spiral path very efficiently covers the vast majority of the arena area. The spiral pattern ensures that the robot does not repeat much of the map with the exception of areas that the robot moves close to obstructions to determine if they are walls or obstacles.

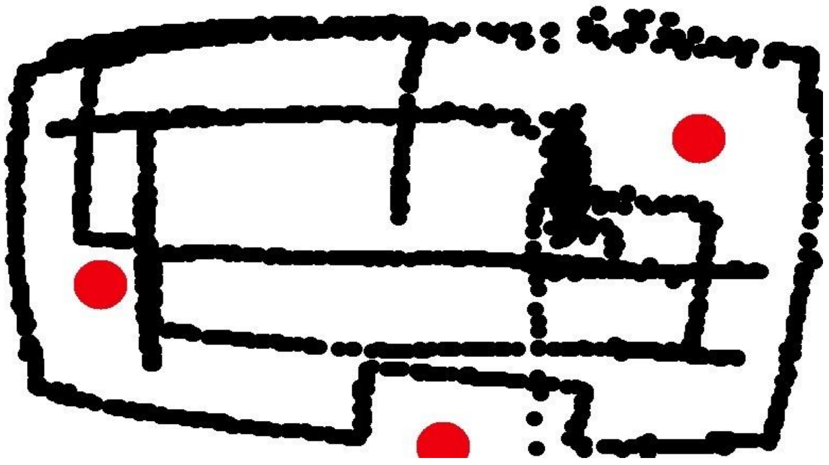


Figure 21: Measured path of robot

7. Discussion

7.1. Future Improvements

In case of incorrect sensor readings in the Begin Obstacle Strafe and Obstacle mode a fail-safe timeout was implemented. One main issue that was faced with the timeout was that an int in the Arduino is only able to hold a value between -32,768 to 32,767. This was a major issue at first because the Arduino only returns a value in milli-seconds which indicates the time since the robot was first powered on (i.e.: Atmega run time) and when we tried to store this value (when after 32 seconds had passed) in an int variable, the program crashed. This problem was solved by dividing the run time by 10. Such a solution is fine for our obstacle course which we completed in approximately 1 minute and 30 seconds. However, if the robot was required to run over a bigger obstacle course (a course that took longer than 5 mins), then this same problem would be faced again. Therefore, a more robust solution of perhaps using a timestamp or relying on a rotating sodar could be used as an improvement for longer courses.

Furthermore, the timeout for the fail-safe was specifically designed for any obstacle with dimensions similar to the rice-cracker cylinder (~100mm diameter). If a different obstacle object are used, the timeout condition could be easily modified to accommodate any new dimension. For example, for larger obstacles the timeout could be increased and for shorter obstacles a shorter timeout could be used. The use of fuzzy logic in this situation would be ideal. However, a better solution would be to rotate the sonar to verify the IR sensor data or even to make the robot temporarily pause over a few seconds and collect more data over a longer period of time thus being able to filter out sustained outliers.

8. Conclusion

The project required a robot to navigate and map a 2 X 1.2 metre arena with obstacles and walls. The three main criteria to assess this project was area coverage, obstacle avoidance and time to complete the course. Our approach to maximise the area coverage was to use a spiral algorithm that made the robot work in towards the centre of the arena with the guidance of two long range IR sensors on the side. Obstacle and wall collision avoidance was implemented using distance readings from the sonar and the two medium range IR sensors situated on the front of the robot. The time goal was achieved by increasing the robot speed by 100 rpm and also by maximising the code to achieve faster update rates of sensor readings.

One of the most effective points about our project was the hardware layout. Testing and plotting the sensors gave us very good understanding of what each sensor was capable of and how to optimise their performance (get rid of dead zone, read further) via their placement on the robot. For example, the IR sensors were capable of accurately reading distances further than what the data specification sheet specified. Discussing the implications of each of the sensor's strengths and weakness, enabled us to quickly limit the number of configurations possible and allowed us to finalise the best sensor arrangement.

A minimalist approach was taken for our sensor accessories and this proved to be very effective. For example, unlike other groups we did not spend time on 3D printing housing parts for our sonar and the IR sensors but rather spent time on figuring out which sensors were the best in which position and applying very good filters in software to get the best possible readings. Although initially we intended to use the servo motor to turn the sonar for obstacle detection, we quickly realised that this added to our time to complete the course. Therefore, we eliminated the need to use the servo motors.

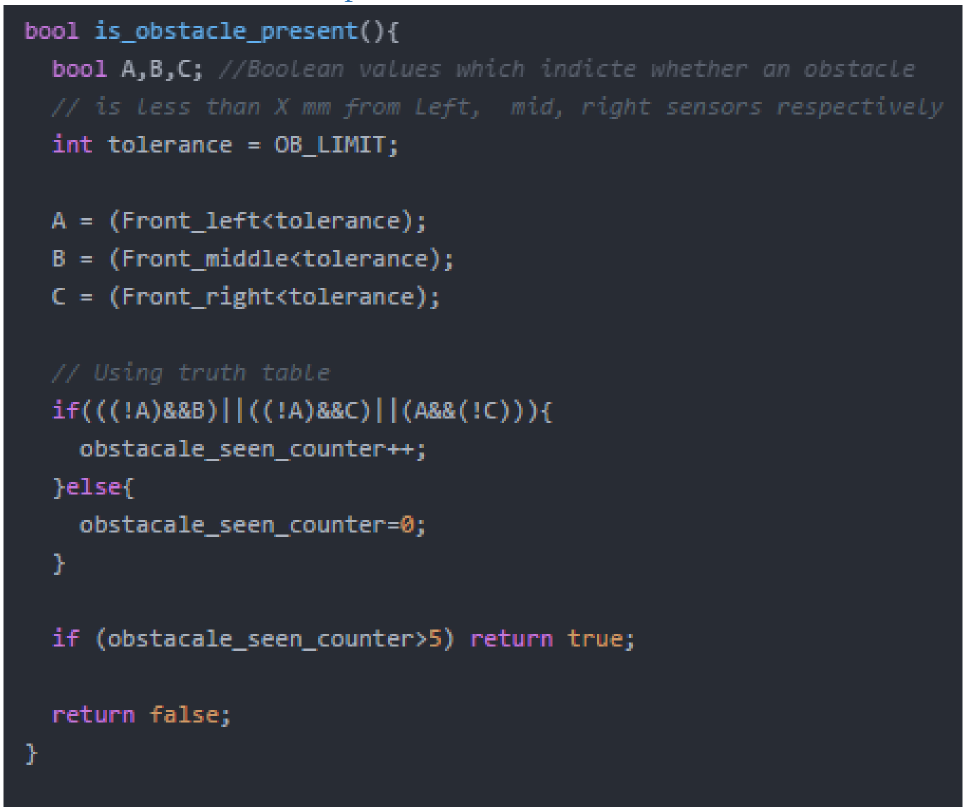
The robot did not miss any areas and there were very few double ups of path making the area coverage not only very high but also very efficient. The robot managed to avoid all obstacles but required some fine tuning in order to achieve this. The robot was also very quick to complete the course and was able to complete it in approximately 1 minute and 30 seconds. The robot is definitely capable of handling the course at a much larger speed than the current finalised course speed. The current code for Startup, Corner and Wall Follow states has been tested and is proven to be robust enough to able to handle a speed at least twice the finalised speed for the spiral path (without obstacles). However, the speed limitation arises because of obstacle detection code which relies on taking at least 30 positive readings to confirm that an obstacle is present. If some optimisation can be done in this area, our robot would be able to complete the course in a much faster time. On the other hand, a simpler change to reduce the time would be to eliminate the final turn which merely double checks that the course is complete.

The mapping for our project was very accurate. However, it required foreknowledge of the robots speed and was able to do it after the robot completed the course using a txt file. Attempts were made to determine the robot's speed dynamically by differentiating the acceleration data from the MPU but was not very reliable. Therefore, the only suitable solution to this problem would be to use a tachometer to enable mapping using dynamic speed.

Overall, the SLAM project was a very enjoyable project that required a clever design of the sensor layout and software states in order to be simple and effective. Our group met both these criteria very well, which ensured that the three main objectives of the project (area coverage, obstacle avoidance, time to complete course) were done very well.

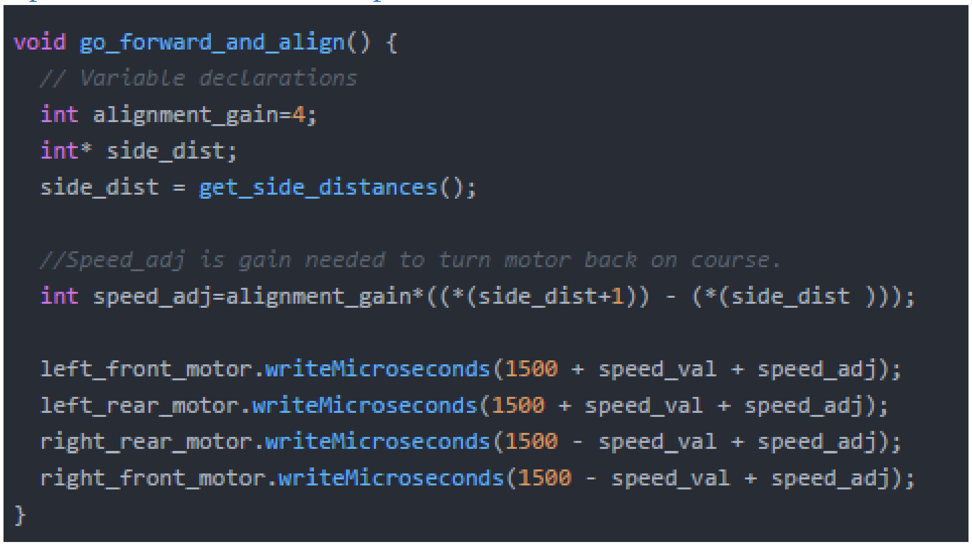
9. Appendix

9.1. Obstacle detection code implementation



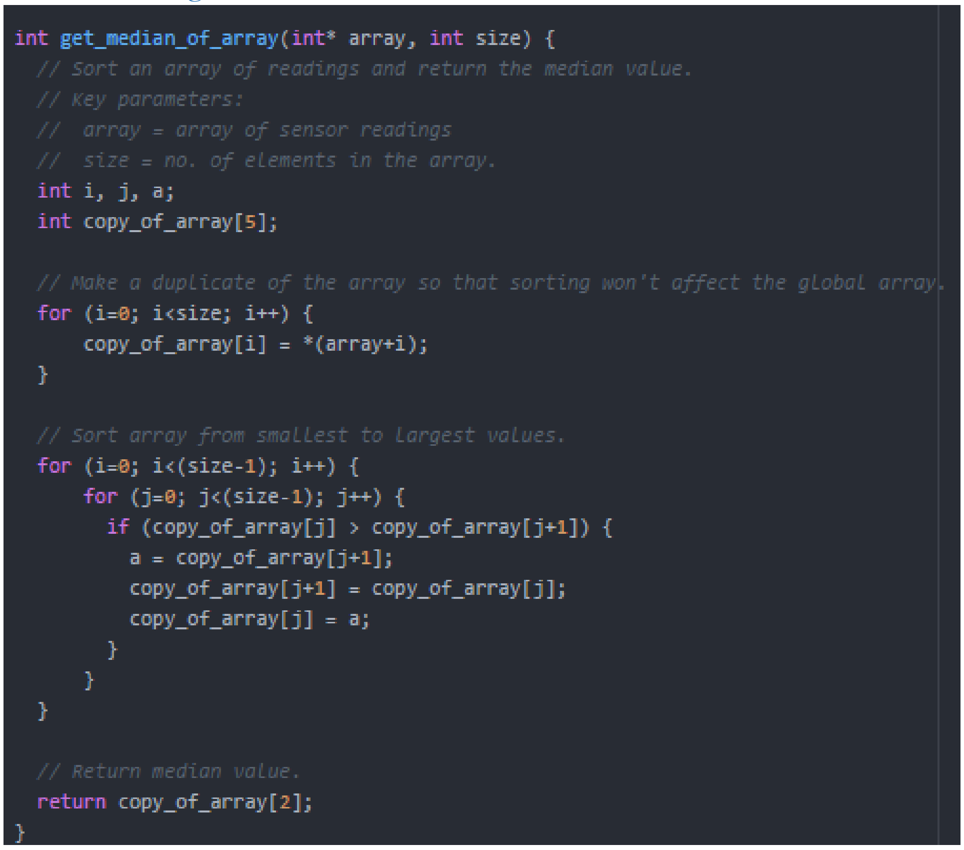
```
bool is_obstacle_present(){
  bool A,B,C; //Boolean values which indicte whether an obstacle
  // is less than X mm from Left,  mid, right sensors respectively
  int tolerance = OB_LIMIT;

  A = (Front_left<tolerance);
  B = (Front_middle<tolerance);
  C = (Front_right<tolerance);

  // Using truth table
  if(((!A)&&B)||((!A)&&C)||(A&&(!C))){
    obstacale_seen_counter++;
  }else{
    obstacale_seen_counter=0;
  }

  if (obstacale_seen_counter>5) return true;

  return false;
}
```

9.2. Proportional control code implementation



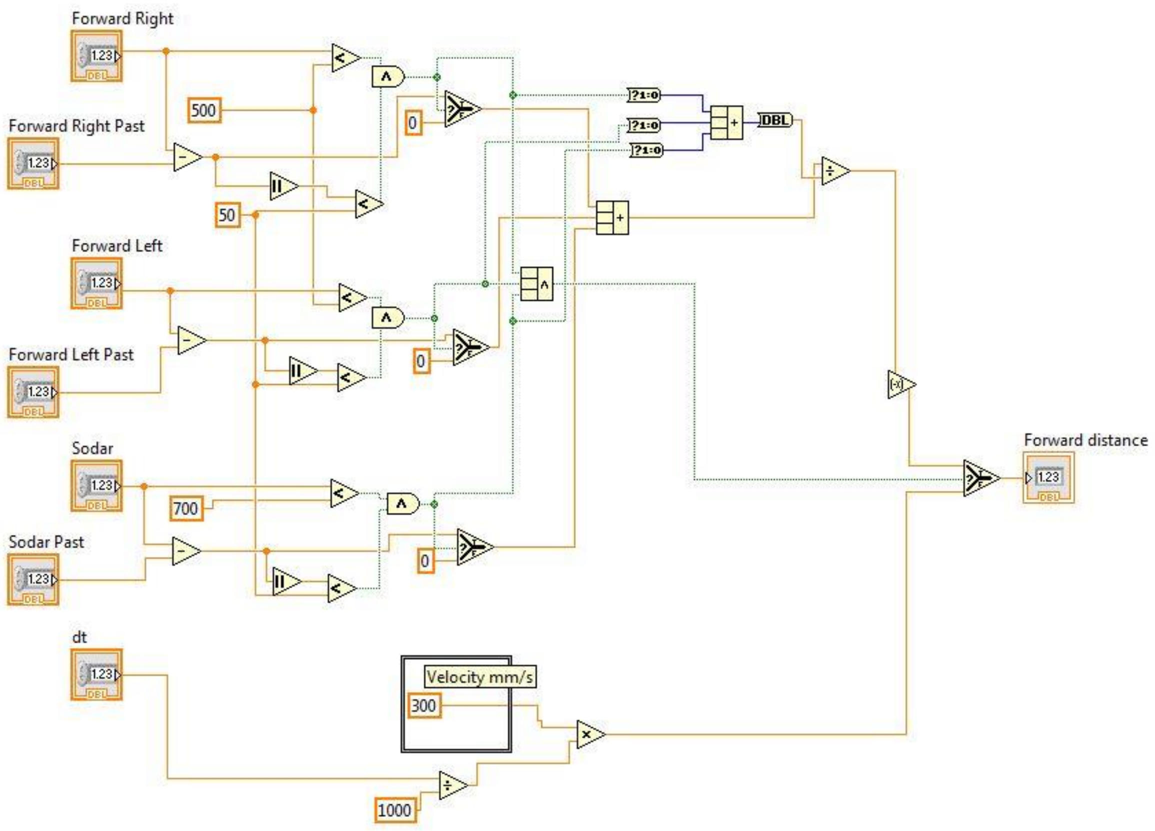
```
void go_forward_and_align() {
  // Variable declarations
  int alignment_gain=4;
  int* side_dist;
  side_dist = get_side_distances();

  //Speed_adj is gain needed to turn motor back on course.
  int speed_adj=alignment_gain*((*(side_dist+1)) - (*(side_dist )));

  left_front_motor.writeMicroseconds(1500 + speed_val + speed_adj);
  left_rear_motor.writeMicroseconds(1500 + speed_val + speed_adj);
  right_rear_motor.writeMicroseconds(1500 - speed_val + speed_adj);
  right_front_motor.writeMicroseconds(1500 - speed_val + speed_adj);
}
```

9.3. Median filter algorithm for filter

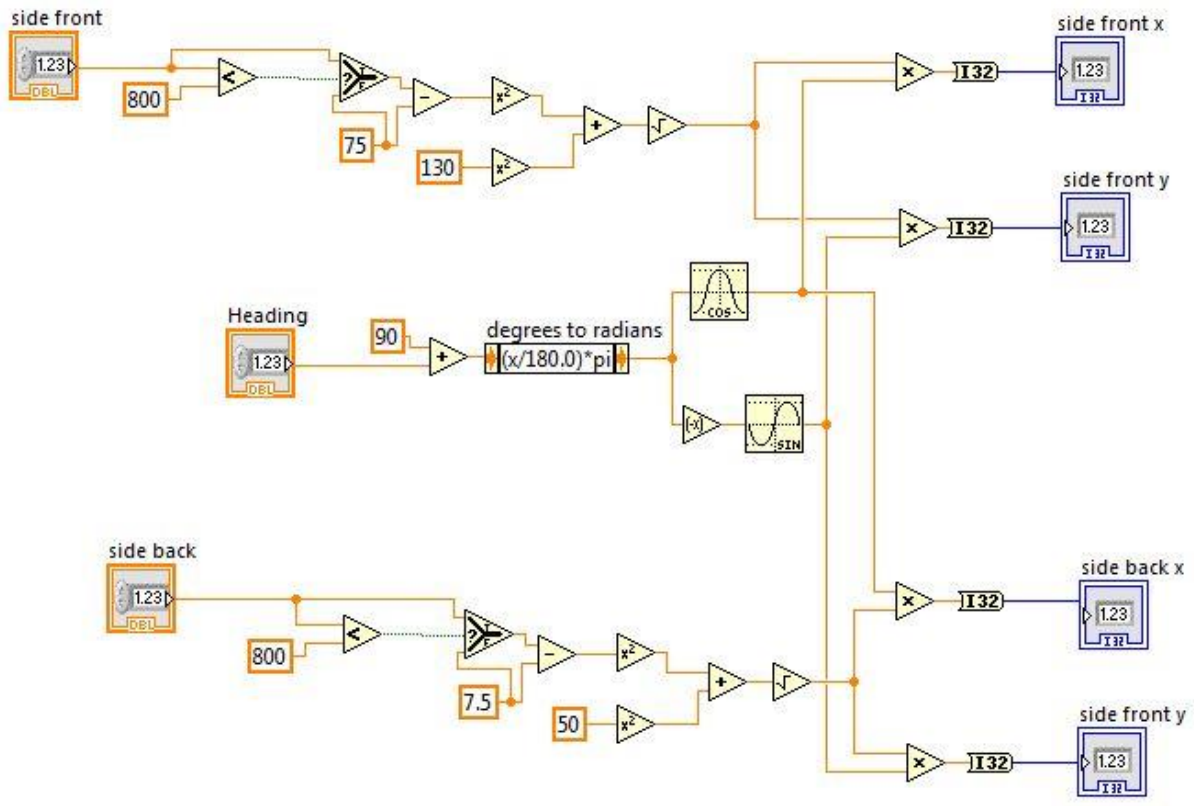


```
int get_median_of_array(int* array, int size) {
  // Sort an array of readings and return the median value.
  // Key parameters:
  //  array = array of sensor readings
  //  size = no. of elements in the array.
  int i, j, a;
  int copy_of_array[5];

  // Make a duplicate of the array so that sorting won't affect the global array.
  for (i=0; i<size; i++) {
      copy_of_array[i] = *(array+i);
  }

  // Sort array from smallest to largest values.
  for (i=0; i<(size-1); i++) {
      for (j=0; j<(size-1); j++) {
        if (copy_of_array[j] > copy_of_array[j+1]) {
          a = copy_of_array[j+1];
          copy_of_array[j+1] = copy_of_array[j];
          copy_of_array[j] = a;
        }
      }
  }

  // Return median value.
  return copy_of_array[2];
}
```

9.4.Position change estimation: LabVIEW implementation



9.5.Yaw estimation: LabVIEW implementation


side front
1.23
DBL
800
75
130
side front x
side front y
Heading
90
degrees to radians
(x/180.0)*pi
side back
800
7.5
50
side back x
side front y
I32