VISION-BASED ROBOT INDOOR NAVIGATION

TEO ZHIN HANG

UNIVERSITI TUNKU ABDUL RAHMAN

VISION-BASED ROBOT INDOOR NAVIGATION

TEO ZHIN HANG

A project report submitted in partial fulfilment of the requirements for the award of Bachelor of Engineering (Honours) Mechatronics Engineering

Lee Kong Chian Faculty of Engineering and Science
Universiti Tunku Abdul Rahman

September 2022

DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at UTAR or other institutions.

Signature	:	[illegible]
Name	:	Teo Zhin Hang
ID No.	:	17UEB00348
Date	:	12/9/2022

APPROVAL FOR SUBMISSION

I certify that this project report entitled **"VISION-BASED ROBOT INDOOR NAVIGATION"** was prepared by **TEO ZHIN HANG** has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Engineering (Honours) Mechatronics Engineering at Universiti Tunku Abdul Rahman.

Approved by,

Signature	:	[illegible]
Supervisor	:	Lee Jer Vui
Date	:	11/9/22

Signature	:	[illegible]
Co-Supervisor	:	Chai Tong Yuen
Date	:	11/9/22

ACKNOWLEDGEMENTS

I would like to thank everyone who contributed to the successful completion of this project. I want to express my gratitude to my research supervisor, Ts Dr Lee Jer Vui, for his invaluable advice, guidance and enormous patience throughout the research development.

In addition, I would also like to express my gratitude to my loving parents and friends who helped and encouraged me. Special thanks to Foo Jing Tong and Teo Zhin Lerk for helping in building the test environment and testing prototype efficiency.

ABSTRACT

This report discusses a vision-based indoor navigation robot training system. The need for manufacturing workers has been increasing throughout the years. Many labours are required in businesses to transfer items and components all around. Many companies still use manual labour for part delivery within the factory, where the completion progress and time depend highly on the workers. With that said, any possible threat may stop the business's operation, making a loss to the company. Indoor navigation robots can close the gap in manual labour. This project aims to develop a vision-based navigation robot using OpenCV and C++ programming language in an indoor environment. The objective of this project is to design, develop and simulate programming code to perform image processing and path planning, integrate programming code with a microcontroller wirelessly, and test and evaluate the performance of the navigation robot. The computer and ESP32 board are the central processing unit for this project to execute path planning and motor command analysis. Node-Red links both processing units together. An algorithm is developed in the computer to achieve image processing, user input, path planning, simulation, and writing of output files. A different algorithm is created in the microcontroller to derive and perform the data delivered from the computer. A navigation robot is built to test the workability and efficiency of the algorithms. In general, the algorithm can provide the nearest path to navigate around the environment without manual assistance. The user will only have to give the program relevant coordinate information and running mode. To conclude, a vision-based indoor navigation robot training system is effectively established. Implementing this system could effectively help increase product delivery within the environment. This system supports one end coordinate or multiple saved coordinates, where a supply chain operation structure can be implemented above it to administer all the operations fully.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS / ABBREVIATIONS

A	ampere
B	byte
c	centi
g	gram
G	giga
Hz	hertz
k	kilo
m	milli
m	meter
M	mega
V	voltage
°	degree
μ	micro
2WD	2-Wheel Drive
ADC	Analogue to Digital Converters
AGV	Automated Guided Vehicles
AMR	Autonomous Mobile Robots
BFS	Breadth-First Search
DC	Direct Current
FIFO	First In, First Out
GPIO	General-Purpose Input/Output
HPA*	Hierarchical Path-finding A*
IDE	Integrated Development Environment
IIC	Inter-Integrated Circuit
IoT	Internet of Things
IP	Internet Protocol
I2C	Inter-Integrated Circuit

LPA*	Lifelong Planning A*
MEMS	Micro-electromechanical systems
MQTT	MQ Telemetry Transport
OpenCV	Open-Source Computer Vision Library
OpenGL	Open Graphics Library
PWM	Pulse-Width Modulation
RPM	Revolutions Per Minute
SCL	Serial Clock Pin
SDL	Serial Data Pin
SDRAM	Synchronous dynamic random-access memory
SRAM	Static random-access memory
USB	Universal Serial Bus
VCC	Voltage Common Collector
VIN	Voltage In

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Introduction to Vision-Based Robot Indoor Navigation

AGVs are widely implemented in industries to increase productivity and manufacturing flexibility (Lydon, 2018). Improvements in navigation techniques boosted the usage of AGV in sectors. AGVs are getting more common in the market and come in different sizes. They can accomplish more tasks in various sectors, but there are a few main applications of the vehicles. In factories, AGVs can be used for parts delivery, product, and pallet handling. Manual labour can be reduced as AGV can easily accomplish these repetitive tasks. Product handling is crucial for specific industries as the materials might be fragile or easily damaged. AGV will be a better transportation choice than manual labour in this case. The automated vehicle's precision, movement, vibration, and acceleration can be computerised to ensure maximum safety during product handling.

Figure 1.1: Implementation of robots in factories (Wise, 2022)

This study is to design a vision-based navigation robot in an environment. The autonomous robot can make full use of the environment. AMR is a more robust selection than AGV. AGV navigates through an environment in predetermined paths where AMR can achieve free movement and real-time path planning (Oitzman, 2021). AMR is the successor of AGV. AMRs can perform tasks that are possible by AGVs and outperform AGVs in navigation. They can navigate through their environment freely and navigate a path between points. Compared to AGVs that will stop if an obstacle is detected, AMRs can navigate around obstacles to reach the end position. AGVs operate in highly controlled environments where all the procedures are nicely defined in different places. AMRs are more flexible and are meant to use in unstructured environments. AMRs are easier to implement in ready production and manufacturing lines without environmental changes.

Different types of navigation robots are developed by researchers where the principles of the techniques are nearly identical. Vision sensors are used to capture images. The computer runs the algorithm for image processing and path planning. Microprocessors execute the paths found by the programming codes. This project aims to build a vision-based navigation robot in an indoor environment with the assistance of a camera. The robot can navigate around the environment freely.

1.2 Importance of the study

Navigation robots are used extensively in manufacturing, logistics, and healthcare lines. The navigation robot must be able to make decisions based on their situations. The robots' decision-making is from the algorithms executed to search a path for the given environment. Many companies face the issue of having to relocate and redesign the environment for automated guided vehicle implementation, increasing cost and space usage. It is still important to introduce a mobile robot that can achieve the given task in an uncontrolled environment with this condition.

1.3 Problem Statement

Many labours are needed in industries to transport items and parts around. Implementing a mobile robot can close the gap in manual labour (Wise, 2022). Transportation done by mobile robots is much more efficient than manual labour, giving a supply chain operation structure. Although introducing mobile robots might be costly initially, it could reduce possible losses to the company. Based on Wise (2022), 2.1 million manufacturing jobs in the United States of America could go unfilled by 2030, costing the manufacturers a total of one trillion dollars in 2030 itself. With a vision-based navigation robot, manufacturers and companies can keep their businesses running by closing the manual labour gaps without having to rebuild the layout of the entire factory.

1.4 Aims and Objectives

This project aims to develop a vision-based navigation robot using OpenCV and C++ programming language in an indoor environment. With the assistance of the camera, the robot can navigate around the environment with the target given by the user. This project has the following objectives:

1. Design, develop and simulate programming code to perform image processing and path planning.
2. Integration of programming code with microcontroller wirelessly.
3. Testing and evaluating the performance of the autonomous robot.

1.5 Scope and Limitations of the Study

This research intends to design a vision-based navigation robot for usage in an indoor environment. In the first part of the project, the programming code of image processing and search algorithms are developed and tested. In the second part, the outcome of the programming is integrated into a microcontroller wirelessly for calibration and testing purposes.

The limitations of this project are mainly from the hardware. While the connection between the computer and microcontroller is established using a Wi-Fi network, the signal around the ESP32 must be strong enough for stable data transmission. The following limitation is the flatness of the test field. As the test field is made of Mahjong paper, some spots might have small folds that might affect the movement of the mobile robot.

1.6 Overview of Project

Software and hardware are both implemented in this project. Software functions to code, simulate and integrate. Hardware tests the workability and efficiency of the software in the environment. The project can be divided into four stages: research, development of software, integration with hardware, and analysis of results.

This report is divided into five chapters. In the first chapter, the introduction of a vision-based navigation robot is discussed along with the importance, problem statement, aim and objectives, and the project's overview. The following chapter is the literature review, where other researchers' ideas on similar topics are reviewed, which provides a better idea of the approach. Chapter 3 is the methodology, where the project approach is discussed alongside the software and hardware selected. The distribution of work is included in this chapter. The next chapter discusses the autonomous robot's coding, architecture, and data analysis. The last chapter concludes the findings of the entire project. Limitations and future improvements are included at the same time.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

A vision-based indoor navigation robot has been a rising trend throughout the years, and different research tested various methods to prove and improve the efficiency of autonomous robots. Other approaches are proposed, but navigation robots always follow a hierarchy, as shown in figure 2.1.

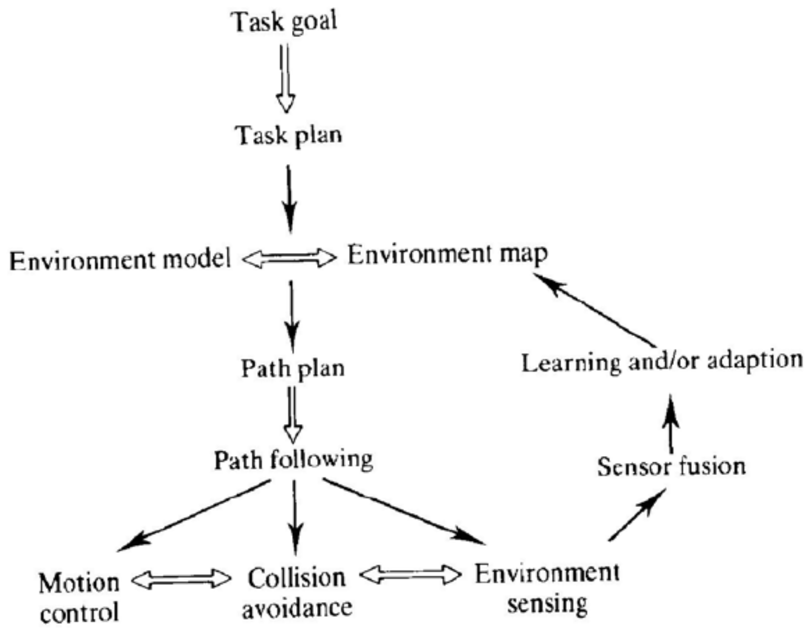


Figure 2.1: Mobile robot control hierarchy (Karastoyanov and Zahariev, 2004)

Taking the mobile robot control hierarchy as a benchmark for vision-based indoor navigation, the robot must first have a set task goal and plan (Singhata, 2021). A webcam captures the image of the environment and transmits it to the computer. The computer translates image information into machine code, and path planning algorithms are executed to find paths for the autonomous robot. The algorithm’s outcome will be transferred to the microcontroller for wheel control, from the starting position to the ending position. The methodology of the vision-based navigation robot is comparable; hence the difference is the image acquisition and processing method, type of search algorithm used, and types of microcontrollers selected.

2.2 Image Acquisition and Processing

In the research by Singhata (2021), a web camera is mounted on a stand and installed above the map. The camera is connected to the computer, where image processing will be done. The web camera selected matches the specification and provides a clear image at a certain height. The image captured is sent to the computer for image processing. In this application, the information of the field is not recorded as the test environment is fixed. The image processing is done to obtain the autonomous robot's current position. Three steps are done to identify the robot’s position, object extraction, thresholding, and template matching.

According to Zidane & Ibrahim (2018), the web camera is set above the test field, covering the entire test environment, as illustrated in figure 2.2. All the details in the test environment are acquired. The robot’s initial position, target position, and obstacles are considered. The captured image is sent into MATLAB software, where the image processing toolbox is used. The sectoring of the images must be chosen correctly to reduce processing time and increase the optimality of results. The image is converted into binary, where 0 represents the obstacles, and 1 signifies the open paths. Morphological operation for removing noises is done to prevent false positives from the image, which might affect the result of the autonomous robot. Another clear image is produced after implementing the morphological operation. The basic algorithm, which is the pathfinding algorithm, can be implemented.

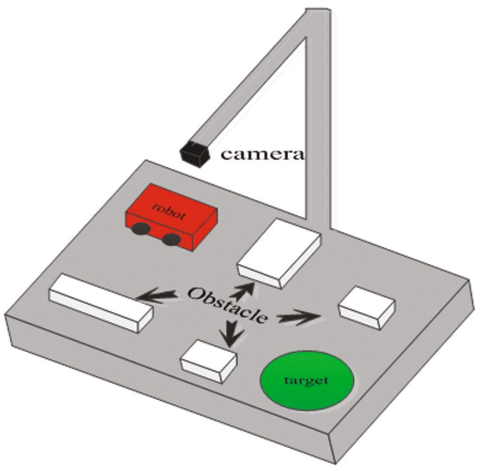


Figure 2.2: Camera setup and Test environment (Zidane and Ibrahim, 2018)

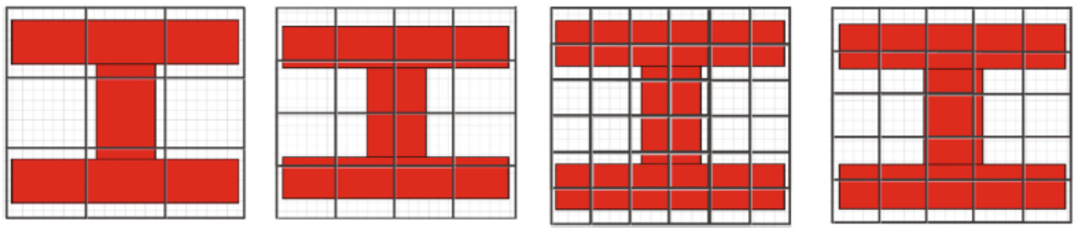


Figure 2.3: Sectoring of Images (Zidane and Ibrahim, 2018)

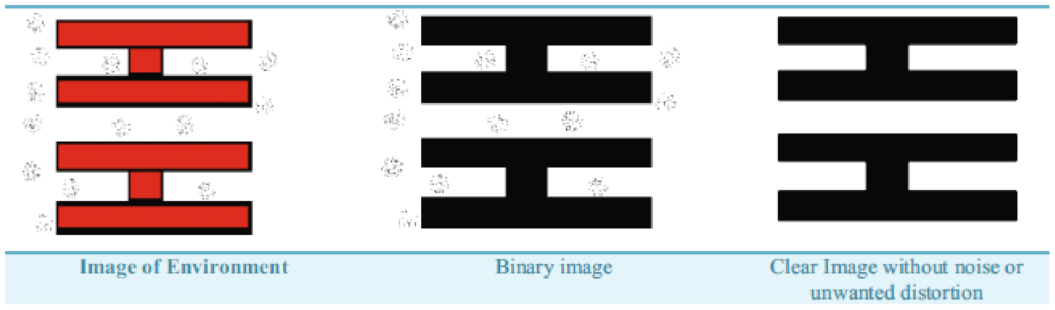


Figure 2.4: Zidane & Ibrahim (2018) image processing steps

Table 2.1: Advantages and Disadvantages between Image Acquisition and Processing Methods

	Advantage	Disadvantage
Singhata (2021)	- Faster processing time - Less memory requirement	- Recognized maps only - Background and robot must not have the same colour
Zidane & Ibrahim (2018)	- Robust to fit different types of maps - More applicable in the industry	- Long processing time if sectoring is not optimised - Requires more memory - All elements must not have the same colour as the background

2.3 Search algorithm

A search algorithm is used for pathfinding, which is the route's navigation between two points, the start position and the end position. This algorithm aims to meet the criteria of achieving the shortest path by evaluating optimality, completeness, space complexity, and time complexity. According to Niederberger et al. (2004), four conditions must be met for path planning algorithms. The path generated by the algorithm should achieve the lowest cost. The algorithm should be correct instead of just achievements on the evaluation criteria of path planning. No human interaction is needed to assist in the path planning of the algorithm. The algorithm should be robust to fit different maps instead of limited map types.

Based on Sidhu (2019), there are two types of pathfinding algorithms: uninformed pathfinding and informed pathfinding. Uninformed pathfinding can be explained as a blind search. It does not have any information about the end node and only searches the adjacent cells until the end node is found. The standard algorithms under this group are BFS and the Dijkstra algorithms. Informed pathfinding is a more advanced method developed from an uninformed pathfinding algorithm. The algorithm considers the end node, where the estimated cost of the end location will be calculated. Informed pathfinding algorithms are mainly uninformed pathfinding algorithms with a heuristic function, which calculates the distance from the current node to the end node. The heuristic functions are developed from Manhattan distance, Euclidian distance, or Octile distance. The more commonly used functions are the Manhattan distance and the Euclidian distance, where the Manhattan distance outperforms the Euclidian distance in terms of execution time (Sharma and Kumar, 2016). The standard algorithms under this group are A*, HPA*, and LPA*.

Referring to Russell and Norvig (2019), BFS explores in all directions equally and is one of the easiest graph search methods. The nodes are explored in levels, one before another, as shown in figure 2.5. FIFO strategy is being applied, where the shallowest node is processed first. The stopping criteria of BFS are either the accomplishment in the detection of the end node or there are no more nodes to be expanded. The BFS algorithm ensures the shortest path from the start node to the end node. The flowchart of BFS can be seen in figure 2.6.

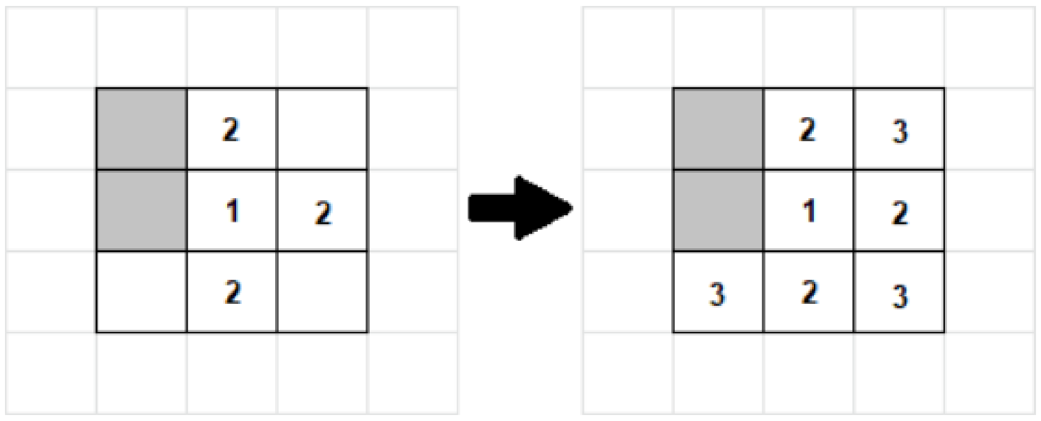


Figure 2.5: BFS illustration (Vargas et al., 2020)

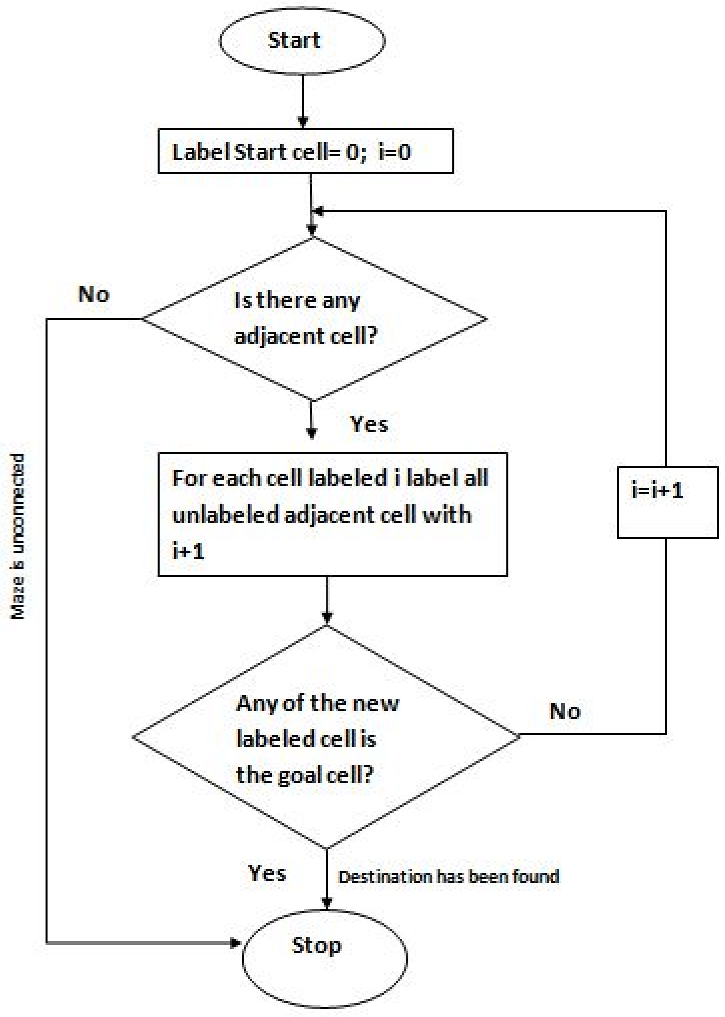

Start
Label Start cell= 0; i=0
Is there any adjacent cell?
No
Yes
Maze is unconnected
For each cell labeled i label all unlabeled adjacent cell with i+1
i=i+1
Any of the new labeled cell is the goal cell?
No
Yes
Destination has been found
Stop

Figure 2.6: Flowchart of BFS (Sadik et al., 2010)

Dijkstra algorithm, also known as uniform cost search, gives priority when exploring paths to the end node (Russell and Norvig, 2019). Compared to

BFS, it does not explore all the adjacent nodes equally but chooses the paths with a lower cost. Dijkstra's algorithm is considered a 'greedy' algorithm as it uses a priority queue, where the node with the highest priority will be processed first. Nodes with lower distances would have higher priorities in the queue. Dijkstra's algorithm stops once the shortest path is found (Mehlhorn and Sanders, 2008). When the cost of the map is the same, the Dijkstra algorithm has the same working principle BFS as shown in figure 2.5.

A* algorithm is slightly different compared to BFS and Dijkstra. Instead of using the start node to calculate the actual distance, it uses the start and end nodes for computation. The actual and estimated distance is calculated. A* is the implementation of Dijkstra's algorithm with heuristic (Martell and Sandberg, 2016). The stopping criteria of A* and Dijkstra's algorithm are the same. It will stop whenever the path is found. The function of the A* algorithm is shown in equation 2.1.

$$f(n) = g(n) + h(n) \tag{2.1}$$

where
$g(n)$ = costs from the start node
$h(n)$ = heuristic

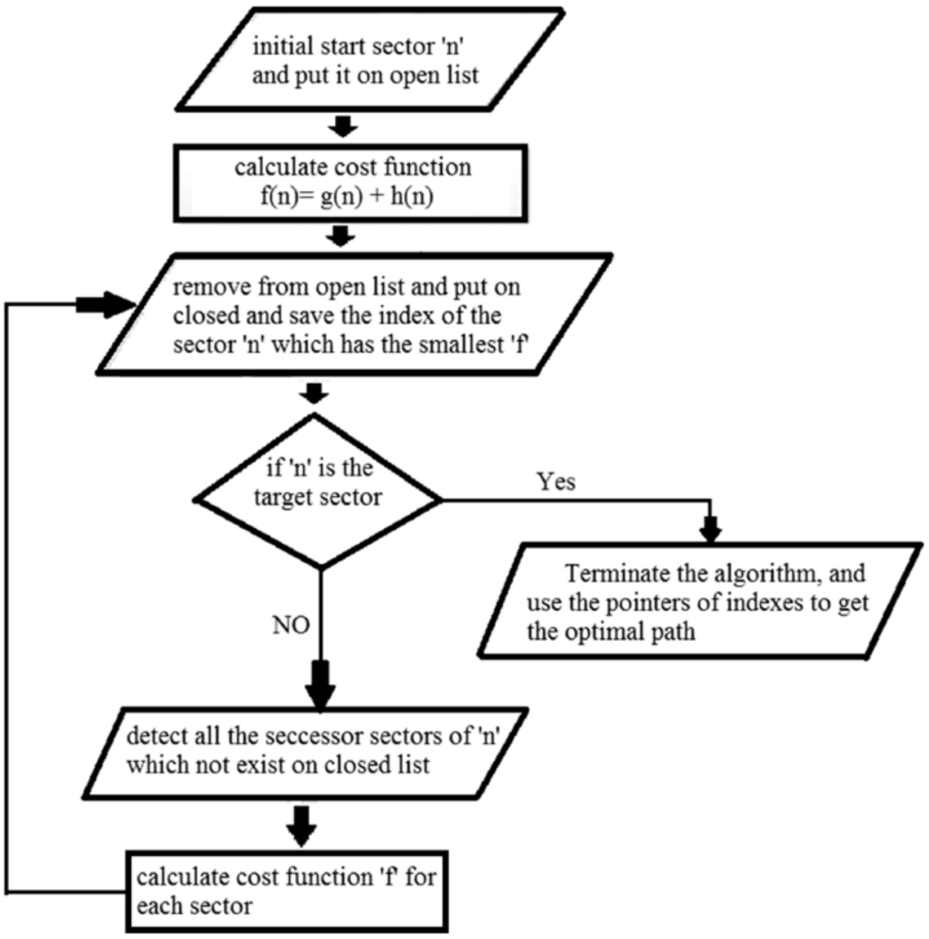

initial start sector 'n' and put it on open list
calculate cost function f(n)= g(n) + h(n)
remove from open list and put on closed and save the index of the sector 'n' which has the smallest 'f'
if 'n' is the target sector
Yes
Terminate the algorithm, and use the pointers of indexes to get the optimal path
NO
detect all the seccessor sectors of 'n' which not exist on closed list
calculate cost function 'f' for each sector

Figure 2.7: Flowchart of A* algorithm (Zidane and Ibrahim, 2018)

In the work of Zarembo and Kodors (2013), the authors reviewed different pathfinding algorithms and made comparisons based on the time and space complexity. According to their findings, the A* algorithm outperforms Dijkstra's execution time when implementing algorithms in different grid sizes. By comparing three different pathfinding algorithms, BFS has a long execution time, making it less applicable in a real-time application for significant grid problems due to its simplicity. But when the algorithm runs through a smaller grid problem, the A* algorithm gives the best outcome in execution time compared to the others. At the same time, BFS comes in second, slightly faster than Dijkstra's algorithm, when applied in a 20x20 grid size in different obstacle arrangements (Fahleraz, 2018).

In terms of navigated nodes, Dijkstra's algorithm navigates through fewer nodes than BFS in more significant grid problems but performs slightly worse than BFS in small grid problems. A* algorithm navigates through much fewer nodes than BFS and Dijkstra's. Dijkstra's algorithm and BFS achieve better optimality outcomes than the A* algorithm (Arshad et al., 2016). From the studies made by Rachmawati and Gustin (2020), the shortest path generated by the A* algorithm and Dijkstra's algorithm is around the same for small maps in terms of execution time. Still, the A* algorithm is more favourable for larger maps as it only searches in the direction of the end node instead of all the adjacent nodes.

Zidane and Ibrahim (2018) implemented both the A* and wavefront expansion algorithms to test the execution time and efficiencies of the algorithm. The wavefront algorithm is the same as BFS. It spreads a wave from the start node propagating forward until it reaches the end node, figure 2.5. The shortest path will be computed. In his studies, A* has a much longer execution time than the wavefront algorithm. The researchers concluded that too many files are needed to calculate the A* algorithm, which takes up a tremendous amount of computer memory in a discrete environment. Hence, A* will be more suitable for higher computational power machines.

Table 2.2: Advantages and Disadvantages of the reviewed search algorithm

	Advantages	Disadvantages
BFS	- Supports multiple locations - Will locate the existent solution - Optimal path	- The cost must be the same - Very long execution time - High memory usage
Dijkstra	- Supports multiple locations - Supports different cost - Optimal path	- Long execution time - Cannot compute negative weights
A*	- Faster execution time - Much lesser traversed nodes - Optimal path	- Huge amount of memory usage - More complex implementation - Highly dependent on heuristics

2.4 Embedded Devices

Based on Vargas et al. (2020), the authors designed their autonomous vehicle using Raspberry Pi. The model selected is Raspberry Pi Model B, which has 512 MB RAM, USB Ports and Ethernet controller. A Wi-Fi dongle is used to establish the connection wirelessly in their application. Raspberry Pi Model B draws low power while having high computational power, making it suitable for various applications. Raspberry Pi is a small computer that is driven by Linux operating system. Raspberry Pi is more extensive in dimension and costs more than other embedded devices, such as microcontrollers. Analog input cannot be used on Raspberry Pi. An external ADC is needed. There are several successors for Raspberry Pi Model B. The newest model is Raspberry Pi 3 Model B+, which supports Wi-Fi and Bluetooth. The obstacle avoidance robot built by Pavithra & Subramanya Goutham (2018) uses Arduino UNO as the microcontroller. Arduino UNO does not have embedded Wi-Fi; hence a Wi-Fi module is needed if we implement it in our system. Arduino with an external Wi-Fi module can be difficult during the configuration stage. Nonetheless, Arduino UNO is a suitable microcontroller for data acquisition. In the research of Mistri (2018), ESP8266 NodeMCU is used. The new design of using NodeMCU instead of the traditional Arduino design is proven to be workable. ESP8266 has an embedded Wi-Fi adapter, which suits the use of IoT platforms. In terms of ADC pins, which are required to convert analogue signal to digital signal, ESP8266 only have one. The number of ADC pins limits the usage and robustness of the ESP8266. ESP32 is the successor of ESP8266, which has more features and can be implemented in our system.

Table 2.3: Specifications of different embedded devices

	Raspberry Pi 3 B+	Arduino UNO	ESP32
Clock Speed	1.4 GHz	16 MHz	80 to 240 MHz
RAM	1 GB SDRAM	2 kB SRAM	520 kB SRAM
Board Power Supply	5 V	5 V	5 V
Flash Memory	MicroSD storage	32 kB	4 MB
Analog Input Pins	0	6	15
Wi-Fi	Yes	No	Yes
Bluetooth	Yes	No	Yes

Table 2.4: Advantages and Disadvantage(s) of different embedded devices

	Advantages	Disadvantage(s)
Raspberry Pi 3 B+	- Open-source - Embedded Wi-Fi adapter, ethernet and Bluetooth - Large range of GPIO pins	- Bigger Size - Expensive - No ADC pins - High power consumption
Arduino UNO	- Open-source - Inexpensive	- No Wi-Fi adapter - Bigger size
ESP32	- Open-source - Embedded Wi-Fi adapter and Bluetooth - Small size - Cheap	None

CHAPTER 3

METHODOLOGY AND WORK PLAN

3.1 System architecture of block diagram

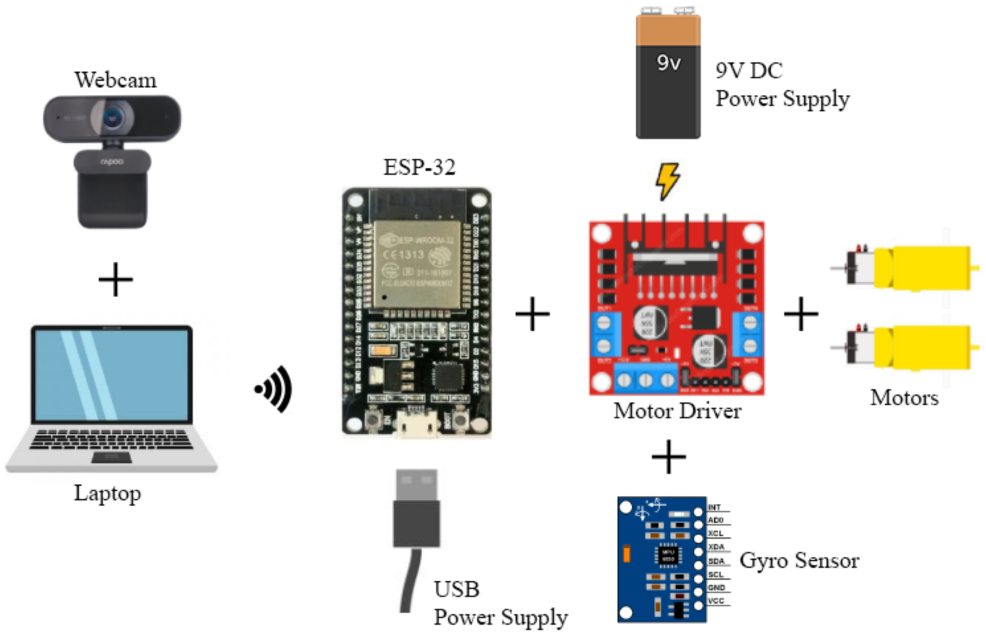


Figure 3.1: Block diagram showing hardware implementation of Navigation Robot

According to the block diagram in Figure 3.1, the input will be captured by a webcam connected to the laptop via USB. The laptop's output file required for the microcontroller to operate the navigation robot is transmitted via MQTT. The connection is established by using Node-RED. ESP-32 is powered by a USB power supply and links to the motor driver and the gyro sensor. A 9 volts DC power supply is needed to power the motor driver. The outputs for the autonomous navigation robot are the motors, which are linked to output one and output two ports in the motor driver. The power of the gyro sensor can be drawn from ESP-32 as the operating voltage of the sensor ranges from 3 volts to 5 volts. The output voltage from the ESP-32 is rated at 3.3 volts.

3.2 Vision-based Indoor Navigation Robot Flowcharts

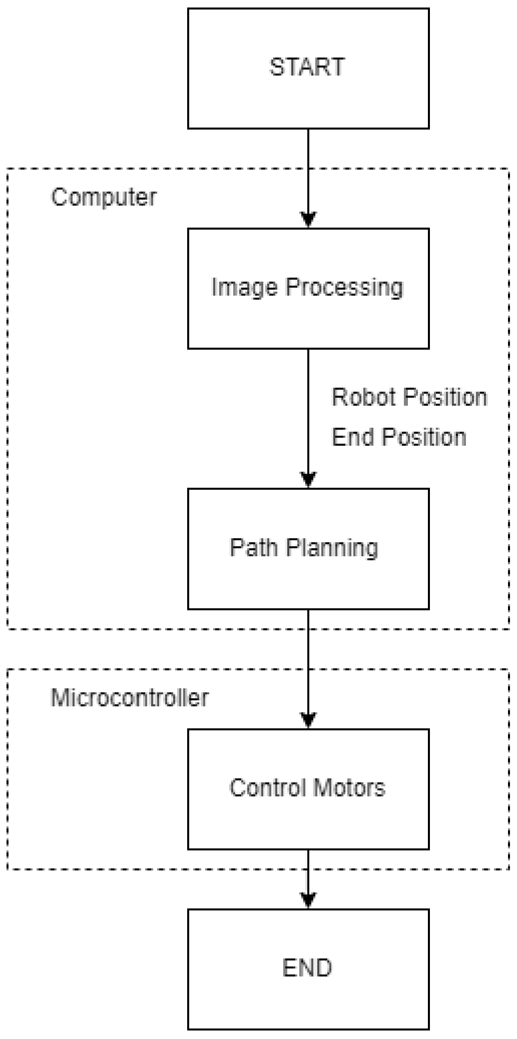


Figure 3.2: Block diagram of Vision-based Navigation Robot

Figure 3.2 visualises the flowchart of the navigation robot. Image processing and path planning are done after image capturing from the web camera. The start position, end position and obstacles are appropriately defined for path planning. The outcome of path planning is sent to the microcontroller to control the motors for the robot’s movement toward the goal set.

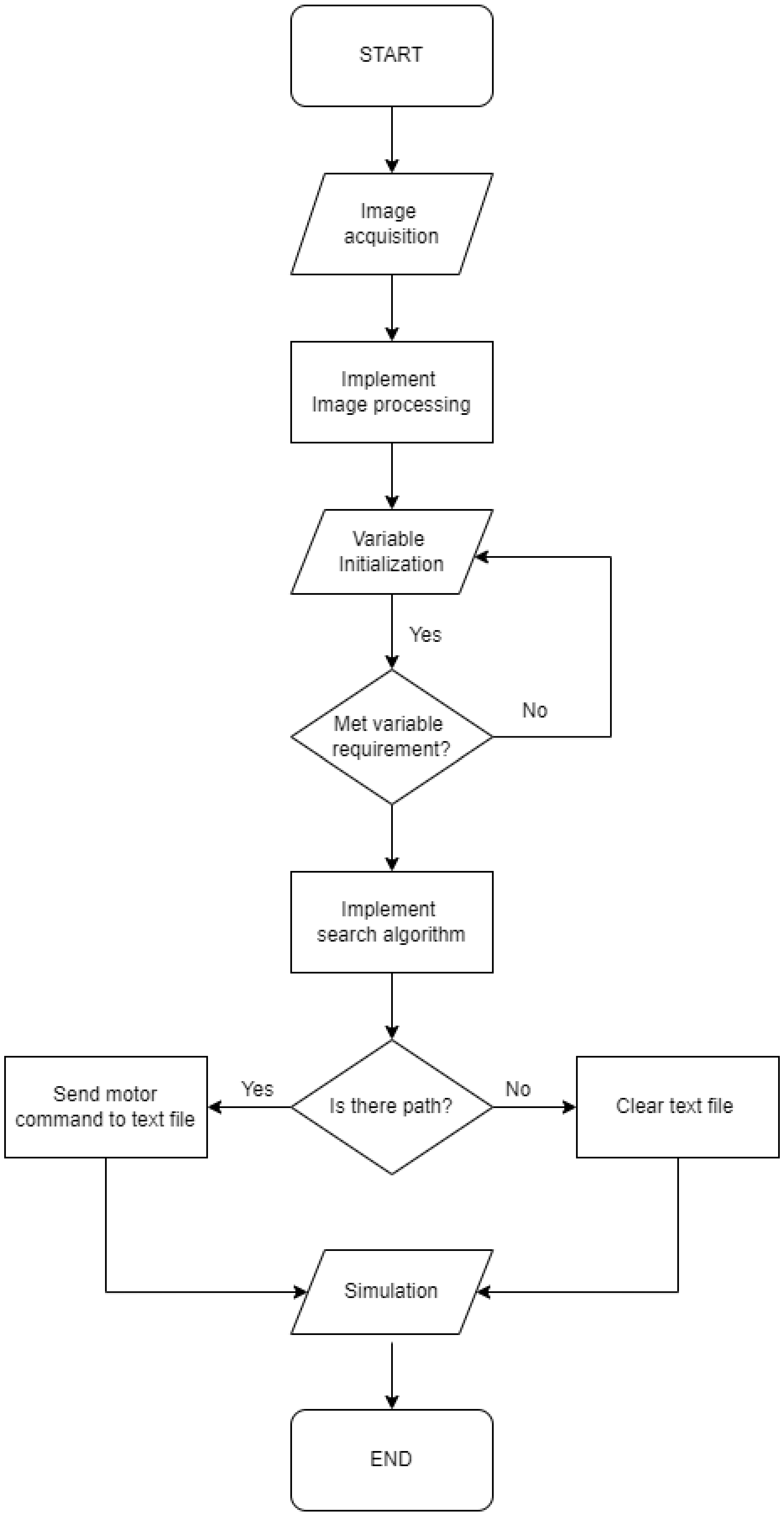

START
Image acquisition
Implement Image processing
Variable Initialization
Yes
Met variable requirement?
No
Implement search algorithm
Send motor command to text file
Yes
Is there path?
No
Clear text file
Simulation
END

Figure 3.3: Sketch of Visual Studio flowchart

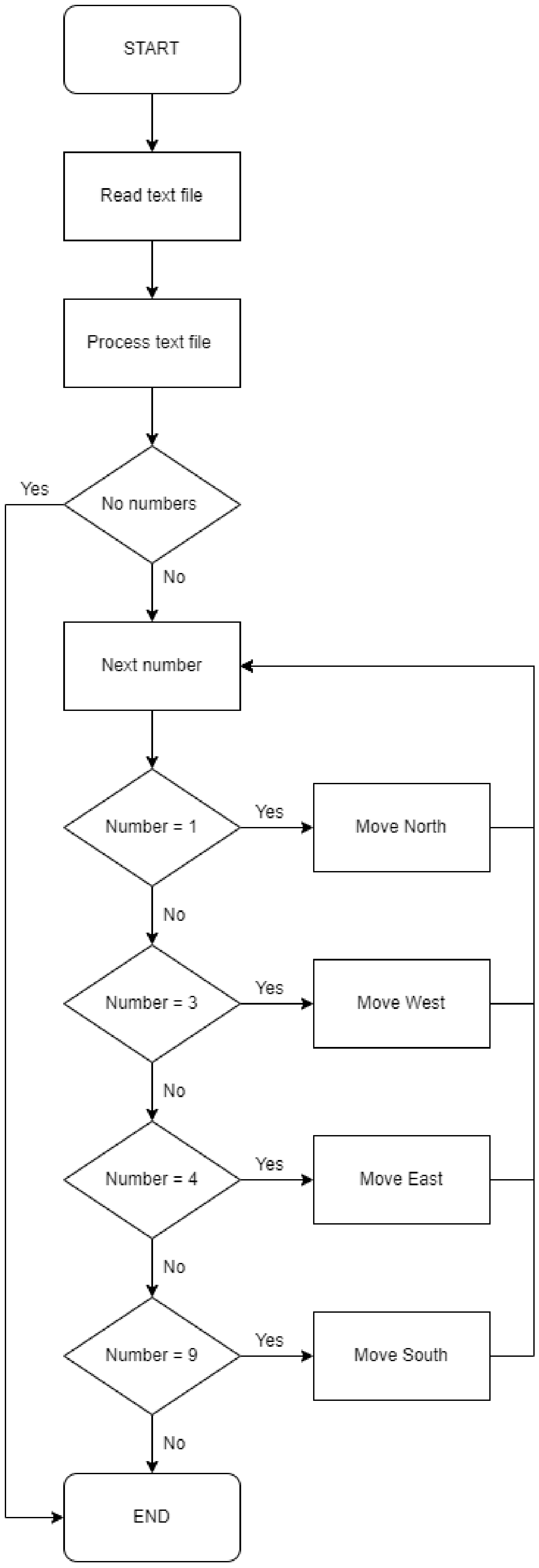


Figure 3.4: Sketch of Microcontroller flowchart

Figure 3.3 visualises the logic flow of the vision-based indoor navigation robot program. The image is captured after the initialisation of the program. The captured image undergoes image processing to convert the image into a programmable logic, which is an array. The boundaries and obstructions in the image can be acquired. The start location of the navigation robot is saved into a variable. After image processing, the user gives the end location input, and a comparison is made to check if the parameters reach the requirement. An input outside the boundaries will not be recorded until a suitable parameter is recorded. The search algorithm can be implemented with all the information, the boundaries and obstacles, the start location, and the end location. The search algorithm will find the optimal path if a solution to the map is provided. When the search algorithm finds the path, the movement towards the end location will be recorded and sent to a text file for routing purposes of the navigation robot. The output text file will be empty if there is no solution where no paths are found. The navigation robot will execute no operation.

Figure 3.4 shows the logic flow of the microcontroller for the system. The microcontroller receives a text file from the computer created by the previous program, and processing is needed to turn the commands given into motor instructions. No numbers in the text file indicate that no path is found in the previous program. The autonomous robot does not need to execute any operations. If numbers are detected in the text file, the microcontroller will turn the number commands into instructions to the motors, where different numbers point to different directions of movement. While all the numbers are executed, no numbers will be next in the string. The autonomous robot should be at the goal location. The process is terminated.

3.3 Hardware

3.3.1 Webcam

Figure 3.5: Rapoo C260 Full HD webcam

Figure 3.5 shows the USB camera used in the system to capture the top view of the field. The USB camera chosen has a high resolution of 1920 x 1080 which meets the requirement of picture quality at a certain distance. It has a 95° wide angle which can capture massive space in the environment. This device is simple to operate and meets the compatibility of different operating systems. No additional drivers are required.

3.3.2 Test field

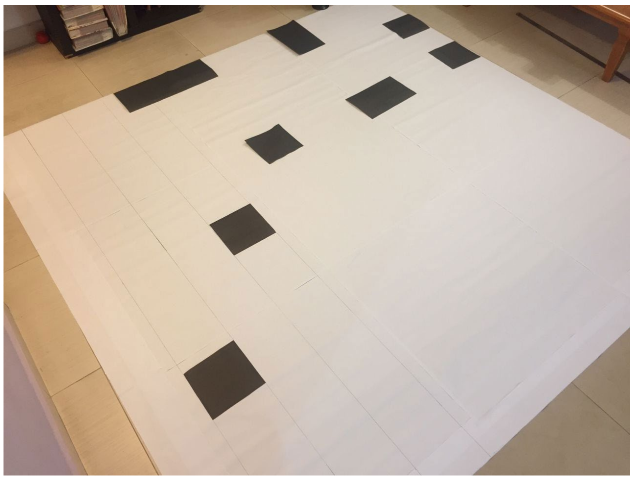


Figure 3.6: Example of the test field

The test field is used to evaluate the performance of the navigation robot. The maze has a 2.4 m x 2.4 m size, which means more maps and variations can be done to assess the robot's movement. Black tiles represent the obstacles, and white tiles represent the path available for the robot's movement.

3.3.3 Robot Chassis and motors

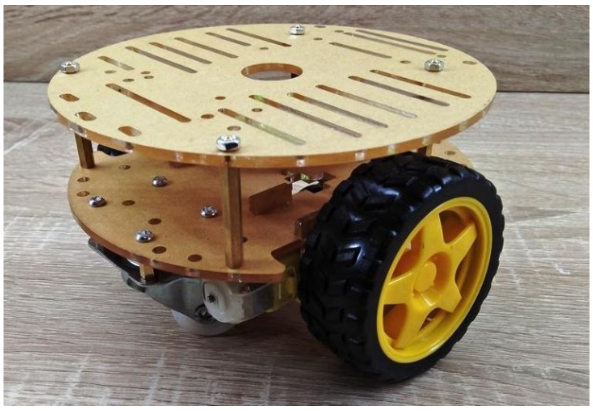


Figure 3.7: 2WD robot chassis

2WD robot chassis will be used in this system. The dimension of the chassis is 140 mm x 146 mm. The 2WD robot chassis, HC02-48, comes with two DC motors (shown in figure 3.8). This DC motor has a rated voltage of 3.3 to 6 volts DC and draws 150 mA of current. The gear reduction for this motor is 1:48, and it can perform forward and backward movements. 125 RPM at 3.3 volts and 250 RPM at 6 volts under no-load conditions. The torque of the motor ranges from 0.15 kg.cm to 0.6 kg.cm. This motor is ideal for this system as it outperforms other small DC motors that encounter high shaft speed and low torque issues. The entire chassis is chosen as it can perform circular movement more accurately than other robot chassis on the market.

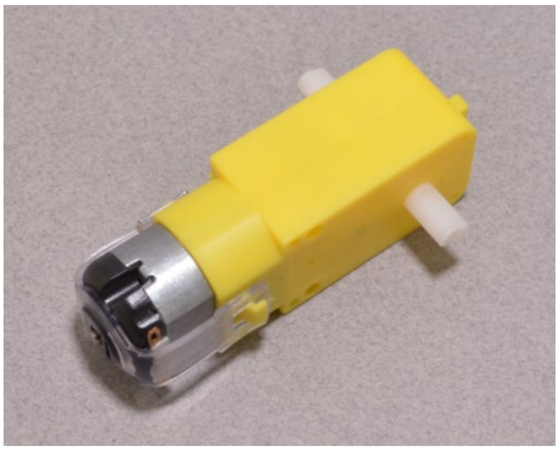


Figure 3.8: DC motor, HC02-48

3.3.4 ESP32

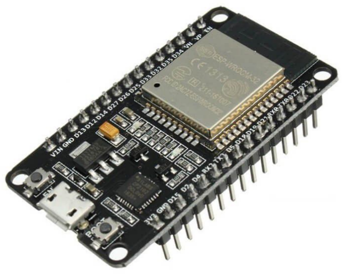


Figure 3.9: ESP32 Microcontroller Unit

ESP32 will be used as the microcontroller of the navigation robot. A comparison was made with other microcontrollers, and ESP32 fits perfectly with this application. This small microcontroller comes with Bluetooth and Wi-Fi adapter, allowing us to connect with the computer wirelessly. ESP32 is open source, and some of the codes posted online could be used as the benchmark for this project.

3.3.5 Motor driver

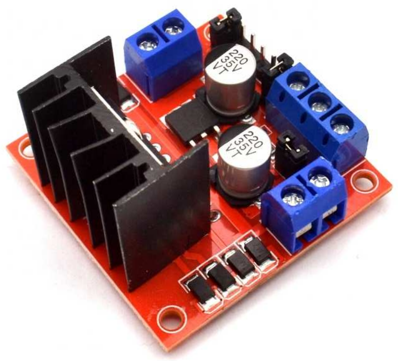


Figure 3.10: L298N motor driver

The L298N motor driver will be implemented in this system. It is common in a microcontroller robot car as it is a high-power motor driver module. The motor driver can operate voltages from 5 volts to 35 volts. There are two channels where both the channels can draw up to 2 Amperes, and which heat sink plays a vital role in providing cooling. This is a dual H-Bridge motor driver, where handling over the rotation direction and speed can be done simultaneously. By varying the PWM output, the rotation speed of the motor changes. The opposite direction can be achieved by inverting the current flowing through the motor using the H-bridge. L298N driver is selected mainly due to its capability to output a higher current, which provides high torque and RPM. The motor driver is lightweight and has a small dimension, which is suitable for the usage of the navigation robot.

3.3.6 Gyroscope and Accelerometer

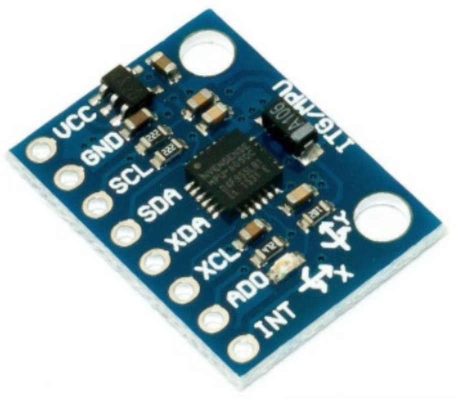


Figure 3.11: MPU6050 sensor

The MPU-6050 module is a combination of a three-axis gyroscope and accelerometer. One chip contains both the MEMS gyro and MEMS accelerometer. It has a 16-bit converter chip, converting analogue to digital for each channel. 16-bit data capture provides high accuracy to the sensor. Acceleration and rotation can be measured with four programmable full-scale ranges, each from the accelerometer and gyroscope. The operating voltage is 3.3 volts, but 5 volts input can be used as the module is equipped with a 3.3 volts regulator, LD3985. This module consumes less power as it only takes 3.6 mA at work and $5\mu A$ during the idle state. IIC communication protocol standard is implemented in the module where two individual addresses are supported. The module's size is small, making it suitable to mount onto the robot chassis. MPU6050 sensor is an assistant in the calibration of the autonomous robot to ensure the accuracy of the motor rotation and motor speed in the test environment.

3.4 Software

3.4.1 Visual Studio

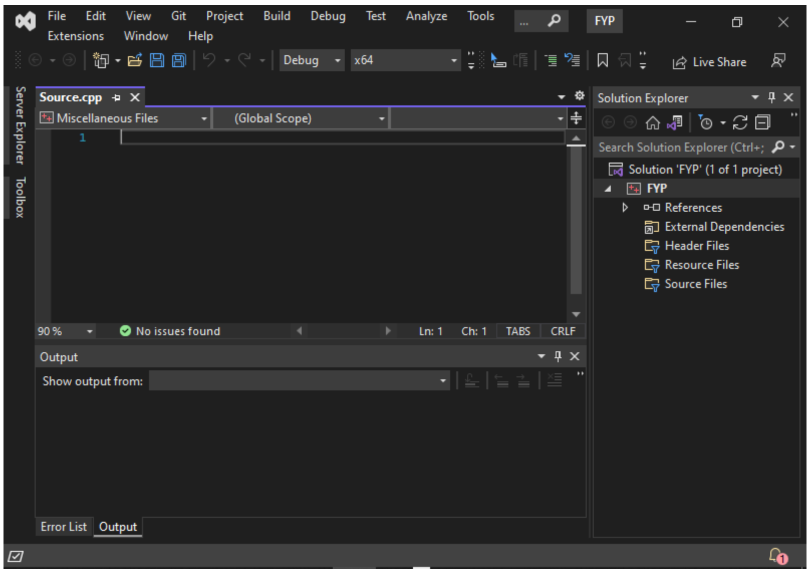
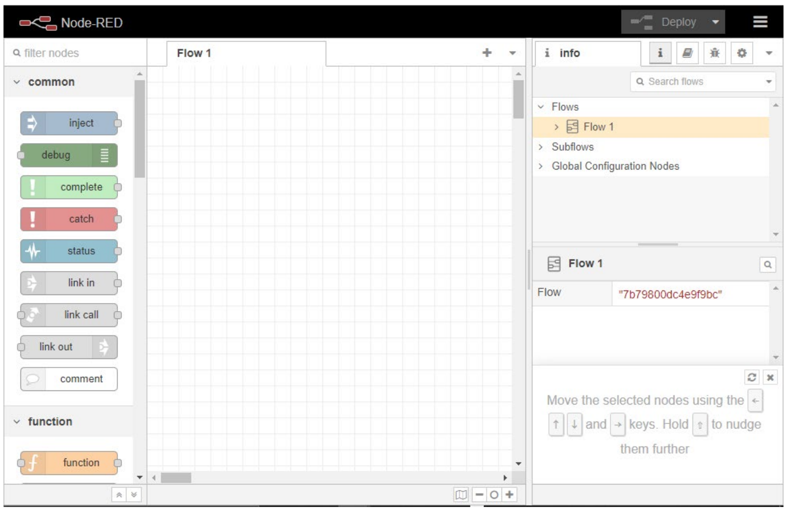


Figure 3.12: Snapshot of Visual Studio 2022

A powerful IDE from Microsoft that can develop computer programs. Handy software for editing, debugging, and building codes. It supports different programming languages, including the language for this project usage, C++ language. This project will use Visual Studio with OpenCV and OpenGL to capture, process images and simulate the results of programs. The IDE is commonly used as a compiler, where compatibility issues will not arise on other devices.

3.4.2 Node-RED

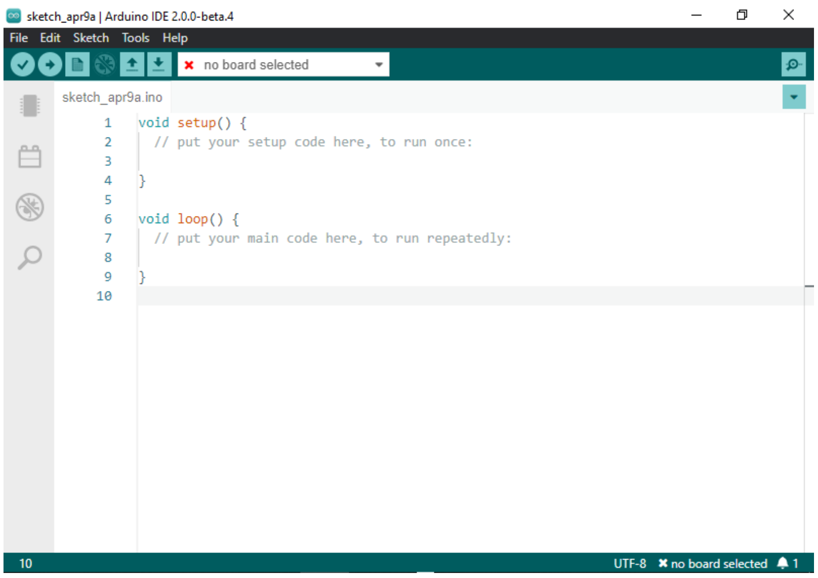


Move the selected nodes using the ← ↑ ↓ and → keys. Hold ⇧ to nudge them further

Figure 3.13: Snapshot of Node-RED

Node-Red is widely used for IoT applications. It registers and supports the connection between different devices. Node-Red provides MQTT subscribe and publish, which is the input and output. This approach is selected to establish a connection between the computer and microprocessor wirelessly.

3.4.3 Arduino IDE



```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}

```

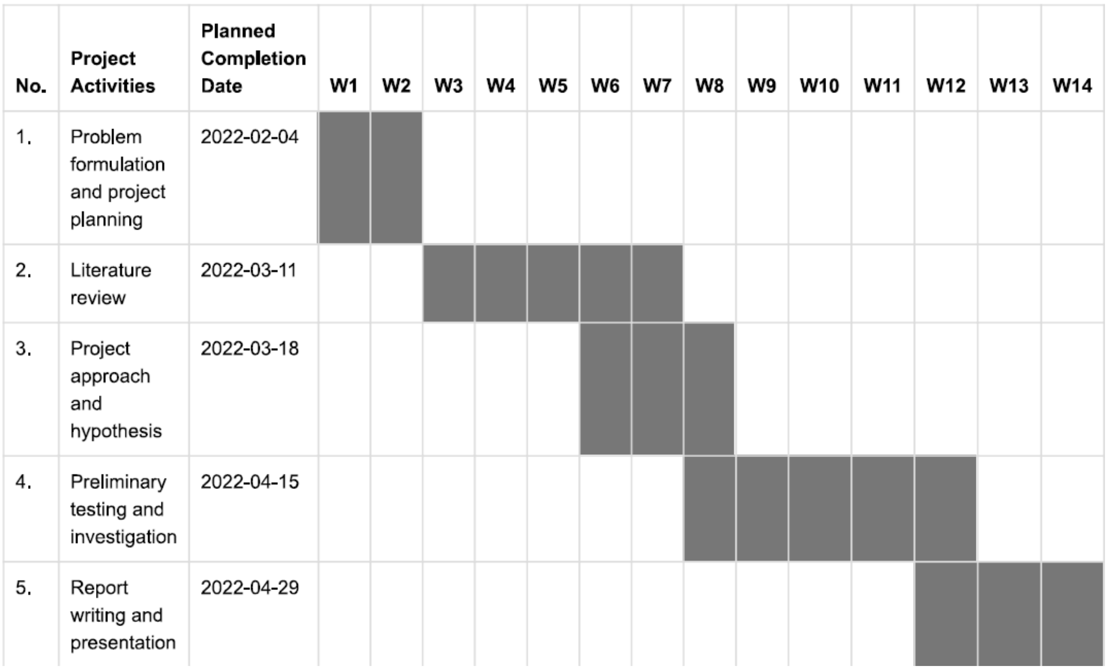
Figure 3.14: Snapshot of Arduino IDE

Arduino IDE is software designed for Arduino usage to edit, debug, compile, upload and communicate. This software aims to convert the C language into machine code. ESP32 is used in this project, where this microcontroller is compatible with Arduino IDE. Arduino is open source, where many references for different projects are available. The codes could be used as a benchmark for the project. In some cases, the codes in open source are more stable as a vast community develops them.

3.5 Workplan

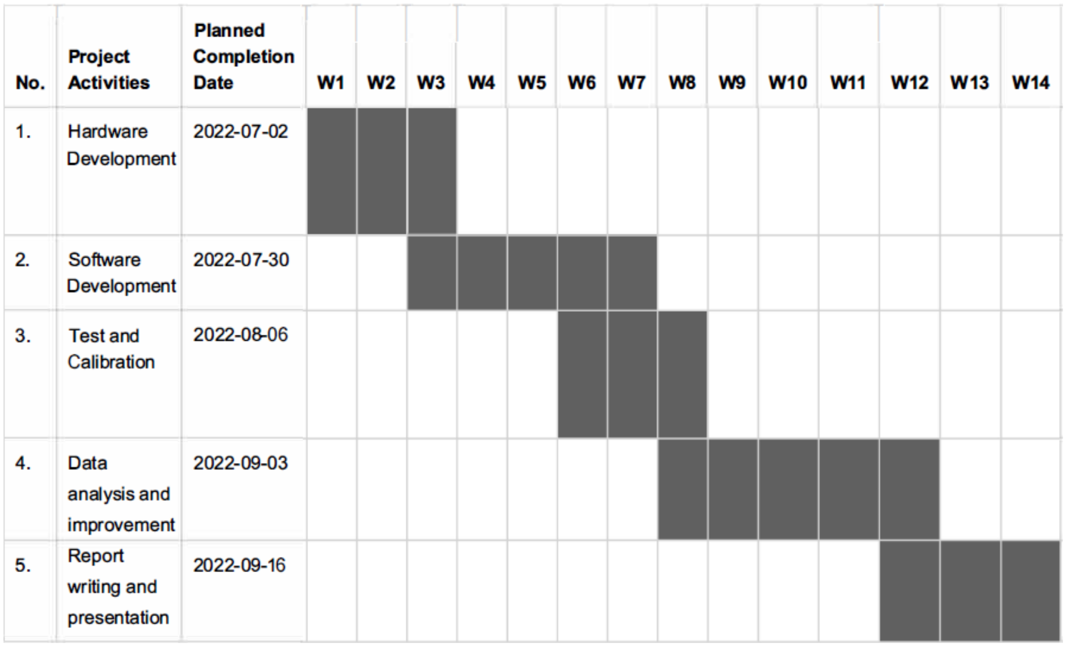
Figure 3.15 shows the Gantt Chart of the final year project part 1. After confirming the final year project title, background research is done based on the selected title. Project planning is completed to verify the scope of the project. Literature reviews of other research are carried out to identify approaches and methods to achieve the outcomes. Discussions were made from the sixth week to the eighth week to consider possible strategies and the expectations of results. Preliminary testing of software development is tested out. Report writing and presentation are focused on the last three weeks of the semester.

Workplan for final year project part 2 can be seen in Figure 3.16. Hardware development, such as the navigation robot, is developed in the first two weeks of the semester. Software development, testing and calibration took the most time as image processing, pathfinding, connection with the microprocessor, microprocessor coding etc., take time to achieve an optimal result. Data analysis and possible improvement are carried out in the next stage. The last three weeks are reserved for report writing and presentation.



No.	Project Activities	Planned Completion Date	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14
1.	Problem formulation and project planning	2022-02-04	■	■												
2.	Literature review	2022-03-11			■	■	■	■	■							
3.	Project approach and hypothesis	2022-03-18						■	■	■						
4.	Preliminary testing and investigation	2022-04-15								■	■	■	■	■		
5.	Report writing and presentation	2022-04-29												■	■	■

Figure 3.15: Gantt chart snapshot from the e-FYP portal (Part-1)



No.	Project Activities	Planned Completion Date	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14
1.	Hardware Development	2022-07-02	■	■	■											
2.	Software Development	2022-07-30			■	■	■	■	■							
3.	Test and Calibration	2022-08-06						■	■	■						
4.	Data analysis and improvement	2022-09-03								■	■	■	■	■		
5.	Report writing and presentation	2022-09-16												■	■	■

Figure 3.16: Gantt chart snapshot from the e-FYP portal (Part-2)

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Circuit Diagram

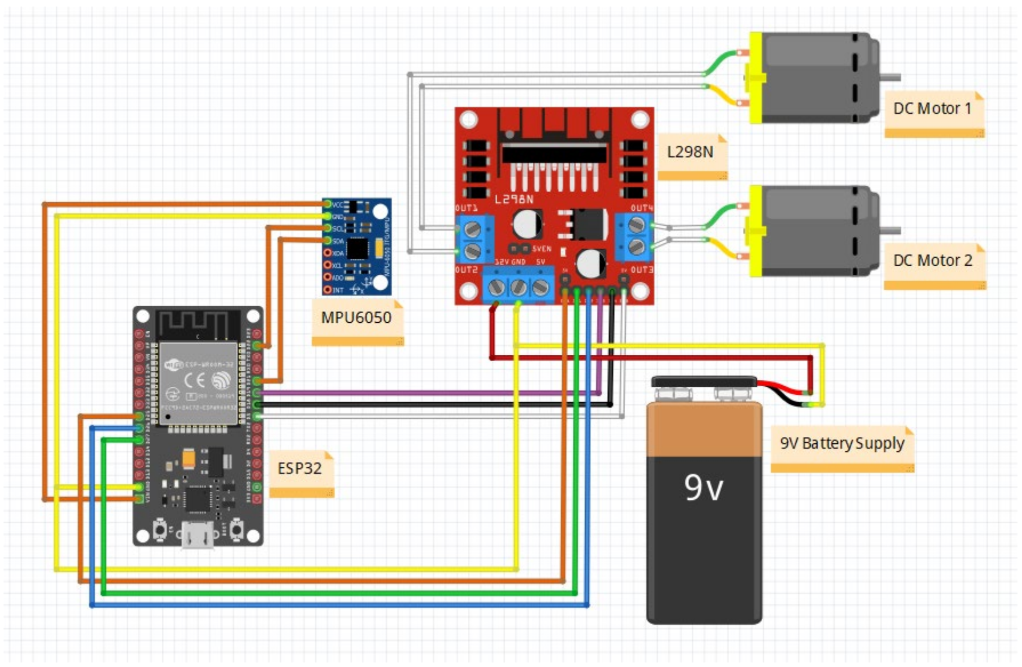


Figure 4.1: Circuit Diagram of Navigation Robot

The navigation robot is built to test the workability of the C++ code written in Visual Studio. Figure 4.1 shows the circuit diagram of the navigation robot. The circuit diagram is illustrated using Fritzing. The system is mainly powered by a USB cable attached to a power bank and a 9 volts battery supply. The USB cable is connected to the ESP32. Based on figure 4.1, orange wires mainly represent the connection of the MPU6050, white wires are the connection between the L298N motor driver and the DC motors, and yellow wires signify the system's grounding.

The connection of the MPU6050 is established by connecting the VCC on the board to the VIN of the ESP32, which also represents a 5 volts power supply. SCL and SDA are wired to D22 and D21, respectively, to establish the I2C protocol between the devices.

The 9 volts battery supply is wired to the 12 volts input port on the L298N motor driver. Outputs 1 and 2 are connected to DC motor 1, and Outputs

2 and 3 are connected to DC motor 2. The direction control pins, IN1, IN2, IN3 and IN4, are connected to the ESP32 individually. IN1 and IN2 control the moving direction for Motor 1, and IN3 and IN4 control Motor 2. ENA and ENB are wired onto D25 and D5, which control the motors' rotational speed. ENA is for DC Motor 1, and ENB is for DC Motor 2. In one of the testing phases, ENA and ENB are jumped directly to the 5 volts adjacent port as the motor's rotational speed is not considered.

The pin connections explained above, connected to the ESP32, can be summarised in Table 4.1.

Table 4.1: Pins connection between ESP32 and Other Modules

Module	Module Pins	ESP32 Pins
MPU6050	VCC	VIN
	SCL	D22
	SDA	D21
L298N	ENA	D25
	ENB	D5
	IN1	D27
	IN2	D26
	IN3	D19
	IN4	D18

4.2 Software Development

4.2.1 Visual Studio

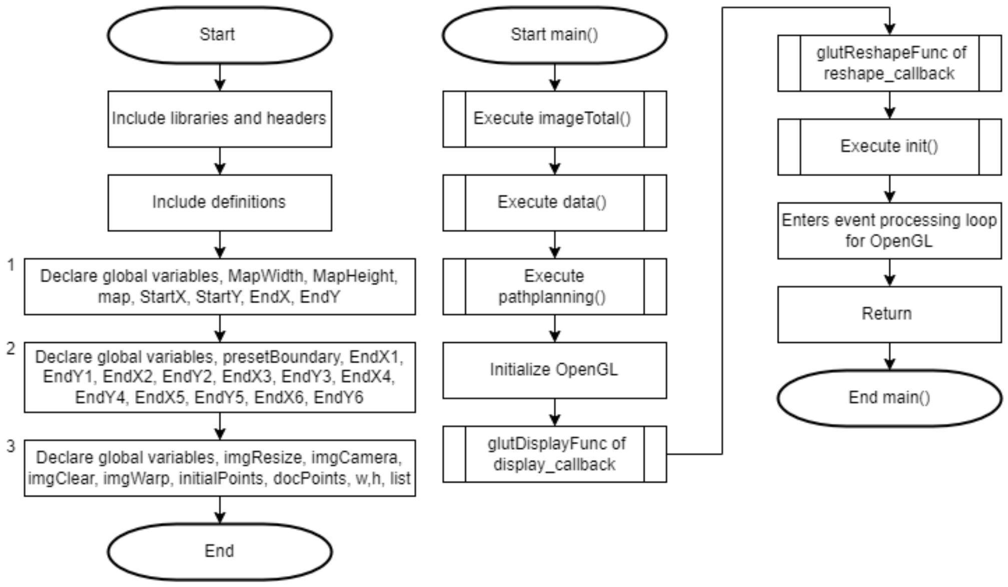

Start
Include libraries and headers
Include definitions
1
Declare global variables, MapWidth, MapHeight, map, StartX, StartY, EndX, EndY
2
Declare global variables, presetBoundary, EndX1, EndY1, EndX2, EndY2, EndX3, EndY3, EndX4, EndY4, EndX5, EndY5, EndX6, EndY6
3
Declare global variables, imgResize, imgCamera, imgClear, imgWarp, initialPoints, docPoints, w,h, list
End
Start main()
Execute imageTotal()
Execute data()
Execute pathplanning()
Initialize OpenGL
glutDisplayFunc of display_callback
glutReshapeFunc of reshape_callback
Execute init()
Enters event processing loop for OpenGL
Return
End main()

Figure 4.2: Flowchart of program and main() function

Libraries and headers are included at the beginning of the program. Essential global variables are labelled with ‘1’ in the flowchart, variables labelled ‘2’ are primarily for the implementation of pre-set coordinates, and variables marked ‘3’ are for image processing of raw images taken by the web camera except for ‘list’. The variable list is for tuples and will be used in path planning. Referring to figure 4.2, the main() function executes imageTotal() initially to capture and process the image for path planning. The end position of the robot in the environment is recorded in the data() function. The data can either be a preset coordinate by the system or a single custom coordinate. Once the data is logged, path planning can be done next. The results of the path planning will be simulated using OpenGL. Based on figure 4.2, from “Initialize OpenGL” to “Enters event processing loop for OpenGL” are the steps for simulation. glutDisplayFunc configures the display callback for the existing window. glutReshapeFunc serves to set up the reshape callback for the current window. The main program returns and stops after the end of the processing loop for OpenGL.

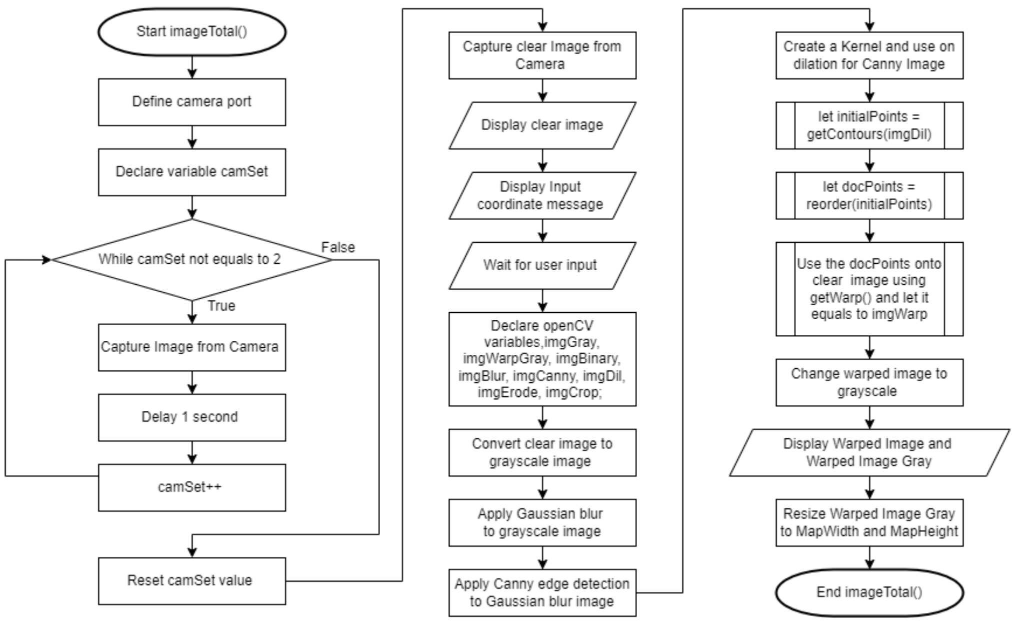

Start imageTotal()
Define camera port
Declare variable camSet
While camSet not equals to 2
False
True
Capture Image from Camera
Delay 1 second
camSet++
Reset camSet value
Capture clear Image from Camera
Display clear image
Display Input coordinate message
Wait for user input
Declare openCV variables,imgGray, imgWarpGray, imgBinary, imgBlur, imgCanny, imgDil, imgErode, imgCrop;
Convert clear image to grayscale image
Apply Gaussian blur to grayscale image
Apply Canny edge detection to Gaussian blur image
Create a Kernel and use on dilation for Canny Image
let initialPoints = getContours(imgDil)
let docPoints = reorder(initialPoints)
Use the docPoints onto clear image using getWarp() and let it equals to imgWarp
Change warped image to grayscale
Display Warped Image and Warped Image Gray
Resize Warped Image Gray to MapWidth and MapHeight
End imageTotal()

Figure 4.3: Flowchart of imageTotal() function

The camera port is defined at the start of the imageTotal() function. Variable camSet is declared for the while loop to create two seconds of delay for the auto-calibration of the camera. The web camera used is the Rapoo C260 Full HD webcam. The image taken on the initialisation of the camera will be too dark for image processing. After two seconds of delay, the camSet value is reset to zero, and a clear image should be taken. The clear image is being displayed on the screen for review purposes. The result of the camera image before and after buffer time is compared in figure 4.4. User input is needed to continue the image processing process. The acquired clear image is processed using OpenCV, and all necessary variables are declared. A clear image is converted to a grayscale image to apply gaussian blur and canny edge detection. Dilation of the canny edge detection image is done after creating a kernel for it. getContour() function is used to get the contours from the dilated image, and reorder() is used to rearrange the initial points found from getContour(). docPoints is referred by the getWarp() function to obtain a perfect warped image of the test environment. The warped image is converted to a grayscale image, and both the warped image and warped grayscale image are displayed to review the step-by-step process of the program. A warped grayscale image is converted to the correct pixel size based on the environment for other operations.

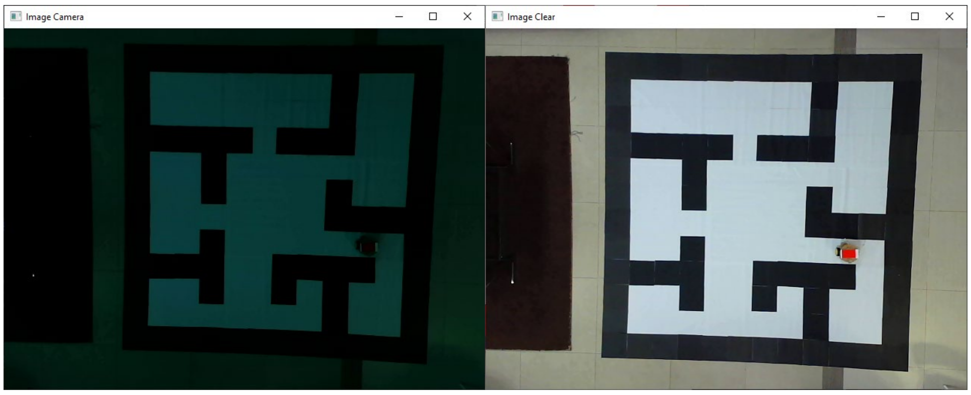


Figure 4.4: Comparison of No Auto Calibration vs Auto Calibration

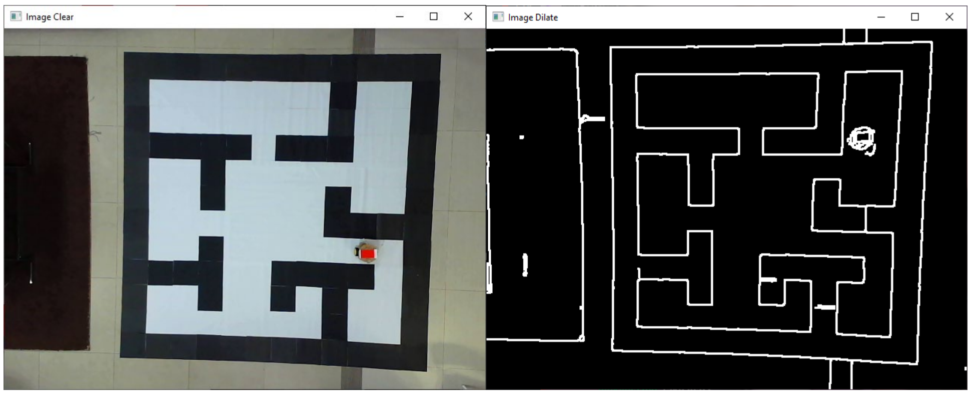


Figure 4.5: Result of Image Clear and Image Dilate from imageTotal()

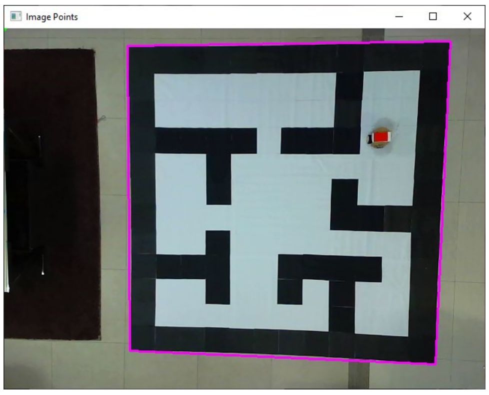


Figure 4.6: Rectangle found and drawn by getContours()

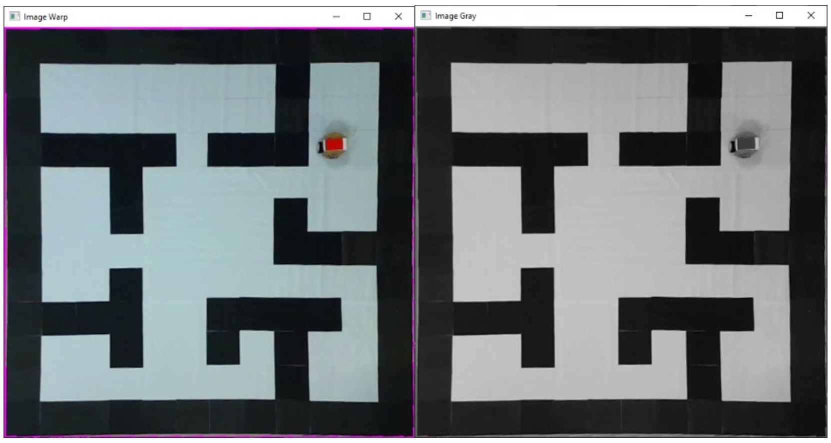


Figure 4.7: Warped image by getWarp() and the grayscale of the image

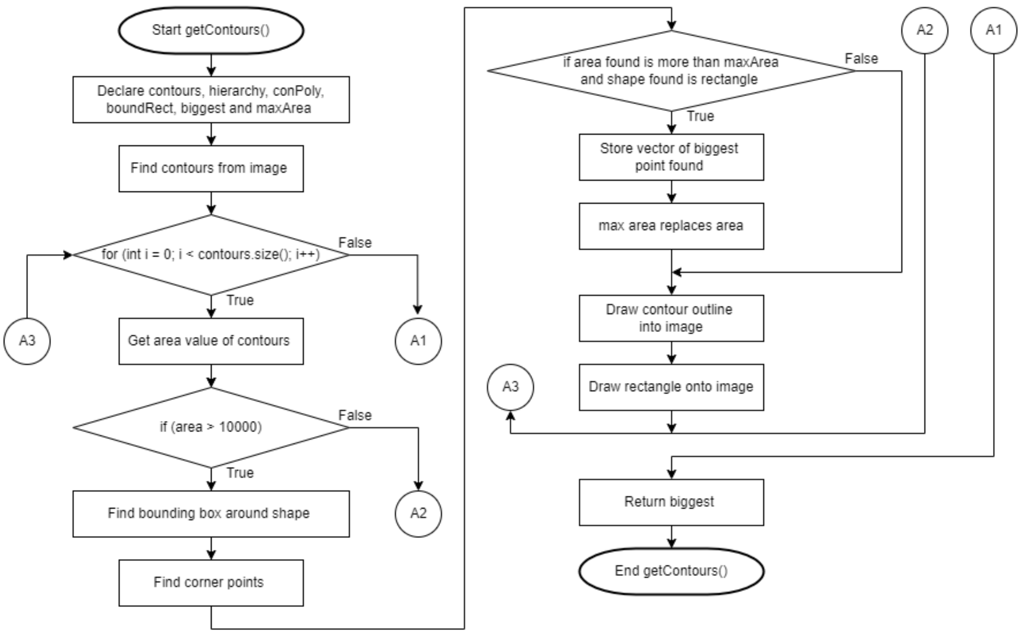

Start getContours()
Declare contours, hierarchy, conPoly, boundRect, biggest and maxArea
Find contours from image
for (int i = 0; i < contours.size(); i++)
False
True
A3
Get area value of contours
A1
if (area > 10000)
False
True
Find bounding box around shape
A2
Find corner points
if area found is more than maxArea and shape found is rectangle
False
True
Store vector of biggest point found
max area replaces area
Draw contour outline into image
Draw rectangle onto image
A3
A2
A1
Return biggest
End getContours()

Figure 4.8: Flowchart of getContours() function

Figure 4.8 shows the flowchart of getContours(), and the function's outcome can be referred to in figure 4.6. The operation started by declaring the local variables required for the process. Contours are found from the binary image using the findContours function. The for loop in the flowchart is used to check through all the found contours and get the area values of the rectangles. If the area of the found contour is less than or equals 10000, the contour will be removed from the results. The for loop process will be repeated to continue checking the next contour. The leftover contour's bounding box and corner points are found for further operations. If the area of the found contour is more than the maxArea and is a rectangle, the vector of the biggest point discovered is stored and let maxArea equals the area. A contour outline and a rectangle are drawn onto the image. As the area found constantly replaces the maxArea, if the value is more significant, the maxArea will eventually equal the most extensive area found in the image. While all the found contours are checked through by the for loop, the vector of the biggest point located is returned.

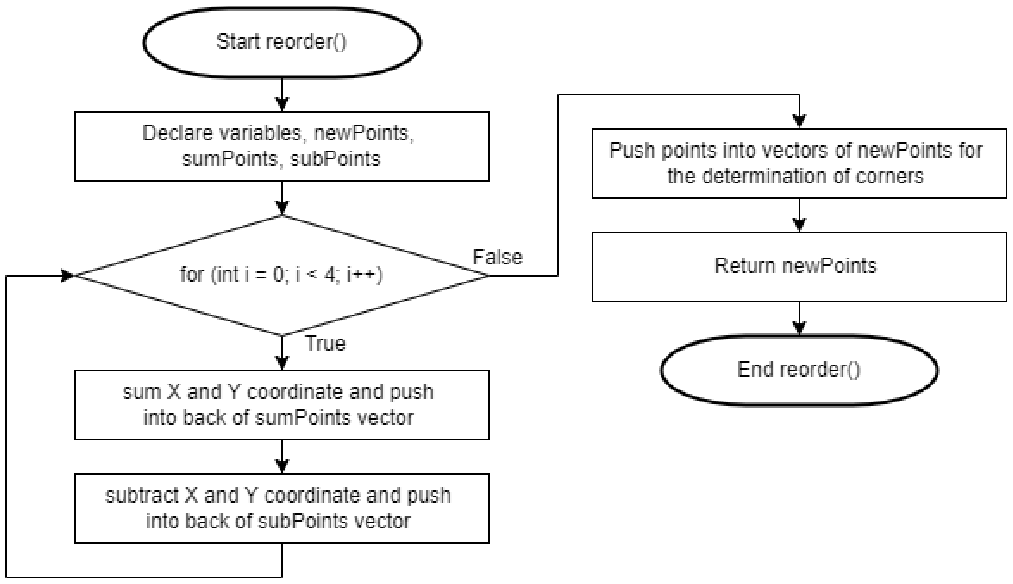

Start reorder()
Declare variables, newPoints, sumPoints, subPoints
for (int i = 0; i < 4; i++)
False
True
sum X and Y coordinate and push into back of sumPoints vector
subtract X and Y coordinate and push into back of subPoints vector
Push points into vectors of newPoints for the determination of corners
Return newPoints
End reorder()

Figure 4.9: Flowchart of reorder() function

Local variables are declared to initialise the reorder() function. ‘For loop’ in the flowchart shown in figure 4.9 checks through the four corner points found in getContours(). When the condition is true, the summation of X and Y coordinates is recorded and stored in the sumPoints vector, and the difference of X and Y coordinates is saved in the subPoints vector. The process repeats until values for all four points are documented. The vector of sumPoints and subPoints are pushed into newPoints to determine corners with the correct numbering for further processing at the following function. The steps can be visualised in figure 4.10. Assuming the coordinate of the most significant contour found is as follows, the box on the left shows the result of sumPoints, the lowest value in the sumPoints will be labelled as ‘0’, and the highest value will be labelled as ‘3’. ‘0’ represents the zero point of the square, and ‘3’ is the point with the maximum width and height of the square. Another two points in the square are labelled with the number ‘1’ and ‘2’, the highest value for subPoints will be ‘1’, and the lowest value is marked with the number ‘2’. newPoints, points ‘1’, ‘2’, ‘3’, and ‘4’, are returned as a vector.

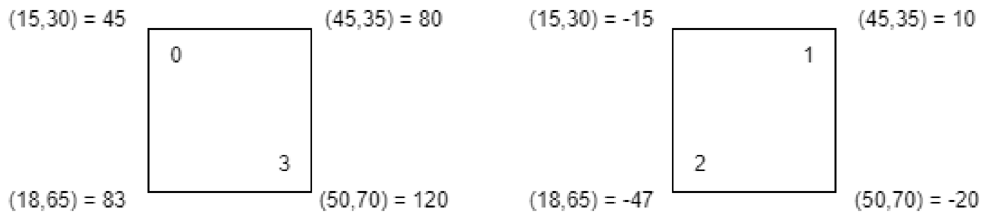


Figure 4.10: Visualisation of technique to get newPoints

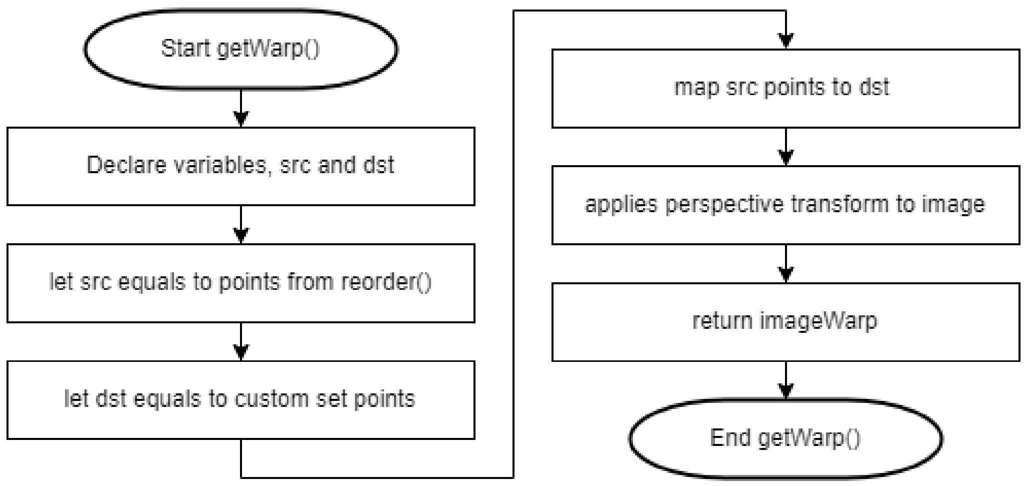

Start getWarp()
Declare variables, src and dst
let src equals to points from reorder()
let dst equals to custom set points
map src points to dst
applies perspective transform to image
return imageWarp
End getWarp()

Figure 4.11: Flowchart of getWarp() function

Two variables are declared in the getWarp() function. The variable src will equal points acquired from reorder() as newPoints, and dst will be equivalent to custom set points for warping. The points in src are mapped with points in dst, and perspective transform is applied to the image. The image is warped and returned as imageWarp. The concept of mapping between src and dst is shown in figure 4.12. The warped result can be found in figure 4.7.

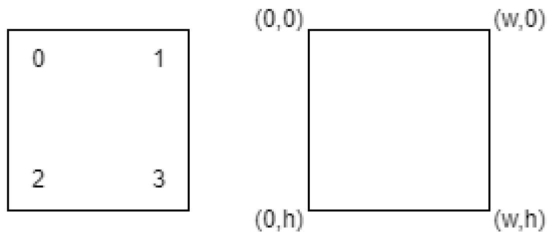


Figure 4.12: Visualisation of technique to map src and dst

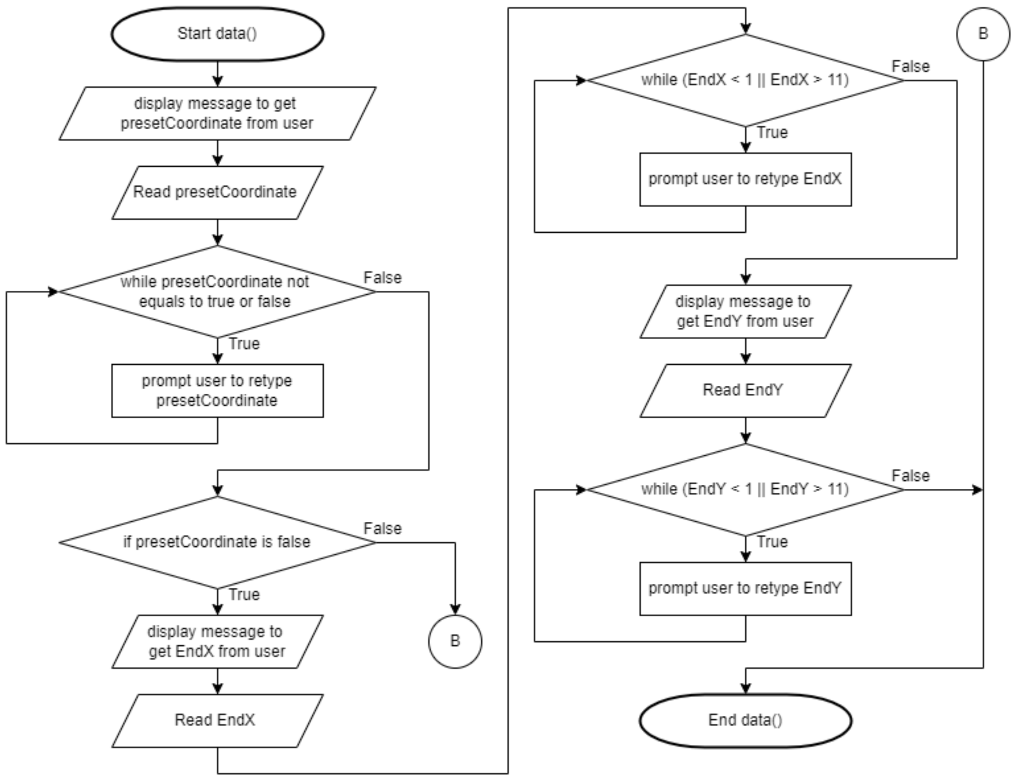

Start data()
display message to get presetCoordinate from user
Read presetCoordinate
while presetCoordinate not equals to true or false
False
True
prompt user to retype presetCoordinate
if presetCoordinate is false
False
True
B
display message to get EndX from user
Read EndX
while (EndX < 1 || EndX > 11)
False
True
prompt user to retype EndX
display message to get EndY from user
Read EndY
while (EndY < 1 || EndY > 11)
False
True
prompt user to retype EndY
B
End data()

Figure 4.13: Flowchart of data() function

At the beginning of the data() function, a message will be displayed to read the user input for presetCoordinate. The variable presetCoordinate is to ask the user if preset coordinates are to be used or if the user will provide a custom coordinate. While the preset coordinate is invalid, the user must retype the option again. When the presetCoordinate input is validated, a condition is used to check the user input. If the “presetCoordinate is false” statement is false, the function ends. If presetCoordinate is true, a message is displayed, and EndX is read. If EndX does not reach the requirement, the user will have to retype the value until it matches the condition. The exact process is repeated for EndY. When both EndX and EndY are recorded appropriately, the data() function will end.

A comparison of using preset coordinates and a custom coordinate is shown in figure 4.14 generated by OpenGL, where more information will be explained in lineInitiate() and unit() functions. The picture on the left shows the end node with a custom coordinate, and the image on the right shows the end nodes with preset coordinates. There will be multiple end nodes when using the preset coordinates and only one end node when using the custom coordinate.

The red blocks represent the obstacles, and the blue blocks represent the end coordinate. The purpose of having numerous end nodes is to relate the project to factory usage. Most factories have preset coordinates for their mobile robot to deliver or retrieve items. Custom coordinates are used for specific coordinate deliveries.

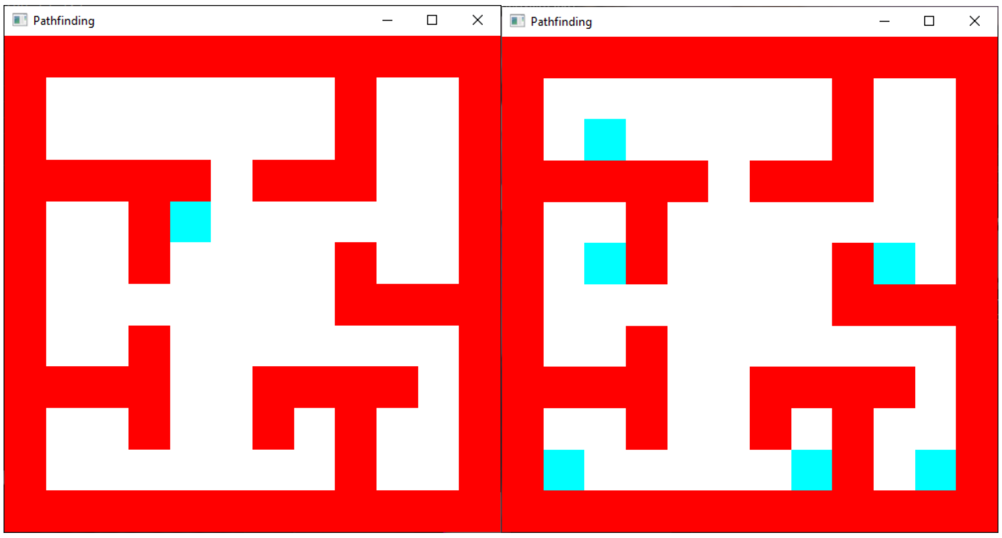


Figure 4.14: Comparison of a custom coordinate with preset coordinates

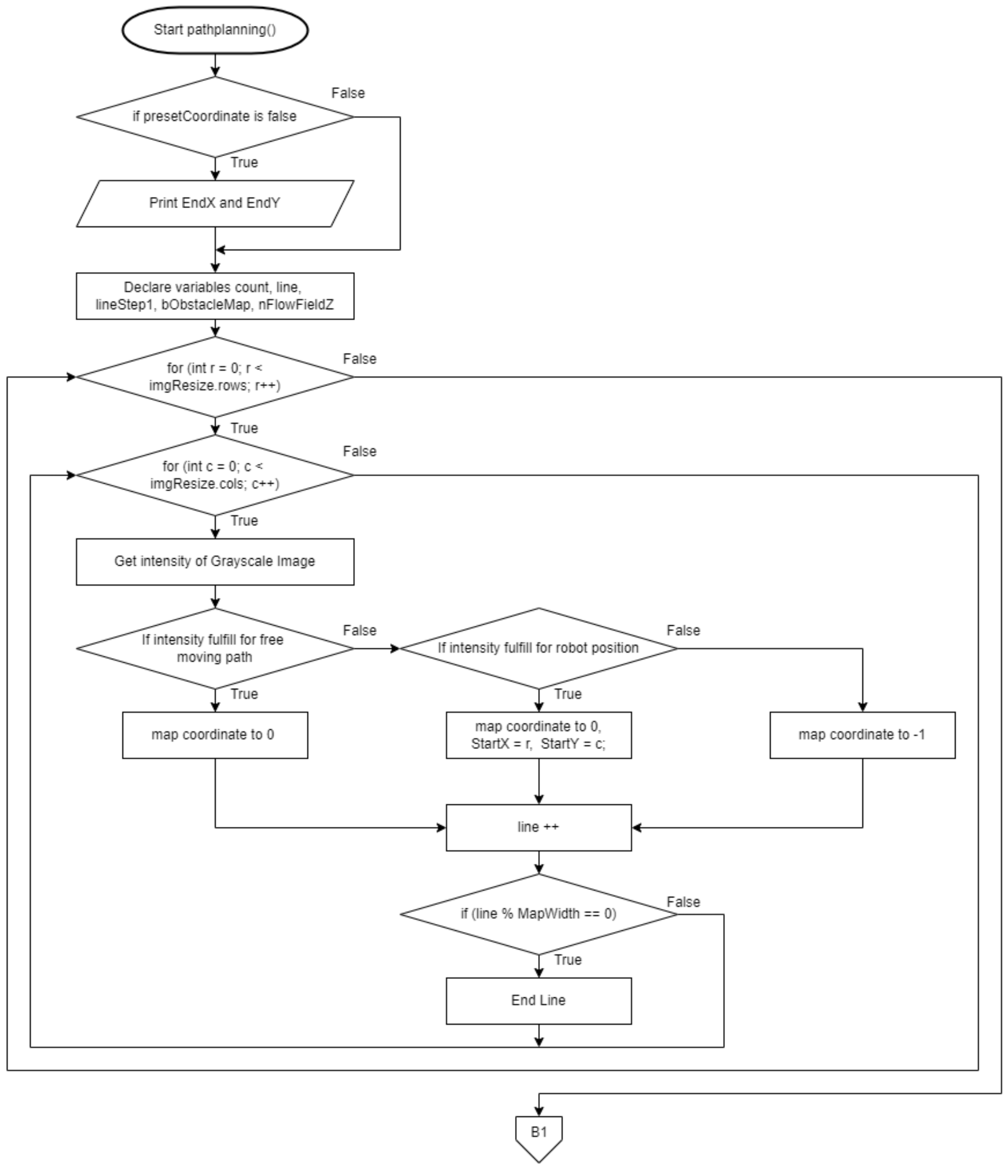
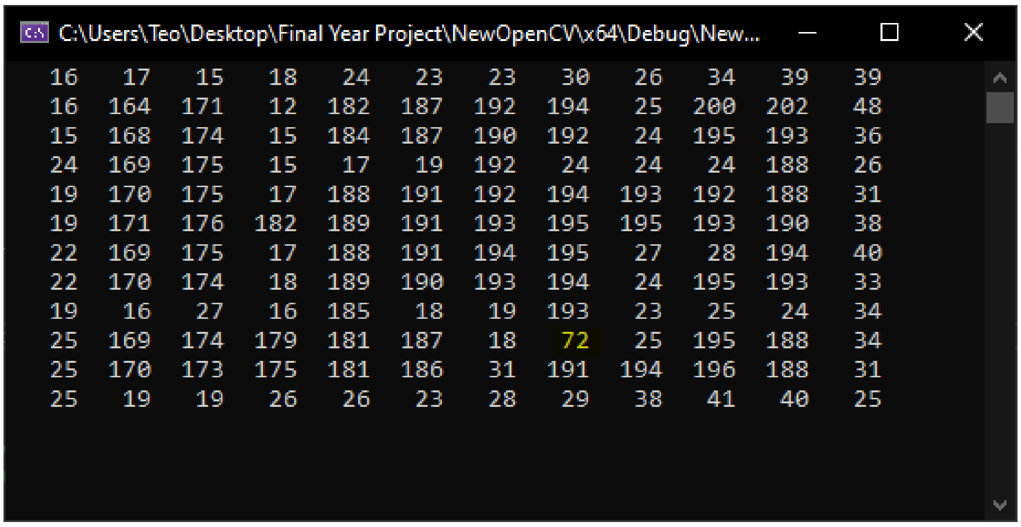

Start pathplanning()
if presetCoordinate is false
False
True
Print EndX and EndY
Declare variables count, line, lineStep1, bObstacleMap, nFlowFieldZ
for (int r = 0; r < imgResize.rows; r++)
False
True
for (int c = 0; c < imgResize.cols; c++)
False
True
Get intensity of Grayscale Image
If intensity fulfill for free moving path
False
True
If intensity fulfill for robot position
False
True
map coordinate to 0
map coordinate to 0, StartX = r, StartY = c;
map coordinate to -1
line ++
if (line % MapWidth == 0)
False
True
End Line
B1

Figure 4.15: Flowchart of pathplanning() function Part A

EndX and EndY will be printed at the start of the pathplanning() function if the ‘presetCoordinate is false’ statement is true. The printing of EndX and EndY is for the custom coordinate only. Variables are declared after the decision. Both the for loops are to repeat the process until all the coordinates from the environment are visited. While visiting the coordinates, the intensity of the grayscale image is recorded to identify the free-moving paths, the robot’s initial position, and obstacles. The free-moving paths and robot’s initial position are mapped with ‘0’ in the two-dimensional array, and obstacles are mapped to ‘-1’. After mapping the robot’s initial position in the two-dimensional array, StartX equals row and StartY equals column. This step records the robot’s initial position for path planning purposes. line++ functions to arrange the two-dimensional array into a manageable and readable style, making it easier to analyse. If the line modulus of the MapWidth is equal to zero, data will be printed on the following line instead. The result can be seen in figure 4.16. This figure shows the printed data of the grayscale intensity of the coordinate in the environment. The value highlighted is the robot’s initial position, which has a grayscale intensity between 60 and 145. Values which fall between 150 to 255 are the accessible moving paths. Other small values are the obstacles. The grayscale intensity is essential to find the best value for further programming purposes. Finding a suitable grayscale intensity enables the camera to work in different lighting conditions. A test is conducted to determine the program’s workability in various lighting settings. The conditions can be seen in Figures 4.17 to 4.20.



```
C:\Users\Teo\Desktop\Final Year Project\NewOpenCV\x64\Debug\New...
   16    17    15    18    24    23    23    30    26    34    39    39
   16   164   171    12   182   187   192   194    25   200   202    48
   15   168   174    15   184   187   190   192    24   195   193    36
   24   169   175    15    17    19   192    24    24    24   188    26
   19   170   175    17   188   191   192   194   193   192   188    31
   19   171   176   182   189   191   193   195   195   193   190    38
   22   169   175    17   188   191   194   195    27    28   194    40
   22   170   174    18   189   190   193   194    24   195   193    33
   19    16    27    16   185    18    19   193    23    25    24    34
   25   169   174   179   181   187    18    72    25   195   188    34
   25   170   173   175   181   186    31   191   194   196   188    31
   25    19    19    26    26    23    28    29    38    41    40    25
```

Figure 4.16: Grayscale intensity at different coordinates

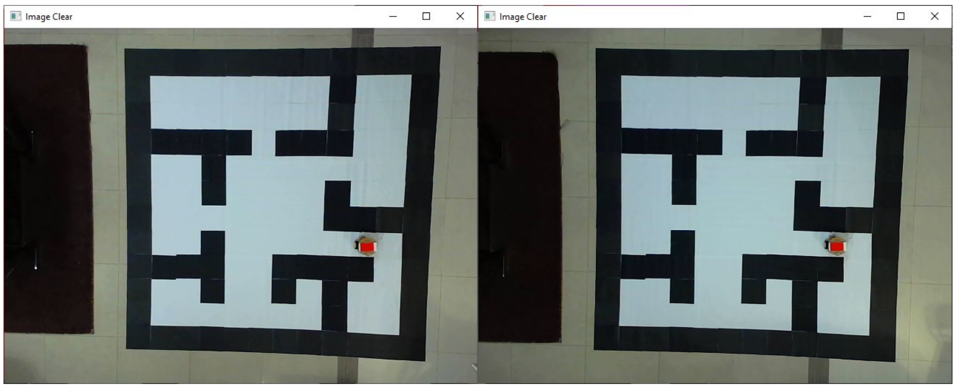


Figure 4.17: Lighting conditions 1 and 2

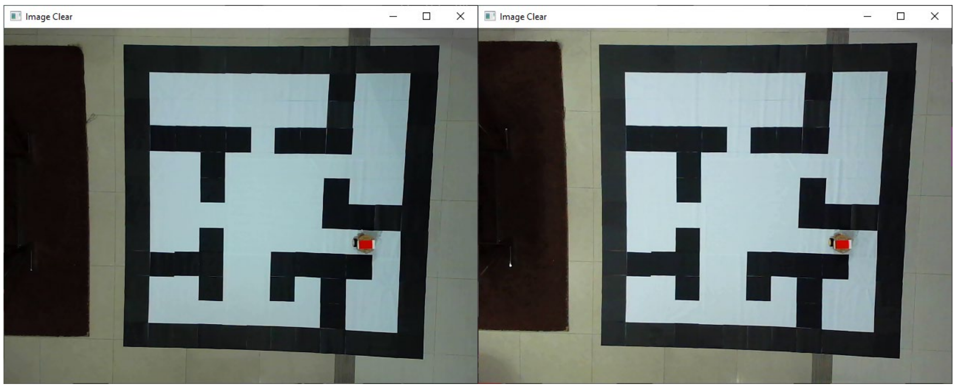


Figure 4.18: Lighting conditions 3 and 4

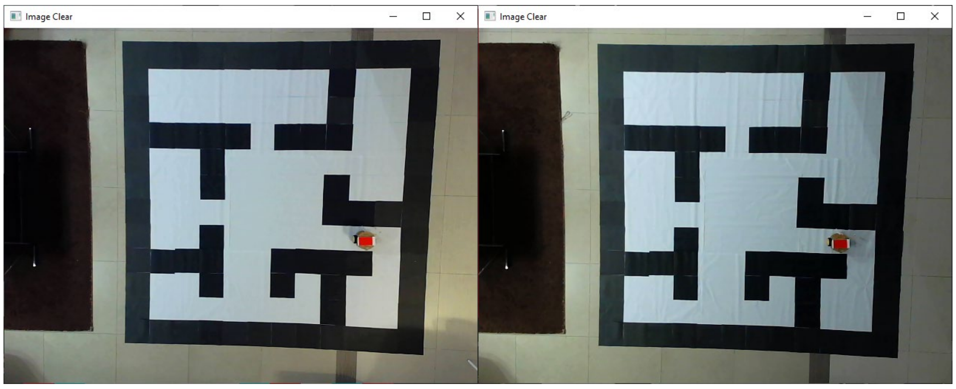


Figure 4.19: Lighting conditions 5 and 6

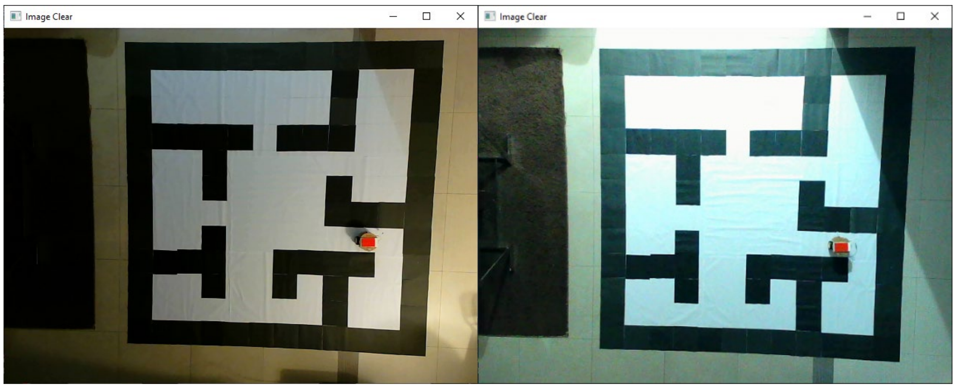


Figure 4.20: Lighting conditions 7 and 8

Lighting conditions 1, 2, and 4 can achieve results with the grayscale intensity configuration in the program. Minor tweaking is needed for lighting conditions 3 and 5, as lighting is not spread evenly onto the test environment. The issue can be solved easily by slight tweaking the program’s values. Lighting conditions 6 and 7 require heavy adjustment as both images are in a low light condition. The light source is far from the environment and only comes from a single direction. A few lines of code to offset the values for further coordinates from the light source can be done to fix the issue of low lighting. Lighting condition 8 is under poor light condition. The only light source is from the outdoor environment. 30% of the results are false positives. Much work is needed to be done to use lighting condition 8 with the program. This test checks the robustness of the code in different lighting, and the result is excellent. Most factories have fixed lighting that is bright enough to light the environment, where the program does not require high robustness. Minor tweaking is only needed when the camera is set up in a space where an obstacle blocks a part of the light source.

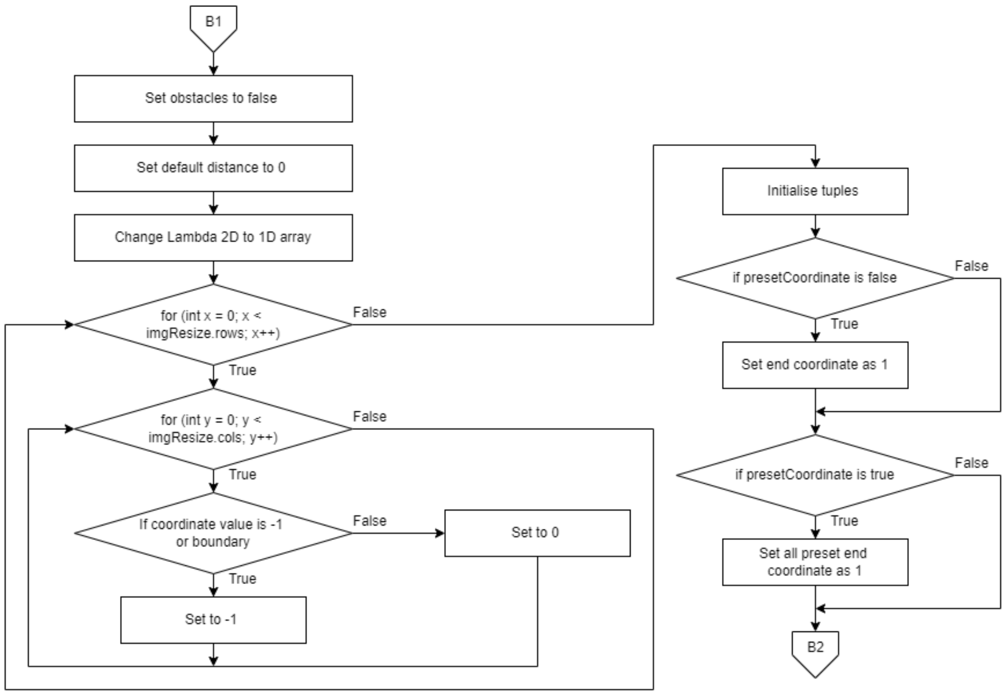

B1
Set obstacles to false
Set default distance to 0
Change Lambda 2D to 1D array
for (int x = 0; x < imgResize.rows; x++)
False
True
for (int y = 0; y < imgResize.cols; y++)
False
True
If coordinate value is -1 or boundary
False
Set to 0
True
Set to -1
Initialise tuples
if presetCoordinate is false
False
True
Set end coordinate as 1
if presetCoordinate is true
False
True
Set all preset end coordinate as 1
B2

Figure 4.21: Flowchart of pathplanning() function Part B

Obstacles are set to false after all the coordinates are mapped to the two-dimensional array in the previous step. Obstacles set to false means that there are no obstacles. The default distance is also set to the value of zero. The two-dimensional array is changed to a one-dimensional array. Two for loops are used to ensure all the coordinates in the environment are being visited. While visiting a coordinate, if the coordinate has the value of '-1' or it is the boundary of the environment, the distance value (nFlowFieldZ) of the coordinate in the environment is set to '-1'. The coordinates that do not fulfil the above requirements will have nFlowFieldZ set to a value of '0'. This part ensures that all coordinate status is saved correctly in the integer pointer, nFlowFieldZ, for the implementation of the search algorithm.

Tuples are initialized by ‘std::list<std::tuple<int, int, int>> nodes, the three integer values represent x, y, and distance. The next process checks the status of presetCoordinates to determine the number of endpoints needed for the environment. If preset coordinates are selected, all the coordinates with the preset value will get the value of ‘1’ in the distance value. If a custom coordinate is chosen, the user input end coordinate will have the value of ‘1’ in the distance value. At this moment, we know that the obstacles are labelled with ‘-1’, free-moving paths are labelled with ‘0’ and end coordinates are labelled with ‘1’.

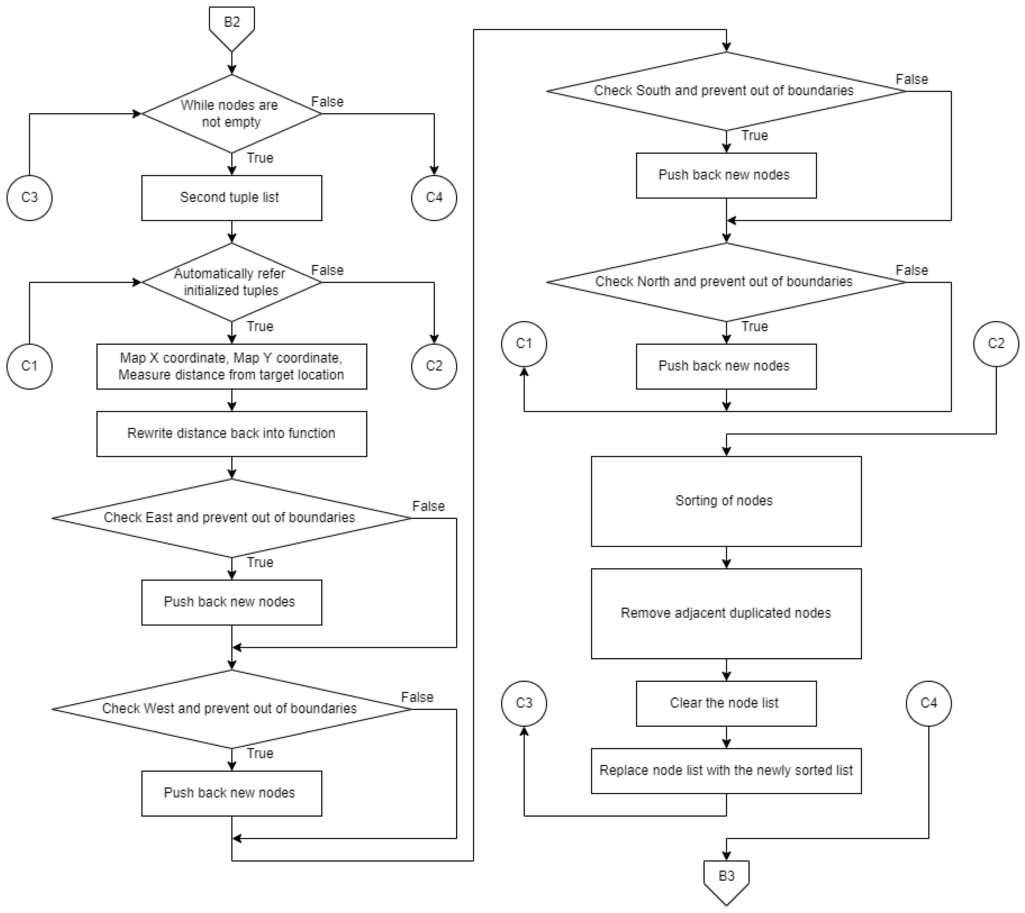

B2
While nodes are not empty
False
True
C3
Second tuple list
C4
Automatically refer initialized tuples
False
True
C1
Map X coordinate, Map Y coordinate, Measure distance from target location
C2
Rewrite distance back into function
Check East and prevent out of boundaries
False
True
Push back new nodes
Check West and prevent out of boundaries
False
True
Push back new nodes
Check South and prevent out of boundaries
False
True
Push back new nodes
Check North and prevent out of boundaries
False
True
C1
Push back new nodes
C2
Sorting of nodes
Remove adjacent duplicated nodes
C3
Clear the node list
C4
Replace node list with the newly sorted list
B3

Figure 4.22: Flowchart of pathplanning() function Part C

Nodes are developed until they are no nodes left undiscovered. Hence a while loop is used to repeat the process. Iterating through nodes will create newly discovered nodes. A second tuple list is implemented to prevent contamination of nodes onto the created tuple list. This prevents the data in the first tuple from being affected throughout the iteration. The second tuple is initialized by std::list<std::tuple<int, int, int>> new_nodes, which differentiates from the first tuple list. While iterating through discovered nodes, neighbouring nodes that are undiscovered or empty will be added to the 'new_nodes' list.

Auto function and a for loop automatically refer to the initialised tuples. X coordinate, Y coordinate, and the distance from the end node are mapped. When a node is added to the new nodes list, the distance value from the end note increases by one. The distance, d, is then rewritten to its initial function, nFlowFieldZ. For the next step, any unmarked neighbour nodes are added by checking the directions of movement for the nodes. Hence, the four main directions are reviewed, north, south, east and west. In verifying the four directions, it is vital to prevent the checking node from going out of bounds. Hence a boundary check is applied. The whole process repeats until the end of the list.

The previous steps create multiple nodes for one location. The list's repeated must be eliminated, or the process will never be complete. The nodes are first sorted, which will rearrange the node that is identical together. As an example, if the nodes are A, B, C, B, C, D, E, F, E, F, the nodes will be sorted accordingly to their order, and the result will be A, B, B, C, C, D, E, E, F, F. This could be done by passing in two arguments if the argument is 'true', swapping takes place. Else, the process is ignored. After sorting, the duplicated nodes have to be removed. To accomplish this step, two arguments are compared if they are the same. This will bring the result from A, B, B, C, C, D, E, E, F, F to A, B, C, D, E, F. The discovered nodes are processed, and the list is cleared for further processing of the next iteration. The node list is substituted with a new sorted list. The end of this procedure brings the process back to the while loop to check if the nodes are not empty. While all nodes are discovered, the following method will continue.

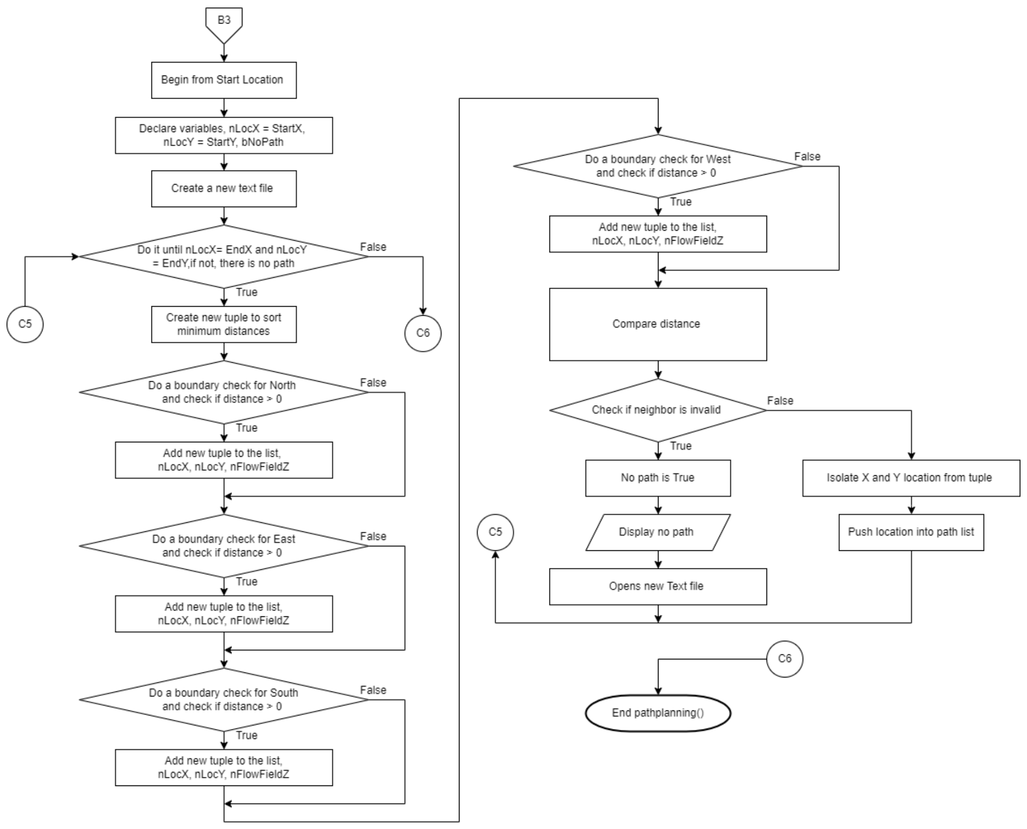

B3
Begin from Start Location
Declare variables, nLocX = StartX, nLocY = StartY, bNoPath
Create a new text file
Do it until nLocX= EndX and nLocY = EndY,if not, there is no path
False
True
C5
C6
Create new tuple to sort minimum distances
Do a boundary check for North and check if distance > 0
Add new tuple to the list, nLocX, nLocY, nFlowFieldZ
Do a boundary check for East and check if distance > 0
Do a boundary check for South and check if distance > 0
Do a boundary check for West and check if distance > 0
Compare distance
Check if neighbor is invalid
No path is True
Display no path
Opens new Text file
Isolate X and Y location from tuple
Push location into path list
End pathplanning()

Figure 4.23: Flowchart of pathplanning() function Part D

Paths have been found from Part C based on the height map, but it is not usable yet as no physical path is drawn to connect the start and end coordinates. A path is needed to be created from the extraction of the height map. Overall, the way of accomplishing this is by taking off from the start location and constructing a path of nodes up to the target location. While processing on the specific node, the neighbour's distance score is compared, and the one with the lowest value is selected.

The start coordinate is pushed into the path list at the beginning of part D, coming from B3. Variables are declared as shown in figure 4.23. A boolean function is implemented to determine if no path is found towards the end location. The boolean to check the presence of the path is set to false on default. A new text file is created for the motor commands, which will be used later for the ESP32.

While loop is executed here to ensure that the parameter 'nLocX' and 'nLocY' are equal to 'EndX' and 'EndY', this is for the case of custom coordinates. For preset coordinates, the working principle is the same, but all the preset coordinates have to be specified in this while loop for validation purposes. At the same time, the boolean function has to be true.

A new tuple list is created to sort the minimum distance in the list. A connectivity test is used, where we can choose from 4-way or 8-way connectivity. In this program, 4-way connectivity will be used as 8-way connectivity might cause a robot to crash with obstacles at turning corners. 8-way connectivity will shorten the travel time of the navigation robot, and improvements in the algorithm can be made to solve the crashing issue. For the testing of the pathfinding algorithm, 8-way connectivity is not as necessary as it can be introduced to the program without difficulty and will not affect the program. Taking the south direction as an example, a new tuple will only be added to the list when the current location of the Y coordinate, nLocY, is lesser than the map height. This is used to prevent the connectivity test from exceeding the borders. Besides border checking, the presence of distance value is also essential. While the condition is reached, a new tuple is added to the list. The tuple format remains the same as before, where the three integers are coordinate X, coordinate Y, and the distance, respectively.

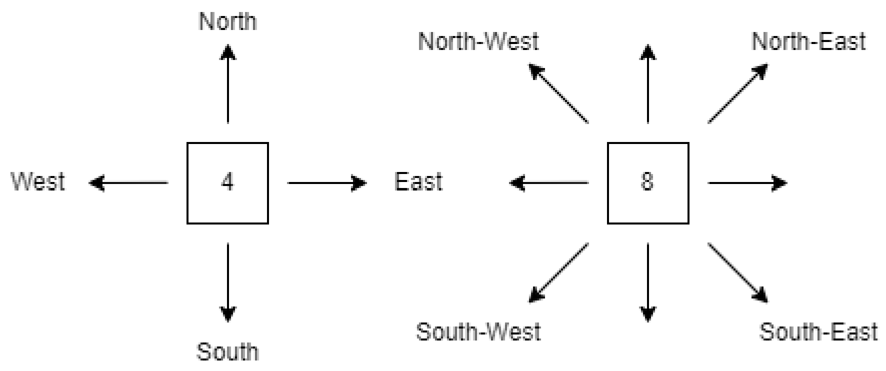


Figure 4.24: 4-way connectivity and 8-way connectivity

After all the tuples are stored in the list, 'listNeighbours', the neighbours are sorted based on distance. The neighbour with the lowest distance will be placed at the beginning of the list. If the list is empty, it indicates that the algorithm finds no path, 'bNoPath'is being set to true, and the while loop stops immediately. A message will be sent to the command window, and nothing will be written into the motor command text file. Else, the X and Y location is isolated from the tuple. The location is pushed into the path list. The new 'nLocX' and 'nLocY' will continue looping through the while loop until the loop condition is false.

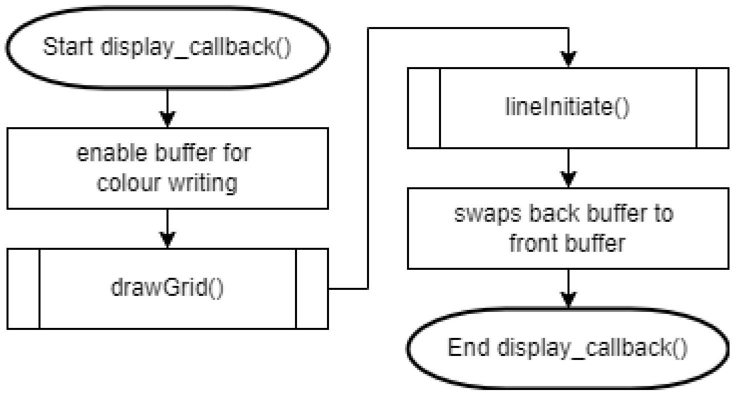


Figure 4.25: Flowchart of display_callback() function

This function enables the buffer for colour writing and works with glutDisplayFunc, as shown in figure 4.25. The drawGrid() function draws grids of the entire enclosed environment to OpenGL. The process, lineinitiate(), joins a line from the starting coordinate to the ending coordinate, which signifies the path found. The function glutSwapBuffers()is used to swap the back buffer to the front buffer.

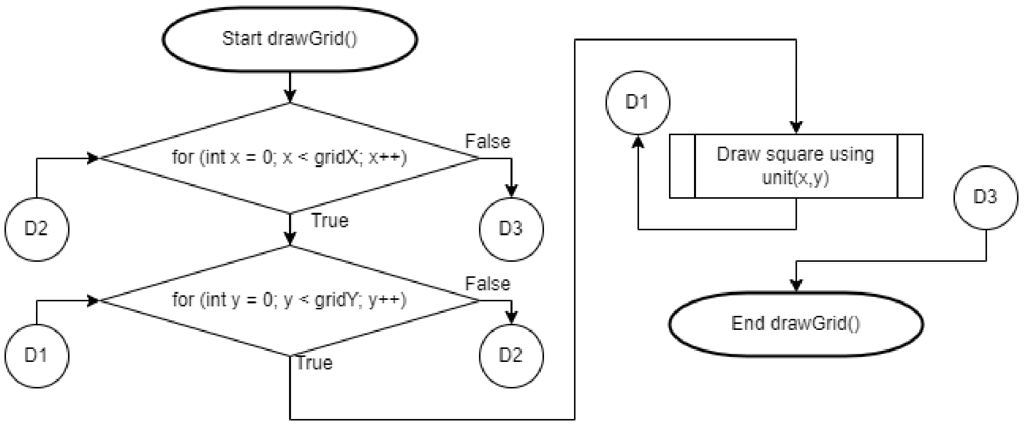

Start drawGrid()
for (int x = 0; x < gridX; x++)
False
True
D2
D3
for (int y = 0; y < gridY; y++)
False
True
D1
D2
D1
Draw square using unit(x,y)
D3
End drawGrid()

Figure 4.26: Flowchart of drawGrid() function

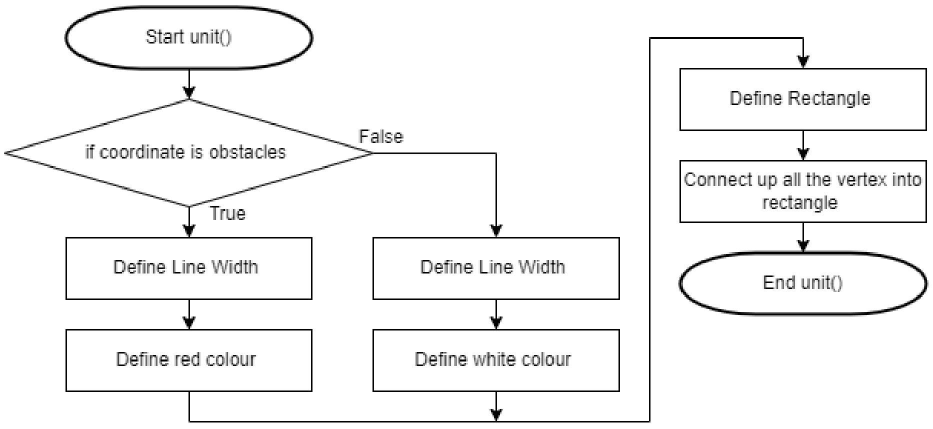


Figure 4.27: Flowchart of unit() function

The drawGrid() functions to iterate through all the rows and columns of the enclosed environment. The square is drawn by the following function, unit(), shown in figure 4.27. While iterating through the locations on the map, if the specific coordinate is an obstacle, red colour is selected. Else, all the others are accessible moving paths. White colour is defined. After choosing a colour, a rectangle is defined, and all the vertex in the rectangle is connected. The process can be visualised in figure 4.28.

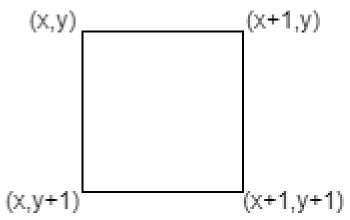


Figure 4.28: Four vertexes to form a rectangle

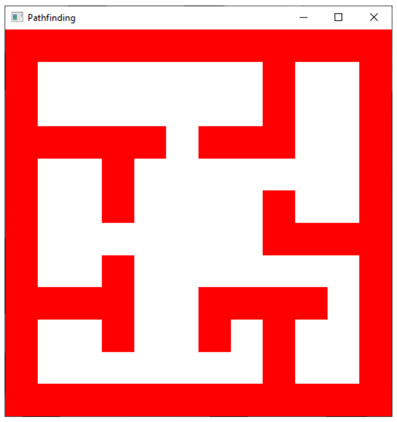


Figure 4.29: Simulation of accessible moving paths and obstacles

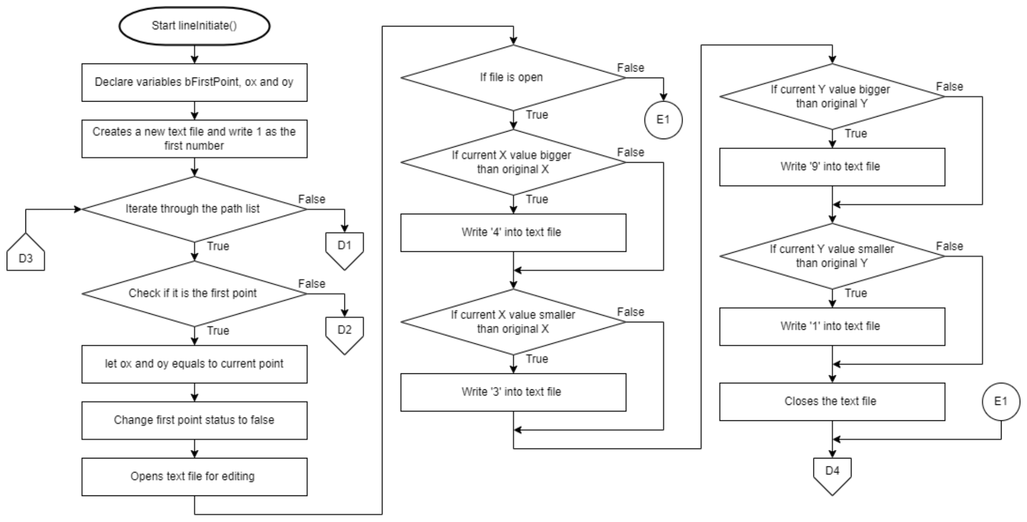

Start lineInitiate()
Declare variables bFirstPoint, ox and oy
Creates a new text file and write 1 as the first number
Iterate through the path list
False
True
D1
D3
Check if it is the first point
False
True
D2
let ox and oy equals to current point
Change first point status to false
Opens text file for editing
If file is open
False
True
E1
If current X value bigger than original X
False
True
Write '4' into text file
If current X value smaller than original X
False
True
Write '3' into text file
If current Y value bigger than original Y
False
True
Write '9' into text file
If current Y value smaller than original Y
False
True
Write '1' into text file
Closes the text file
E1
D4

Figure 4.30: Flowchart of lineInitiate() function Part A

The function lineInitiate() is applied to join the path from the start coordinate to the end coordinate with a line. The bFirstPoint boolean is set to true initially as the progress started from the first point. After the first point, the boolean will be converted to false, and the remaining points can be processed. Variables ox and oy are stored with the initial X and Y coordinates of the environment. The value of ox and oy will change with ‘a.first’ and ‘a.second’ for the remaining points until the end coordinate.

A text file is created for the motor command, and the value ‘1’ is written to the first value in the file. Value ‘1’ is needed as the navigation robot will always face the north direction on start-up. Iteration is done through the path list, and a condition is used to check if the coordinate is the first point. If the condition is true, variables ox and oy equal the current point. The first point status, bFirstPoint, is set to false. The output text file, the motor command file, is open for editing.

Four conditions are checked when the file is available. It functions to verify the direction of movement. The visualisation of the process can be seen in figure 4.31. The numbers ‘1’, ‘3’, ‘4’, and ‘9’ signifies the movement towards North, West, East, and South direction. The numbers are calculated for easy processing in ESP32. For example, the motor command will always start with the value ‘1’, as discussed above. If the current Y value is more significant than the original Y, ‘9’ is written into the motor command text file. The text file starts with two numbers, ‘19’. Visualisation of the numbers can be seen in figure 4.32. After all the conditions are validated, the text file will close.

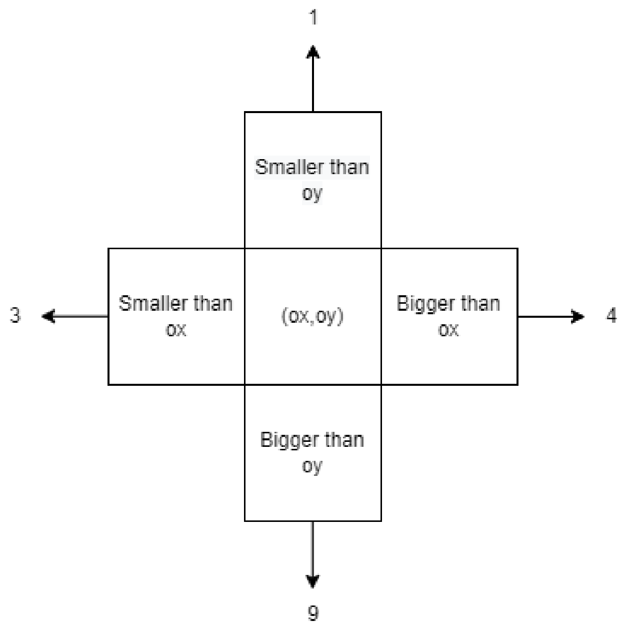
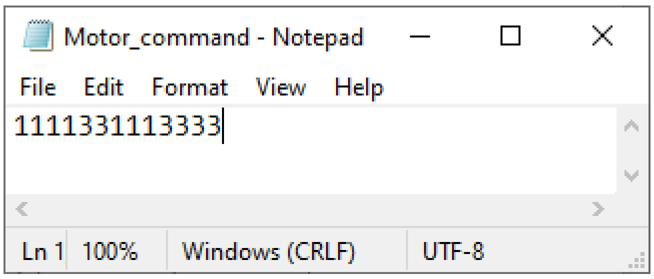


Figure 4.31: Four execution conditions and motor command numbers direction



```
1111331113333
```

Figure 4.32: Example of a motor command text file

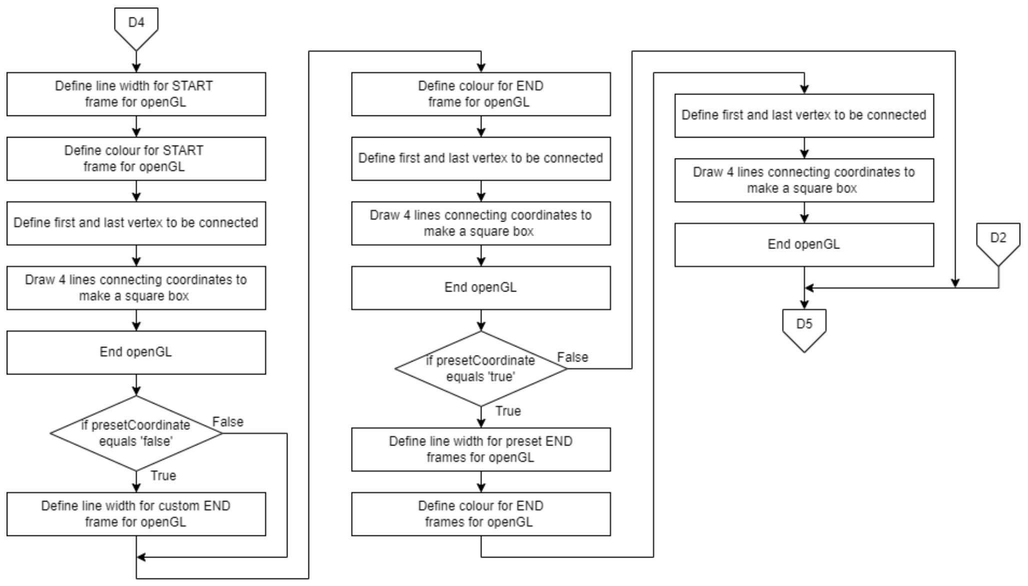

D4
Define line width for START frame for openGL
Define colour for START frame for openGL
Define first and last vertex to be connected
Draw 4 lines connecting coordinates to make a square box
End openGL
if presetCoordinate equals 'false'
False
True
Define line width for custom END frame for openGL
Define colour for END frame for openGL
Define first and last vertex to be connected
Draw 4 lines connecting coordinates to make a square box
End openGL
if presetCoordinate equals 'true'
False
True
Define line width for preset END frames for openGL
Define colour for END frames for openGL
Define first and last vertex to be connected
Draw 4 lines connecting coordinates to make a square box
End openGL
D2
D5

Figure 4.33: Flowchart of lineInitiate() function Part B

After the process from D4 in figure 4.30, the algorithm is still in the first point, which is the starting location. Based on figure 4.33, the start coordinate and end coordinate are not implemented yet. Since the process is still in the start coordinate, the starting frame can be drawn onto the simulation. The approach here will be similar to figure 4.27's unit() function. The line width and colour are defined, and the start frame is set to green. The visualisation in figure 4.28 is recreated here. The identical method is used for the end location. As the end location is not in the current iteration, the coordinate is called directly from the 'EndX' and 'EndY' global variables. A similar mechanism is used to generate blue boxes for the end coordinate. There will only be one end frame developed when there is no pre-set coordinate, only the user input coordinate. If the pre-set coordinate is true, multiple end frames will be drawn. The simulation can be seen in figure 4.14. While finished executing the codes in lineInitiate() function Part B, everything required for the first point and simulation is completed. This part will not be revisited until the restart of the program.

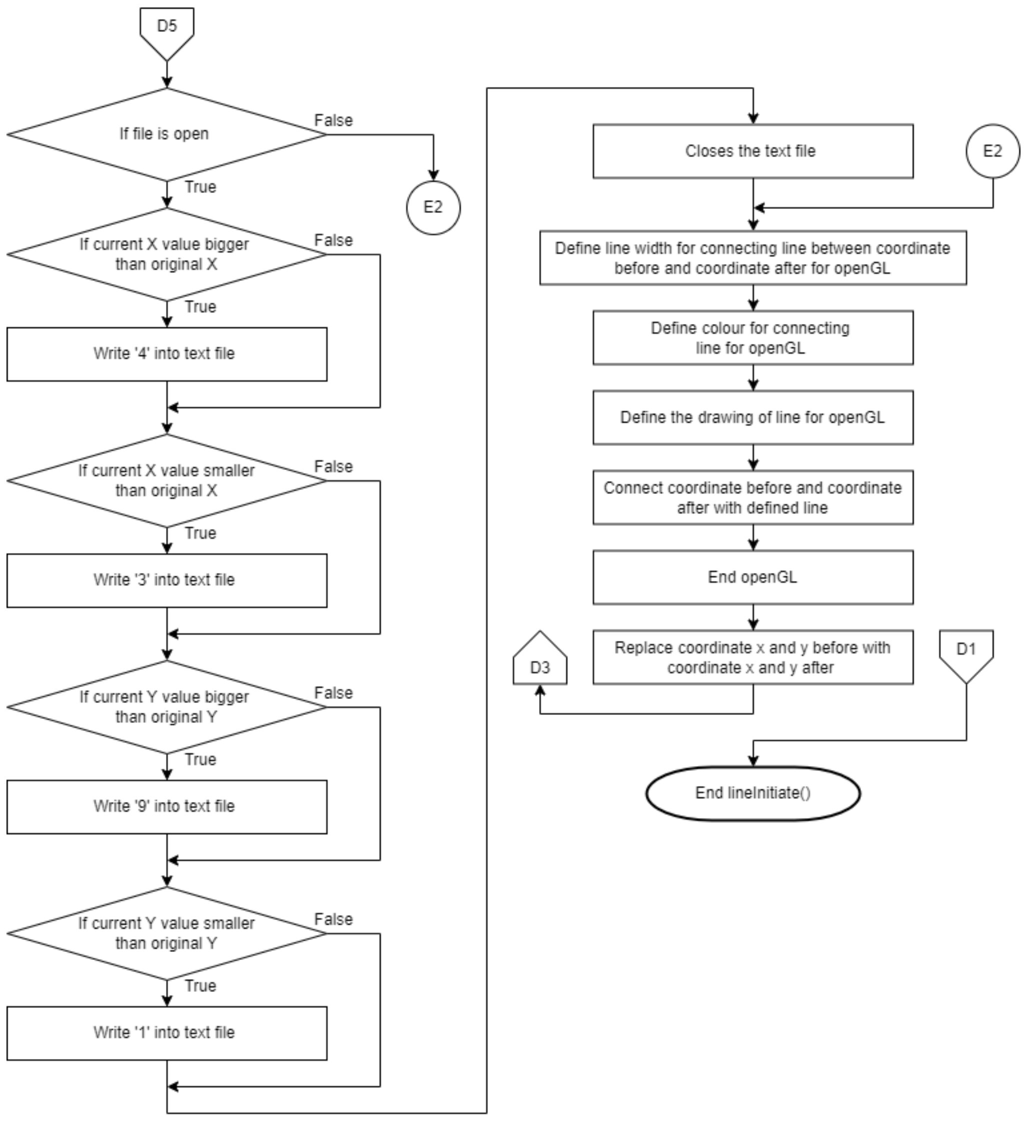

D5
If file is open
False
True
E2
If current X value bigger than original X
Write '4' into text file
If current X value smaller than original X
Write '3' into text file
If current Y value bigger than original Y
Write '9' into text file
If current Y value smaller than original Y
Write '1' into text file
Closes the text file
E2
Define line width for connecting line between coordinate before and coordinate after for openGL
Define colour for connecting line for openGL
Define the drawing of line for openGL
Connect coordinate before and coordinate after with defined line
End openGL
D3
Replace coordinate x and y before with coordinate x and y after
D1
End lineInitiate()

Figure 4.34: Flowchart of lineInitiate() function Part C

Part C of the lineInitiate() function executes from the second point until the last. It works similarly to Part A, as shown in figure 4.30. All the steps from D5 to the end of E2 are the same. After E2, the line width and colour are defined as black. The line is defined in OpenGL and drawn from the coordinate before to the coordinate after. OpenGL is ended, and the coordinates x and y before are replaced with the ones after. The process will continue iterating until the end coordinate. When the line is drawn up to the end coordinate, the lineInitiate() function ends.

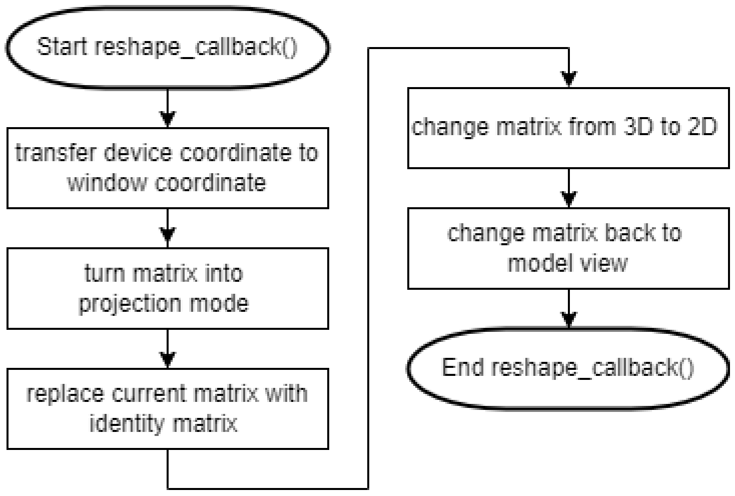

Start reshape_callback()
transfer device coordinate to window coordinate
turn matrix into projection mode
replace current matrix with identity matrix
change matrix from 3D to 2D
change matrix back to model view
End reshape_callback()

Figure 4.35: Flowchart of reshape_callback()

The function in figure 4.35 is mandatory for OpenGL. The reshape_callback() function has a few stages, where the first step is to shift the device coordinate to the window coordinate. Matrix mode is then transformed into projection mode mainly to change the 3-dimensional view to a 2-dimensional view. The current matrix is replaced with an identity matrix before switching to an orthogonal projection.

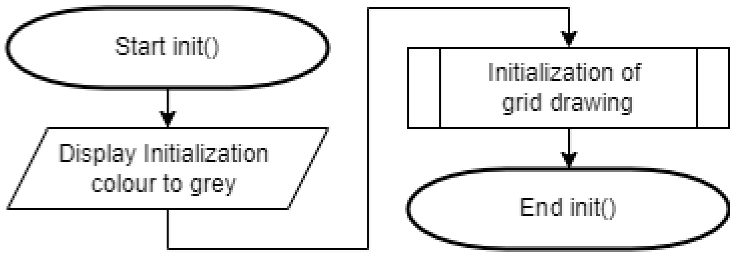

Start init()
Display Initialization colour to grey
Initialization of grid drawing
End init()

Figure 4.36: Flowchart of init()

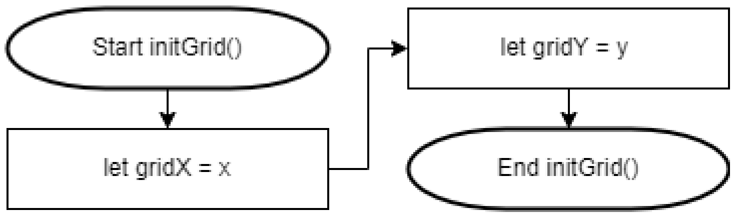


Figure 4.37: Flowchart of initGrid()

Based on figure 4.36, the init() function is performed. This function works with the initGrid(), which mainly defines a function for the initialisation process. From figure 4.36, the initialisation colour is grey, which signifies that the backdrop colour of the simulation is set to grey.

4.2.2 Node-Red

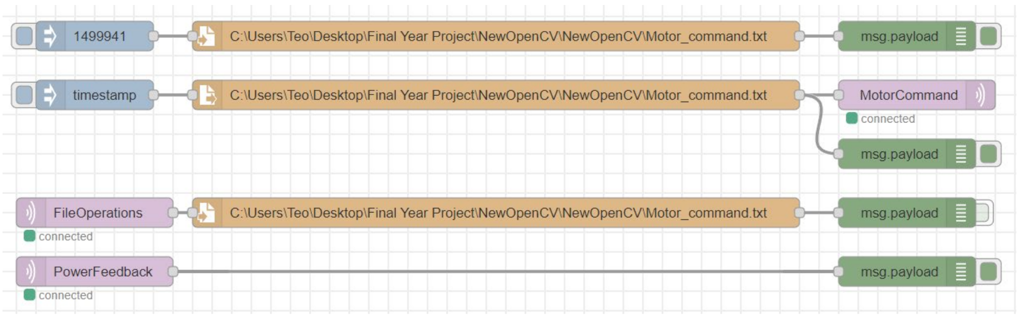

1499941
C:\Users\Teo\Desktop\Final Year Project\NewOpenCV\NewOpenCV\Motor_command.txt
msg.payload
timestamp
C:\Users\Teo\Desktop\Final Year Project\NewOpenCV\NewOpenCV\Motor_command.txt
MotorCommand
connected
msg.payload
FileOperations
connected
C:\Users\Teo\Desktop\Final Year Project\NewOpenCV\NewOpenCV\Motor_command.txt
msg.payload
PowerFeedback
connected
msg.payload

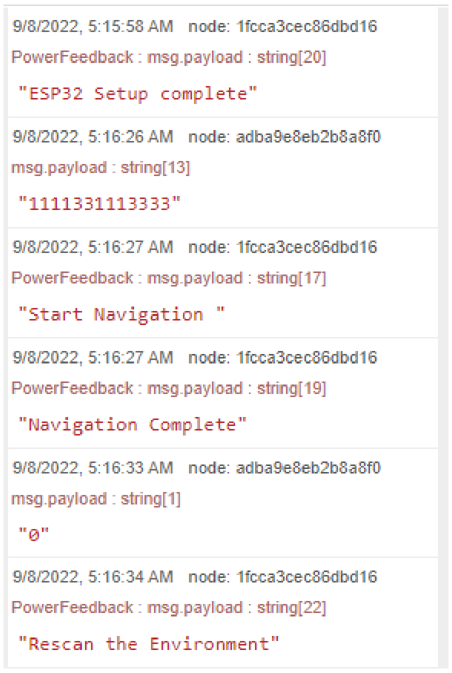
Figure 4.38: Node-Red flow

Node-red functions to integrate between the computer and the microcontroller. The motor command text file output from the folder in the computer can be connected directly to the ESP32 via MQTT. The node in orange is the file location in the computer. Node ‘msg.payload’ is used to read the transmission text or numbers. In the first line, the series of numbers is for the testing phase only. Different series of numbers that fulfil the motor command can be inserted to check the workability and accuracy of the navigation robot without having to start the path-finding program. The series of numbers replace the value in the motor command text file.

The second line is the integration of the program. While the timestamp button is activated, the motor command in the text file will be delivered to the ESP32 via MQTT. ‘MotorCommand’ is the topic. The microcontroller will have to subscribe to this topic to retrieve relevant data.

The third line is subscribed to the topic ‘FileOperations’. This is a vital feedback as it replaces the series of numbers in the motor command text file with a single digit, ‘0’. This prevents the robot from executing the same code again. When the program in the computer does not edit the file, the text file remains unchanged. Hence, it is essential to have this safety feature to prevent unnecessary collisions from happening.

The final line is subscribed to the topic ‘PowerFeedback’. This topic is vital as it transmits important messages to the user. Based on figure 4.39, most of the messages are from the topic. ‘ESP 32 Setup complete’ is to notify the user that the navigation robot is prepared for navigation. When the timestamp is triggered, a ‘Start Navigation’ message is sent to Node-Red, and the navigation robot starts to move in the environment based on the motor command. When the navigation is completed, ‘the Navigation Complete’ message will appear in Node-Red. If the user trigger to execute the program again without running the vision-based program, the ‘Rescan the Environment’ message will appear in Node-Red. This can be accomplished by rewriting ‘0’ into the text file. When ‘0’ is scanned, no process will be carried out and rescan message will be displayed in Node-Red. The message which shows a series of numbers is the motor command. The message is displayed for reclarification purposes for the user.



```
9/8/2022, 5:15:58 AM   node: 1fcca3cec86dbd16
PowerFeedback : msg.payload : string[20]
"ESP32 Setup complete"

9/8/2022, 5:16:26 AM   node: adba9e8eb2b8a8f0
msg.payload : string[13]
"1111331113333"

9/8/2022, 5:16:27 AM   node: 1fcca3cec86dbd16
PowerFeedback : msg.payload : string[17]
"Start Navigation "

9/8/2022, 5:16:27 AM   node: 1fcca3cec86dbd16
PowerFeedback : msg.payload : string[19]
"Navigation Complete"

9/8/2022, 5:16:33 AM   node: adba9e8eb2b8a8f0
msg.payload : string[1]
"0"

9/8/2022, 5:16:34 AM   node: 1fcca3cec86dbd16
PowerFeedback : msg.payload : string[22]
"Rescan the Environment"
```

Figure 4.39: Node-Red debug

4.2.3 ESP32

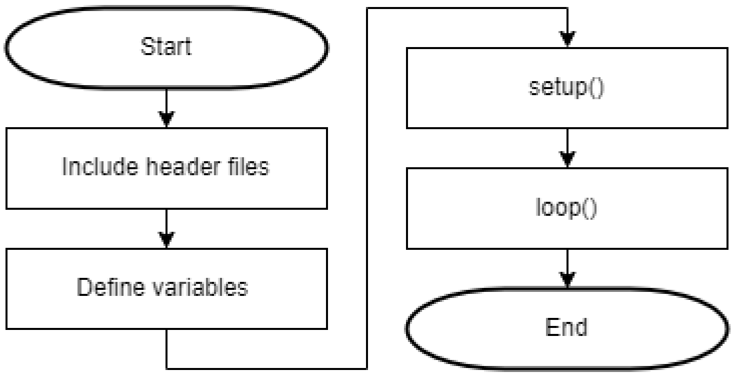


Figure 4.40: Flowchart of ESP32

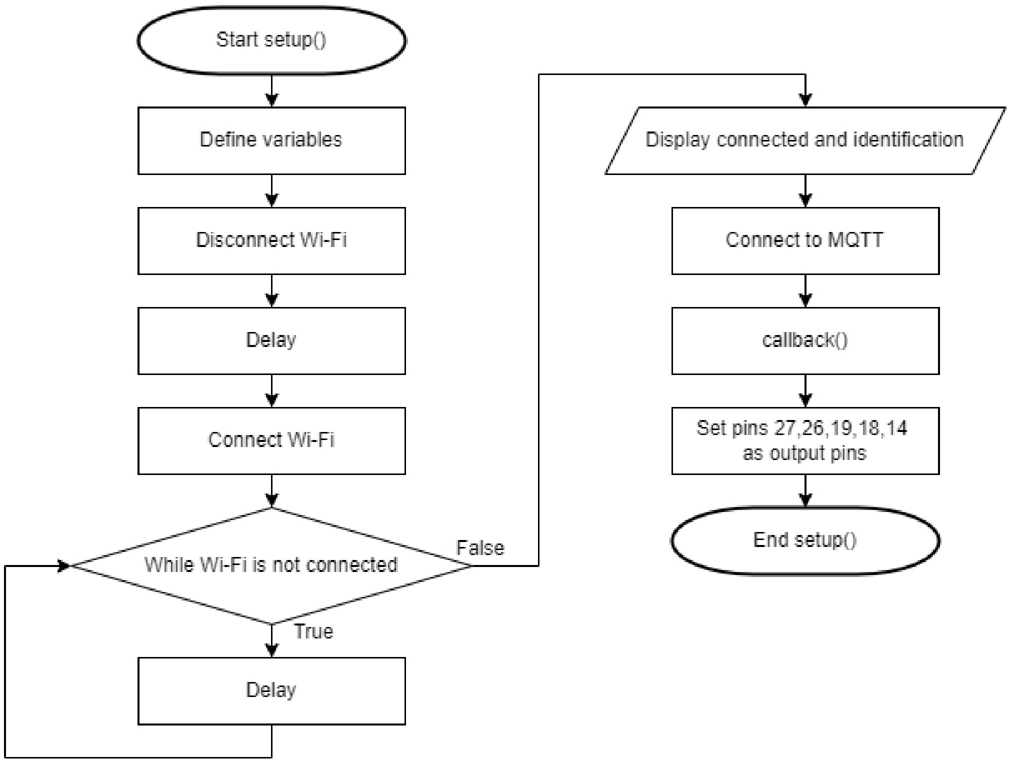


Figure 4.41: Flowchart of ESP32 setup() function

Figure 4.40 is the main flow of the ESP32. Header files are included, and variables are defined for other processes. The setup() function is called, which starts by defining variables. Wi-Fi is disconnected on every startup to make sure that the ESP32 is not under a connected network. While the Wi-Fi is not connected, a short interval waiting time is given. When the Wi-Fi is connected, the connected message and Wi-Fi details will be generated on the output screen in the IDE. The callback() function will occur after establishing the MQTT connection. Pins 27, 26, 19, 18 and 14 are output pins.

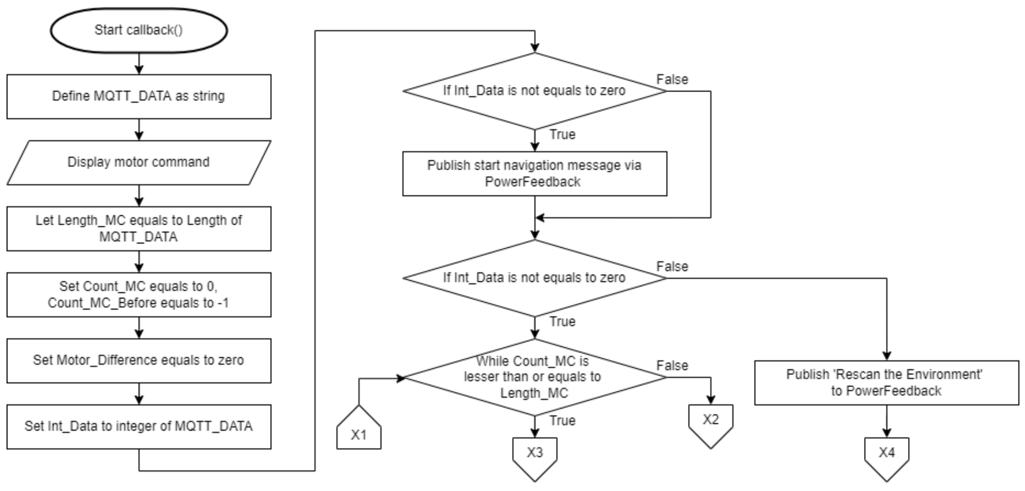

Start callback()
Define MQTT_DATA as string
Display motor command
Let Length_MC equals to Length of MQTT_DATA
Set Count_MC equals to 0, Count_MC_Before equals to -1
Set Motor_Difference equals to zero
Set Int_Data to integer of MQTT_DATA
If Int_Data is not equals to zero
False
True
Publish start navigation message via PowerFeedback
If Int_Data is not equals to zero
False
True
While Count_MC is lesser than or equals to Length_MC
False
True
Publish 'Rescan the Environment' to PowerFeedback
X1
X2
X3
X4

Figure 4.42: Flowchart of ESP32 setup() function Part A

MQTT_DATA is defined as a string. The motor command is imported into ESP32 in MQTT_DATA. The motor command is displayed in the serial monitor. The integer Length_MC equals the length of MQTT_DATA. Counter for motor command and one digit before, Count_MC and Count_MC_Before, are defined. The variable Motor_Difference is also set to zero. Let Int_Data equal to MQTT_DATA to check the status of the motor command text file. If the value in the text file is not equal to zero, the ‘Start Navigation’ message is published to Node-Red via PowerFeedback. The navigation process will start. When the Count_MC variable is lesser than or equal to Length_MC, it signifies that not all the values are processed yet. The process will repeat until all the digits in the text file are fully processed. When Int_data equals zero, the text file had been processed beforehand, and the environment had to be inspected again before handling the motor command. The ‘Rescan the Environment’ message is published to Node-Red via the PowerFeedback topic. The navigation robot will carry out no process if this statement is shown.

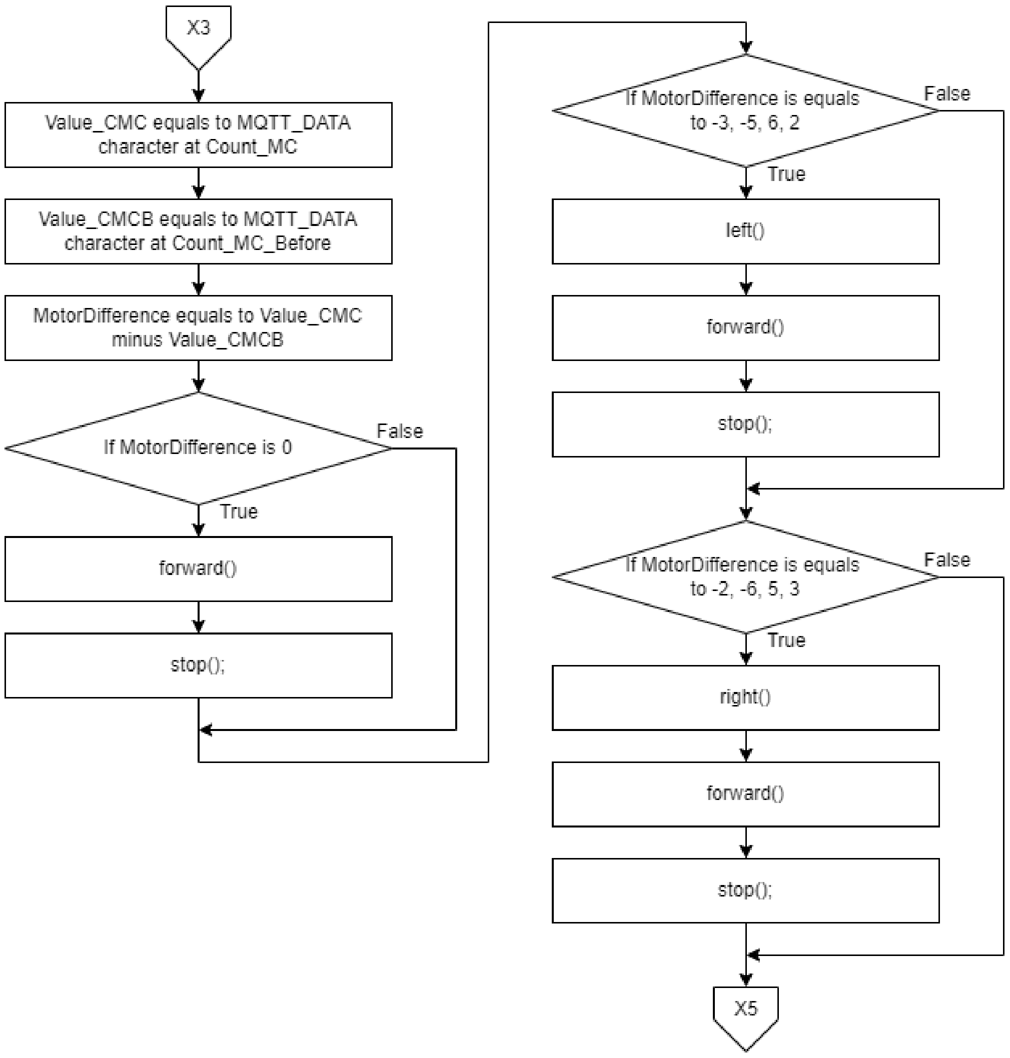

X3
Value_CMC equals to MQTT_DATA character at Count_MC
Value_CMCB equals to MQTT_DATA character at Count_MC_Before
MotorDifference equals to Value_CMC minus Value_CMCB
If MotorDifference is 0
False
True
forward()
stop();
If MotorDifference is equals to -3, -5, 6, 2
False
True
left()
forward()
stop();
If MotorDifference is equals to -2, -6, 5, 3
False
True
right()
forward()
stop();
X5

Figure 4.43: Flowchart of ESP32 setup() function Part B

After X3, integer Value_CMC is set to MQTT_DATA character at Count_MC. Value_CMCB is set to the value of the MQTT_DATA character at Count_MC_Before. MotorDifference takes the result of subtraction for Value_CMC and Value_CMCB. If the value for difference in motor command at each character is equal to 0, the navigation robot moves forward and stops. If the motor difference equals -3, -5, 6 and 2, the navigation turns to the right, moves forward and stops. Similarly, if the values are -2, -6, 5, and 3, the navigation robot will turn left, move forward, and stop. The methodology behind the difference in value for different directions is shown in figure 4.44. This is why the inspection program outputs a value of only 1, 3, 4, and 9 in the text file. The functions for forward(), stop(), left() and right(), can be seen in figure 4.45.

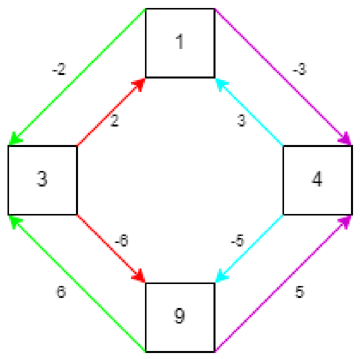


Figure 4.44: Direction of movement for Navigation Robot

1,3,4, and 9 are the movement direction of the navigation robot. Outputs signify the four directions: North, West, East, and South. When the robot moves North at the coordinate before, the number will be '1'. If the next digit in the command is '3', the navigation robot must turn to the left-hand side. Subtraction of 3 from 1 results in -2. We can set -2 to be the command to turn the navigation robot towards the left-hand side direction and move forward. The same calculation is used for other points. Numbers 1, 3, 4, and 9 are used as they are unique when the difference between values is calculated.

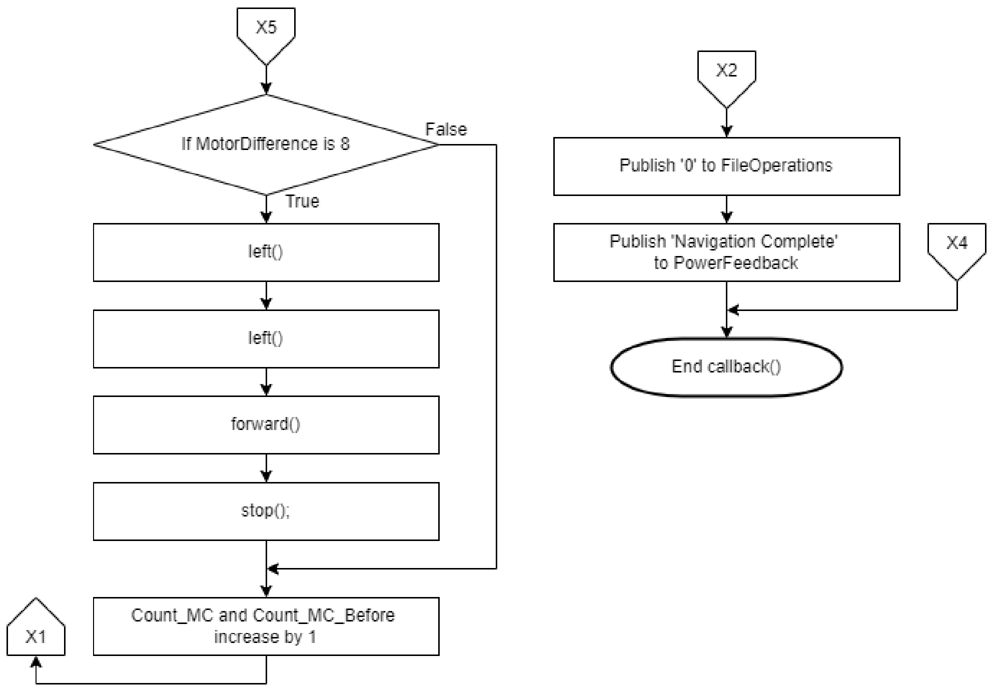

X5
If MotorDifference is 8
False
True
left()
left()
forward()
stop();
Count_MC and Count_MC_Before increase by 1
X1
X2
Publish '0' to FileOperations
Publish 'Navigation Complete' to PowerFeedback
X4
End callback()

Figure 4.45: Flowchart of ESP32 setup() function Part C

Based on figure 4.45, if the motor difference has a value of 8, the navigation robot has to make a 180 degrees turn. This event will only happen from the first to the second digit. When the 'if statement' is completed, both the Count_MC and Count_MC_Before increase by 1. The while loop repeats until all the numbers in the motor command are processed. Once the process mentioned is completed, '0' is published to the FileOperations topic. This is to prevent a repetition of the same motor command without rescanning the environment. 'Navigation Complete' message is published to PowerFeedback topic.

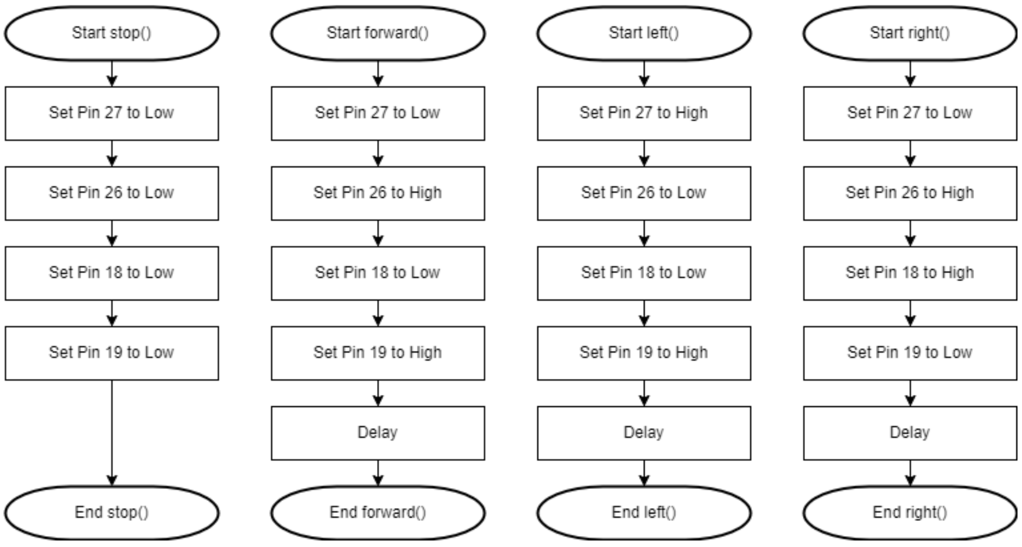


Figure 4.46: Flowchart of ESP32 stop(), forward(), right(), and left() functions

These four functions mainly control the navigation robot's movement. Pin 27 and 26 are for DC motor 1, and Pin 18 and 19 are for DC motor 2. The navigation robot will move forward when Pin 27 and 19 are set to high. When all the Pins are set to low, both motors stop. A combination of pins can achieve turning left and turning right. A delay is needed to ensure the movement of the navigation robot in the environment reaches the next tile. A gyrator is used and tested in this phase. The accuracy of the gyrator is not as accurate as setting a delay to the process. Hence, only the delay is implemented in the final phase of the ESP32 program.

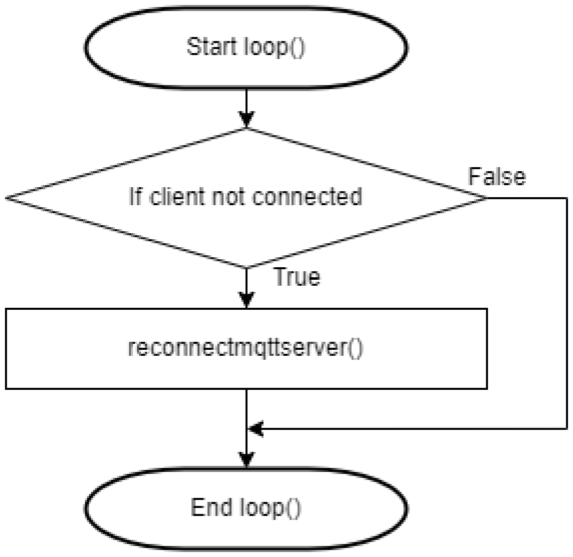


Figure 4.47: Flowchart of ESP32 loop() function

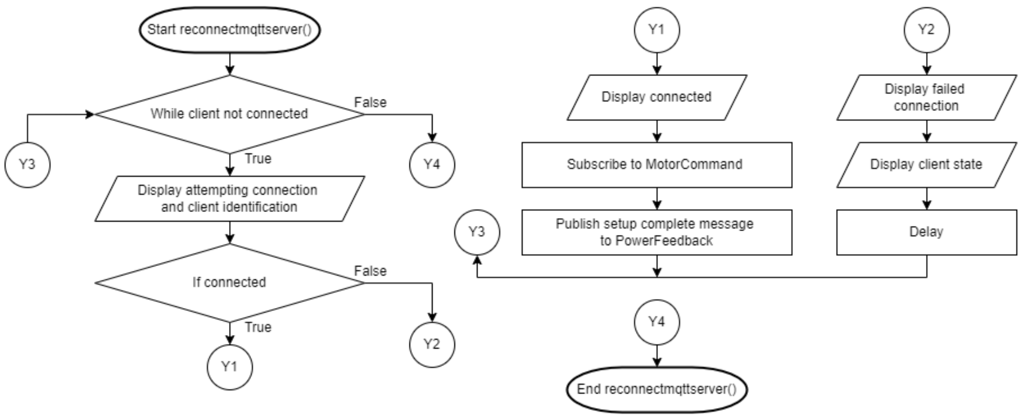

Start reconnectmqttserver()
While client not connected
False
True
Y3
Y4
Display attempting connection and client identification
If connected
False
True
Y1
Y2
Y1
Display connected
Subscribe to MotorCommand
Publish setup complete message to PowerFeedback
Y2
Display failed connection
Display client state
Delay
Y3
Y4
End reconnectmqttserver()

Figure 4.48: Flowchart of ESP32 reconnectmqttserver() function

The ESP32 loops around the connection of the MQTT server. If the client is connected, the loop ends. If the client is not connected, reconnectmqttserver() function is addressed. While the client is not connected, attempting connection and client identification are printed. If the client is connected, a message will be displayed, 'MotorCommand' topic will be subscribed. A message to notify the user that the ESP32 setup is complete will be sent to Node-Red via the topic, PowerFeedback. The unsuccessful connection message will be displayed if the client fails to connect. The client's state is displayed. The function will end when the client is connected.

4.3 Project Prototype

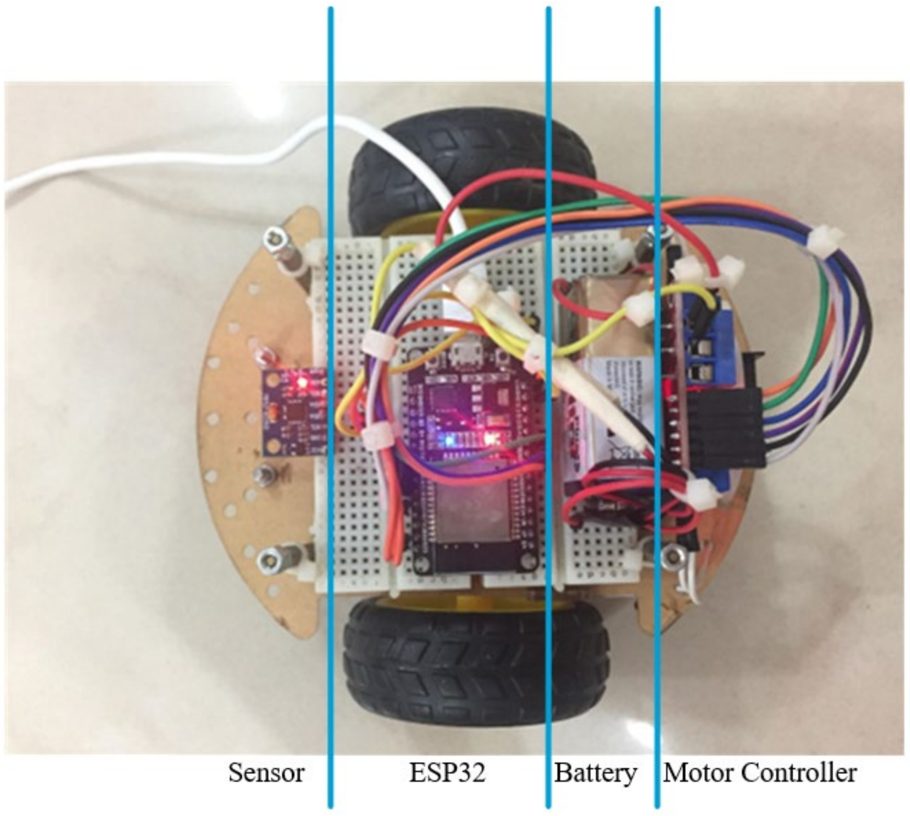


Figure 4.49: Complete system in navigation robot

The best way to mount ESP32 onto the 150 mm length by 105 mm width navigation robot chassis is by placing the microcontroller connecting two 82 mm by 35 mm breadboards together. Two breadboards fit perfectly in the slot between the four pillars of the robot. The front of the robot faces the left, and the gyrator, which has a dimension of 20 mm by 16 mm, is placed at the front. Other sensors are given space to be added at the front of the navigation robot. The ESP32, with a dimension of 57 mm by 28 mm, is in the middle of two breadboards. The width of ESP32 is almost identical to the breadboard, which makes it impossible to connect both sides of the pins on the same board. Figure 4.49 shows that the battery is held between the microcontroller and the motor controller. To prevent the battery from moving around, it is fixed in place using jumper wires on the side. From the top view, the battery has a length of 485 mm and a width of 175 mm. The battery height, along with the breadboard, is 275 mm, exceeding the height of the four pillars on the navigation robot. Another mounting method for the top layer of the navigation robot must be considered. The motor controller is mounted with cable ties at the far end of the chassis. It is positioned where the height is 55 mm, exceeding the four pillars, width and length to be 27 mm and 44 mm. The width and length of the motor controller are sufficient to fit into the back end of the chassis. Cable ties are used to tightly hold the breadboard onto the chassis with the support of the four pillars. The jumper wires used have the same colour as figure 4.1. All wires can be pinpointed easily during improvement and maintenance. All the jumper wires are adequately organised with cable ties to keep the robot aesthetically clean.

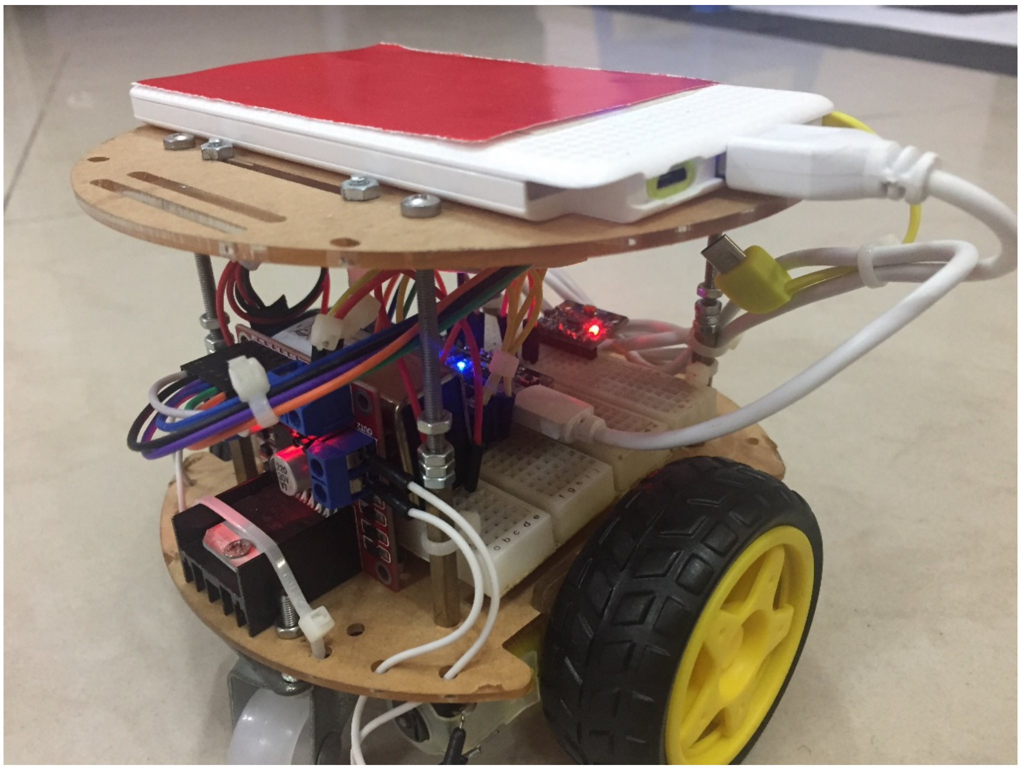


Figure 4.50: Complete assembly of navigation robot

Figure 4.50 shows the fully assembled navigation robot from the back right direction. As discussed, the height of both the battery and the motor controller exceeds the height of the pillars. Four additional screws are needed to extend the height of the pillars. A power bank, which is connected to the ESP32, is mounted above the second layer of the chassis. It is held in place by the screws and nuts surrounding it. Multiple test runs prove that the power bank will stay in position while navigating the environment.

Some of the standard parts of the navigation robot are replaced and mounted by a longer screw for stability. While assembling following the standard parts procedure, the two guiding wheels of the navigation robot did not have the same height as the DC motors. During the navigation process, the robot will either lean in one direction, making the impact of two DC motors unequal. This affects the accuracy of the robot and causes it to drift off the standard control. Full forward might turn out to be slightly tilting towards the side. The issue is solved by mounting the guiding wheels with longer screws with the aid of a spirit level.

Figure 4.50 shows that the chassis holes are utilised to properly organise the wires and cable ties. Passing wires across the holes avoids unintentional disconnection when transporting the robot.

4.4 Performance of Vision-Based Program

The vision-based program undergoes different stages of improvement throughout the project. The program is used to test pictures initially to experiment with the workability of the algorithm. A real-time image is added subsequently to achieve the project objective.

4.4.1 Maze Solving

Different mazes were tested using the programming code. The program can solve the maze by navigating a path from start to end and looking for the correct path. OpenGL generates the figure with obstacles in red.

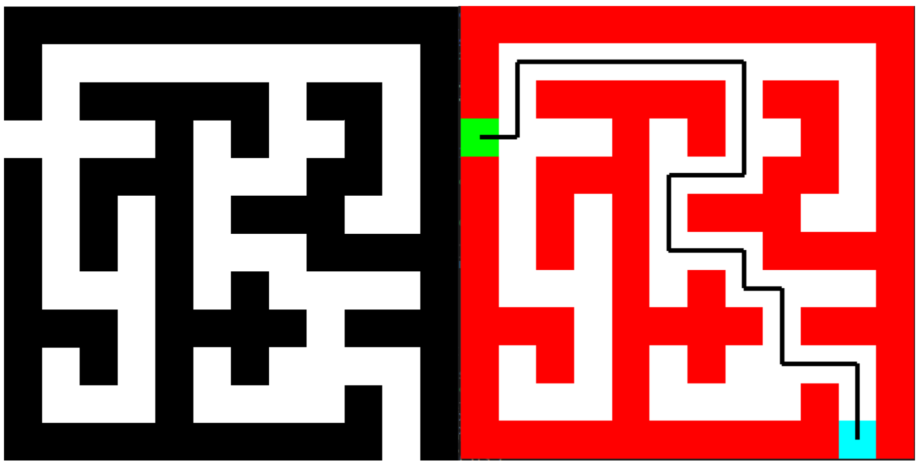


Figure 4.51: Path to solve the maze

4.4.2 Enclosed Environment Preliminary Results

An enclosed environment makes the robot navigation closer to the real world. The first and second enclosed environments are created to simulate a scenario closer to a real-world environment from the top view. More spaces are provided in the environment instead of a one-lane road.

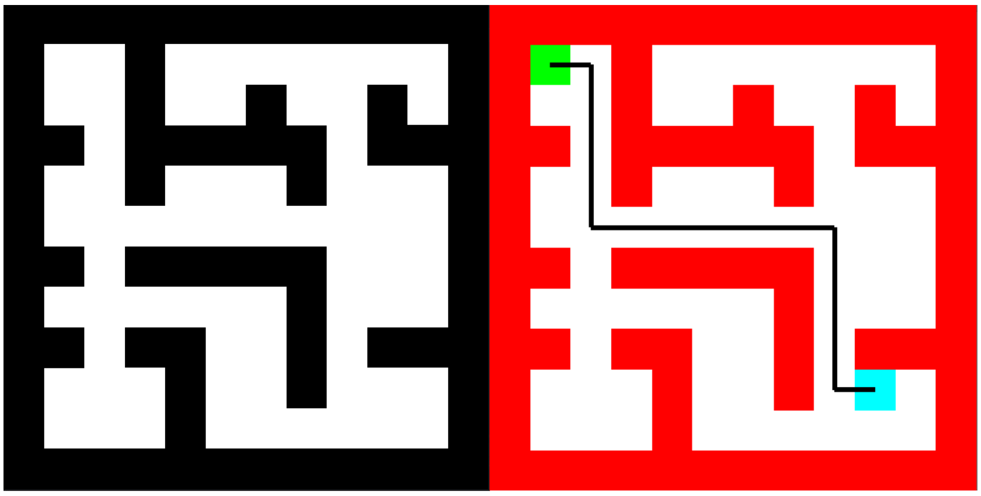
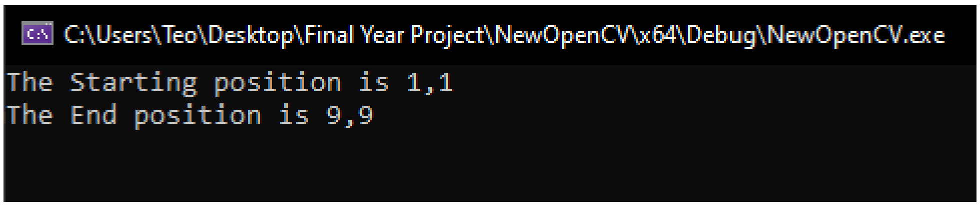


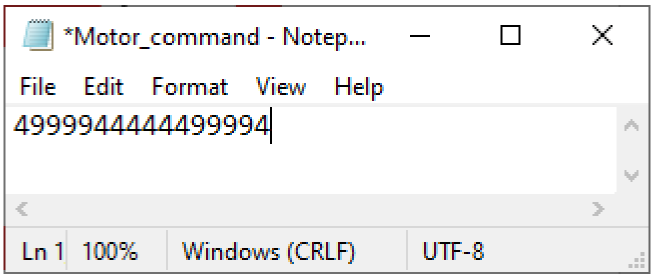
Figure 4.52: Path of solving the first enclosed environment


C:\Users\Teo\Desktop\Final Year Project\NewOpenCV\x64\Debug\NewOpenCV.exe

```
The Starting position is 1,1
The End position is 9,9
```

Figure 4.53: Starting and ending position for the first enclosed environment

The start position can be edited in the programming code, and the end position will be input by the user after the program’s execution. The message in figure 4.53 shows the environment's starting and ending positions.


*Motor_command - Notep...
File Edit Format View Help
Ln 1 100% Windows (CRLF) UTF-8

```
4999944444499994
```

Figure 4.54: Motor command for the first enclosed environment

The program stops after the path is found and generates the motor command in a text file. The numbers signify the movement of the robot. ‘1’ commands the robot to move the robot in the north direction, ‘3’ commands the robot to move to the west, ‘4’ commands the robot to move to the east, and ‘9’ controls the robot to move to the south. The series of numbers is shown in figure 4.54. The numbers represent the robot movement generated from figure 4.52. Figure 4.55 shows the second enclosed environment, and figures 4.56 and 4.57 show the message and motor command.

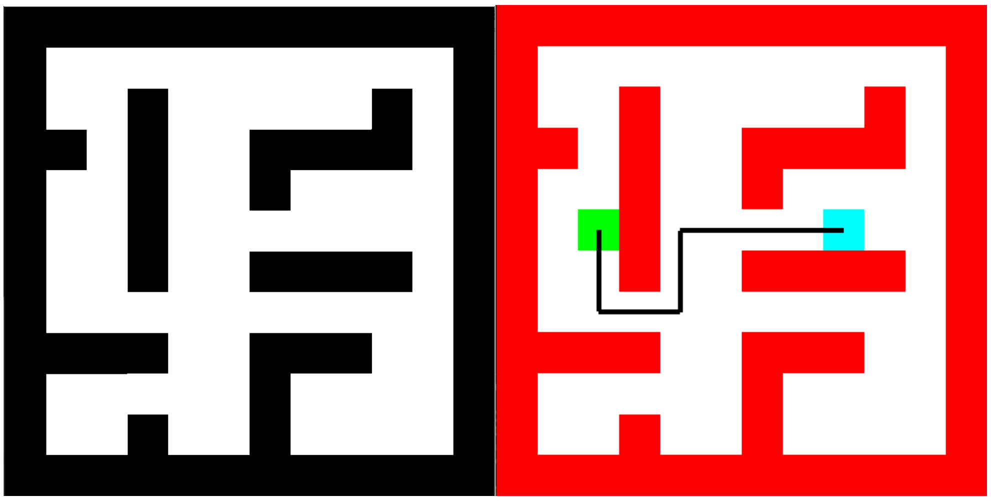
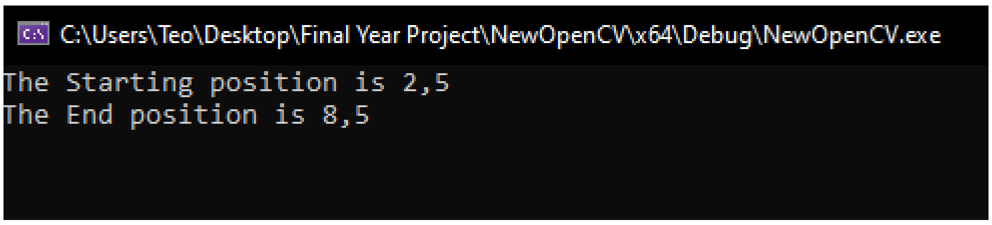
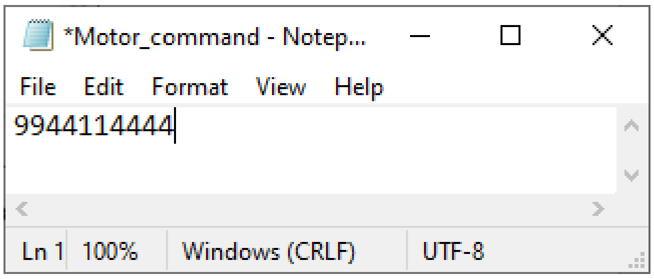


Figure 4.55: Path of solving the second enclosed environment


C:\Users\Teo\Desktop\Final Year Project\NewOpenCV\x64\Debug\NewOpenCV.exe

```
The Starting position is 2,5
The End position is 8,5
```

Figure 4.56: Starting and ending position for the second enclosed environment


*Motor_command - Notep...
File Edit Format View Help
Ln 1 100% Windows (CRLF) UTF-8

```
9944114444
```

Figure 4.57: Motor command for the second enclosed environment

The robot should not execute any command when obstacles enclose the starting point. While running a test for this enclosed environment with no path, another test is carried out. Testing of the thresholding to turn the grayscale image into a binary image. The greyed blocks are still detected in the OpenGL simulation. Figure 4.58 shows that no path is generated when obstacles surround the starting point. If no path is generated, the output message will notify the user that no path is generated and does not output anything in the motor command text file.

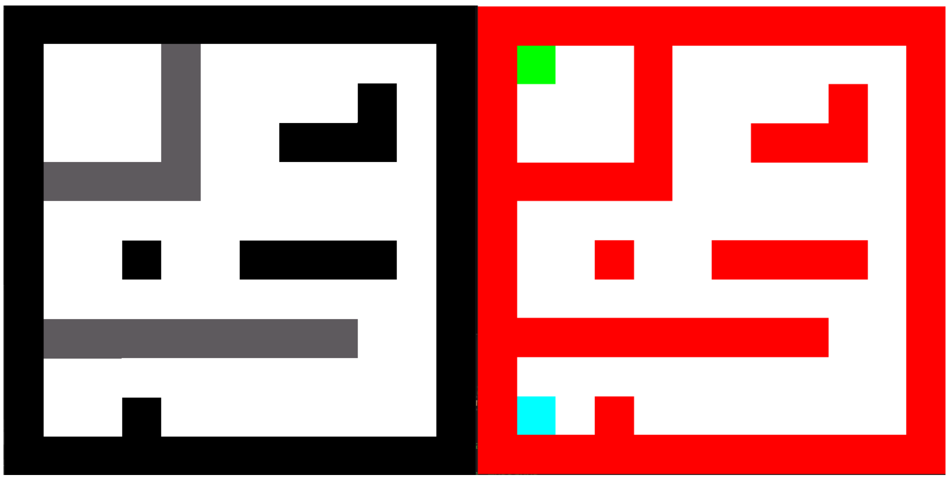
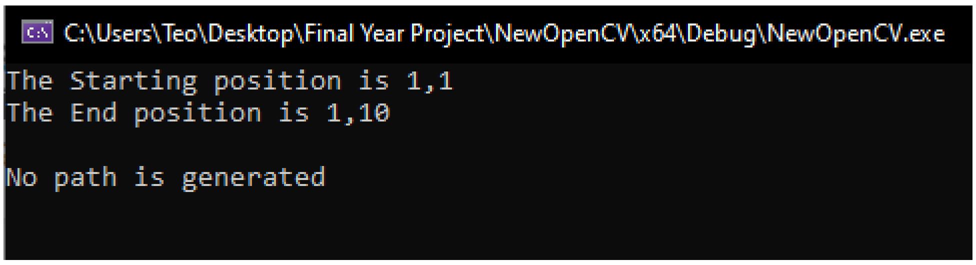


Figure 4.58: Solving no path enclosed environment


C:\Users\Teo\Desktop\Final Year Project\NewOpenCV\x64\Debug\NewOpenCV.exe

```
The Starting position is 1,1
The End position is 1,10

No path is generated
```

Figure 4.59: Positions and message for no path enclosed environment

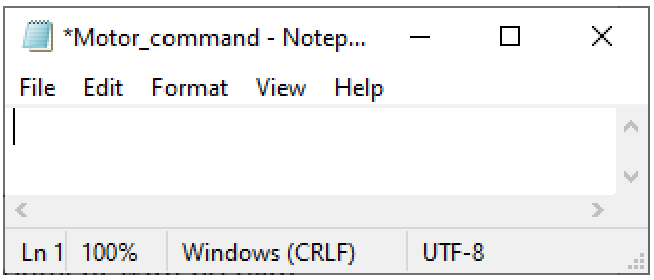

*Motor_command - Notep...
File Edit Format View Help
Ln 1 100% Windows (CRLF) UTF-8

Figure 4.60: Motor command for no path enclosed environment

4.4.3 Enclosed Environment complete program

Vision is added to this stage of the program. It is integrated with the algorithm from the previous test to check workability and efficiency. Practical images require post-processing before executing path planning to capture the environment perfectly. The complete outcome from a practical image is shown under level four subsections.

4.4.3.1 Enclosed environment without pre-set coordinates

A custom coordinate is used in this section of the program. The user provides the custom coordinate within the boundary of the enclosed environment. There will only be one end coordinate in the program when there are no pre-set coordinates.

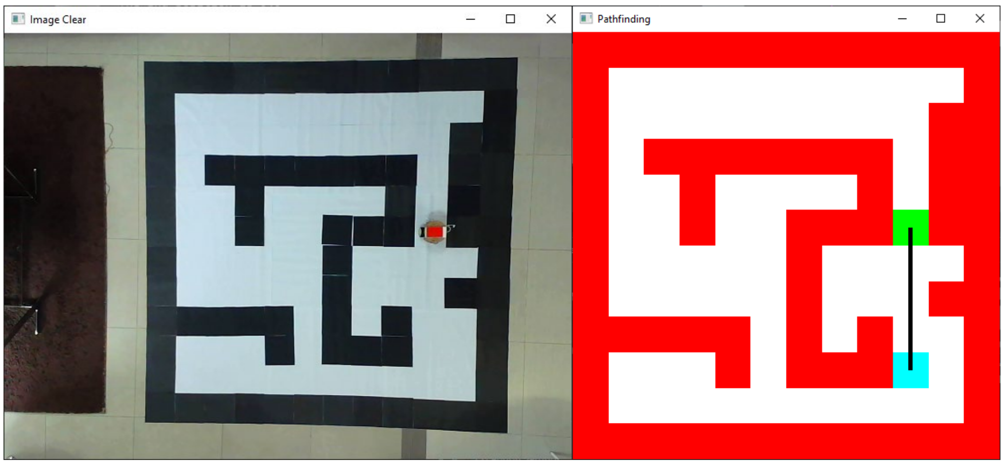


Figure 4.61: Path to solve the first practical enclosed environment

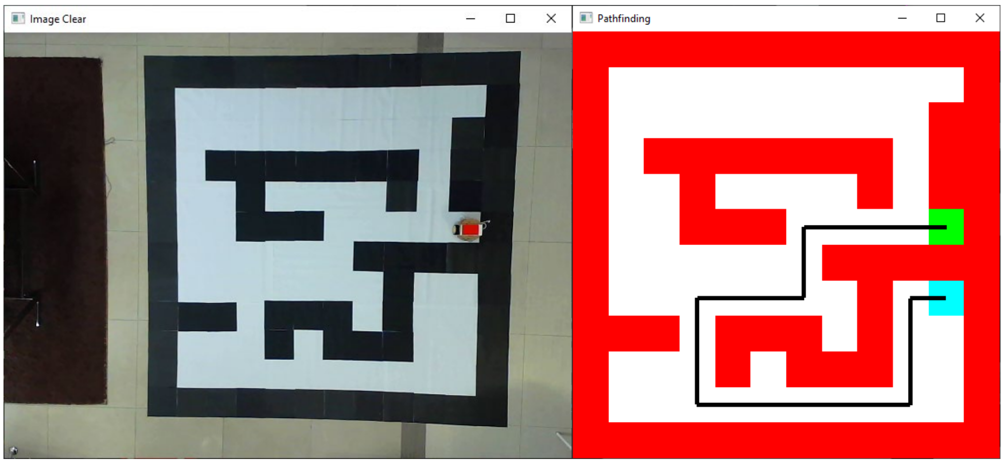


Figure 4.62: Path to solve the second practical enclosed environment

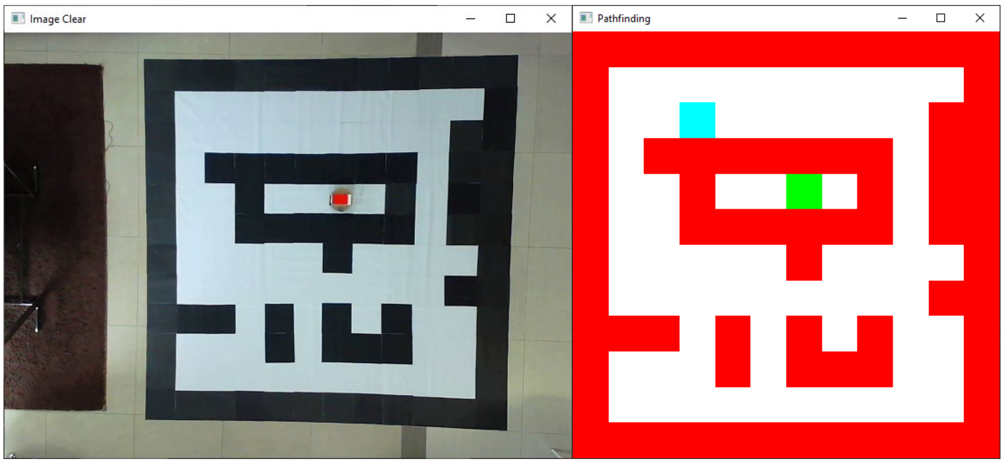


Figure 4.63: Path to solve the third practical enclosed environment

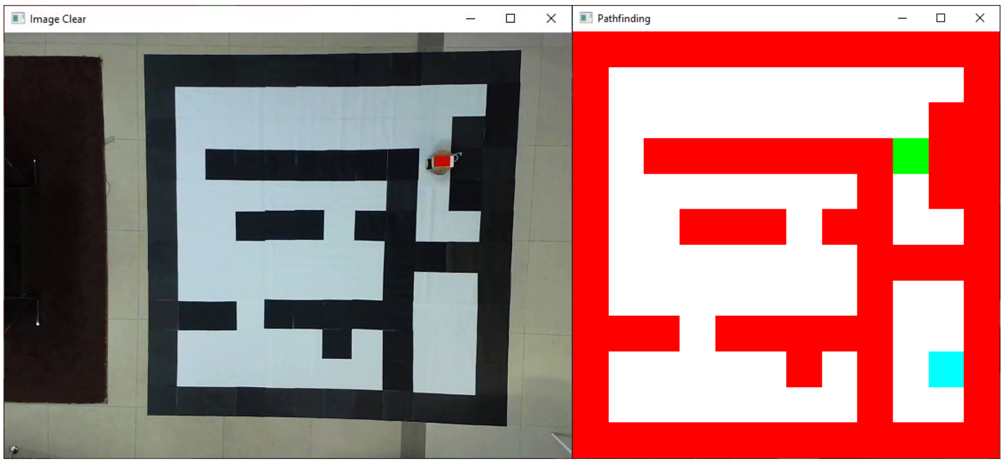


Figure 4.64: Path to solve the fourth practical enclosed environment

Figures 4.61 and 4.62 show a path between the start and end coordinates. Figures 4.63 and 4.64 show that no possible path is found in the environment, and no path will be drawn. The success rate for scanning the environment is 95% out of 40 tries. The main reason for failure in the scan is the interruption on the outermost of the boundary. If an object is placed on the border, the rectangle algorithm cannot find the square, which causes the scan to fail. If the scan is successful, an enclosed environment without pre-set coordinates has a success rate of 100% from 20 tries. No path scans also have a success rate of 100% out of 10 attempts.

4.4.3.2 Enclosed environment with pre-set coordinates

Pre-set coordinates are used in this section of the program. The pre-set coordinates are saved in the program. There will be multiple end coordinates in the program when there are pre-set coordinates. The program will search for the closest end coordinate and connects it with a path. This is very useful for the navigation robot to find the nearest station in the physical world.

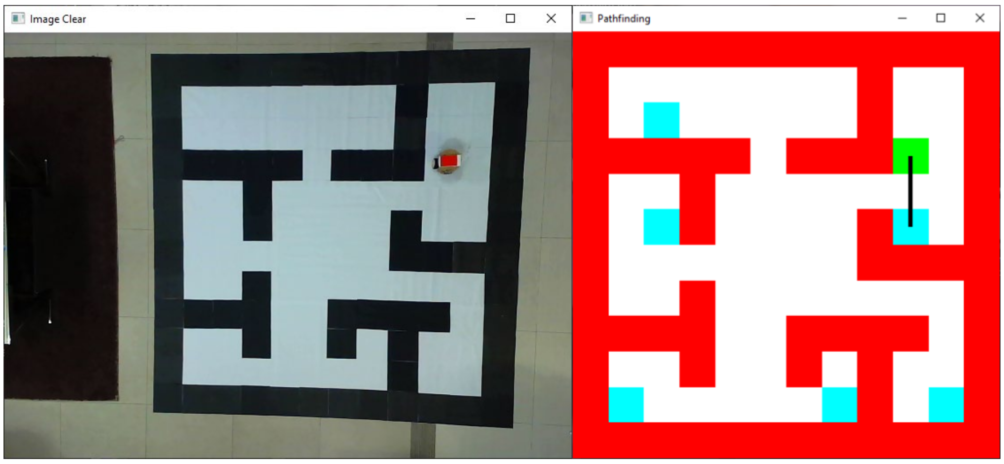


Figure 4.65: Path to solve the fifth practical enclosed environment

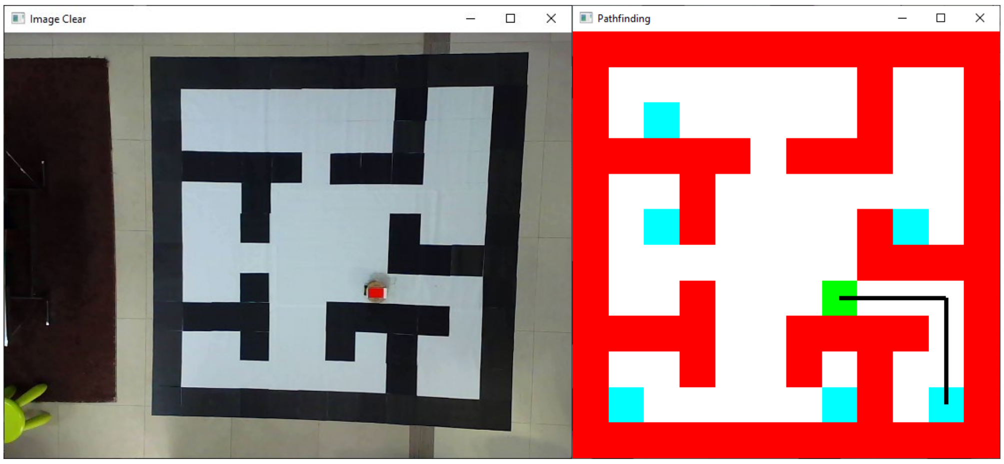


Figure 4.66: Path to solve the sixth practical enclosed environment

Figures 4.65 and 4.66 shows that the simulation can find the nearest path from multiple coordinates. The success rate of the algorithm is 100% out of 15 tries. An uninformed search algorithm can only accomplish this. An informed search algorithm needs to have a precise end coordinate.

4.4.3.3 Enclosed environment program

The program is fully automated and does not require tuning to switch from custom coordinates to pre-set coordinates. Based on simulations, the accuracy of the algorithm is very high and will be able to achieve the objective if the environment is scanned without false positives. False positives will only occur when the environment is in poor light condition, as shown in figure 4.6. Increment in the size of the enclosed environment will require changing numbers in rows and columns in the program but will not affect the accuracy and path planning of the algorithm. Accuracy remains the same when the enclosed environment is expanded. The processing time for the enclosed environment program will increase. The motor command simulated will still be accurate as the paths found will be in the midpoint of the cell, preventing it from offsetting from the actual path.

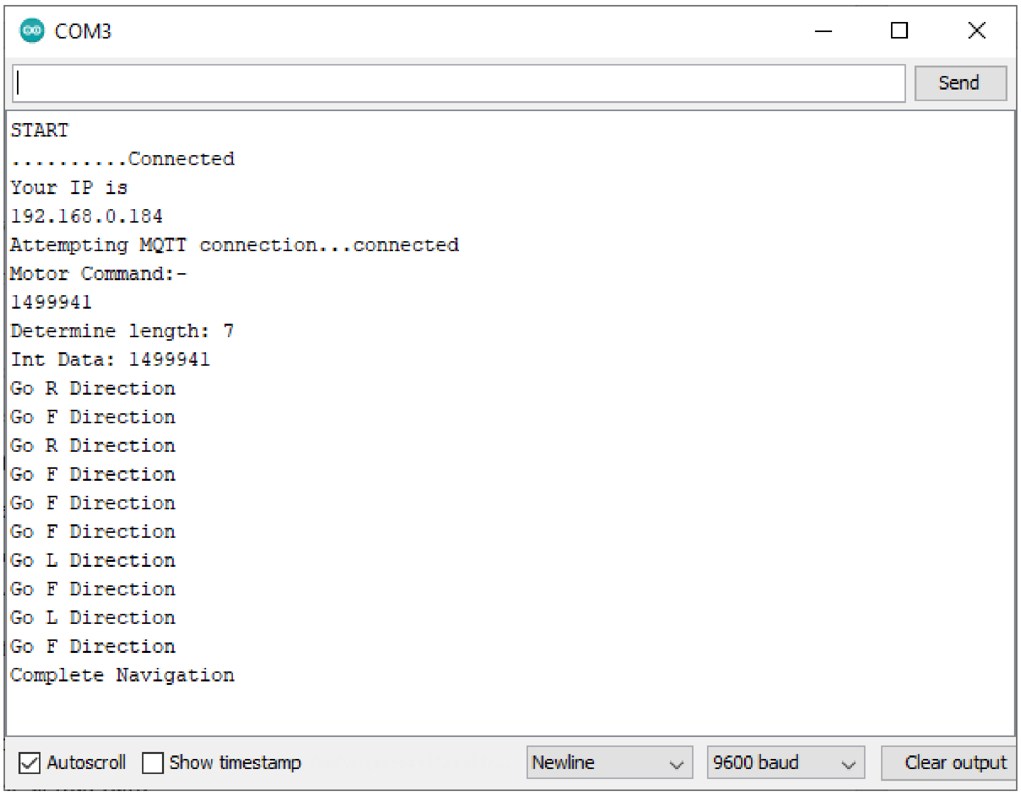
Table 4.2: Summary table of simulation results for enclosed environment

Evaluation Criteria	Success	Failure	Success Rate (%)
Scanning Environment	38	2	95
Single Point End coordinate	20	0	100
Multiple Points End coordinate	15	0	100
No path on Single Point	6	0	100
No path on Multiple Points	4	0	100

4.5 Usability test

A usability test is carried out at the final stage of the project. This is to test the efficiency of the developed training system. The prototype in figure 4.50 is utilised in the enclosed environment. In the usability test, two sets of results will be presented as one set is the improvement for the other. The first set of results is the initial solution, where the gyrator is considered. The second set of results has the gyrator replaced with delay time. The navigation robot performs better in this setting.

4.5.1 Processing of motor command



```
START
..........Connected
Your IP is
192.168.0.184
Attempting MQTT connection...connected
Motor Command:-
1499941
Determine length: 7
Int Data: 1499941
Go R Direction
Go F Direction
Go R Direction
Go F Direction
Go F Direction
Go F Direction
Go L Direction
Go F Direction
Go L Direction
Go F Direction
Complete Navigation
```

Figure 4.67: Serial monitor output for ESP32

Figure 4.67 shows the connection process and navigation process of the robot. When the ESP32 is started, Wi-Fi and MQTT are connected. The IP address shown in the serial monitor is for the ESP32. The motor command is successfully sent from the computer to ESP32, as shown in the line "Motor Command: -". On rare occasions, the MQTT connection might be disconnected due to service interference from the broker. Interruption of MQTT will not disrupt the ongoing navigation process and will return online after a short amount of time.

Lines starting with 'Go' are the navigation commands. R direction, L direction, and F direction indicates the right (), left(), and forward() function in the ESP32 code. As discussed in subsection 4.2.1, the robot will always face the North direction at start-up. First '1' indicates the facing direction of the robot. The navigation can be visualised and validated manually by drawing the movement flow of the robot.

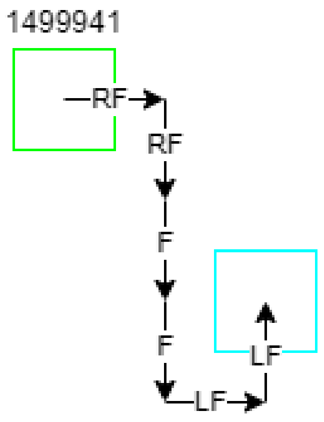


Figure 4.68: Navigation path of motor command

The motor command shown in figures 4.67 and 4.68 is both ‘1499941’. The number series also means moving to the east for one time, to the south three times, to the east again, and to the north. The navigation command does not know the orientation of the robot. It will process the motor command by controlling the motors to turn left and right. Initially, the robot is facing the north. The robot must turn right to face the east direction and move forward to arrive at the next block. The following motor command is ‘9’, where the navigation command will direct the motors to turn right again, facing the south direction. The navigation command will continue forward three times, moving the position three blocks towards the south. At ‘4’, the robot will have to move towards the east direction again. The turning direction is different from before now. Instead of turning right, the robot will have to turn left to move in the east direction. After left and forward, the motor turns left and moves forward again to reach the end position. Overall, the robot will move Right, Forward, Right, Forward, Forward, Forward, Left, Forward, Left, Forward. The navigation command matches the command in figure 4.68.

The same test shown in figure 4.68 is done 13 times to check the movement directions generated by ESP32 in the serial monitor. All 13 tests output the appropriate results, making processing motor commands in the ESP32 100% accurate.

4.5.2 Navigation robot with gyrator

Table 4.3: Results of Navigation Robot with Gyrator

Evaluation Criteria	Success	Failure	Success Rate (%)
Static Single Turn	4	7	40
Static Multiple Turns	2	7	22
Short distance	6	7	46
Long distance	1	8	11

Table 4.3 shows the results of the navigation robot while using the gyrator to achieve 90 degrees turn. A low success rate was achieved as the accuracy of the gyrator was terrible. The gyrator is perfectly tuned and can make 90 degrees turn at the first corner. It will quickly lose accuracy on the second turn. It could achieve 90 ± 3 degrees on a right turn but will shoot up to 90 ± 15 degrees on a left turn. This causes unusual success on a single turn, and the prototype will most likely fail in multiple turns.

The success in single turns on the same spot is all contributed by turning to the right. There are also times when turning to the right side fails. For static multiple turns, all the successful attempts are achieved by turning to the right side only. Those with a combination of left and right, or turning left only, fail badly. Short distances have a higher success rate as half of the testing is done by turning towards the right-hand side. Those that turn to the left-hand side has a very bad success rate. Long distance usually combines multiple turns to achieve the target. Inaccuracy in long distances is very high as the left-hand turn has bad accuracy, and the right-hand turn still has a relatively high tolerance, where the sum of offset can go up to 10 degrees.

The success rate is calculated by summing up the successful attempts for short and long distances and dividing by the total attempts. From the results in table 4.3, we can conclude that the success rate of the navigation robot using a gyrator is relatively low at only 32%. Other methods must be tested to increase the accuracy of the navigation robot.

4.5.3 Navigation robot with time delay

Table 4.4: Results of Navigation Robot with time delay

Evaluation Criteria	Success	Failure	Success Rate (%)
Static Single Turn	9	1	90
Static Multiple Turns	8	2	80
Short distance	9	1	90
Long distance	7	3	70

Table 4.4 shows the outcomes of the navigation robot while utilising the time delay to accomplish 90 degrees turn. A higher success rate is achieved as the motor delay can be set and executed without external disturbance. The delay time is initially tested with the help of the gyrator to check if the angle is precisely 90 degrees. The idea is withdrawn as the gyrator display false positives, and manual testing of delay time is carried out. After completion of calibration, the time delay is applied to the microcontroller code in Arduino IDE. Static single turns and multiple turns are tested and observed. Successful turns are determined by two rulers placed perpendicularly. The test is successful if the navigation robot can go straight parallelly with the ruler after the turn.

Static single turn has a success rate of 90%, where the accuracy is considerably high. Short-distance navigation will mostly succeed when the static single-turn result is good, as both assessments are highly related. Short distance usually makes only one or two turns throughout the navigation. Inaccuracy in short distances will not affect as much as a sum up of inaccuracy for long distances. External factors influence failure in the static single turn, where the test environment is on a surface with a slightly different height. Failure in short-distance navigation is affected by the initial orientation of the robot at the beginning of the navigation. If the robot is not faced to the north accurately, the whole movement of the robot might offset slightly to either direction. The navigation robot must be placed at the stipulated position with the correct orientation to keep the result accurate.

The success rate of static multiple turns is 80%, and long-distance navigation is 70%. Static multiple turns results will directly affect the outcome for long-distance navigation. Failure in static multiple turns has the same reason as a static single turn, where the surface has a slightly different height. Travelling long distances has a lower success rate as the summation of inaccuracy in time delay increases. The environmental factors might also be a cause for inaccuracy. The further it travels, the other consideration must be taken into account. An ideal case is not possible.

From the results in table 4.4, we can conclude that the success rate of the navigation robot using a time delay for 90 degrees turn is relatively high, at 80%. A better chassis with a more rigid structure can be introduced to increase accuracy further. Bending of Perspex might happen during assembly, which might also be a reason for inaccuracy.

CHAPTER 5

CONCLUSION

5.1 Conclusion

The vision-based indoor navigation robot training system is successfully developed using OpenCV and C++ programming language. The robot can navigate around the environment with the support of the camera and the user-specified target. The algorithm can provide the nearest path to navigate around the environment without manual assistance. The user will only have to give the program relevant coordinate information and running mode. The computer and ESP32 board are the central processing unit for this project to perform path planning and motor command analysis. Node-Red connects both processing units via MQTT.

An algorithm is developed in Visual Studio on the computer as functionalities can be written and customised in the IDE. The IDE provides excellent flexibility in image processing, user input, path planning, simulation, and writing of output files. Next, Node-Red is a great platform to transfer data from the computer into the microcontroller. Data and user reminder messages can be easily shared across platforms, which creates a great connection between control units. Another algorithm is developed in the ESP32 to derive and execute the data delivered from the computer. Arduino IDE is used to code the ESP32 to fulfil the task.

A few software and hardware problems arise throughout the training system's testing phase. The issues are resolved and reviewed in Chapter 4, under results and discussion. As the vision-based indoor navigation robot is developed in a training system, many improvements are accomplished by adding extra codes or substituting the existing algorithm with a new one. Not just the advancements in software, enhancements in hardware are also documented in Chapter 4.

The objective of this project is to design, develop and simulate programming code to perform image processing and path planning, integrate programming code with a microcontroller wirelessly, and test and evaluate the performance of the navigation robot. All three objectives of the project are achieved.

5.2 Limitations of the project

The project does have a few limitations. Firstly, the navigation robot does not have a feedback mechanism when moving near an obstacle. A collision between the navigation robot and the obstacle will likely happen if other moving objects are in the environment. Without the feedback mechanism, the navigation robot will continue running the DC motors even though a crash occurred.

Next, the camera mounted above the environment could only scan the limited space. The implementation might only be effective indoors in small areas or on high roofs. It will be ideal if the scannable area can be expanded.

Lastly, the algorithm can process multiple end coordinates, but only one start coordinate exists. In big factories, a single robot is not enough, and more robots will be added to the environment sooner or later. Increasing the number of robots could improve the efficiency of the factory.

5.3 Recommendations for improvements

As this is a training system for a vision-based indoor navigation robot, several improvements can be made. Most of the recommendations revolve around the limitations of this project.

To bring the environment closer to the usual indoor environment for factories, the number of pixels in the environment should be increased. More information can be extracted from an image with high pixel counts. The established algorithm currently is the benchmark for the training system. Increasing the pixel values can be built above the model.

Subsequently, an ultrasonic sensor can be introduced into the navigation robot to measure the distance between the navigation robot and the obstacles. If there are new objects in the environment that are not scanned by the camera previously, the ultrasonic sensor can detect the object's presence and send a stop signal to the DC motors. A signal can be sent to the vision-based program in the computer to capture a new image of the environment. The new obstacle will be added to the environment.

Other than that, it is discussed in the limitations that the camera's field of view will restrict the environment. To resolve this issue, integration of maps between cameras can be established. Cameras are placed above different environments. The borders of the environment are connected to the following environment. The environment can be joined virtually in the software, creating the primary environment from sub-environments. This increases the size of the scannable area.

Finally, additional start coordinates can be introduced into the algorithm. As reviewed in the limitations, more than one robot will be employed in big factories. Increasing start coordinates and prevention of collision between navigation robots can be applied.

REFERENCES

Arshad, R., Arslan Shahid, M., Khan, D. and Hammad Hussain Shah, S., 2016. *Study and analysis of shortest path algorithms*. 2nd International Multi-Disciplinary Conference. Gujrat, Pakistan, 19-20 December 2016.

Fahleraz, F., 2018. *A comparison of BFS, Dijkstra's and A* algorithm for grid-based path-finding in mobile robots*. [Online] Available at: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Makalah/Makalah-IF2211-2018-016.pdf> [Accessed 12 April 2022].

Karastoyanov, D. and Zahariev, R., 2004. A navigation system and task planning in a mobile robot for inspection. *Problems of Engineering Cybernetics and Robotics*. 54, pp. 22-29.

Kaur Sidhu, H., 2019. *Performance evaluation of pathfinding algorithms.* Master. University of Windsor.

Lydon, B., 2018, *Automated guided vehicles improve production - ISA* [Online]. Available at: <https://www.isa.org/intech-home/2018/july-august/features/automated-guided-vehicles-improve-production> [Accessed: 18 April 2022].

Martell, V. and Sandberg, A., 2016. *Performance evaluation of A* algorithms*. Degree. Blekinge Institute of Technology.

Mehlhorn, K. and Sanders, P., 2008. *Algorithms and data structures: the basic toolbox*, Springer Berlin Heidelberg.

Mistri, R.K., 2018. Wi-Fi control robot using Node MCU. *International Journal of Engineering Development and Research*, 6(2), pp.325–328.

Niederberger, C., Radovic, D. and Gross, M., 2004. Generic path planning for real-time applications. *Proceedings of Computer Graphics International Conference, CGI*, pp.299–306.

Oitzman, M., 2021, *What's the difference between an AMR and an AGV?* [Online]. Available at: <https://mobilerobotguide.com/2021/08/06/whats-the-difference-between-an-amr-and-an-agv/> [Accessed: 18 April 2022].

Pavithra, A.C. and Subramanya Goutham, V., 2018. Obstacle avoidance robot using arduino. *International Journal of Engineering Research & Technology* , 6(13).

Rachmawati, D. and Gustin, L., 2020. Analysis of Dijkstra's algorithm and A∗ algorithm in shortest path problem. *Journal of Physics: Conference Series*.

Russell, S. and Norvig, P., 2019. *Artificial Intelligence a modern approach fourth edition* 4th ed. Hoboken: Pearson.

Sadik, A.M.J. et al., 2010. *A comprehensive and comparative study of maze-solving techniques by implementing graph theory*. 2010 International Conference on Artificial Intelligence and Computational Intelligence. Sanya, China, 23-24 October 2010. Dhaka: Institute of Electrical and Electronics Engineers.

Sharma, S.K. and Kumar, S., 2016. *Comparative analysis of Manhattan and Euclidean distance metrics using A* algorithm*. Journal of Research in Engineering and Applied Science. 4(1), pp. 196 – 198.

Singhata, N., 2021. Autonomous mobile robot using vision system and ESP8266 Node MCU board. *Current Applied Science and Technology*, 21(3), pp. 467 – 480.

Vargas, A.J.O., Serrano, J.E.C., Acuna, L.C. and Martinez-Santos, J.C., 2020. *Path Planning for Non-Playable Characters in Arcade Video Games using the Wavefront Algorithm*. IEEE Games, Multimedia, Animation and Multiple Realities, GMAX 2020. Barranquilla, Colombia, 17-18 September 2020. Institute of Electrical and Electronics Engineers.

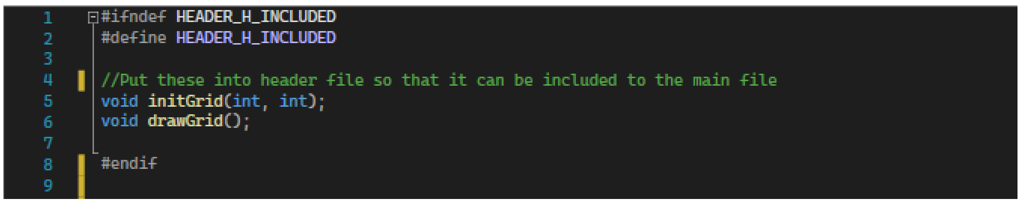
Wise, M., 2022, *Could AMRs close the labor gap completely? - mobile robot guide* [Online]. Available at: <https://mobilerobotguide.com/2022/02/11/could-amrs-close-the-labor-gap-completely/> [Accessed 18 April 2022].

Zarembo, I. and Kodors, S., 2013. *Pathfinding algorithm efficiency analysis in 2D grid. Environment. Technology.* Resources Proceedings of the 9th International Scientific and Practical Conference. Rēzekne, Latvia, 20-22 June 2013. Rēzeknes Augstskola.

Zidane, I.M. and Ibrahim, K., 2018. *Wavefront and a-star algorithms for mobile robot path planning*. International Conference on Advanced Intelligent Systems and Informatics. Cairo, Egypt, 1-3 September 2018. Springer Verlag.

APPENDICES

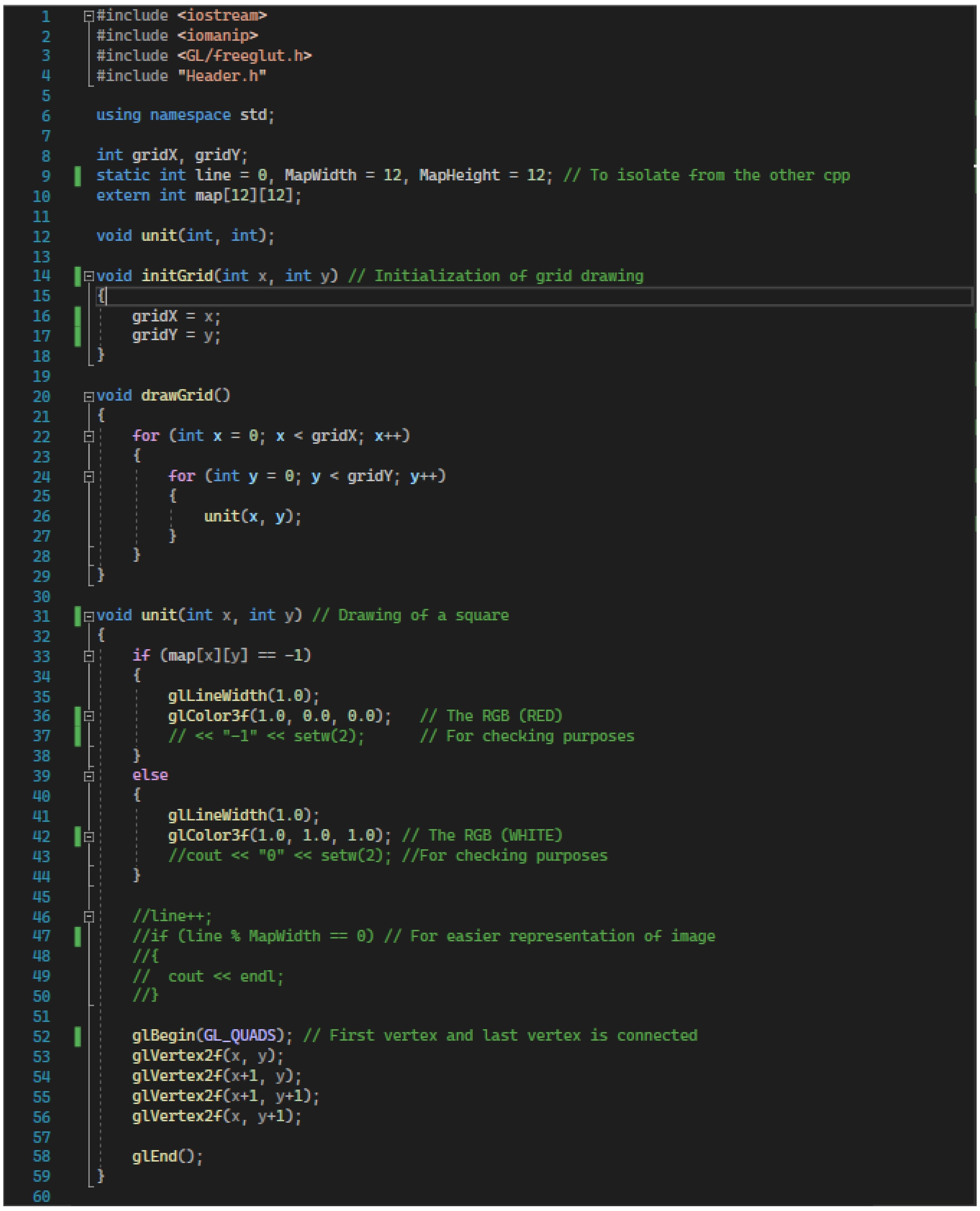
Appendix A: C++ code for Header.h



```cpp
#ifndef HEADER_H_INCLUDED
#define HEADER_H_INCLUDED

//Put these into header file so that it can be included to the main file
void initGrid(int, int);
void drawGrid();

#endif

```

Figure A-1: Header.h file

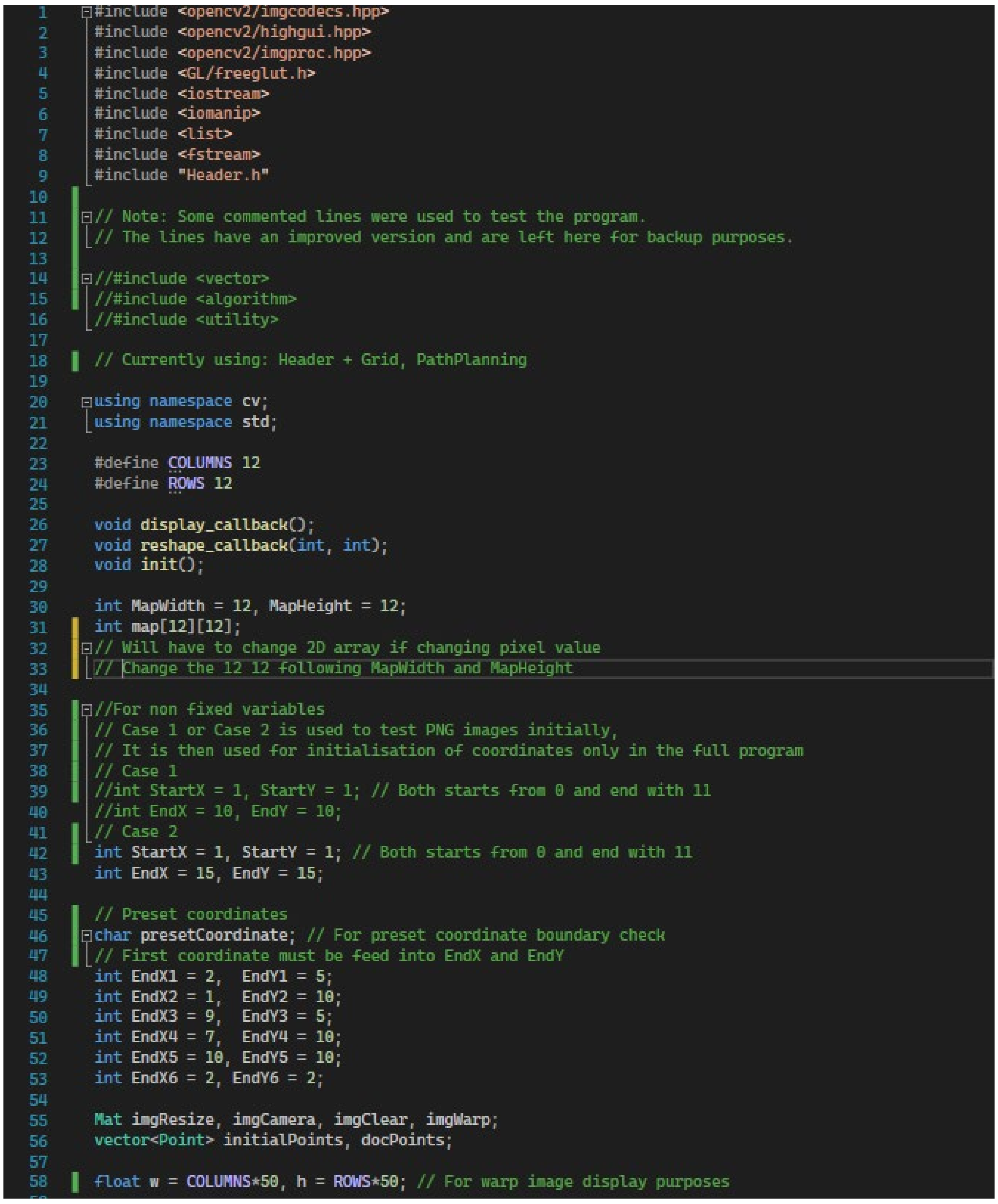
Appendix B: C++ code for Grid.cpp



```cpp
#include <iostream>
#include <iomanip>
#include <GL/freeglut.h>
#include "Header.h"

using namespace std;

int gridX, gridY;
static int line = 0, MapWidth = 12, MapHeight = 12; // To isolate from the other cpp
extern int map[12][12];

void unit(int, int);

void initGrid(int x, int y) // Initialization of grid drawing
{
    gridX = x;
    gridY = y;
}

void drawGrid()
{
    for (int x = 0; x < gridX; x++)
    {
        for (int y = 0; y < gridY; y++)
        {
            unit(x, y);
        }
    }
}

void unit(int x, int y) // Drawing of a square
{
    if (map[x][y] == -1)
    {
        glLineWidth(1.0);
        glColor3f(1.0, 0.0, 0.0);   // The RGB (RED)
        // << "-1" << setw(2);      // For checking purposes
    }
    else
    {
        glLineWidth(1.0);
        glColor3f(1.0, 1.0, 1.0); // The RGB (WHITE)
        //cout << "0" << setw(2); //For checking purposes
    }

    //line++;
    //if (line % MapWidth == 0) // For easier representation of image
    //{
    //  cout << endl;
    //}

    glBegin(GL_QUADS); // First vertex and last vertex is connected
    glVertex2f(x, y);
    glVertex2f(x+1, y);
    glVertex2f(x+1, y+1);
    glVertex2f(x, y+1);

    glEnd();
}

```

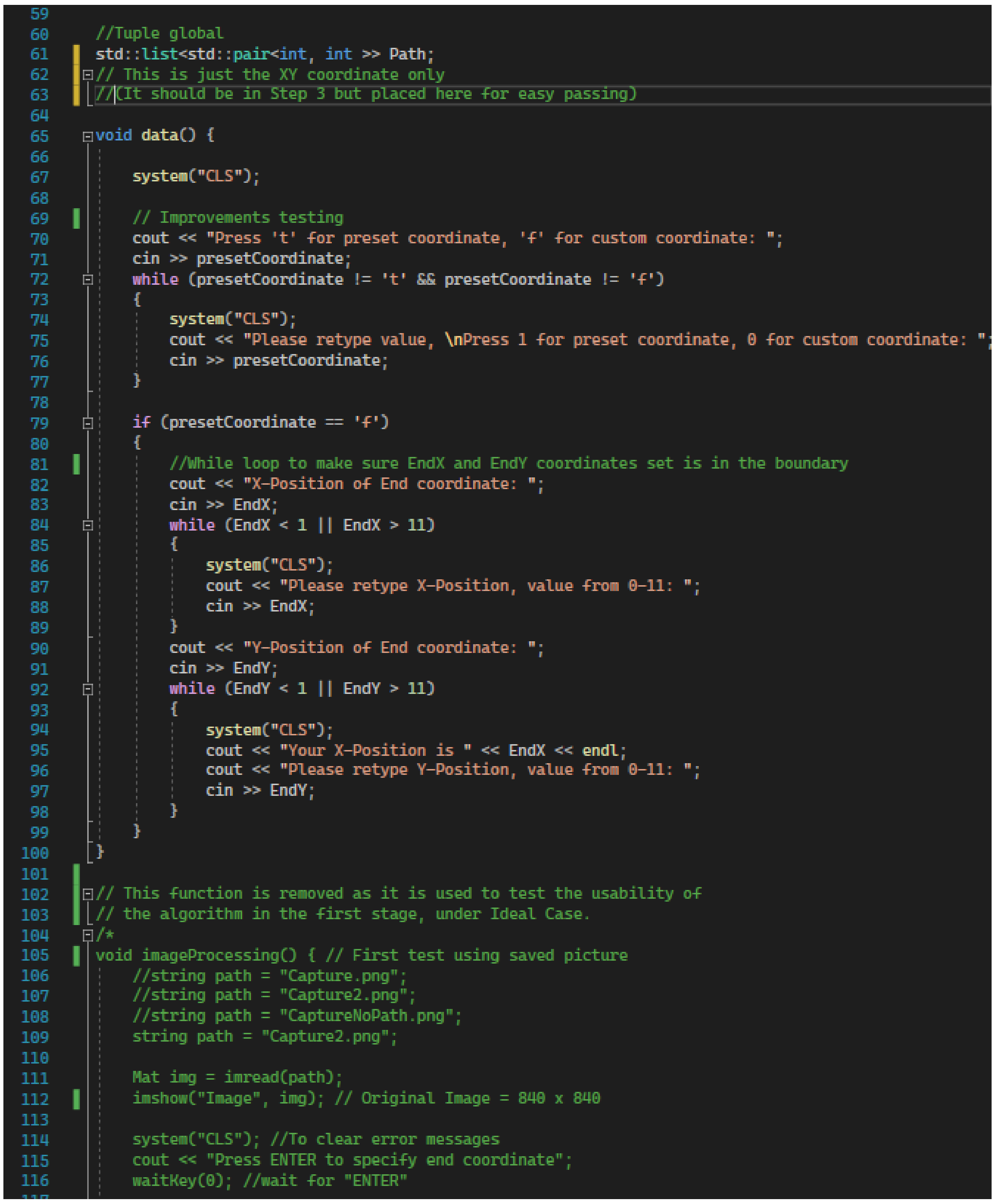
Figure B-1: Grid.cpp Program

Appendix C: C++ code for Pathplanning.cpp



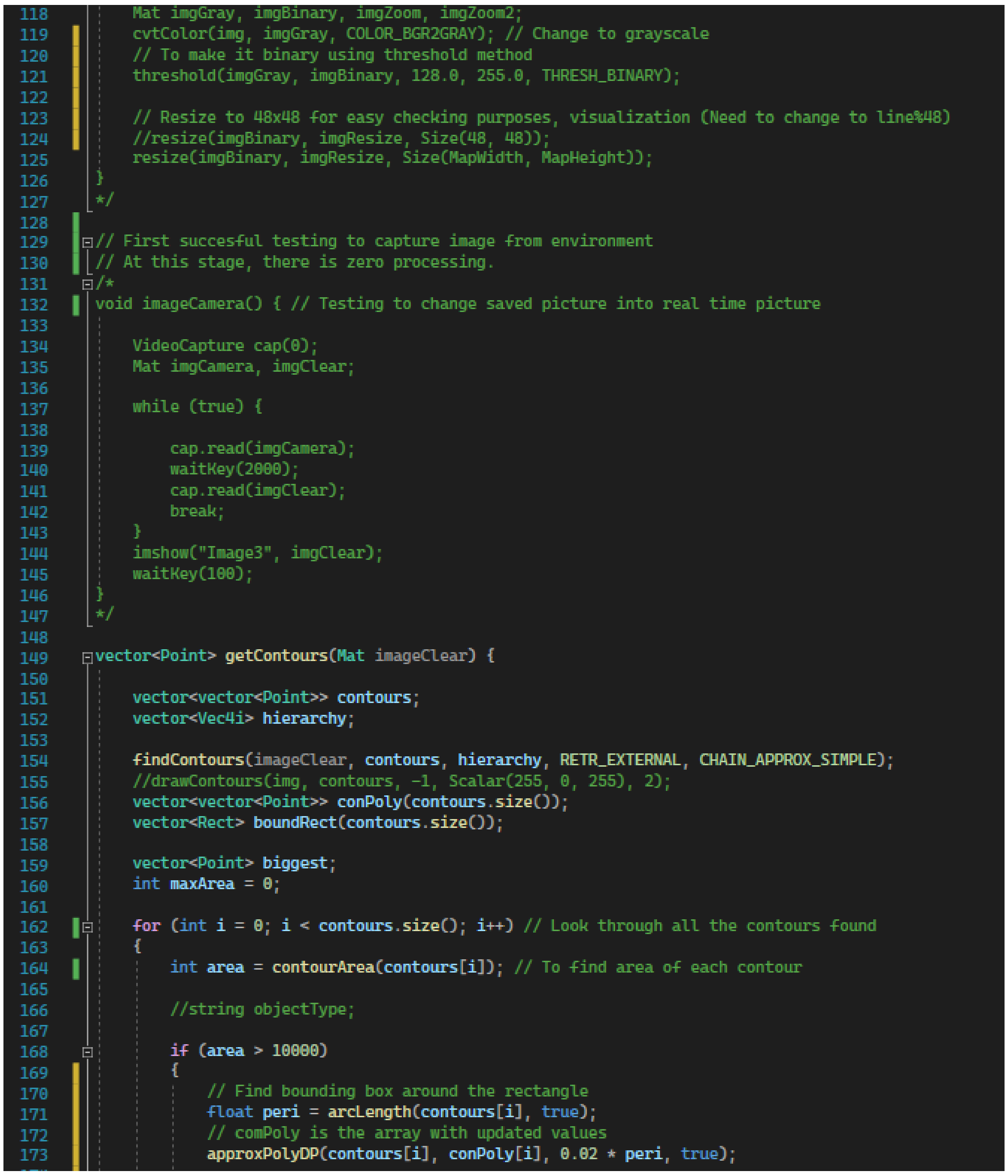
```cpp
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <GL/freeglut.h>
#include <iostream>
#include <iomanip>
#include <list>
#include <fstream>
#include "Header.h"

// Note: Some commented lines were used to test the program.
// The lines have an improved version and are left here for backup purposes.

//#include <vector>
//#include <algorithm>
//#include <utility>

// Currently using: Header + Grid, PathPlanning

using namespace cv;
using namespace std;

#define COLUMNS 12
#define ROWS 12

void display_callback();
void reshape_callback(int, int);
void init();

int MapWidth = 12, MapHeight = 12;
int map[12][12];
// Will have to change 2D array if changing pixel value
// Change the 12 12 following MapWidth and MapHeight

//For non fixed variables
// Case 1 or Case 2 is used to test PNG images initially,
// It is then used for initialisation of coordinates only in the full program
// Case 1
//int StartX = 1, StartY = 1; // Both starts from 0 and end with 11
//int EndX = 10, EndY = 10;
// Case 2
int StartX = 1, StartY = 1; // Both starts from 0 and end with 11
int EndX = 15, EndY = 15;

// Preset coordinates
char presetCoordinate; // For preset coordinate boundary check
// First coordinate must be feed into EndX and EndY
int EndX1 = 2,  EndY1 = 5;
int EndX2 = 1,  EndY2 = 10;
int EndX3 = 9,  EndY3 = 5;
int EndX4 = 7,  EndY4 = 10;
int EndX5 = 10, EndY5 = 10;
int EndX6 = 2, EndY6 = 2;

Mat imgResize, imgCamera, imgClear, imgWarp;
vector<Point> initialPoints, docPoints;

float w = COLUMNS*50, h = ROWS*50; // For warp image display purposes
```

Figure C-1: Pathplanning.cpp Program (1)



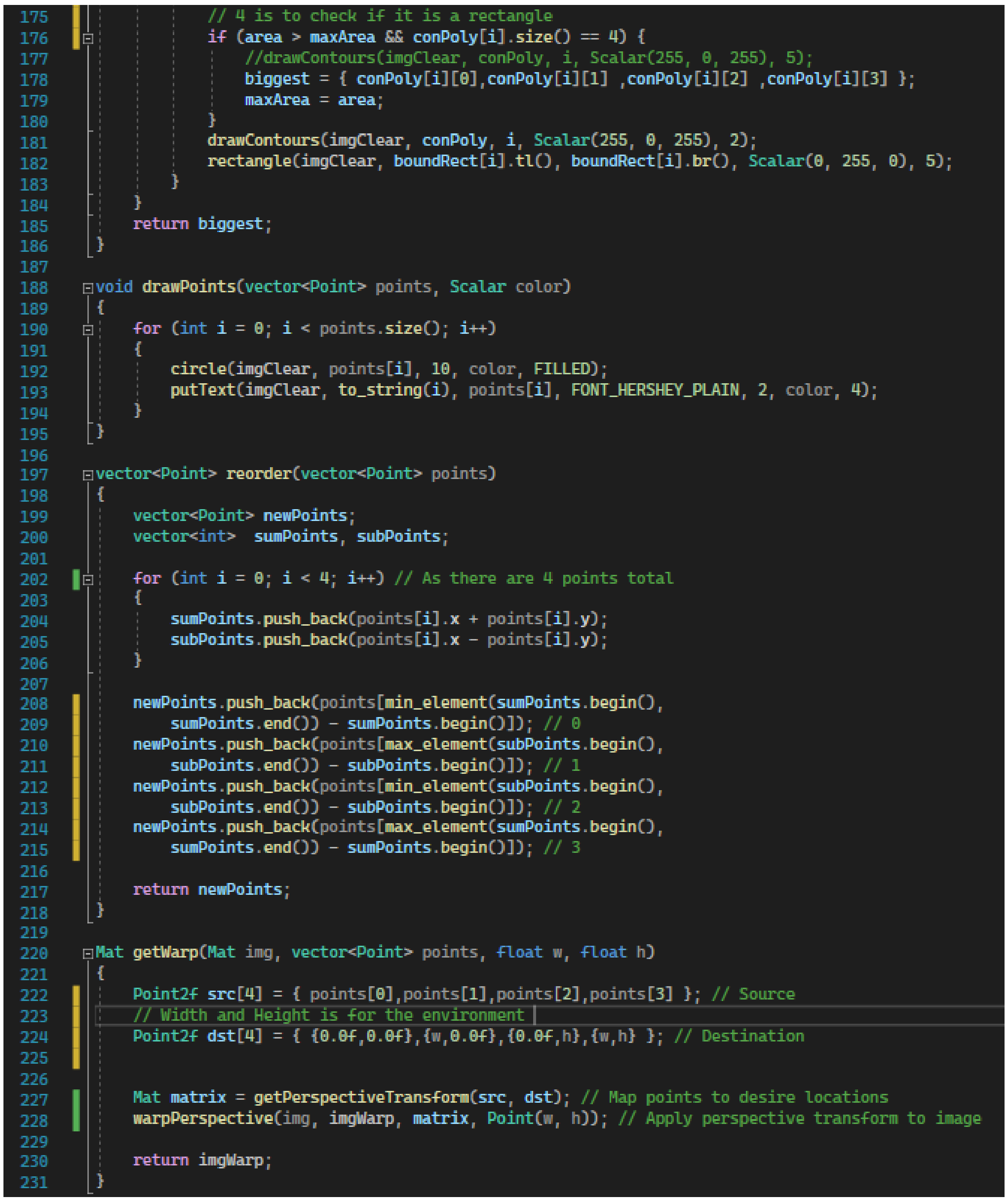
```cpp

//Tuple global
std::list<std::pair<int, int >> Path;
// This is just the XY coordinate only
//(It should be in Step 3 but placed here for easy passing)

void data() {

    system("CLS");

    // Improvements testing
    cout << "Press 't' for preset coordinate, 'f' for custom coordinate: ";
    cin >> presetCoordinate;
    while (presetCoordinate != 't' && presetCoordinate != 'f')
    {
        system("CLS");
        cout << "Please retype value, \nPress 1 for preset coordinate, 0 for custom coordinate: "
        cin >> presetCoordinate;
    }

    if (presetCoordinate == 'f')
    {
        //While loop to make sure EndX and EndY coordinates set is in the boundary
        cout << "X-Position of End coordinate: ";
        cin >> EndX;
        while (EndX < 1 || EndX > 11)
        {
            system("CLS");
            cout << "Please retype X-Position, value from 0-11: ";
            cin >> EndX;
        }
        cout << "Y-Position of End coordinate: ";
        cin >> EndY;
        while (EndY < 1 || EndY > 11)
        {
            system("CLS");
            cout << "Your X-Position is " << EndX << endl;
            cout << "Please retype Y-Position, value from 0-11: ";
            cin >> EndY;
        }
    }
}

// This function is removed as it is used to test the usability of
// the algorithm in the first stage, under Ideal Case.
/*
void imageProcessing() { // First test using saved picture
    //string path = "Capture.png";
    //string path = "Capture2.png";
    //string path = "CaptureNoPath.png";
    string path = "Capture2.png";

    Mat img = imread(path);
    imshow("Image", img); // Original Image = 840 x 840

    system("CLS"); //To clear error messages
    cout << "Press ENTER to specify end coordinate";
    waitKey(0); //wait for "ENTER"
```

Figure C-2: Pathplanning.cpp Program (2)



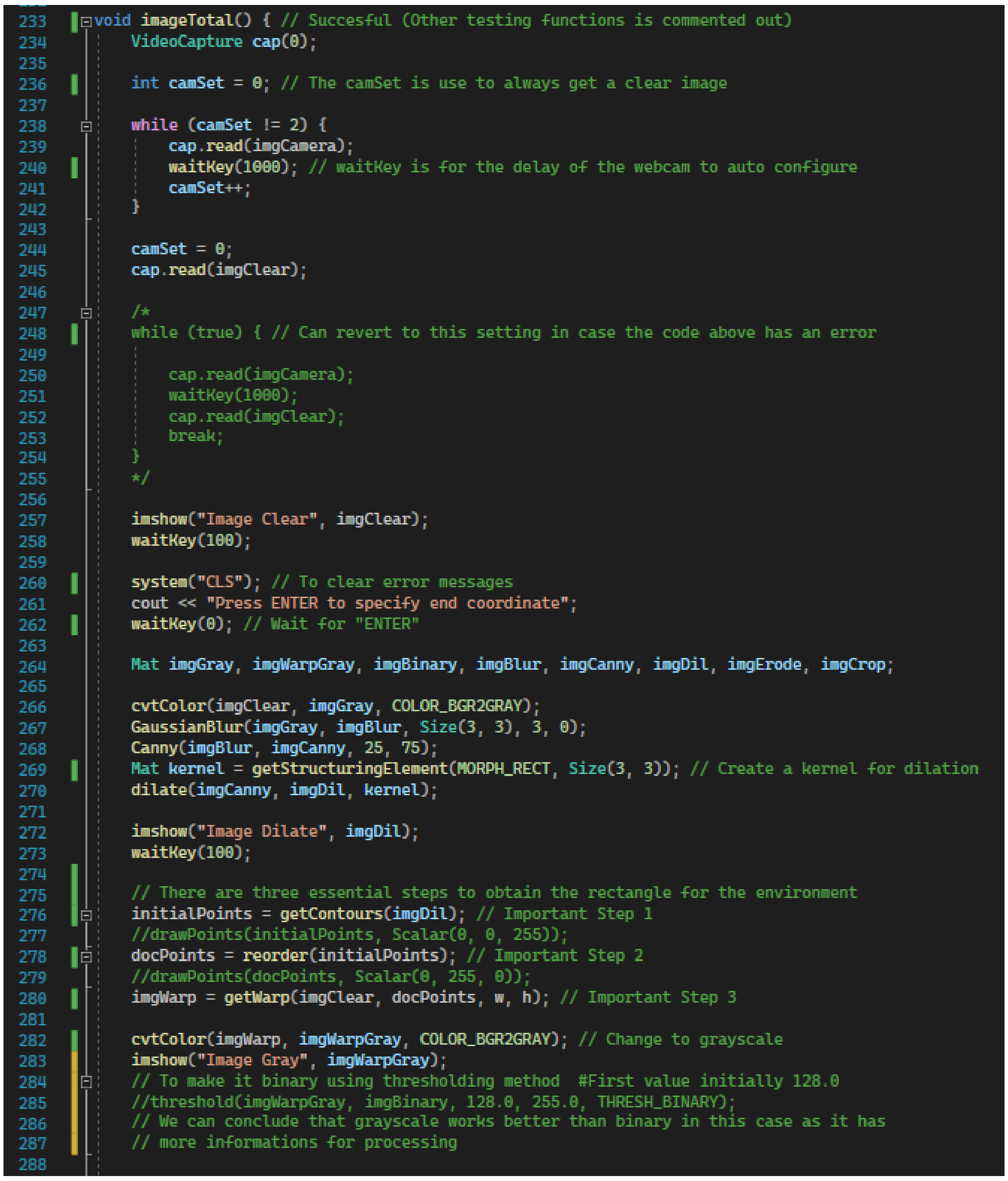
```
    Mat imgGray, imgBinary, imgZoom, imgZoom2;
    cvtColor(img, imgGray, COLOR_BGR2GRAY); // Change to grayscale
    // To make it binary using threshold method
    threshold(imgGray, imgBinary, 128.0, 255.0, THRESH_BINARY);

    // Resize to 48x48 for easy checking purposes, visualization (Need to change to line%48)
    //resize(imgBinary, imgResize, Size(48, 48));
    resize(imgBinary, imgResize, Size(MapWidth, MapHeight));
}
*/

// First succesful testing to capture image from environment
// At this stage, there is zero processing.
/*
void imageCamera() { // Testing to change saved picture into real time picture

    VideoCapture cap(0);
    Mat imgCamera, imgClear;

    while (true) {

        cap.read(imgCamera);
        waitKey(2000);
        cap.read(imgClear);
        break;
    }
    imshow("Image3", imgClear);
    waitKey(100);
}
*/

vector<Point> getContours(Mat imageClear) {

    vector<vector<Point>> contours;
    vector<Vec4i> hierarchy;

    findContours(imageClear, contours, hierarchy, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE);
    //drawContours(img, contours, -1, Scalar(255, 0, 255), 2);
    vector<vector<Point>> conPoly(contours.size());
    vector<Rect> boundRect(contours.size());

    vector<Point> biggest;
    int maxArea = 0;

    for (int i = 0; i < contours.size(); i++) // Look through all the contours found
    {
        int area = contourArea(contours[i]); // To find area of each contour

        //string objectType;

        if (area > 10000)
        {
            // Find bounding box around the rectangle
            float peri = arcLength(contours[i], true);
            // conPoly is the array with updated values
            approxPolyDP(contours[i], conPoly[i], 0.02 * peri, true);
```

Figure C-3: Pathplanning.cpp Program (3)



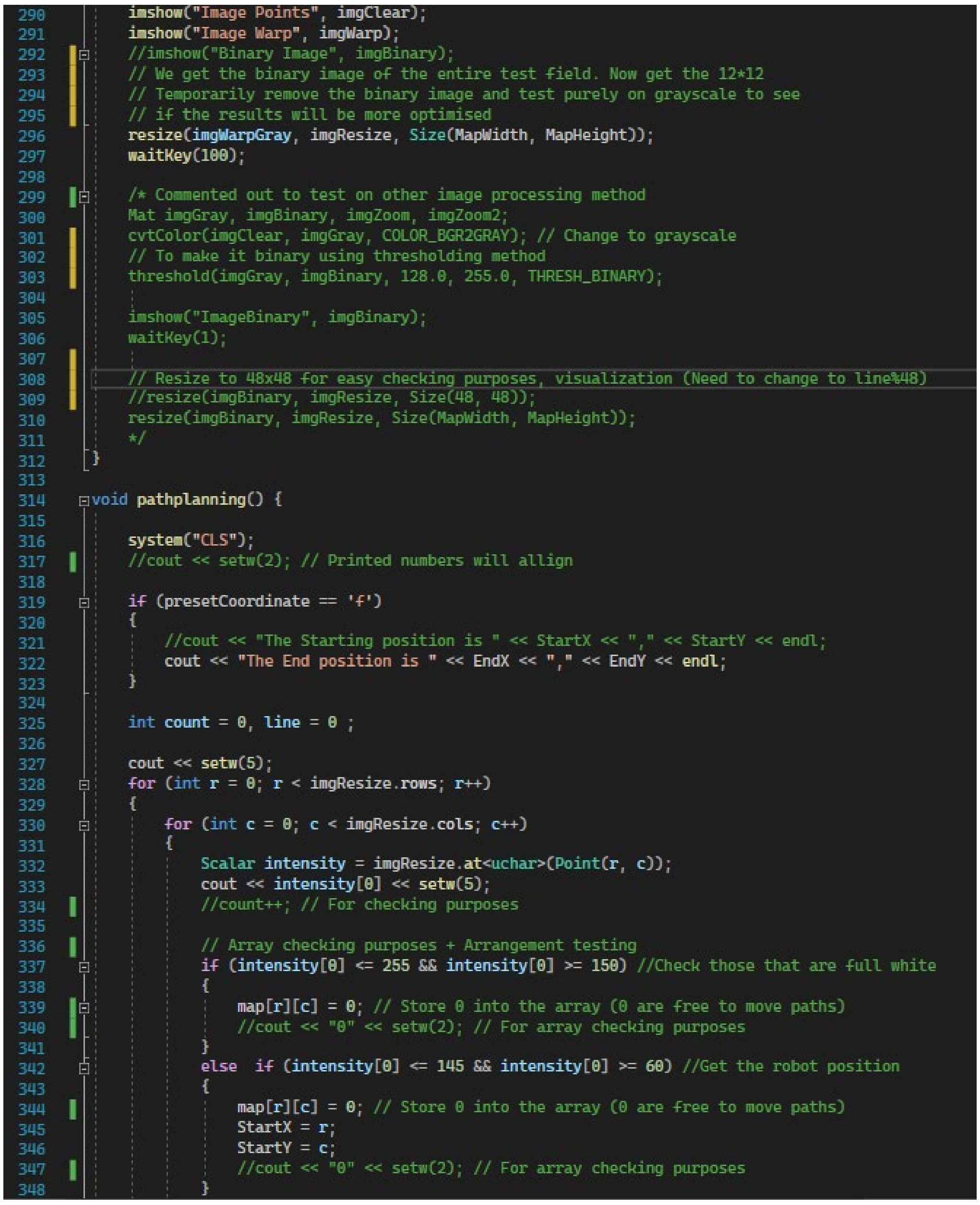
```
            // 4 is to check if it is a rectangle
            if (area > maxArea && conPoly[i].size() == 4) {
                //drawContours(imgClear, conPoly, i, Scalar(255, 0, 255), 5);
                biggest = { conPoly[i][0],conPoly[i][1] ,conPoly[i][2] ,conPoly[i][3] };
                maxArea = area;
            }
            drawContours(imgClear, conPoly, i, Scalar(255, 0, 255), 2);
            rectangle(imgClear, boundRect[i].tl(), boundRect[i].br(), Scalar(0, 255, 0), 5);
        }
    }
    return biggest;
}

void drawPoints(vector<Point> points, Scalar color)
{
    for (int i = 0; i < points.size(); i++)
    {
        circle(imgClear, points[i], 10, color, FILLED);
        putText(imgClear, to_string(i), points[i], FONT_HERSHEY_PLAIN, 2, color, 4);
    }
}

vector<Point> reorder(vector<Point> points)
{
    vector<Point> newPoints;
    vector<int>  sumPoints, subPoints;

    for (int i = 0; i < 4; i++) // As there are 4 points total
    {
        sumPoints.push_back(points[i].x + points[i].y);
        subPoints.push_back(points[i].x - points[i].y);
    }

    newPoints.push_back(points[min_element(sumPoints.begin(),
        sumPoints.end()) - sumPoints.begin()]); // 0
    newPoints.push_back(points[max_element(subPoints.begin(),
        subPoints.end()) - subPoints.begin()]); // 1
    newPoints.push_back(points[min_element(subPoints.begin(),
        subPoints.end()) - subPoints.begin()]); // 2
    newPoints.push_back(points[max_element(sumPoints.begin(),
        sumPoints.end()) - sumPoints.begin()]); // 3

    return newPoints;
}

Mat getWarp(Mat img, vector<Point> points, float w, float h)
{
    Point2f src[4] = { points[0],points[1],points[2],points[3] }; // Source
    // Width and Height is for the environment
    Point2f dst[4] = { {0.0f,0.0f},{w,0.0f},{0.0f,h},{w,h} }; // Destination


    Mat matrix = getPerspectiveTransform(src, dst); // Map points to desire locations
    warpPerspective(img, imgWarp, matrix, Point(w, h)); // Apply perspective transform to image

    return imgWarp;
}
```

Figure C-4: Pathplanning.cpp Program (4)



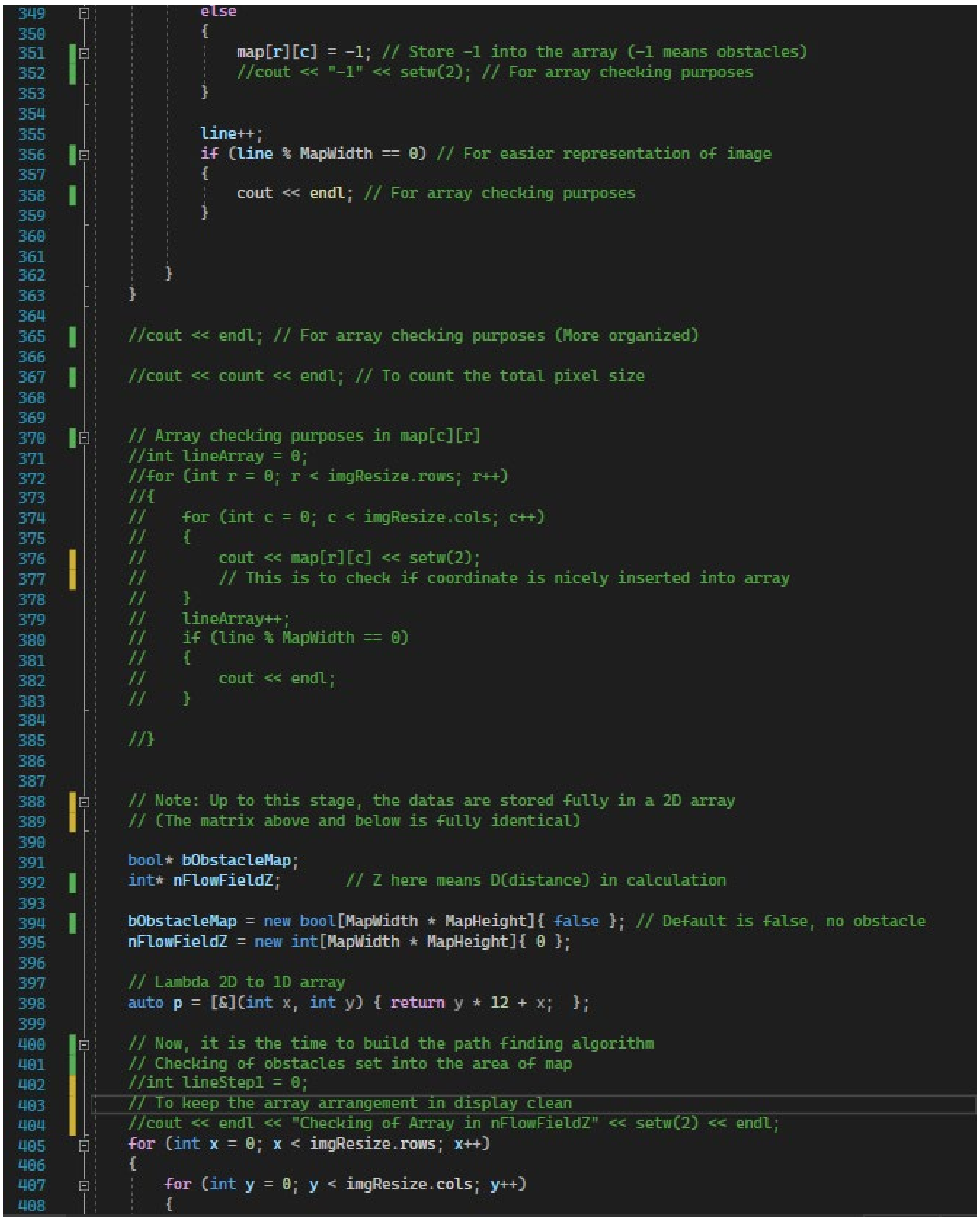
```cpp
void imageTotal() { // Succesful (Other testing functions is commented out)
    VideoCapture cap(0);

    int camSet = 0; // The camSet is use to always get a clear image

    while (camSet != 2) {
        cap.read(imgCamera);
        waitKey(1000); // waitKey is for the delay of the webcam to auto configure
        camSet++;
    }

    camSet = 0;
    cap.read(imgClear);

    /*
    while (true) { // Can revert to this setting in case the code above has an error

        cap.read(imgCamera);
        waitKey(1000);
        cap.read(imgClear);
        break;
    }
    */

    imshow("Image Clear", imgClear);
    waitKey(100);

    system("CLS"); // To clear error messages
    cout << "Press ENTER to specify end coordinate";
    waitKey(0); // Wait for "ENTER"

    Mat imgGray, imgWarpGray, imgBinary, imgBlur, imgCanny, imgDil, imgErode, imgCrop;

    cvtColor(imgClear, imgGray, COLOR_BGR2GRAY);
    GaussianBlur(imgGray, imgBlur, Size(3, 3), 3, 0);
    Canny(imgBlur, imgCanny, 25, 75);
    Mat kernel = getStructuringElement(MORPH_RECT, Size(3, 3)); // Create a kernel for dilation
    dilate(imgCanny, imgDil, kernel);

    imshow("Image Dilate", imgDil);
    waitKey(100);

    // There are three essential steps to obtain the rectangle for the environment
    initialPoints = getContours(imgDil); // Important Step 1
    //drawPoints(initialPoints, Scalar(0, 0, 255));
    docPoints = reorder(initialPoints); // Important Step 2
    //drawPoints(docPoints, Scalar(0, 255, 0));
    imgWarp = getWarp(imgClear, docPoints, w, h); // Important Step 3

    cvtColor(imgWarp, imgWarpGray, COLOR_BGR2GRAY); // Change to grayscale
    imshow("Image Gray", imgWarpGray);
    // To make it binary using thresholding method  #First value initially 128.0
    //threshold(imgWarpGray, imgBinary, 128.0, 255.0, THRESH_BINARY);
    // We can conclude that grayscale works better than binary in this case as it has
    // more informations for processing
```

Figure C-5: Pathplanning.cpp Program (5)



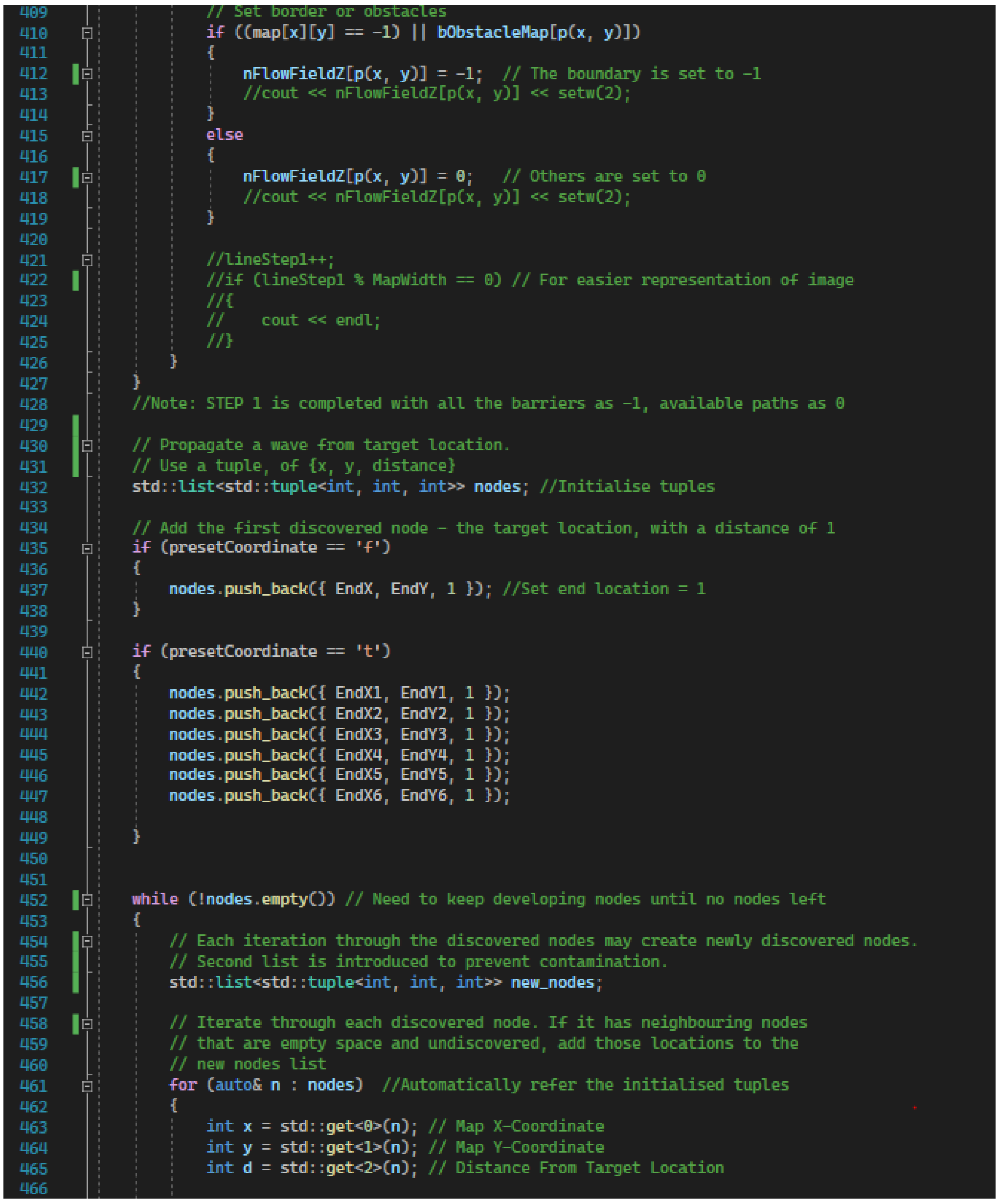
```
    imshow("Image Points", imgClear);
    imshow("Image Warp", imgWarp);
    //imshow("Binary Image", imgBinary);
    // We get the binary image of the entire test field. Now get the 12*12
    // Temporarily remove the binary image and test purely on grayscale to see
    // if the results will be more optimised
    resize(imgWarpGray, imgResize, Size(MapWidth, MapHeight));
    waitKey(100);

    /* Commented out to test on other image processing method
    Mat imgGray, imgBinary, imgZoom, imgZoom2;
    cvtColor(imgClear, imgGray, COLOR_BGR2GRAY); // Change to grayscale
    // To make it binary using thresholding method
    threshold(imgGray, imgBinary, 128.0, 255.0, THRESH_BINARY);

    imshow("ImageBinary", imgBinary);
    waitKey(1);

    // Resize to 48x48 for easy checking purposes, visualization (Need to change to line%48)
    //resize(imgBinary, imgResize, Size(48, 48));
    resize(imgBinary, imgResize, Size(MapWidth, MapHeight));
    */
}

void pathplanning() {

    system("CLS");
    //cout << setw(2); // Printed numbers will allign

    if (presetCoordinate == 'f')
    {
        //cout << "The Starting position is " << StartX << "," << StartY << endl;
        cout << "The End position is " << EndX << "," << EndY << endl;
    }

    int count = 0, line = 0 ;

    cout << setw(5);
    for (int r = 0; r < imgResize.rows; r++)
    {
        for (int c = 0; c < imgResize.cols; c++)
        {
            Scalar intensity = imgResize.at<uchar>(Point(r, c));
            cout << intensity[0] << setw(5);
            //count++; // For checking purposes

            // Array checking purposes + Arrangement testing
            if (intensity[0] <= 255 && intensity[0] >= 150) //Check those that are full white
            {
                map[r][c] = 0; // Store 0 into the array (0 are free to move paths)
                //cout << "0" << setw(2); // For array checking purposes
            }
            else  if (intensity[0] <= 145 && intensity[0] >= 60) //Get the robot position
            {
                map[r][c] = 0; // Store 0 into the array (0 are free to move paths)
                StartX = r;
                StartY = c;
                //cout << "0" << setw(2); // For array checking purposes
            }
```

Figure C-6: Pathplanning.cpp Program (6)



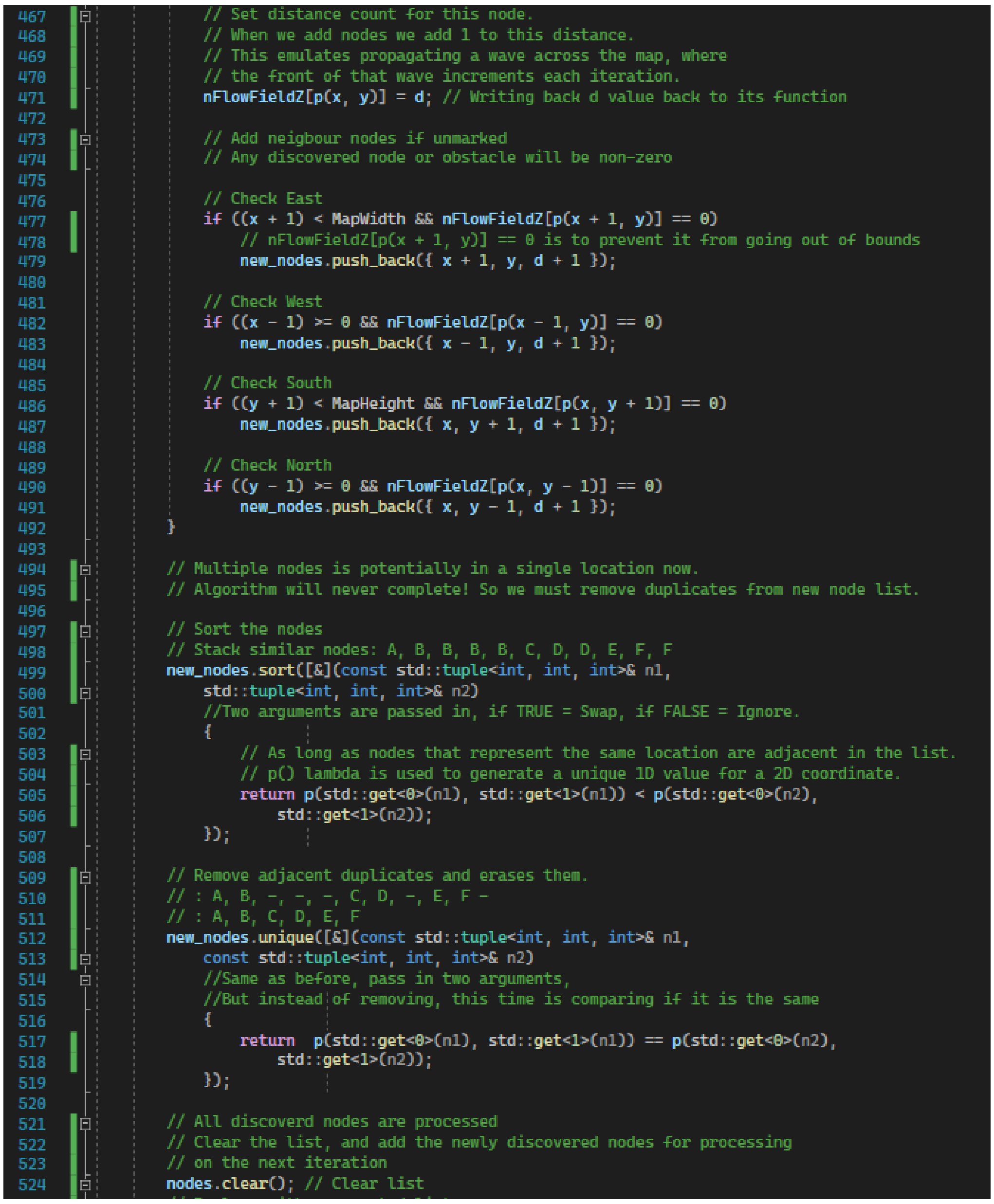
```
            else
            {
                map[r][c] = -1; // Store -1 into the array (-1 means obstacles)
                //cout << "-1" << setw(2); // For array checking purposes
            }

            line++;
            if (line % MapWidth == 0) // For easier representation of image
            {
                cout << endl; // For array checking purposes
            }


        }
    }

    //cout << endl; // For array checking purposes (More organized)

    //cout << count << endl; // To count the total pixel size


    // Array checking purposes in map[c][r]
    //int lineArray = 0;
    //for (int r = 0; r < imgResize.rows; r++)
    //{
    //    for (int c = 0; c < imgResize.cols; c++)
    //    {
    //        cout << map[r][c] << setw(2);
    //        // This is to check if coordinate is nicely inserted into array
    //    }
    //    lineArray++;
    //    if (line % MapWidth == 0)
    //    {
    //        cout << endl;
    //    }

    //}


    // Note: Up to this stage, the datas are stored fully in a 2D array
    // (The matrix above and below is fully identical)

    bool* bObstacleMap;
    int* nFlowFieldZ;       // Z here means D(distance) in calculation

    bObstacleMap = new bool[MapWidth * MapHeight]{ false }; // Default is false, no obstacle
    nFlowFieldZ = new int[MapWidth * MapHeight]{ 0 };

    // Lambda 2D to 1D array
    auto p = [&](int x, int y) { return y * 12 + x;  };

    // Now, it is the time to build the path finding algorithm
    // Checking of obstacles set into the area of map
    //int lineStep1 = 0;
    // To keep the array arrangement in display clean
    //cout << endl << "Checking of Array in nFlowFieldZ" << setw(2) << endl;
    for (int x = 0; x < imgResize.rows; x++)
    {
        for (int y = 0; y < imgResize.cols; y++)
        {
```

Figure C-7: Pathplanning.cpp Program (7)



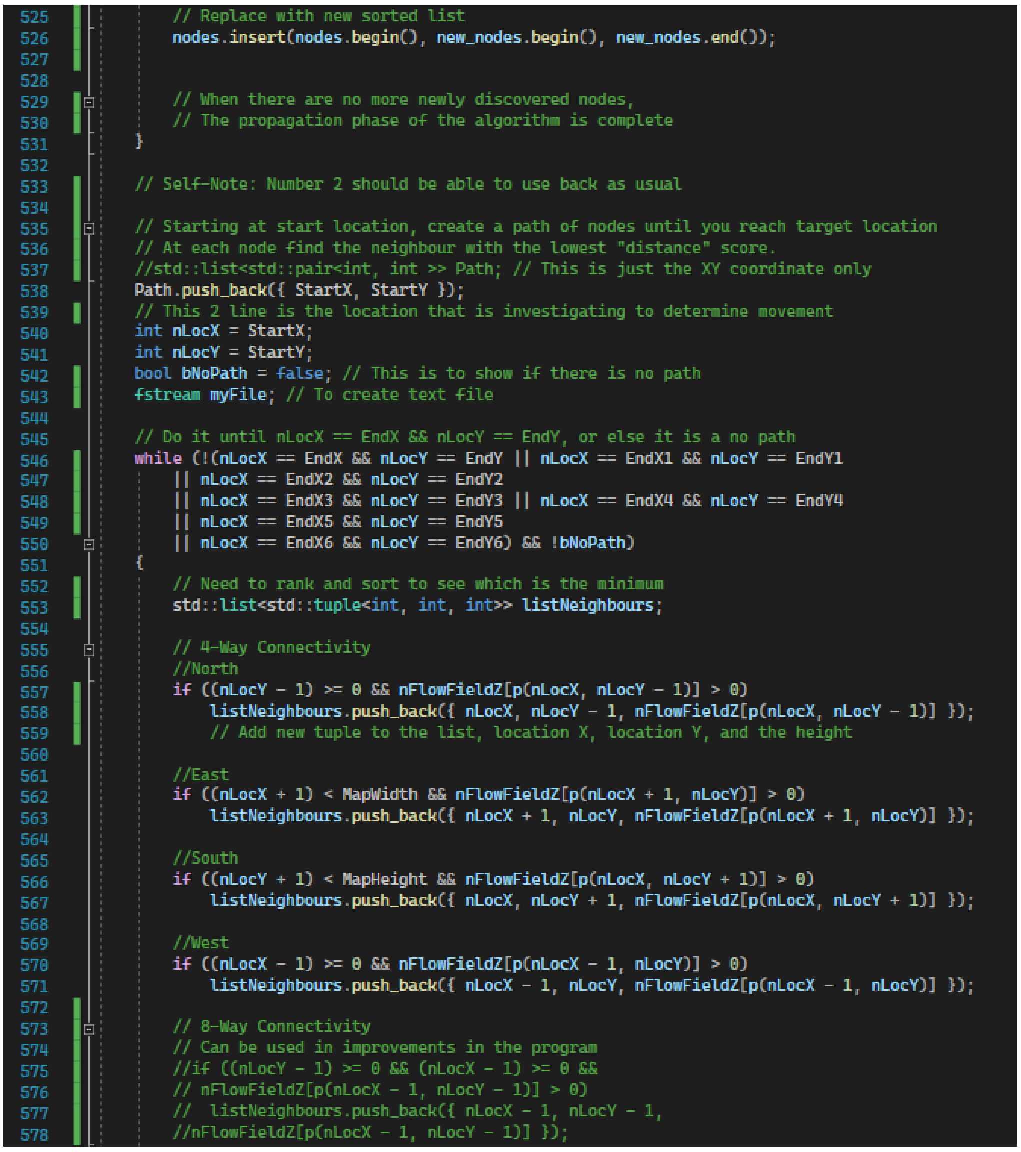
```
            // Set border or obstacles
            if ((map[x][y] == -1) || bObstacleMap[p(x, y)])
            {
                nFlowFieldZ[p(x, y)] = -1;  // The boundary is set to -1
                //cout << nFlowFieldZ[p(x, y)] << setw(2);
            }
            else
            {
                nFlowFieldZ[p(x, y)] = 0;   // Others are set to 0
                //cout << nFlowFieldZ[p(x, y)] << setw(2);
            }

            //lineStep1++;
            //if (lineStep1 % MapWidth == 0) // For easier representation of image
            //{
            //    cout << endl;
            //}
        }
    }
    //Note: STEP 1 is completed with all the barriers as -1, available paths as 0

    // Propagate a wave from target location.
    // Use a tuple, of {x, y, distance}
    std::list<std::tuple<int, int, int>> nodes; //Initialise tuples

    // Add the first discovered node - the target location, with a distance of 1
    if (presetCoordinate == 'f')
    {
        nodes.push_back({ EndX, EndY, 1 }); //Set end location = 1
    }

    if (presetCoordinate == 't')
    {
        nodes.push_back({ EndX1, EndY1, 1 });
        nodes.push_back({ EndX2, EndY2, 1 });
        nodes.push_back({ EndX3, EndY3, 1 });
        nodes.push_back({ EndX4, EndY4, 1 });
        nodes.push_back({ EndX5, EndY5, 1 });
        nodes.push_back({ EndX6, EndY6, 1 });

    }


    while (!nodes.empty()) // Need to keep developing nodes until no nodes left
    {
        // Each iteration through the discovered nodes may create newly discovered nodes.
        // Second list is introduced to prevent contamination.
        std::list<std::tuple<int, int, int>> new_nodes;

        // Iterate through each discovered node. If it has neighbouring nodes
        // that are empty space and undiscovered, add those locations to the
        // new nodes list
        for (auto& n : nodes)  //Automatically refer the initialised tuples
        {
            int x = std::get<0>(n); // Map X-Coordinate
            int y = std::get<1>(n); // Map Y-Coordinate
            int d = std::get<2>(n); // Distance From Target Location

```

Figure C-8: Pathplanning.cpp Program (8)



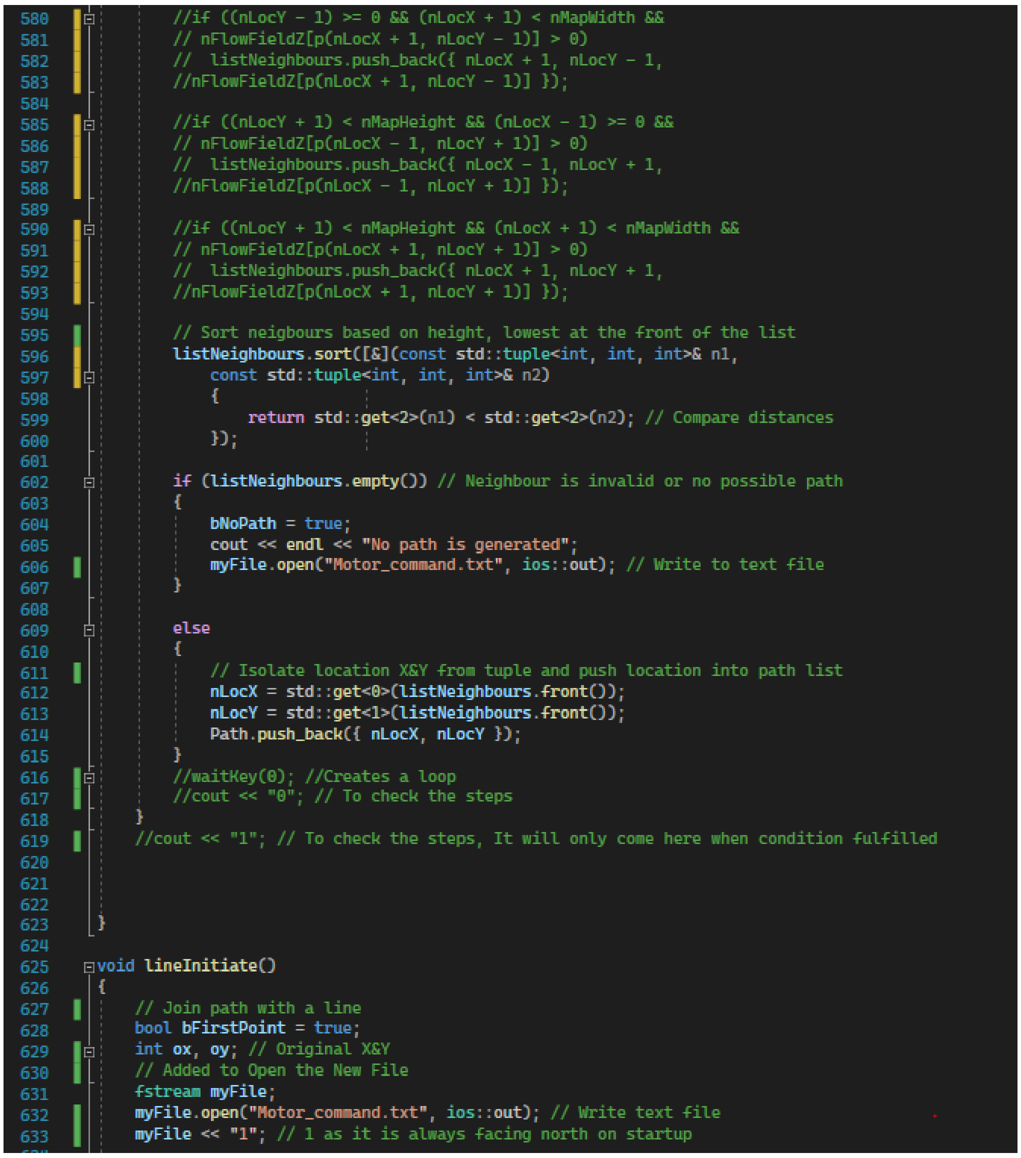
```
            // Set distance count for this node.
            // When we add nodes we add 1 to this distance.
            // This emulates propagating a wave across the map, where
            // the front of that wave increments each iteration.
            nFlowFieldZ[p(x, y)] = d; // Writing back d value back to its function

            // Add neigbour nodes if unmarked
            // Any discovered node or obstacle will be non-zero

            // Check East
            if ((x + 1) < MapWidth && nFlowFieldZ[p(x + 1, y)] == 0)
                // nFlowFieldZ[p(x + 1, y)] == 0 is to prevent it from going out of bounds
                new_nodes.push_back({ x + 1, y, d + 1 });

            // Check West
            if ((x - 1) >= 0 && nFlowFieldZ[p(x - 1, y)] == 0)
                new_nodes.push_back({ x - 1, y, d + 1 });

            // Check South
            if ((y + 1) < MapHeight && nFlowFieldZ[p(x, y + 1)] == 0)
                new_nodes.push_back({ x, y + 1, d + 1 });

            // Check North
            if ((y - 1) >= 0 && nFlowFieldZ[p(x, y - 1)] == 0)
                new_nodes.push_back({ x, y - 1, d + 1 });
        }

        // Multiple nodes is potentially in a single location now.
        // Algorithm will never complete! So we must remove duplicates from new node list.

        // Sort the nodes
        // Stack similar nodes: A, B, B, B, B, C, D, D, E, F, F
        new_nodes.sort([&](const std::tuple<int, int, int>& n1,
            std::tuple<int, int, int>& n2)
            //Two arguments are passed in, if TRUE = Swap, if FALSE = Ignore.
            {
                // As long as nodes that represent the same location are adjacent in the list.
                // p() lambda is used to generate a unique 1D value for a 2D coordinate.
                return p(std::get<0>(n1), std::get<1>(n1)) < p(std::get<0>(n2),
                    std::get<1>(n2));
            });

        // Remove adjacent duplicates and erases them.
        // : A, B, -, -, -, C, D, -, E, F -
        // : A, B, C, D, E, F
        new_nodes.unique([&](const std::tuple<int, int, int>& n1,
            const std::tuple<int, int, int>& n2)
            //Same as before, pass in two arguments,
            //But instead of removing, this time is comparing if it is the same
            {
                return  p(std::get<0>(n1), std::get<1>(n1)) == p(std::get<0>(n2),
                    std::get<1>(n2));
            });

        // All discoverd nodes are processed
        // Clear the list, and add the newly discovered nodes for processing
        // on the next iteration
        nodes.clear(); // Clear list
```

Figure C-9: Pathplanning.cpp Program (9)



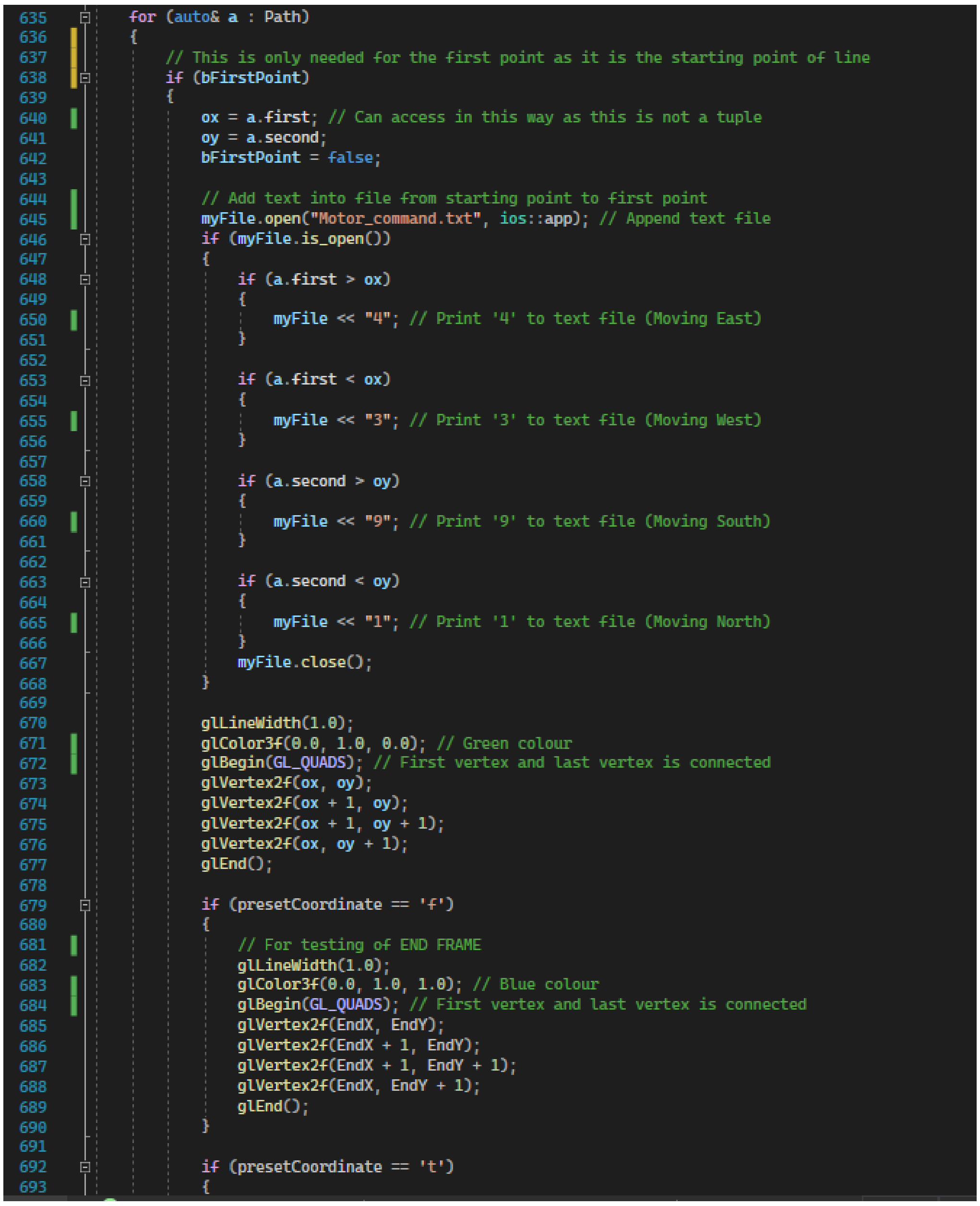
```
        // Replace with new sorted list
        nodes.insert(nodes.begin(), new_nodes.begin(), new_nodes.end());


        // When there are no more newly discovered nodes,
        // The propagation phase of the algorithm is complete
    }

    // Self-Note: Number 2 should be able to use back as usual

    // Starting at start location, create a path of nodes until you reach target location
    // At each node find the neighbour with the lowest "distance" score.
    //std::list<std::pair<int, int >> Path; // This is just the XY coordinate only
    Path.push_back({ StartX, StartY });
    // This 2 line is the location that is investigating to determine movement
    int nLocX = StartX;
    int nLocY = StartY;
    bool bNoPath = false; // This is to show if there is no path
    fstream myFile; // To create text file

    // Do it until nLocX == EndX && nLocY == EndY, or else it is a no path
    while (!(nLocX == EndX && nLocY == EndY || nLocX == EndX1 && nLocY == EndY1
        || nLocX == EndX2 && nLocY == EndY2
        || nLocX == EndX3 && nLocY == EndY3 || nLocX == EndX4 && nLocY == EndY4
        || nLocX == EndX5 && nLocY == EndY5
        || nLocX == EndX6 && nLocY == EndY6) && !bNoPath)
    {
        // Need to rank and sort to see which is the minimum
        std::list<std::tuple<int, int, int>> listNeighbours;

        // 4-Way Connectivity
        //North
        if ((nLocY - 1) >= 0 && nFlowFieldZ[p(nLocX, nLocY - 1)] > 0)
            listNeighbours.push_back({ nLocX, nLocY - 1, nFlowFieldZ[p(nLocX, nLocY - 1)] });
            // Add new tuple to the list, location X, location Y, and the height

        //East
        if ((nLocX + 1) < MapWidth && nFlowFieldZ[p(nLocX + 1, nLocY)] > 0)
            listNeighbours.push_back({ nLocX + 1, nLocY, nFlowFieldZ[p(nLocX + 1, nLocY)] });

        //South
        if ((nLocY + 1) < MapHeight && nFlowFieldZ[p(nLocX, nLocY + 1)] > 0)
            listNeighbours.push_back({ nLocX, nLocY + 1, nFlowFieldZ[p(nLocX, nLocY + 1)] });

        //West
        if ((nLocX - 1) >= 0 && nFlowFieldZ[p(nLocX - 1, nLocY)] > 0)
            listNeighbours.push_back({ nLocX - 1, nLocY, nFlowFieldZ[p(nLocX - 1, nLocY)] });

        // 8-Way Connectivity
        // Can be used in improvements in the program
        //if ((nLocY - 1) >= 0 && (nLocX - 1) >= 0 &&
        // nFlowFieldZ[p(nLocX - 1, nLocY - 1)] > 0)
        //  listNeighbours.push_back({ nLocX - 1, nLocY - 1,
        //nFlowFieldZ[p(nLocX - 1, nLocY - 1)] });
```

Figure C-10: Pathplanning.cpp Program (10)



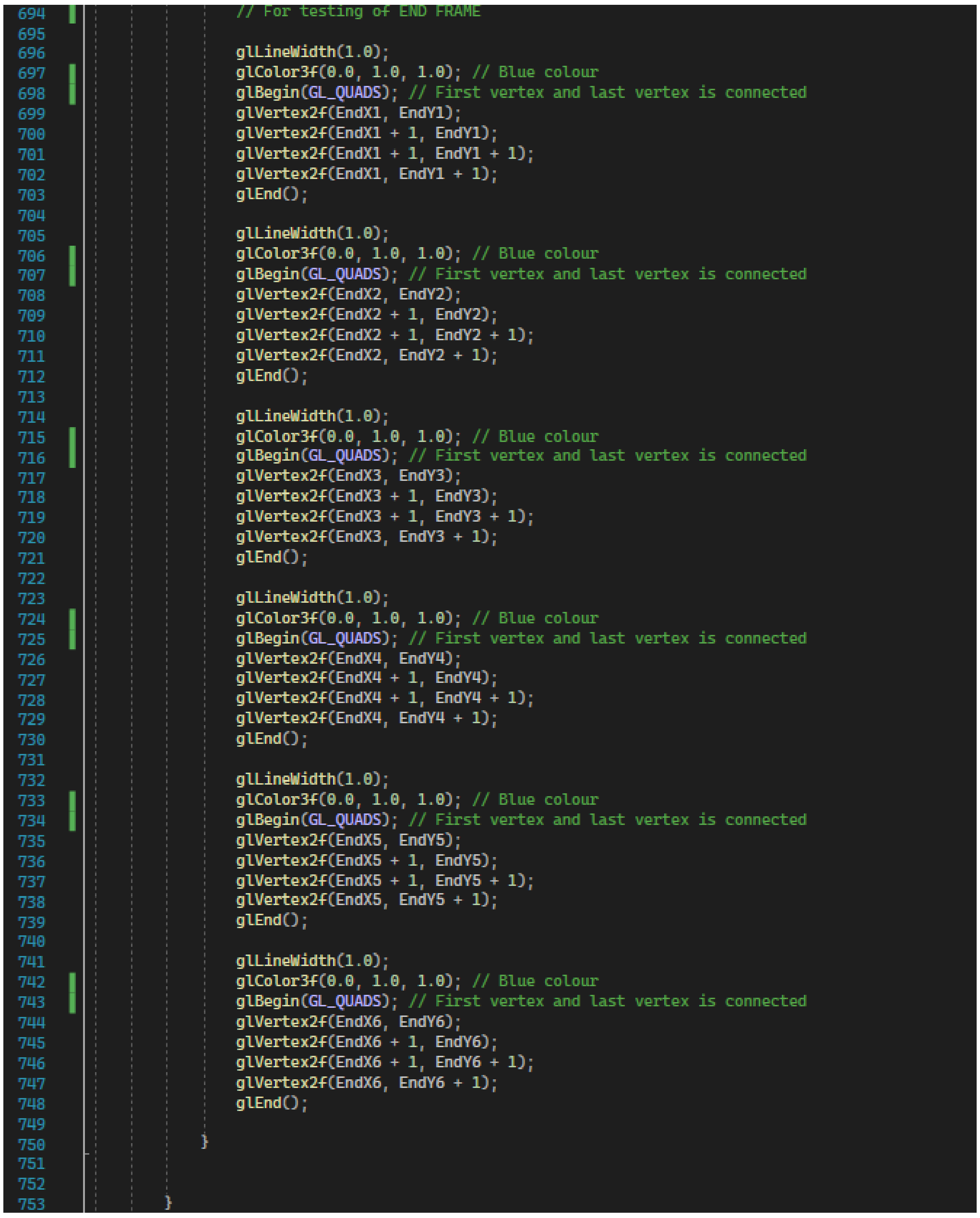
```
        //if ((nLocY - 1) >= 0 && (nLocX + 1) < nMapWidth &&
        // nFlowFieldZ[p(nLocX + 1, nLocY - 1)] > 0)
        //  listNeighbours.push_back({ nLocX + 1, nLocY - 1,
        //nFlowFieldZ[p(nLocX + 1, nLocY - 1)] });

        //if ((nLocY + 1) < nMapHeight && (nLocX - 1) >= 0 &&
        // nFlowFieldZ[p(nLocX - 1, nLocY + 1)] > 0)
        //  listNeighbours.push_back({ nLocX - 1, nLocY + 1,
        //nFlowFieldZ[p(nLocX - 1, nLocY + 1)] });

        //if ((nLocY + 1) < nMapHeight && (nLocX + 1) < nMapWidth &&
        // nFlowFieldZ[p(nLocX + 1, nLocY + 1)] > 0)
        //  listNeighbours.push_back({ nLocX + 1, nLocY + 1,
        //nFlowFieldZ[p(nLocX + 1, nLocY + 1)] });

        // Sort neigbours based on height, lowest at the front of the list
        listNeighbours.sort([&](const std::tuple<int, int, int>& n1,
            const std::tuple<int, int, int>& n2)
            {
                return std::get<2>(n1) < std::get<2>(n2); // Compare distances
            });

        if (listNeighbours.empty()) // Neighbour is invalid or no possible path
        {
            bNoPath = true;
            cout << endl << "No path is generated";
            myFile.open("Motor_command.txt", ios::out); // Write to text file
        }

        else
        {
            // Isolate location X&Y from tuple and push location into path list
            nLocX = std::get<0>(listNeighbours.front());
            nLocY = std::get<1>(listNeighbours.front());
            Path.push_back({ nLocX, nLocY });
        }
        //waitKey(0); //Creates a loop
        //cout << "0"; // To check the steps
    }
    //cout << "1"; // To check the steps, It will only come here when condition fulfilled




}

void lineInitiate()
{
    // Join path with a line
    bool bFirstPoint = true;
    int ox, oy; // Original X&Y
    // Added to Open the New File
    fstream myFile;
    myFile.open("Motor_command.txt", ios::out); // Write text file
    myFile << "1"; // 1 as it is always facing north on startup
```

Figure C-11: Pathplanning.cpp Program (11)



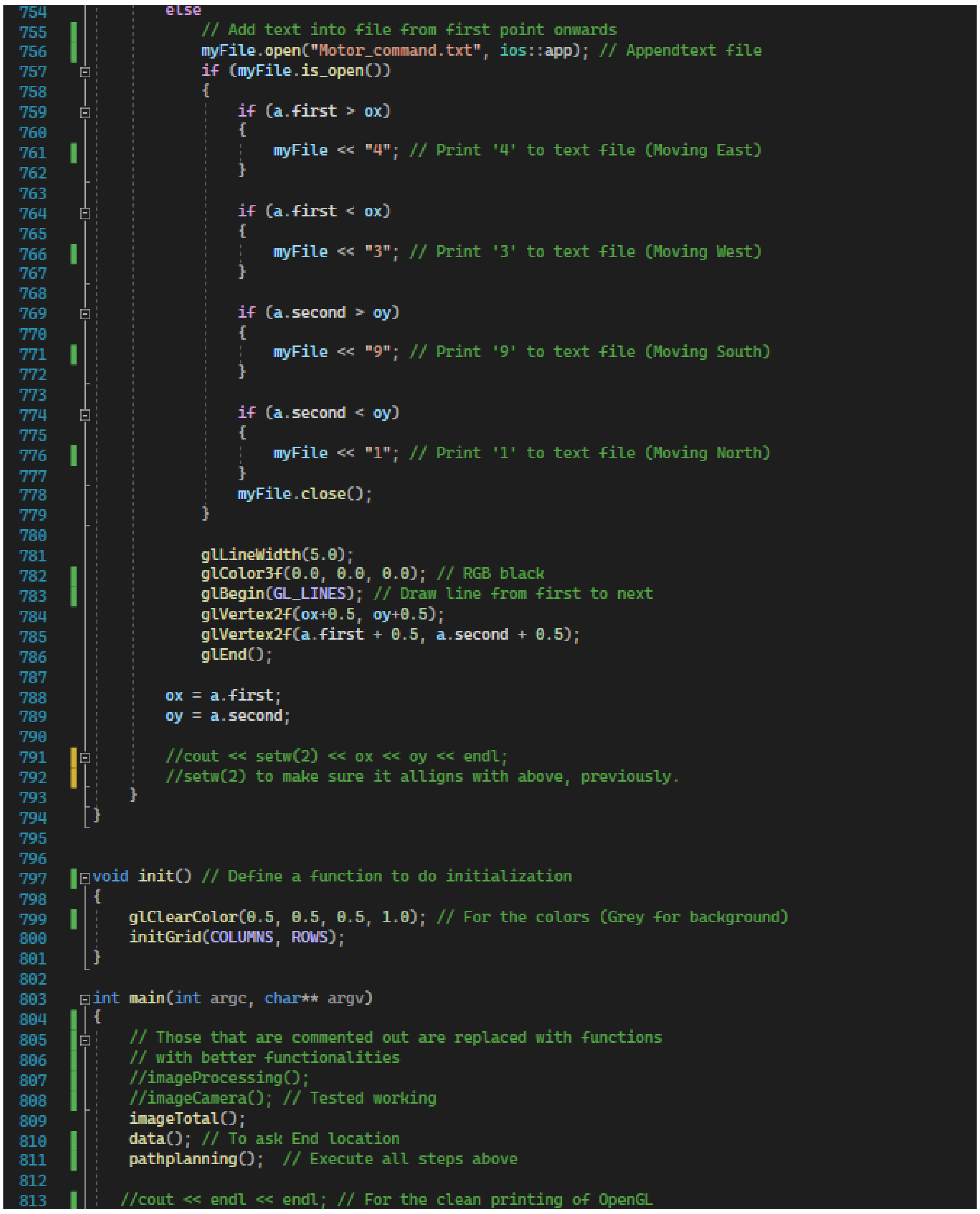
```cpp
    for (auto& a : Path)
    {
        // This is only needed for the first point as it is the starting point of line
        if (bFirstPoint)
        {
            ox = a.first; // Can access in this way as this is not a tuple
            oy = a.second;
            bFirstPoint = false;

            // Add text into file from starting point to first point
            myFile.open("Motor_command.txt", ios::app); // Append text file
            if (myFile.is_open())
            {
                if (a.first > ox)
                {
                    myFile << "4"; // Print '4' to text file (Moving East)
                }

                if (a.first < ox)
                {
                    myFile << "3"; // Print '3' to text file (Moving West)
                }

                if (a.second > oy)
                {
                    myFile << "9"; // Print '9' to text file (Moving South)
                }

                if (a.second < oy)
                {
                    myFile << "1"; // Print '1' to text file (Moving North)
                }
                myFile.close();
            }

            glLineWidth(1.0);
            glColor3f(0.0, 1.0, 0.0); // Green colour
            glBegin(GL_QUADS); // First vertex and last vertex is connected
            glVertex2f(ox, oy);
            glVertex2f(ox + 1, oy);
            glVertex2f(ox + 1, oy + 1);
            glVertex2f(ox, oy + 1);
            glEnd();

            if (presetCoordinate == 'f')
            {
                // For testing of END FRAME
                glLineWidth(1.0);
                glColor3f(0.0, 1.0, 1.0); // Blue colour
                glBegin(GL_QUADS); // First vertex and last vertex is connected
                glVertex2f(EndX, EndY);
                glVertex2f(EndX + 1, EndY);
                glVertex2f(EndX + 1, EndY + 1);
                glVertex2f(EndX, EndY + 1);
                glEnd();
            }

            if (presetCoordinate == 't')
            {
```

Figure C-12: Pathplanning.cpp Program (12)



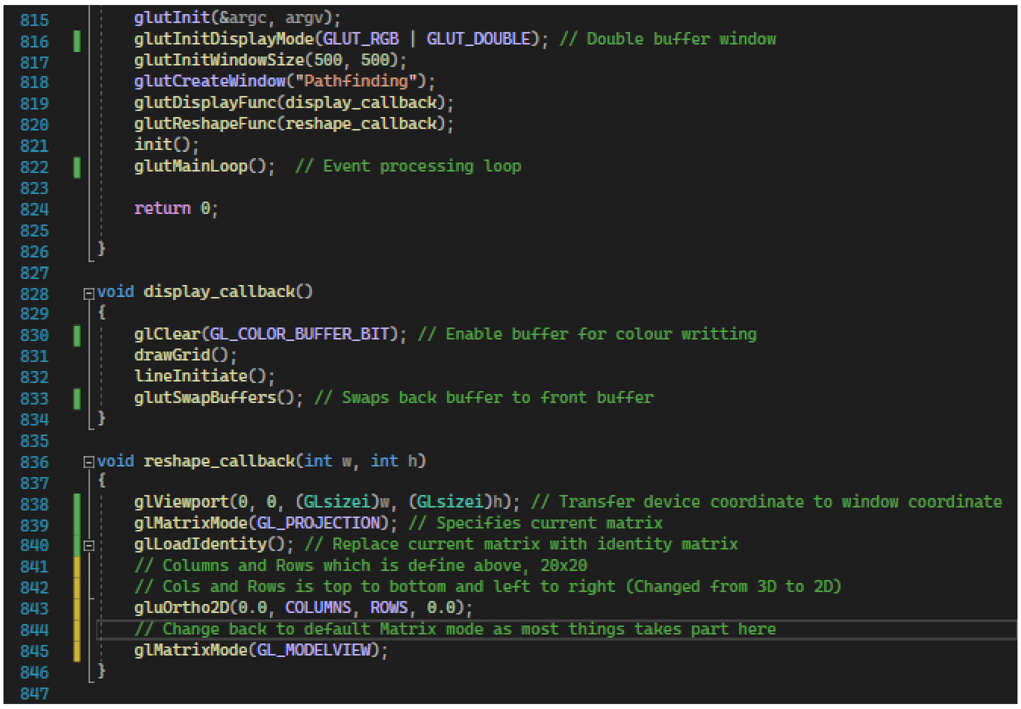
```
                // For testing of END FRAME

                glLineWidth(1.0);
                glColor3f(0.0, 1.0, 1.0); // Blue colour
                glBegin(GL_QUADS); // First vertex and last vertex is connected
                glVertex2f(EndX1, EndY1);
                glVertex2f(EndX1 + 1, EndY1);
                glVertex2f(EndX1 + 1, EndY1 + 1);
                glVertex2f(EndX1, EndY1 + 1);
                glEnd();

                glLineWidth(1.0);
                glColor3f(0.0, 1.0, 1.0); // Blue colour
                glBegin(GL_QUADS); // First vertex and last vertex is connected
                glVertex2f(EndX2, EndY2);
                glVertex2f(EndX2 + 1, EndY2);
                glVertex2f(EndX2 + 1, EndY2 + 1);
                glVertex2f(EndX2, EndY2 + 1);
                glEnd();

                glLineWidth(1.0);
                glColor3f(0.0, 1.0, 1.0); // Blue colour
                glBegin(GL_QUADS); // First vertex and last vertex is connected
                glVertex2f(EndX3, EndY3);
                glVertex2f(EndX3 + 1, EndY3);
                glVertex2f(EndX3 + 1, EndY3 + 1);
                glVertex2f(EndX3, EndY3 + 1);
                glEnd();

                glLineWidth(1.0);
                glColor3f(0.0, 1.0, 1.0); // Blue colour
                glBegin(GL_QUADS); // First vertex and last vertex is connected
                glVertex2f(EndX4, EndY4);
                glVertex2f(EndX4 + 1, EndY4);
                glVertex2f(EndX4 + 1, EndY4 + 1);
                glVertex2f(EndX4, EndY4 + 1);
                glEnd();

                glLineWidth(1.0);
                glColor3f(0.0, 1.0, 1.0); // Blue colour
                glBegin(GL_QUADS); // First vertex and last vertex is connected
                glVertex2f(EndX5, EndY5);
                glVertex2f(EndX5 + 1, EndY5);
                glVertex2f(EndX5 + 1, EndY5 + 1);
                glVertex2f(EndX5, EndY5 + 1);
                glEnd();

                glLineWidth(1.0);
                glColor3f(0.0, 1.0, 1.0); // Blue colour
                glBegin(GL_QUADS); // First vertex and last vertex is connected
                glVertex2f(EndX6, EndY6);
                glVertex2f(EndX6 + 1, EndY6);
                glVertex2f(EndX6 + 1, EndY6 + 1);
                glVertex2f(EndX6, EndY6 + 1);
                glEnd();

            }


        }
```

Figure C-13: Pathplanning.cpp Program (13)



```
            else
                // Add text into file from first point onwards
                myFile.open("Motor_command.txt", ios::app); // Appendtext file
                if (myFile.is_open())
                {
                    if (a.first > ox)
                    {
                        myFile << "4"; // Print '4' to text file (Moving East)
                    }

                    if (a.first < ox)
                    {
                        myFile << "3"; // Print '3' to text file (Moving West)
                    }

                    if (a.second > oy)
                    {
                        myFile << "9"; // Print '9' to text file (Moving South)
                    }

                    if (a.second < oy)
                    {
                        myFile << "1"; // Print '1' to text file (Moving North)
                    }
                    myFile.close();
                }

                glLineWidth(5.0);
                glColor3f(0.0, 0.0, 0.0); // RGB black
                glBegin(GL_LINES); // Draw line from first to next
                glVertex2f(ox+0.5, oy+0.5);
                glVertex2f(a.first + 0.5, a.second + 0.5);
                glEnd();

            ox = a.first;
            oy = a.second;

            //cout << setw(2) << ox << oy << endl;
            //setw(2) to make sure it alligns with above, previously.
        }
}


void init() // Define a function to do initialization
{
    glClearColor(0.5, 0.5, 0.5, 1.0); // For the colors (Grey for background)
    initGrid(COLUMNS, ROWS);
}

int main(int argc, char** argv)
{
    // Those that are commented out are replaced with functions
    // with better functionalities
    //imageProcessing();
    //imageCamera(); // Tested working
    imageTotal();
    data(); // To ask End location
    pathplanning();  // Execute all steps above

   //cout << endl << endl; // For the clean printing of OpenGL
```

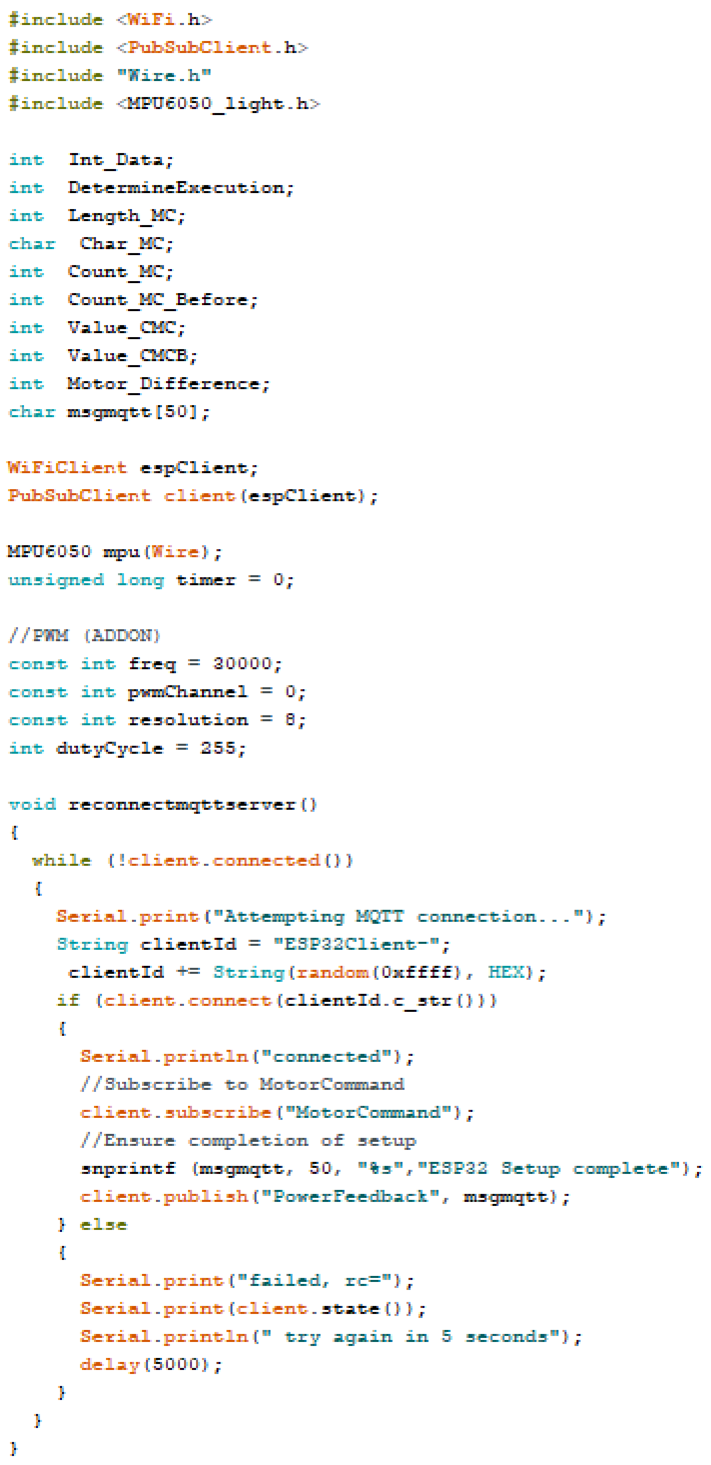
Figure C-14: Pathplanning.cpp Program (14)



```
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE); // Double buffer window
    glutInitWindowSize(500, 500);
    glutCreateWindow("Pathfinding");
    glutDisplayFunc(display_callback);
    glutReshapeFunc(reshape_callback);
    init();
    glutMainLoop();  // Event processing loop

    return 0;

}

void display_callback()
{
    glClear(GL_COLOR_BUFFER_BIT); // Enable buffer for colour writting
    drawGrid();
    lineInitiate();
    glutSwapBuffers(); // Swaps back buffer to front buffer
}

void reshape_callback(int w, int h)
{
    glViewport(0, 0, (GLsizei)w, (GLsizei)h); // Transfer device coordinate to window coordinate
    glMatrixMode(GL_PROJECTION); // Specifies current matrix
    glLoadIdentity(); // Replace current matrix with identity matrix
    // Columns and Rows which is define above, 20x20
    // Cols and Rows is top to bottom and left to right (Changed from 3D to 2D)
    gluOrtho2D(0.0, COLUMNS, ROWS, 0.0);
    // Change back to default Matrix mode as most things takes part here
    glMatrixMode(GL_MODELVIEW);
}
```

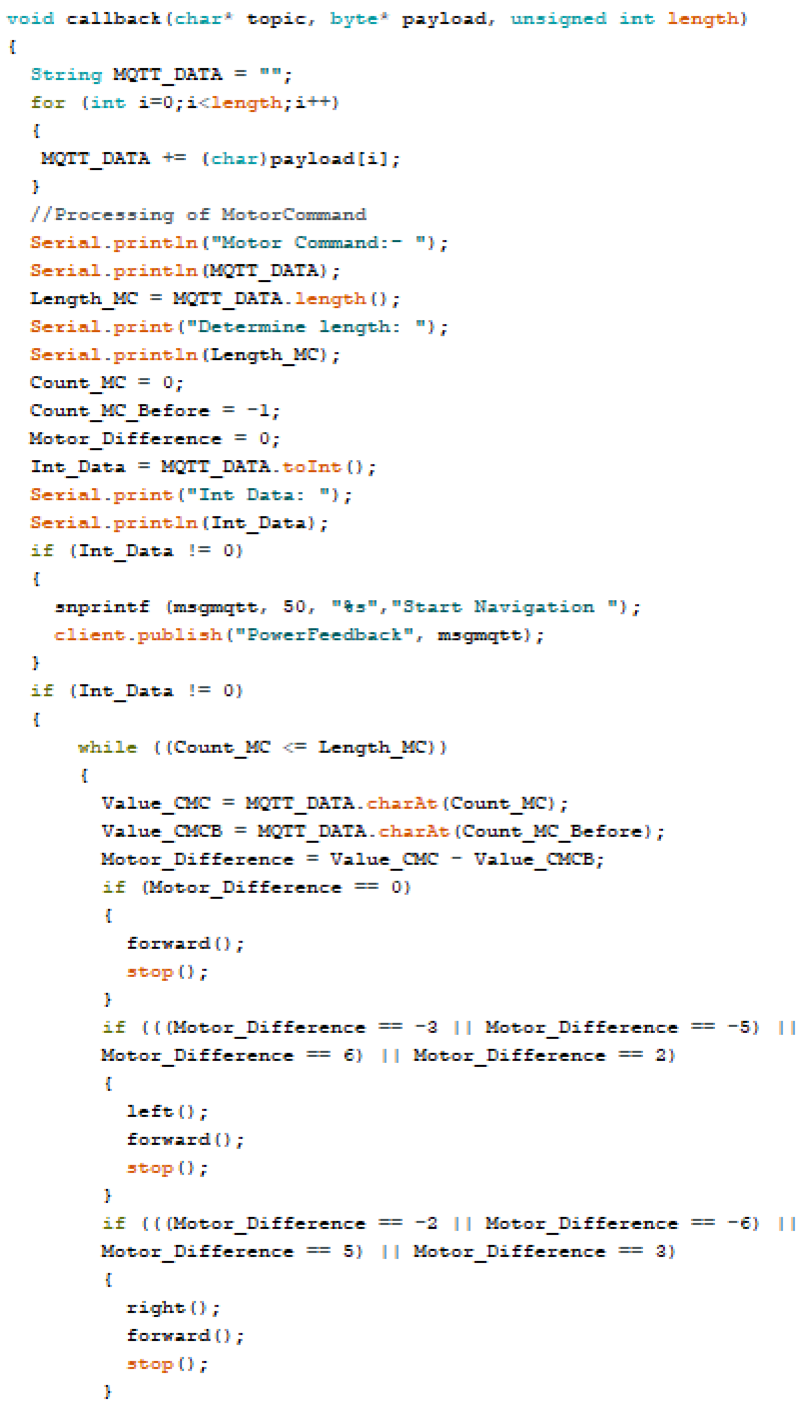
Figure C-15: Pathplanning.cpp Program (15)

Appendix D: Arduino code for ESP32



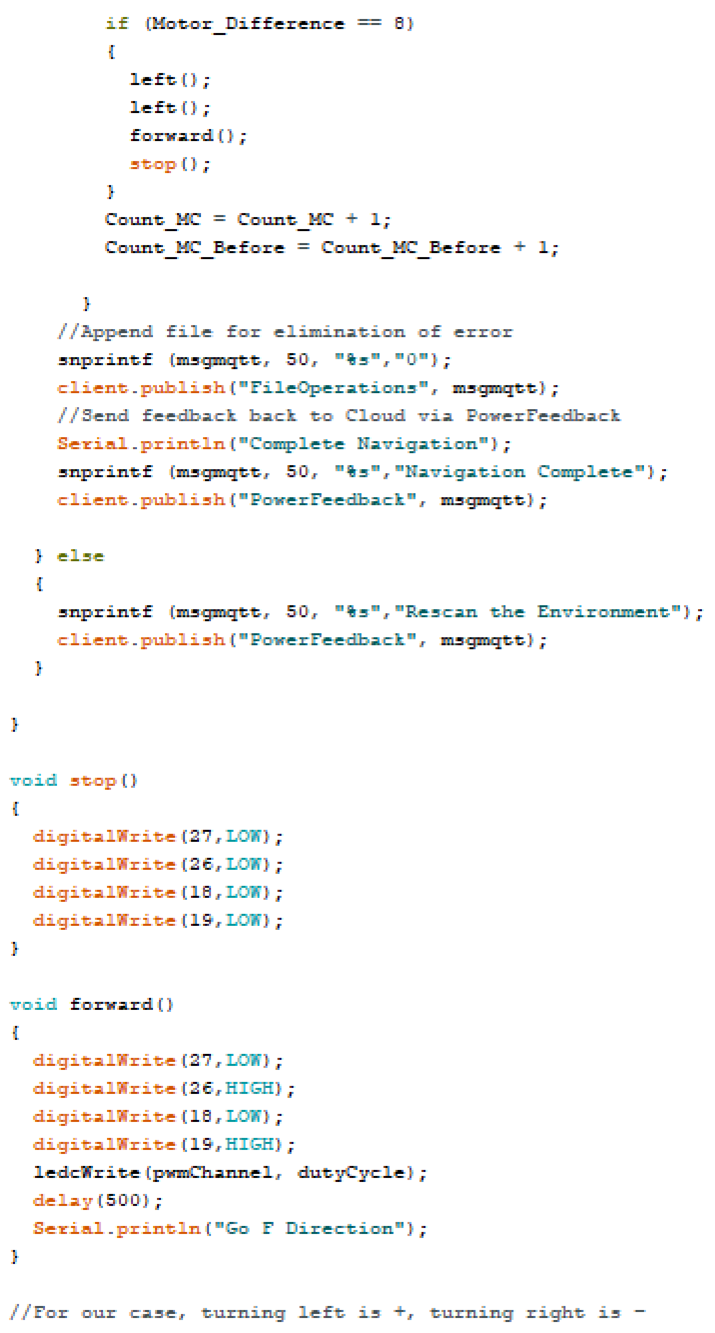
```
#include <WiFi.h>
#include <PubSubClient.h>
#include "Wire.h"
#include <MPU6050_light.h>

int  Int_Data;
int  DetermineExecution;
int  Length_MC;
char  Char_MC;
int  Count_MC;
int  Count_MC_Before;
int  Value_CMC;
int  Value_CMCB;
int  Motor_Difference;
char msgmqtt[50];

WiFiClient espClient;
PubSubClient client(espClient);

MPU6050 mpu(Wire);
unsigned long timer = 0;

//PWM (ADDON)
const int freq = 30000;
const int pwmChannel = 0;
const int resolution = 8;
int dutyCycle = 255;

void reconnectmqttserver()
{
  while (!client.connected())
  {
    Serial.print("Attempting MQTT connection...");
    String clientId = "ESP32Client-";
     clientId += String(random(0xffff), HEX);
    if (client.connect(clientId.c_str()))
    {
      Serial.println("connected");
      //Subscribe to MotorCommand
      client.subscribe("MotorCommand");
      //Ensure completion of setup
      snprintf (msgmqtt, 50, "%s","ESP32 Setup complete");
      client.publish("PowerFeedback", msgmqtt);
    } else
    {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      delay(5000);
    }
  }
}
```

Figure D-1: ESP32 Program (1)



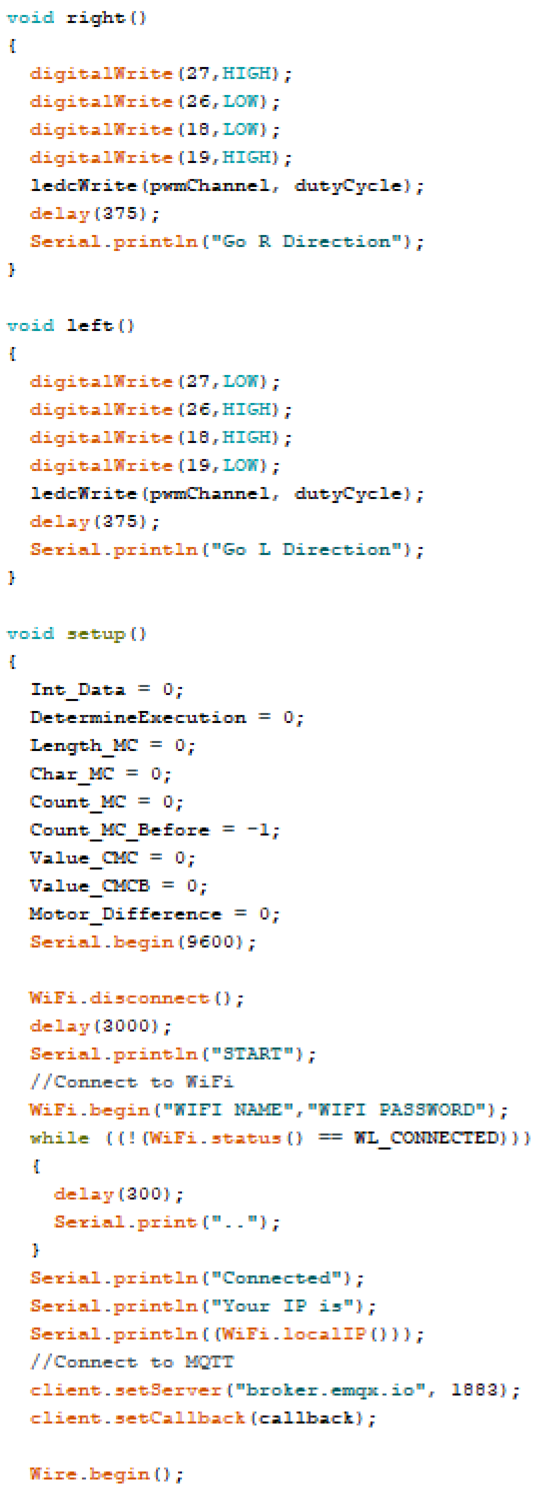
```
void callback(char* topic, byte* payload, unsigned int length)
{
  String MQTT_DATA = "";
  for (int i=0;i<length;i++)
  {
   MQTT_DATA += (char)payload[i];
  }
  //Processing of MotorCommand
  Serial.println("Motor Command:- ");
  Serial.println(MQTT_DATA);
  Length_MC = MQTT_DATA.length();
  Serial.print("Determine length: ");
  Serial.println(Length_MC);
  Count_MC = 0;
  Count_MC_Before = -1;
  Motor_Difference = 0;
  Int_Data = MQTT_DATA.toInt();
  Serial.print("Int Data: ");
  Serial.println(Int_Data);
  if (Int_Data != 0)
  {
    snprintf (msgmqtt, 50, "%s","Start Navigation ");
    client.publish("PowerFeedback", msgmqtt);
  }
  if (Int_Data != 0)
  {
      while ((Count_MC <= Length_MC))
      {
        Value_CMC = MQTT_DATA.charAt(Count_MC);
        Value_CMCB = MQTT_DATA.charAt(Count_MC_Before);
        Motor_Difference = Value_CMC - Value_CMCB;
        if (Motor_Difference == 0)
        {
          forward();
          stop();
        }
        if (((Motor_Difference == -3 || Motor_Difference == -5) ||
        Motor_Difference == 6) || Motor_Difference == 2)
        {
          left();
          forward();
          stop();
        }
        if (((Motor_Difference == -2 || Motor_Difference == -6) ||
        Motor_Difference == 5) || Motor_Difference == 3)
        {
          right();
          forward();
          stop();
        }
```

Figure D-2: ESP32 Program (2)



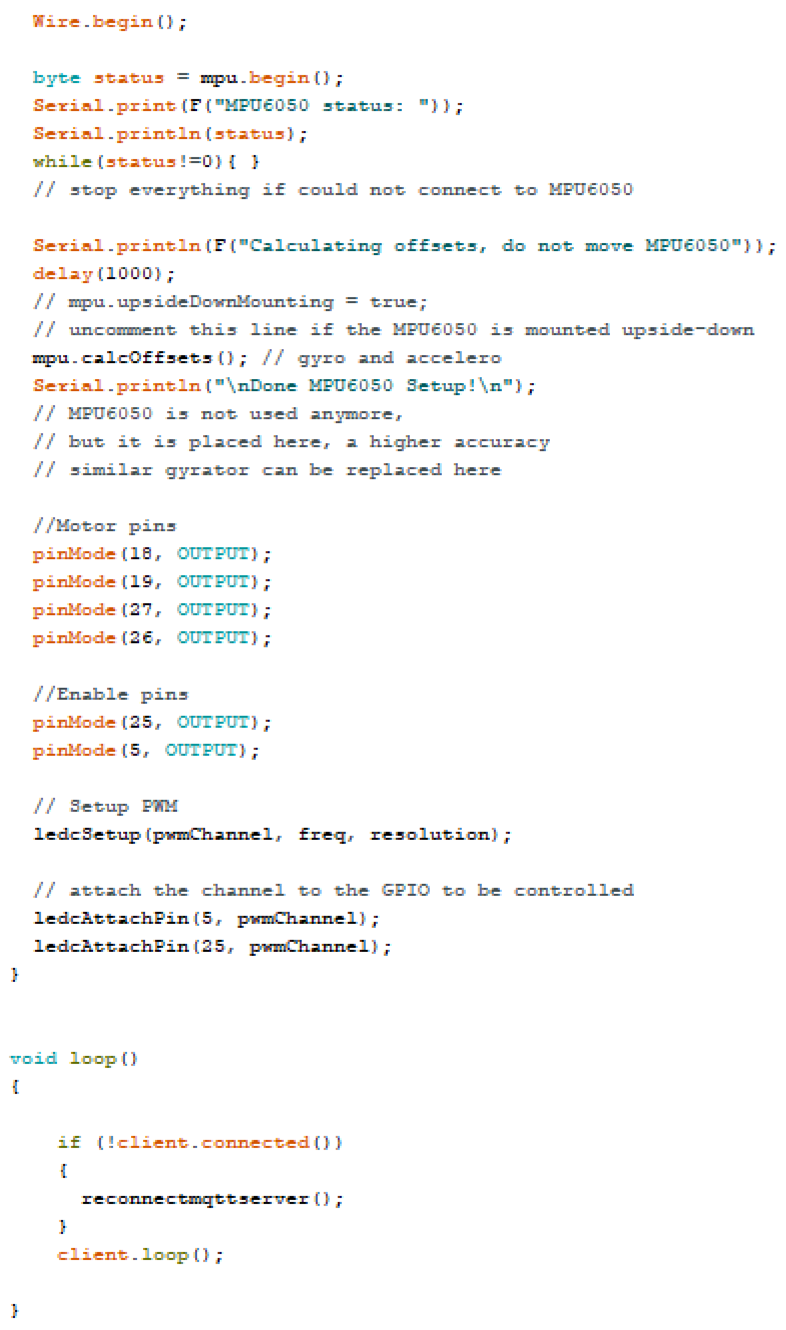
```
        if (Motor_Difference == 8)
        {
          left();
          left();
          forward();
          stop();
        }
        Count_MC = Count_MC + 1;
        Count_MC_Before = Count_MC_Before + 1;

      }
    //Append file for elimination of error
    snprintf (msgmqtt, 50, "%s","0");
    client.publish("FileOperations", msgmqtt);
    //Send feedback back to Cloud via PowerFeedback
    Serial.println("Complete Navigation");
    snprintf (msgmqtt, 50, "%s","Navigation Complete");
    client.publish("PowerFeedback", msgmqtt);

  } else
  {
    snprintf (msgmqtt, 50, "%s","Rescan the Environment");
    client.publish("PowerFeedback", msgmqtt);
  }

}

void stop()
{
  digitalWrite(27,LOW);
  digitalWrite(26,LOW);
  digitalWrite(18,LOW);
  digitalWrite(19,LOW);
}

void forward()
{
  digitalWrite(27,LOW);
  digitalWrite(26,HIGH);
  digitalWrite(18,LOW);
  digitalWrite(19,HIGH);
  ledcWrite(pwmChannel, dutyCycle);
  delay(500);
  Serial.println("Go F Direction");
}

//For our case, turning left is +, turning right is -
```

Figure D-3: ESP32 Program (3)



```
void right()
{
  digitalWrite(27,HIGH);
  digitalWrite(26,LOW);
  digitalWrite(18,LOW);
  digitalWrite(19,HIGH);
  ledcWrite(pwmChannel, dutyCycle);
  delay(375);
  Serial.println("Go R Direction");
}

void left()
{
  digitalWrite(27,LOW);
  digitalWrite(26,HIGH);
  digitalWrite(18,HIGH);
  digitalWrite(19,LOW);
  ledcWrite(pwmChannel, dutyCycle);
  delay(375);
  Serial.println("Go L Direction");
}

void setup()
{
  Int_Data = 0;
  DetermineExecution = 0;
  Length_MC = 0;
  Char_MC = 0;
  Count_MC = 0;
  Count_MC_Before = -1;
  Value_CMC = 0;
  Value_CMCB = 0;
  Motor_Difference = 0;
  Serial.begin(9600);

  WiFi.disconnect();
  delay(3000);
  Serial.println("START");
  //Connect to WiFi
  WiFi.begin("WIFI NAME","WIFI PASSWORD");
  while ((!(WiFi.status() == WL_CONNECTED)))
  {
    delay(300);
    Serial.print("..");
  }
  Serial.println("Connected");
  Serial.println("Your IP is");
  Serial.println((WiFi.localIP()));
  //Connect to MQTT
  client.setServer("broker.emqx.io", 1883);
  client.setCallback(callback);

  Wire.begin();
```

Figure D-4: ESP32 Program (4)



```
  Wire.begin();

  byte status = mpu.begin();
  Serial.print(F("MPU6050 status: "));
  Serial.println(status);
  while(status!=0){ }
  // stop everything if could not connect to MPU6050

  Serial.println(F("Calculating offsets, do not move MPU6050"));
  delay(1000);
  // mpu.upsideDownMounting = true;
  // uncomment this line if the MPU6050 is mounted upside-down
  mpu.calcOffsets(); // gyro and accelero
  Serial.println("\nDone MPU6050 Setup!\n");
  // MPU6050 is not used anymore,
  // but it is placed here, a higher accuracy
  // similar gyrator can be replaced here

  //Motor pins
  pinMode(18, OUTPUT);
  pinMode(19, OUTPUT);
  pinMode(27, OUTPUT);
  pinMode(26, OUTPUT);

  //Enable pins
  pinMode(25, OUTPUT);
  pinMode(5, OUTPUT);

  // Setup PWM
  ledcSetup(pwmChannel, freq, resolution);

  // attach the channel to the GPIO to be controlled
  ledcAttachPin(5, pwmChannel);
  ledcAttachPin(25, pwmChannel);
}


void loop()
{

    if (!client.connected())
    {
      reconnectmqttserver();
    }
    client.loop();

}
```

Figure D-5: ESP32 Program (5)