|  |  |
| --- | --- |
| School of Electronic Engineering and Computer Science | **Final Report**  **Programme of study:**  Electrical and Electronic Engineering  **Project Title:**  **Vision-Based Indoor Positioning System**  **Supervisor:**  Tijana Timotijevic  **Student Name:**  Syed Uzair Maghrabi  Date: 6/05/25 |
| Final Year  Undergraduate Project 2024/25 |
| QMLogo |

Abstract

This paper explores developing and implementing a vision-based indoor positioning system (IPS) using a camera for autonomous robot navigation. The system uses ArUco markers for camera calibration and object tracking, leveraging computer vision techniques via OpenCV. Key concepts covered include path finding using and obstacle avoidance facilitated by graph-based algorithms such as A\* and an interactive model of the system using the 2D visual framework Pygame.

This review emphasises the advantages of computer vision systems over traditional radio-based IPS systems, particularly in dynamic environments prone to signal interference and signal blocking. This study also examines the integration of Wi-Fi communication for real-time command relay and discusses real-time PID control systems that can react to changes in the environment. Challenges such as visualising and detecting obstructions in real-time will be undertaken by colour and contour filtering using OpenCV modules. Additionally, another challenge will be the ability to seamlessly integrate all the different frameworks that will be used in a complete and optimised API to reduce computational performance and increase efficiency.

This review also addresses the difficulty of scaling the system to larger environments and maintaining precision despite the potential for environment variability, such as moving obstacles, light variability, or camera recalibration needs.

**C**

**ontents**

[Chapter 1: Introduction 5](#_Toc197417205)

[1.1 Background 5](#_Toc197417206)

[1.2 Problem Statement 6](#_Toc197417207)

[1.3 Aim 6](#_Toc197417208)

[1.4 Objectives 7](#_Toc197417209)

[1.5 Research Questions 8](#_Toc197417210)

[Chapter 2: Literature Review 9](#_Toc197417211)

[2.1 Localisation Using Computer Vision 9](#_Toc197417212)

[2.2 Challenges in computer vision 11](#_Toc197417213)

[2.3 Mapping Using Pathfinding Algorithms 12](#_Toc197417214)

[Chapter 3: Requirements 15](#_Toc197417215)

[3.1 Functional and Non-Functional Requirements 15](#_Toc197417216)

[3.2 Hardware Requirements 17](#_Toc197417217)

[3.3 Software Requirements 17](#_Toc197417218)

[Chapter 4: Methodology 19](#_Toc197417219)

[4.1 State Machine 19](#_Toc197417220)

[4.2 Environment and Robot Hardware 20](#_Toc197417221)

[4.3 Computing Camera matrix 21](#_Toc197417222)

[4.4 Obstacle Detection using Colour Filtering 22](#_Toc197417223)

[4.5 Generating Pygame UI Layout 23](#_Toc197417224)

[4.6 A\* Algorithm 23](#_Toc197417225)

[4.7 B-Spline Interpolation of Points 23](#_Toc197417226)

[4.8 PID Control System 24](#_Toc197417227)

[4.9 Forward Kinematics 25](#_Toc197417228)

[4.10 MQTT Broker 25](#_Toc197417229)

[Chapter 5: Implementation 26](#_Toc197417230)

[5.1 Camera Calibration Code 26](#_Toc197417231)

[5.2 ArUco generation code 27](#_Toc197417232)

[5.3 Object Detection 27](#_Toc197417233)

[5.4 Pygame Initialisation 29](#_Toc197417234)

[5.5 A\* pathfinding 31](#_Toc197417235)

[5.6 B-spline Interpolation 32](#_Toc197417236)

[5.7 PID Control System 33](#_Toc197417237)

[Chapter 6: Testing 35](#_Toc197417238)

[6.1 PID tuning 35](#_Toc197417239)

[6.2 Number of waypoints 35](#_Toc197417240)

[6.3 Object Detection Testing 36](#_Toc197417241)

[Chapter 7: Evaluation 37](#_Toc197417242)

[7.1 PID Response 37](#_Toc197417243)

[7.2 FPS 38](#_Toc197417244)

[Chapter 8: Conclusion 40](#_Toc197417245)

[8.1 Project Aims 40](#_Toc197417246)

[8.2 Project Requirements 41](#_Toc197417247)

[8.3 Research Questions 43](#_Toc197417248)

[8.4 Future Improvements 43](#_Toc197417249)

[8.5 Challenges 44](#_Toc197417250)

[References 45](#_Toc197417251)

[Appendix A: ArUco Generator 47](#_Toc197417252)

[Appendix B: Camera Matrix Code 48](#_Toc197417253)

[Appendix C: Parallel Autonomous Robot Code 50](#_Toc197417254)

[Appendix C: Arduino Code 66](#_Toc197417255)

# Introduction

## Background

The vision for autonomous robots is to create systems that operate independently in dynamic environments, performing tasks efficiently and reliably. Autonomous robots are designed to reduce the dependency on human intervention, allowing them to perform complicated and repetitive tasks in various industries such as manufacturing, healthcare and agriculture.

Autonomous Ground Vehicles include robotic lawnmowers, self-driving cars, and warehouse robots. They use external inputs from sources such as radio-based sensors (such as the UWB) or Image Resonance Systems (such as LiDAR) to navigate their environments and make intellectual decisions based on the environment around them. In this paper, we will be using a vision-based system to map out the environment, allowing our robot the make intellectual decisions based on its objective and the changes in the environment around it.

The robot I will be building is designed specifically to be used in a manufacturing environment where products need to be transported from the starting point to the destination. The robot must be able to react and change its direction in response to a dynamic environment, so a constant stream of information will be required. This information will be delivered to the robot through an external source (i.e., a computer) via Wi-Fi.

The robot will feature an interactive UI, which will give users the ability to plan routes and visualise the environment. I will use the 2D visual game module Pygame for this purpose. Data from the camera will be processed into a grid arrangement, which will be mapped out to an interactive 2D map of the environment using Pygame. Additionally, we will apply the A\* pathfinding algorithm to the grid to generate the necessary path. This path will be shown on the interactive display to show the robot's trajectory. I will use a custom-tuned PID algorithm to ensure that the robot follows the path accurately and efficiently using waypoints.

## Problem Statement

Most indoor positioning systems (IPS) rely on radio-based technologies such as Ultra-Wideband (UWB) or Bluetooth Low Energy (BLE) to determine a robot’s position relative to its environment. While these systems provide reasonable accuracy, they come with significant drawbacks. One limitation is the reliance on multiple radio beacons for triangulation, which increases deployment costs and setup time, particularly in large or complex environments. Furthermore, these radio beacons are incapable of detecting physical obstacles within the environment, making them unsuitable for dynamic scenarios where real-time obstacle avoidance is necessary. Additionally, radio signals are susceptible to interference and obstructions, leading to inaccuracies in position tracking, which can compromise the robot's ability to navigate effectively and safely. This highlights the need for an alternative approach that combines accurate localisation with robust environmental awareness.

## Aim

This project aims to design and develop an affordable indoor positioning system (IPS) to control an autonomous robot capable of operating in dynamic environments such as manufacturing, agriculture, and healthcare. This system will prioritise cost-effectiveness while maintaining high accuracy and adaptability to complex, changing conditions. The project will involve creating a simplified small-scale prototype to serve as a proof of concept, allowing for rigorous testing and evaluation. The insights gained will inform potential scaling of the design for full-scale deployment in real-world applications, ultimately contributing to improved efficiency and automation in various industries.

## Objectives

### Build Environment

Construct the environment for an autonomous robot to move around in. Corners will be designed in CAD modelling software and 3d printed. Once printed, the corners will be joined together using metal rods to create the rectangular environment. The modular design of the environment is to ensure quick prototyping. A Camera will be mounted in a top-down view, looking down at the environment.

### Generate ARUCO Markers

Generate and print out ARUCO Markers and stick them on each corner of the rectangular environment; these will be used for perspective transformation. Another ARUCO marker will be stuck to the robot to ensure quick detection and tracking.

### Robot Construction

Build the robot using electronic components and a 3d printed chassis, we will need motors and motor drivers as well. These components will be connected to an ESP32. Information will be sent between the System and the ESP32 using the MQTT communication protocol.

### Localization

Use the camera to capture the real-time movement of the robot and the environment. Correct for any distortion caused by the lens of the camera by finding the camera and the distortion matrix.

### Mapping

Using the ARUCO Markers as corners on the grid, perform a perspective transformation to ensure the frame is perfectly level, then split the environment into a 2D grid structure so pathfinding algorithms can be used.

### Pathfinding

Use the A\* Algorithm to determine the most efficient path from the robot to the destination and send information to the robot via Wi-Fi. Enforce a PID controller to ensure the robot does not stray from the course.

## Research Questions

1. **Core Functionality and Design Evaluation**

How effectively can a vision-based system using a 2D mapping approach enable an autonomous vehicle to navigate and respond to a dynamic environment?

1. **Scalability**

What are the key challenges and modifications required to scale up a small-scale vision-based autonomous robot to a large-scale manufacturing environment?

1. **Algorithm Efficiency**

How will the implementation of the A\* pathfinding algorithm and a PID control system enable real-time processing?

1. **Environmental Adaptation**

To what extent can the proposed vision-based system adapt to changes in a dynamic manufacturing environment, such as moving obstacles or changing routes

# Literature Review

## Localisation Using Computer Vision

### Localisation in autonomous robots.

**(Alam, M 2023**) Localisationis a vital task in an autonomous robot. To navigate, robots will need to initialise an effective localisation strategy, map their surroundings, and carry out various tasks autonomously. Localisation involves determining the robot's position concerning the environment in real-time. This process is crucial for making precise and informed decisions.

Localisation in robotics involves integrating numerous data sources, such as sensors and other relevant information, to estimate a robot's position in a predefined environment. There are 3 main approaches to localisation

1. Odometry-based Localisation – This requires precise wheel encoders to measure the rotation and angle of the wheel to determine the direction and the distance the robot has travelled; however, the location of the robot will not be known concerning the environment unless a starting point is predefined. Additionally, this method is prone to drift over time due to wheel slipping and other inconsistencies; therefore, prediction-based filters such as the Kalman filter are employed to predict the correct trajectory of the robot.
2. Sensor-based Localisation: This approach integrates data from various sensors (e.g., LiDAR, cameras, IMU) to estimate the robot's position. Sensors such as LiDAR and cameras are also used to create maps of the environment, allowing simultaneous mapping and localisation of the robot. Therefore, more complex routines can be programmed for the robot to follow.
3. Beacon-based Localisation: This approach involves placing known beacons or markers in the environment. The robot uses these beacons the determine its position relative to the beacon. This is the main principle used in UWB IPS; time-of-flight sensors are used to triangulate the robot's position between the beacons. This makes this approach extremely accurate.

### Challenges in Localisation

In Ideal circumstances, the data recorded from the environment would be perfect. But in real-world settings, this is not the case [2]. Sensors, such as UWB, may be inaccurate in a dynamic environment, blocking out signals and preventing localisation. Additionally, none of these methods provides any information about the environment.

Other difficulties include deviating from the robot's intended path, such as wheel slippage and collisions in odometry-based localisation. Once the robot deviates from its path, an additional recovery algorithm must be implemented to correct the robot’s trajectory. Additionally, in Lidar-based localisation, the robot will need to take additional time to search and map out the environment, which can be inefficient.

Furthermore, maps frequently depict the environment as a collection of static objects. This is a problem as the environment will likely contain more barriers. These barriers can either be static (like a box on the floor) or dynamic (like another robot working in tandem). These issues need to be properly addressed by localisation techniques.

### Map representation

To perform independent tasks, the robot will require a map of the environment. Since a robot cannot process all the data in the real world due to its limited memory, a simplified representation of the environment is necessary to enhance computational efficiency. The challenge lies in minimising the number of features from the real world while retaining the most vital information. Obtaining a 2D model of the map is a common strategy, as adding another dimension would lead to computational overload. This method is optimal as the robot will operate exclusively in two dimensions.

### Computer Vision and OpenCV

To achieve the desired outcome and mitigate the challenges, we will utilise computer vision for localisation. Computer vision, a field of artificial intelligence, employs techniques such as machine learning and neural networks to enable computers to derive meaningful information from digital inputs like images and videos. For this task, we will use the widely recognised computer vision library, Opencv.

**(OpenCV 2018)** Opencv is an open-source computer vision and machine learning software library that provides robust capabilities such as tracking moving objects, identifying objects, and extracting models from a 3d landscape. These functions will be processed on a laptop, ensuring efficient computation and real-time operation.

Opencv will require a constant stream of information, so we will use a wide-angle camera mounted in a top-down position, directly overlooking the environment. The benefits of this approach include:

1. Wide Field of View: A top-down perspective offers a comprehensive view of the environment, which will allow us to capture more area in a single frame. This is ideal for monitoring large spaces.
2. Simplified navigation: From a top-down perspective, obstacles are more visible to identify and track.
3. Better Safety monitoring: In dynamic environments, this perspective allows for quick detection of unexpected hazards or changes.
4. Reduced Processing Complexity: With a top-down view, the geometric relationships are simpler to process, as distances and angles can be measured directly on the 2D map

## Challenges in computer vision

### Camera distortion

As we will be using a wide-angle camera to record the environment, the received image will have some level of lens distortion that we need to process before converting the 3d image into a 2D grid format. There are two major types of lens distortion effects, which can be seen in Figure 1:

1. Radial distortion: This type of distortion usually occurs due to unequal bending of light. The rays bend more near the edges of the lens than the rays near the centre of the lens. This can give the effect of straight lines in the image being curved. The light ray gets displaced radially inward or outward from its ideal location before hitting the image sensor. There are two types of radial distortion effect **(Sadekar 2020)**
   1. Barrel distortion – This corresponds to negative radial displacement.
   2. A blue and black grid

      Description automatically generatedPincushion distortion – this corresponds to positive radial displacement.

Figure : Different types of distortion

1. Tangential distortion: This usually occurs when the frame screen is at an angle concerning the lens. Thus, the image will seem tilted and stretched.

Distortion can introduce errors in the shape, size, and position of objects, which can affect tasks such as object positioning and tracking. To ensure seamless control, we aim to make the image frame accurately replicate the real world. Objects within grids may be disproportionally affected by distorted leading to misclassification in calculating error values or failure to detect them.

### Varying Lighting Conditions

The light source is crucial for computer vision systems, and varying light conditions can create significant challenges, particularly in real-world situations where natural or artificial light fluctuates throughout the day.

If the light intensity is too strong, we may encounter overexposure, causing parts of the image to look washed out or faded, making it harder to extract information. This is especially an issue for ArUco detection, where the whites may appear to be overexposed, increasing the risk of a missed detection.

On the other hand, if the light intensity is too weak, underexposure can occur, making objects harder to detect due to low contrast with the floor. This is particularly problematic for object detection methods like contour analysis and colour detection, where a clear distinction between the object and background is crucial.

If the light varies at different angles, shadows may appear in the frame. Since we are using a top-down perspective, these shadows can make objects seem like they are in areas where they aren’t.

## Mapping Using Pathfinding Algorithms

Mapping in autonomous robots is the process of creating a representation of the environment that the robot can perceive. The goal of mapping is to generate a map that accurately captures the spatial layout, obstacles, and other relevant features of the surroundings.

Mapping is a fundamental capability for robots to navigate and operate autonomously in their environment, using mapping we can plan paths, avoid obstacles and make intelligent decisions based on its surroundings.

To map features like objects and the robot itself, we will need to employ object detection methods.

### Object detections

There are a variety of object tracking methods we could utilise in our Opencv program.

#### Contour analysis

When we connect all the points along the boundary of an object, we obtain a contour. A contour represents the intensity of boundary pixels that share the same colour and brightness. Using contour detection, we can identify the outlines of objects within our frame and easily localise them in an image. (6).

Using contours, we can identify obstacles within the frame. However, if shadows are present, the contour detection function may also detect them as obstacles, leading to inconsistencies in the system.

#### Colour detection

Colour detection works by creating a mask using the HSV values of the colour you are trying to identify. This mask is overlaid onto the original image, and any matching colours will appear in the masked frame.

Since we are detecting colour, this method can be extended beyond just identifying obstacles to recognising areas of interest. For example, an autonomous cleaning robot designed to detect a specific material by its colour can easily identify target areas. Similarly, a golf ball collection robot could identify and locate white golf balls on the field.

While shadows do not affect the performance of colour detection, variations in light intensity can cause overexposure or underexposure, which may shift the colour parameters set in the mask.

In this project, we will use colour detection, due to its more accurate performance.

### Pathfinding using A\*

Robotic pathfinding is a process where robots use algorithms to navigate their environment efficiently, there are multiple approaches we can use when choosing the best pathfinding algorithm **(Patel, A 1997)** .

#### Dijkstra’s Algorithm

Dijkstra’s algorithm calculates the shortest path by searching nodes expanding from the starting node, always choosing the node with the smallest known distance. It symmetrically explores all parts to the destination and guarantees the shortest route, but it can be computationally expensive because it doesn’t prioritise direction towards the destination.

#### Best-First Search

Best-First Search uses a heuristic approach to estimate the distance to the destination and selects the node that appears closest to the destination. While it is typically faster than Dijkstra’s Algorithm, it may follow misleading paths and does not guarantee the shortest route, especially in environments containing obstacles.

#### A\* Algorithm

Figure : A\* Algorithm searching for a path around an obstacle

We will be using the A\* algorithm as it takes the best quality from both Dijkstra and Best-First. It balances Dijkstra's favouring of vertices that are close to the starting point and Best-First favouring vertices that are close to the goal. G(n) represents the exact costs of the path from the starting point to any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the destination.A graph with a rectangle and a line

AI-generated content may be incorrect.

In Figure 2, the yellow vertices (h) represent the vertices that are far away from the destination, while the blue vertices (g) represent the vertices that are far away from the starting point. The A\* algorithm balances these two factors as it creates a path from the starting point to the destination. Once it reaches the destination, it evaluates all potential paths and selects the one with the lowest f(n)

As we are employing obstacle avoidance methods, the A\* algorithm will be used

# Requirements

## Functional and Non-Functional Requirements

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Category** | **ID** | **Requirement** | **Type** | **Priority** |
| **General** | G1 | Must be able to perform localising, pathfinding and obstacle avoidance seamlessly | Functional | High |
|  | G2 | One a new obstacle obstructs the robot’s current path, a new path will be generated. | Non-Functional | Medium |
|  | G3 | Use a 3d printed mount to orient the camera into a top-down position | Functional | High |
|  | G5 | 3d printed arena needs to be modular to enable rapid testing in a variety of environments | Non-Functional | Medium |
| **User** | U1 | The user should be able to identify the robot and the finishing point on the graph. | Functional | High |
|  | U2 | The user should be able to select the finishing node and any obstacles | Functional | High |
|  | U3 | Users will be notified if an obstacle has crossed the path of the robot | Non-Functional | Medium |
|  | U4 | The user interface should be streamlined and easy to use, and understand | Non-Functional | Low |
|  | U5 | The user interface should run in parallel to the object detection and image processing algorithm. | Non-Functional | High |
|  | U6 | The user will be informed whether the connection with esp32 is established. | Non-Functional | Medium |
| **Robotic car** | R1 | Should be designed to be as compact as possible to ensure the area is a large enough environment | Non-Functional | High |
|  | R2 | Uses ESP32 to receive information from the software and control the wheels | Functional | High |
|  | R3 | Motor Drivers are needed to ensure the robot's bi-directional compatibility. | Non-Functional | High |
|  | R4 | The robot should have a long battery life for prolonged testing | Non-Functional | Low |
|  | R5 | A robot must have low latency when responding to commands from the system | Non-Functional | High |
|  | R6 | The battery source must be rechargeable to minimise environmental impact | Non-Functional | Low |
| **Software** | S1 | Must be written in Python to ensure rapid prototyping | Non-Functional | Medium |
|  | S2 | The system must be implemented using OpenCV, Pygame and Arduino IDE software. | Functional | High |
|  | S3 | The system should utilise multiprocessing for parallel development | Non-Functional | High |
|  | S | The system should run at least 30fps to ensure real-time operation | Non-Functional | Medium |
|  | S4 | All code must be annotated and easy to understand and follow | Non-Functional | Medium |
|  | S5 | The system should detect and track green objects in the arena and update the grid. | Functional | High |

## Hardware Requirements

### ESP32

The ESP32 is a low-cost, low-power microcontroller developed by Espressif. It features a variety of input and output peripherals, such as capacitive touch, ADCS, DACS, UART, SPI, I2C, and PWM. These pins will be used to control the motor driver. Additionally, the ESP32 has built-in Wi-Fi and Bluetooth capabilities, enabling wireless communication for remote command execution within our system.

### DRV8833 Motor Driver

**(Staff 2023)** The DRV8833 is an integrated H-Bridge driver IC that is optimised for motor driving applications. It's able to simultaneously control two DC brushed motors. Therefore, we will need 2 Motor Drivers for 4-wheel control. The motor driver has a very small and compact design, which will allow us to further compact the design of the robot

### N20 Micro Gear Motor with Rubber Wheels

The N20 is a small, geared, brushed motor that allows the robot to move through its environment. The rubber wheels enhance grip on the floor, reducing slippage and improving stability, leading to better control of the system during path-following.

## Software Requirements

### Windows Operating System

A Windows computer with multiple cores is necessary to utilise multiprocessing for parallelising processes. Additionally, it allows us to use the Python programming language as well as the Arduino IDE.

### Python

We will use Python as the primary programming language for object detection and the A\* algorithm model. Python allows us to easily utilise Opencv modules, enabling quick adjustments to the algorithm.

### NumPy

We will use the NumPy library to handle numerical computations efficiently in our project. NumPy provides powerful array operations and mathematical functions, enabling faster data processing and optimizing of the A\* algorithm. Its seamless integration with other Python libraries makes it essential for high-performance computing

### OpenCV

The Opencv module will allow us to utilise image processing and object detection for our project. Opencv provides a wide range of tools for real-time computer vision tasks, allowing us to efficiently analyse and manipulate frames.

### Pygame

We will use the Pygame library to create a visual simulation of our system. Pygame provides an easy-to-use framework for rendering graphics, handling user input, and managing animations. This will allow us to visualise the movement of the robot and test the A\* algorithm in a simulated environment before real-world implementation.

### Fusion 360

We will use the CAD-based software to design our structured environment as well as the robot

# Methodology

## State Machine

### System State Machine

A screenshot of a computer

AI-generated content may be incorrect.

Figure : System State Machine diagram

### Sequential vs Parallel execution

**(GeeksforGeeks 2022)** We can see in Figure 3 that the System State Machine is made from 3 connected sequences, the middle sequence handles the camera calibration and object detection sequences, A secondary process handles the Pygame UI and the A\* pathfinding and another process handles the PID controller and Wi-Fi communication.

**Sequential Execution**

Sequential execution refers to the process of executing one instruction at a time, in a strict step-by-step order. Each task must be completed before the next one begins. Since this approach relies on a single processor, it can become inefficient or slow, especially in complex or large-scale systems where multiple operations need to be handled simultaneously.

#### Parallel Execution

Parallel execution involves processing multiple instructions simultaneously, allowing tasks to be carried out concurrently. This approach significantly reduces execution time, especially in complex systems. Unlike sequential execution, where tasks are handled one after another, parallel execution enables the system to perform many operations at once, leading to greater efficiency and responsiveness. Since this system will react and function in real time, minimising execution time is crucial for a smooth system.

In our system, we will use two separate processes to work simultaneously with our main sequential code. One of the process functions is to display an interactive display of the environment, showing robot, object and goal positions. Additionally, the A\* pathfinding algorithm will also be processed. We will use two queues to send information to and from the main loop to receive and send information. The second process will run the PID controller and send PWM values through the MQTT broker to the ESP32, another queue is used to send information to this process. Unlike sequential cores, sequential execution can use additional cores to facilitate multiple processes, resulting in a rapid response.

## Environment and Robot Hardware

### Environment

The Hardware Setup of this project consists of a structured environment, using 3d printed corners, designed in Fusion 360, and metal poles to create a rectangular frame, which is to simulate a scaled-down working environment as mentioned in the research questions. Additionally, the modular design allows for rapid disassembly and reassembly for effective research. A long PVC pipe is inserted into one of the middle sections of the environment, which connects it to the 3d printed camera mount using a press fit design.

An ultrawide camera is mounted on the camera mount, providing a top-down perspective of the environment. This setup offers a comprehensive view, capturing the entire area, including obstacles, and helps prevent occlusions between the robot and any obstacles. The 2D area captured by the camera will be converted into a grid format using signal processing, enabling easy analysis and control of the robot's operations. In real-world scenarios, the camera could be mounted on the roof of a building or from a drone flying above the area, providing autonomous control regardless of the environment.

### Robot

The ESP32 microcontroller serves as the heart of the robot, utilising its built-in Wi-Fi and Bluetooth modules for wireless communication between the system and the robot. PWM outputs on the ESP32's GPIO pins will be used to control the motor speed, enabling precise control over the robot's direction and speed variations.

We will use the DRV8833 motor driver to enable bi-directional control of the motors through an H-bridge configuration. Each DRV8833 can control two brushed DC motors, so two motor drivers will be required. Additionally, the compact size of the motor driver allows for a smaller chassis, making it easier to manoeuvre the robot around the environment. We used four small, geared motors to enable movement around the arena. Since the system relies heavily on object detection and signal processing, the robot's speed is not a critical factor.

## Computing Camera matrix

### Distortion

For the prototyping process, we will use an ultrawide camera to capture a large area without requiring a high mounting position. However, using an ultrawide camera introduces some complications that must be addressed before object detection can be performed. To maximise the captured area, the camera stream will be distorted; this distortion must be corrected before processing to ensure accurate positioning.

A black and white checkered board

AI-generated content may be incorrect.

Figure : Checkboard pattern

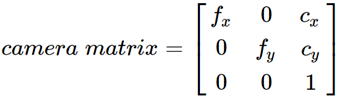
To correct these, we use a checkerboard such as Figure 4. We use a checkboard because it provides a well-defined, high-contrast, rectangular grid of known geometry. Camera calibration is performed using Opencv **(OpenCV 2025**), which provides the intrinsic parameters such as focal length and optical centres, and extrinsic parameters for 3d-to-2d coordinate transformations. A camera matrix is calculated to rectify the image before further processing, as shown in Figure 5:

Figure : Matrix

Extrinsic parameters relate to rotation and translation vectors, which translate the coordinates of a 3d point to a 2D coordinate system.

### ArUco Markers

We will stick one ArUco marker, such as Figure 6, on each corner of the environment, and the system will detect the ArUco code and measure the coordinates of its vertices. We can use these coordinates to apply the perspective transformation to the image. Additionally, we will stick an ArUco code onto the top of the robot, not only to track its location within the environment but also to measure the angle between the heading of the robot to the path trajectory. This will be useful later when developing the pathfinding control system.

### Perspective transform

A perspective transform is applied to achieve a true drop-down view. Using ArUco markers placed at the corners of the environment, like the one in Figure 6. The system identifies the markers and adjusts the image accordingly, enabling accurate 2D grid conversion for localisation and navigation **(docs.opencv.org. 2025)**.

A black and white logo

AI-generated content may be incorrect.

Figure : ArUco Code

## Obstacle Detection using Colour Filtering

### Object detection methods

To detect objects in the arena, we will use colour detection to filter out green objects and identify them as obstacles. Compared to other methods, such as contour detection, colour-based object detection offers more reliable results. Contour detection involves tracking the outline of an image, which can lead to the environment and robot being incorrectly identified as objects, whereas colour filtering is more efficient and produces more accurate results. However, one limitation is that this method will only detect green objects.

### Colour detection process

Frames from the camera are first converted into the HSV colour space. HSV allows for easier colour filtering and detection under varying lighting conditions compared to RGB, which can be significantly influenced by changes in lighting. This is particularly important because, during testing, my model will encounter different lighting environments, yet the system needs to perform optimally in all conditions.

## Generating Pygame UI Layout

We use the Pygame UI to allow manual input into the system, such as selecting the finishing node. It also visually displays current information, including the environment size, object areas, and robot position. This UI will be initialised in a separate process to ensure the display remains active throughout the system's runtime.

Obstacles will be shown as red nodes, the robot’s location as a green node, and the selected finishing node as a blue node. There will be two buttons: one to clear the selected finishing node and another to start the pathfinding process. The visual interface is updated in real time, allowing users to observe the robot’s navigation path as it is computed or adjusted dynamically

## A\* Algorithm

Once a valid finishing node is selected, we will deploy an A\* algorithm to detect the quickest path from the starting node to the finishing node. When deciding which next node to travel to, we need to calculate the g-score, h-score and f-score. The g-score is the actual distance you have travelled from the starting node. For example, for any of the neighbours of the starting node, the g-score will be 1, and their neighbours will have a g-score of 2, and so on. The h-score, or heuristic score, is the estimated cost from the current node to the finishing node. We will use the Manhattan distance to estimate the distance:

Manhattan Distance=∣x1​−x2​∣+∣y1​−y2​∣

Each node's f-score (g + h) determines its priority in the search. Nodes with the lowest f-score are explored first, resulting in a balance between path accuracy and speed. If an object blocks the path, the algorithm recalculates an alternative route automatically, allowing dynamic object avoidance.

Once a path is discovered, it is returned as a list of waypoints. These are further refined using B-spline interpolation.

## B-Spline Interpolation of Points

**(CFD 2025)** Currently, our list of waypoints is blocky and angular. If we were to plot these points as our waypoints, we would travel from one centre of a square to the next, which would result in a choppy operation. To correct for this, we will interpolate our points across a B-spline graph.

A cubic B-spline provides a balance between smoothness and control. The original waypoints computed by the A\* algorithm are used as control points, and the spline curve is constructed between these points using a knot vector. Once a B-spline curve is generated, it is sampled to obtain a new, smoothed list of waypoints. This will allow the robot to follow a natural curve instead of a rigid and angular path like in Figure 7.

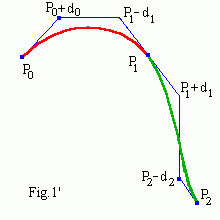


Figure : B-spline curve

## PID Control System

For the autonomous control of the robot, I will design a PID control system. From the state diagram above, we can see that the control system is in a completely separated process. By decoupling the controller from the rest of the system, we are able to react to any changes to the environment and update our PID controller in real time leading to a fast and responsive operation **(Ni.com. 2025)**.

Our B-spline waypoints are sent into a queue to the process, these are the inputs of our system, additionally, the position of the robot will also be sent. Using these two values, we can determine the error of the system and design a PID controller to correct for these values. This information is then sent back to the robot. Usually, in PID controllers, information is sent back to the start of the controller to form a closed loop; however, in our case, this is not needed, as any changes to the position of the robot will be recorded by our main sequential loop and sent back to the start of the controller.

Once we have calculated our errors, we can input these values into our PID component. There are 3 components we need to consider, the proportional, integral and derivative response.

The proportional components depend only on the distance error, the proportional gain determines the ratio of the output response to the error signal. A greater proportion component will result in a greater correction. However, if the gain is increased further, it increases the chance for oscillations in the system.

The integral component sums the error term over time; the integral response will slowly increase over time until the error is zero, so the effect is to drive the Steady-State error to zero. Steady-state error is the final difference between the process variable and the set point.

The derivative component will respond accordingly to the rate of change of the process variable, so if the process variable is increasing very rapidly, the derivative component causes the output to decrease. Most practical control system uses a small derivative component as the derivative response is highly sensitive to noise.

Once we input our distance and heading error, the PID controller will output the linear and angular velocity, respectively.

## Forward Kinematics

Once we have worked out the linear and angular velocity, we need to calculate the forward kinematics so the system, so we know how to drive the motors to reach a desired position.

Forward kinematics is the process of calculating the position and motion of a robot based on the commands sent to the actuators. In our context, Forward Kinematics will tell us how fast we need to spin each wheel to achieve a particular angular and linear velocity. The Forward Kinematics for a differential drive robot are:

The Is the wheelbase of the robot, since our linear and angular velocity is measured in We need to compute an equation to convert centimetre length to pixel length. Since we can obtain the pixel coordinates of the AruCo marker, we can relate the length of a side in pixel coordinates to the length of the same side in centimetres. Using this, we can compute a pixels\_per\_cm variable.

Now that we have the desired left and right wheel velocities, we can convert these values into PWM values compatible with our robot to drive to the destination.

## MQTT Broker

We will send information from the system to the ESP32 over Wi-Fi. There are many different messaging protocols we could use, but for this instance, we will use the MQTT messaging protocol **(EMQX Team 2025**) .

MQTT has defined itself as one of the best IoT protocols due to its specific features that cater towards IoT applications

IoT devices are often limited in terms of processing power. MQTT’s minimal overhead and small packet size make it perfect for these devices, enabling efficient communication even with limited capabilities.

MQTT follows a publish and subscribe pattern where the publisher and subscriber are decoupled, as the MQTT broker is responsible for establishing connections and routing and distributing messages. Both clients can publish topics to subscribe to and send messages, enabling effective data exchange and Bi-directional communication.

We will install the Mosquitto Broker on our local machine to enable communication between our devices

# Implementation

## Camera Calibration Code

To correct for lens distortion from the wide-angle camera, we calibrated the system using Opencv. A checkerboard pattern was used to extract 3d real-world coordinates to 2D image points. The *cv2.calibrate.Camera()* function is used to obtain the camera matrix and distortion coefficients. once found, we can apply to undistort each frame before further processing **(OpenCV 2025)**.

camera\_matrix = np.array([[650.07504615, 0, 624.17607378],  
 [0, 650.3368611, 366.05976318],  
 [0, 0, 1]])  
  
 dist\_coeffs = np.array([[-0.39029839, 0.188932, 0.00151121, -0.00146645, -0.04711856]])

Now, in the main code now that we have the camera matrix and distortion coefficients, so we can undistort our distorted input image (Figure 8).

undistorted\_frame = cv2.undistort(distorteted\_frame, camera\_matrix, dist\_coeffs)

A blue cubes on a brown and white rug

AI-generated content may be incorrect.A close-up of a carpet

AI-generated content may be incorrect.Which will produce this output (Figure 9)

Figure :Undistorted Image

Figure : Distorted Image

## ArUco generation code

For ArUco marker generation, we used the *DICT\_6X6\_250* dictionary and created markers with *cv2.aruco.In generateImageMarker(),* we can see an example with Figure 10. These markers were placed in the corners of the environment for perspective warping and on the robot for position tracking **(GeeksforGeeks 2024)**

aruco\_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT\_6X6\_250)

marker\_image = cv2.aruco.generateImageMarker(aruco\_dict, marker\_id=15, side\_pixels=200)

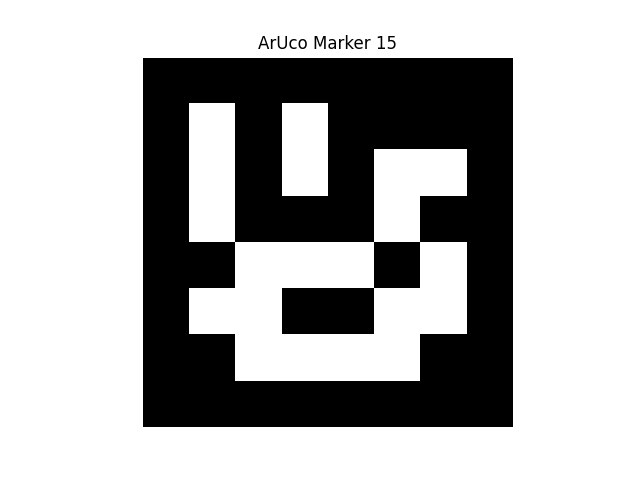
In this case, we will generate the ArUco marker: 

Figure : ArUco marker

## Object Detection

To detect any objects, we will use colour detections to identify green objects in the time frame, however, we first need to process the frame.

cropped\_frame\_HSV\_unblured = cv2.cvtColor(cropped\_frame, cv2.COLOR\_BGR2HSV)  
cropped\_frame\_HSV = cv2.GaussianBlur(cropped\_frame\_HSV\_unblured, (5, 5), 0)

The first line converts our frame into HSV format, which is necessary for colour detection, and the second line applies a Gaussian blur onto our frame. A Gaussian blur is a smoothing technique that reduces noise and detail by applying a Gaussian function to blur the image. When detecting objects, the camera may introduce small noise variations under adverse lighting conditions, which can cause false positives. Blurring helps reduce random pixel-level noise, making further detection more accurate.

green\_lower\_1 = np.array([35, 20, 40]) # Adjusted to avoid over-detection & flickering  
 green\_upper\_1 = np.array([85, 255, 255])  
  
  
# Create masks to filter out the green color  
 green\_mask = cv2.inRange(cropped\_frame\_HSV, green\_lower\_1, green\_upper\_1)  
  
  
 kernel = np.ones((5, 5), np.uint8)  
 green\_mask = cv2.morphologyEx(green\_mask, cv2.MORPH\_OPEN, kernel)  
 green\_mask = cv2.morphologyEx(green\_mask, cv2.MORPH\_CLOSE, kernel)

We then set two arrays to describe the upper and lower limits for our colour detection; any colour within these limits will be identified. A mask will be created that individually checks each pixel in the frame and compares its HSV value to the range we described. If it is within range, the pixel is set to white; and if not set to black. The function returns a binary image with any obstacles highlighted.

The next set of code cleans up the binary mask and removes unwanted noise, the MORPH\_OPEN function removes any white noise outside the main object and the MORPH\_CLOSE fills in any black holes inside any white object

num\_labels, labels, stats, \_ = cv2.connectedComponentsWithStats(green\_mask)  
  
min\_area = 200  
  
for i in range(1, num\_labels):  
 if stats[i, cv2.CC\_STAT\_AREA] < min\_area:  
 green\_mask[labels == i] = 0

To fully remove any false positives, we will apply contouring filtering to our mask. Since our mask only includes our real obstacles and noise, by measuring the contours of the objects, we can differentiate between real obstacles and noise.

We will set the minimum area of the object to 200, and iterate over the areas of all the objects; any object that has an area less than 200 is ignored. As a result, we obtain a clean frame free from any remaining noise.

green\_pixels = cv2.findNonZero(green\_mask)  
obstacle\_array = np.zeros((coloumns, rows))  
if green\_pixels is not None:  
 for point in green\_pixels:  
 x, y = point[0] # point[0] contains [x, y] from each non-zero pixel  
  
 grid\_x = x // 80  
 grid\_y = y // 80  
  
  
  
 if 0 <= grid\_y < coloumns and 0 <= grid\_x < rows:  
 obstacle\_array[grid\_y, grid\_x] = 1

Now we identify all non-zero pixels and map them onto a grid array to mark obstacles, Firstly, we create a zero array with the appropriate columns and row dimensions. We then loop through all the green pixels and calculate the respective grid node using our grid dimensions, which in this case are 80 pixels. After a respective grid format is calculated, we insert a 1 into the array. Finally, we will have an array representing the areas of objects in the environment.

## Pygame Initialisation

We will run our Pygame UI in a separate process. The way we send parameters into processes is different from working with functions. Instead of passing variables into the function definition, we will pass two queues, one for the input and another for outputting back into the main code.

def pathfinding\_algorithm(queue\_in, queue\_out):  
 pygame.init()

Before using any Pygame functionality, we must call pygame.int() to initialise Pygame’s internal modules.

screen = pygame.display.set\_mode((SCREEN\_WIDTH, SCREEN\_HEIGHT))  
start\_img = pygame.image.load('button.png').convert\_alpha()  
clear\_img = pygame.image.load('button (1).png').convert\_alpha()

We will be using two buttons in our Pygame UI, one for clearing the path and another for selecting the finishing node. Therefore, we need to load the PNG of both buttons into Pygame.

class Button():  
 def \_\_init\_\_(self, x, y, image, scale):

We create a button class to group all properties and behaviours of a button into one reusable structure, this keeps our main game loop easier to manage as the button logic will be found within the class. Since we are creating two buttons using a class makes it easy to duplicate the logic of the previous button.

def draw(self):  
 action = False  
 pos\_1 = pygame.mouse.get\_pos()  
  
 if self.rect.collidepoint(pos\_1):  
 if pygame.mouse.get\_pressed()[0] == 1 and not self.clicked:  
 self.clicked = True  
 action = True  
  
 if pygame.mouse.get\_pressed()[0] == 0:  
 self.clicked = False

The code snippet above describes the logic of a button press; the position of the mouse is recorded into pos\_1. If pos\_1 is over the image of the respective button, it will check whether the button has been pressed. If the mouse has been pressed, a flag will be set, which can be used as a trigger to influence another function.

class Grid():  
 def \_\_init\_\_(self, obstacle\_array, ROWS, COLOUMNS):

Similarly, we will also create a Grid class to initialise our grid.

def draw\_grid(self):  
 (x\_pos, y\_pos) = pygame.mouse.get\_pos()  
 if (x\_pos <= self.COLOUMNS \* CELL\_SIZE and y\_pos <= self.ROWS \* CELL\_SIZE) and not self.destination:  
 if pygame.mouse.get\_pressed()[0] == 1:  
 x\_grid = int(x\_pos // CELL\_SIZE)  
 y\_grid = int(y\_pos // CELL\_SIZE)  
 if 0 <= y\_grid < self.ROWS and 0 <= x\_grid < self.COLOUMNS:  
 if obstacle\_array[y\_grid, x\_grid] in [1, 2]:  
 print("error")  
 else:  
 self.y\_final = y\_grid  
 self.x\_final = x\_grid  
 self.destination = True

When selecting a finishing node, the logic we will use will be like the logic we used for the button. We will record our mouse and ensure it's within the boundaries of our graph. If the mouse is pressed, we record the position and compute which node in the grid we selected and set the flag self.desitnation to True. Additionally, if the node we selected contains our robot or an obstacle, the system will print out an “error” message.

if self.destination:  
 self.obstacle\_array[self.y\_final, self.x\_final] = 3

Once we have input a valid finishing node, we will input a 3 into our obstacle array. Now our obstacle array contains all key information of the environment: the obstacles, the robot position, and the finishing node. Finally, we can draw out our grid.

for row in range(self.ROWS):  
 for col in range(self.COLOUMNS):  
 if self.obstacle\_array[row, col] == 1:  
 color = RED  
 elif self.obstacle\_array[row, col] == 2:  
 color = GREEN  
 elif self.obstacle\_array[row, col] == 3:  
 color = BLUE  
 elif self.obstacle\_array[row, col] == 4:  
 color = YELLOW  
 else:  
 color = WHITE  
  
 pygame.draw.rect(screen, color, (col \* CELL\_SIZE, row \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))  
 pygame.draw.rect(screen, BLACK, (col \* CELL\_SIZE, row \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE), 1)

To draw our grid onto the Pygame display we will iterate through our obstacle array, each number in the array stands for a unique component in the environment, 1 is an obstacle and its grid is coloured red, 2 is the robot position and is coloured green, 3 is the finishing node and is coloured blue and 4 is the path which is coloured in yellow, Figure 11 shows what the Pygame UI would look like. The code below draws a singular square node, using the position in the array.

A screenshot of a game

AI-generated content may be incorrect.

Figure : Pathfinding

## A\* pathfinding

When a starting point and finishing point are initialised, we can begin the A\* algorithm

def astar(grid, start, end):  
 ROWS, COLS = grid.shape  
 open\_set = []  
  
 #insert start node with a cost of 0 into the heap open\_set  
 heapq.heappush(open\_set, (0, start))  
 came\_from = {}  
 #g\_score dictionary measures the distance travelled from the start node to the current node  
 g\_score = {start: 0}  
 #f\_score dictionary predicts the total cost from start to end through this node  
 f\_score = {start: heuristic(start, end)}

Our parameters are the obstacle array, so we know where the obstacles are, and the starting and finishing points. The rows and columns of the array can be found through the function grid.shape, *the open\_set* is a heap that includes the f-scores of any visited nodes. A heap has a special function that sets the variable with the lowest value to the front, which is useful in A\* as the node with the lowest f-score is the closest node to the finishing point and consequently the next node the investigate. The *came\_from* tuple will record our path when it is concluded. We will then initialise our *g\_score*, since we are at the starting node, the *g\_score* will be zero, as we haven’t moved anywhere, we will also initialise our *f\_score* list.

def heuristic(a, b):  
 # Manhattan distance  
 return abs(a[0] - b[0]) + abs(a[1] - b[1])

This function calculates the *F-score* using the Manhattan distance between the current node and the end. It computes and returns the sum of the absolute difference between the x & y coordinates.

def get\_neighbors(pos):  
 (y, x) = pos  
 directions = [(-1,0), (1,0), (0,-1), (0,1)]  
 For dy, dx in directions:  
 ny, nx = y + dy, x + dx  
 if 0 <= ny < ROWS and 0 <= nx < COLS and grid[ny, nx] != 1:  
 #yield allows you to return the value of the next neighbour without forgetting the others  
 yield (ny, nx)

To check the neighbours of our current node, we use an additional list that adds or subtracts from the current node’s coordinates to investigate its neighbours. We will then calculate the *h-score* of each neighbour and insert it into our heap.

while open\_set:  
 #returns the next best node to explore from the heap (lowest f\_score  
 \_, current = heapq.heappop(open\_set)  
 if current == end:  
 path = []  
 #fills out path array with came\_from list  
 while current in came\_from:  
 path.append(current)  
 current = came\_from[current]  
  
 path = path[::-1]  
 path.insert(0, start)  
 return path  
 #adds the current node into the visited set  
 visited.add(current)

We receive the first node from the heap, this will return the node closest to the destination and set that node as the current. If that node isn’t the finishing node, we iterate over the sequence again. However, If that node is the finishing node, then we conclude our A\* algorithm and insert the path into a *path* list and return it into our Pygame grid to be depicted on our grid.

## B-spline Interpolation

The waypoints produced by the A\* algorithm will be angular and blocky due to its grid-based structure, making them inefficient for waypoint control. To produce smoother motion, we apply a B-spline interpolation using the *splprep* and *splev* functions. This takes our original grid coordinates as control points and generates a smooth curve by parameterising the points and sampling at finer intervals.

tck, u = splprep([x, y], s=100.0)

u\_fine = np.linspace(0, 1, 25)

x\_smooth, y\_smooth = splev(u\_fine, tck)

## PID Control System

Now that we have a smoothed list of waypoints, we can move on to the PID control system. We will need to send our new revised list of waypoints and the current robot position to complete a full control loop.

try:  
 current\_waypoints, robot\_coords = queue1\_in.get(timeout=0.05)  
 if current\_waypoints == new\_waypoints:  
 pass  
 elif current\_waypoints is None:  
 control\_flag = False  
 else:  
 control\_flag = True  
except Empty:  
 pass

The controller continuously compares the robot’s current position and heading with the target waypoint and calculates two errors: the distance between the two points and the difference between the headings. The time interval between control loops is also recorded, as it is used for integral and derivative control.

current\_time = time.time()  
dt = max(current\_time - prev\_time, 1e-6)  
prev\_time = current\_time  
x\_robot, y\_robot, theta = pose\_calculation(robot\_coords)  
dx = x\_goal - x\_robot  
dy = y\_goal - y\_robot  
distance\_error = math.hypot(dx, dy)  
  
desired\_heading = math.atan2(dy, dx)  
heading\_error = desired\_heading - theta  
heading\_error\_new = math.atan2(math.sin(heading\_error), math.cos(heading\_error))

Using our errors, the PID controller computes the linear and angular velocity needed to guide the robot towards the waypoint. Two separate controllers are used – one for linear distances and one for heading. These outputs are then used to compute the wheel speeds using our forward kinematics equations.

left\_wheel = linear\_velocity + (angular\_velocity \* wheel\_base\_pixels / 2)  
right\_wheel = linear\_velocity - (angular\_velocity \* wheel\_base\_pixels / 2)  
  
left\_pwm = velocity\_to\_pwm(left\_wheel)  
right\_pwm = velocity\_to\_pwm(right\_wheel)

Once the left and right wheel speeds are calculated, they are converted into PWM values compatible with our DRV833 motor driver.

goal\_threshold = 12

if distance\_error < goal\_threshold:

distance\_integral = 0

heading\_integral = 0

prev\_distance\_error = 0

prev\_heading\_error = 0

i += 1

Once the robot has reached within 12 pixels of the waypoint, all errors are initialised to zero, and we iterate ‘I’ along our list of waypoints. This process repeats itself until it reaches its destination.

# Testing

## PID tuning

To ensure smooth and accurate path following, the PID controller was manually tuned through an iterative process. Initial values were decided depending on the robot’s general response. Adjustments were determined based on the system’s behaviour in real-time, focusing on reducing overshoot, improving response time and maintaining stability. Since we are using two controllers, there are two sets of PID variables we need to tune.

For distance control:

Kp\_distance = 0.315

Ki\_distance = 0.02

Kd\_distance = 0.03

During my testing, I noticed that the faster the robot travelled, the harder it was for the system to detect the robot's ArUco marker, which would cause the robot to become obscured and the system to stop sending information. This would cause the robot to abruptly start and stop during operation, this is why a low proportional gain was chosen. A small integral gain helped eliminate steady-state error, while a low derivative improved damping and reduced overshoot.

For heading control:

Kp\_heading = 1.73

Ki\_heading = 0.01

Kd\_heading = 0.1

With an initial smaller proportional gain, I noticed that the robot wouldn’t turn unless the error got too large, which would cause a stop and turn operation when travelling in a straight line. Therefore, a higher proportional gain was required to ensure rapid correction of heading errors. The integral and derivative terms were kept small to fine-tune the correction and prevent oscillations.

## Number of waypoints

During the B-spline sampling function, we can edit the number of waypoints sampled by editing the ‘s’ variable.

u\_fine = np.linspace(0, 1, s)

By using a higher value of ‘s’ such as 25, the robot can perform tighter manoeuvres around obstacles. This attribute is desirable due to our relatively small area to manoeuvre in. We can see in Figure 12 the waypoints in the shape of a B-spline from the robot to the destination.

A toy car with wheels and dots

AI-generated content may be incorrect.

Figure : Robot

## Object Detection Testing

To improve the reliability of the object detection system, especially under noisy conditions, I implemented a post-processing filter system based on the counter area of any potential “obstacles”. As mentioned earlier, due to light abstractions and HSV mismatches, I observed small noise artefacts that would present themselves as false positives.

To eliminate these false positives, I used *cv2.connectedComponentsWithStats()* with our *green\_mask* to return the area of any objects detected. A minimum threshold of 200 pixels was set and any object with an area less than that threshold will be ignored.

This approach significantly improves detection quality and grid mapping accuracy, especially in situations where there are undesirable lighting conditions. After applying the threshold, we should receive a clean, stable frame to map any obstacles. Figure 13 shows the *green\_mask* before the contour filters, We can see there are large white areas (obstacles) and smaller white areas (noise). Figure 14 shows *green\_mask* after the contour filtering; now we are left with just the obstacles.

‘

# A white object in the sky AI-generated content may be incorrect.A white dots in the sky AI-generated content may be incorrect.Evaluation

Figure : green mask without false positives

Figure : green mask with false positives

## PID Response

To measure the effectiveness of my control system, I will use *matplotlib.pyplot* I plotted the two errors with respect to time. The output returns two graph, one for the distance error and the other for the heading error.

**A graph showing a number of data

AI-generated content may be incorrect.**

Figure 15: PID error vs time graph

We can see in Figure 15, in the distance PID response, the robot’s error starts high, then rapidly drops to approximately 30 pixels before spiking again. This happens because 30 pixels is the defined goal threshold — once the robot gets within that range, the system considers the waypoint reached and moves on to the next one. Since the robot is still moving as this change happens, the distance to the new waypoint increases, causing that sudden jump in error. As the robot gets closer to its target, its speed (or rate of error change) naturally decreases. In a few cases, it even passes the 30-pixel mark before slowing down, which shows a quick rise time but also a bit of overshoot. We can fix this issue by lowering the proportional gain to attempt to reduce the acceleration. Increasing the derivative gain can help resist sudden changes like fast closing distance, which can help reduce overshoot and stabilise stopping.

In the heading PID response, the robot's heading starts very high (0.6 radians) and quickly drops down to near zero, indicating a low rise time. However, past the 2-second mark, we can see the error oscillating around 0 radians, indicating a high settling time. The oscillating may be caused by an overly sensitive system reacting too quickly to minor heading changes. We can fix this issue by lowering the proportional gain slightly to avoid overcorrection and increasing the derivative gain to dampen oscillations. We can also introduce a dead zone where the error is within a small range, treat it as zero to avoid unnecessary adjustments

## FPS

For a system to be used for real-time operations, the FPS needs to be high. In our scenario, as we are reacting to any obstacles during the operation, the FPS need to be high enough to detect the object while giving enough time to redirect the robot to avoid collision. During standard operation, I recorded an FPS of approximately 25 fps. Although for optimal real-time performance, this fps is a little bit low, with the standard in industry being 30 fps. However, the robot can still accurately react to dynamic changes in its environment.

In my control loop my integral and derivative components depend on *dt* which will be the time interval between two frames. A low fps would create a large *dt* which may introduce errors such as over shooting. Additionally, the rate information is sent to the ESP32 also depends on the frame rate. Therefore, a low fps would cause the rate of information transfer to decrease, which may result in slow or sluggish behaviour.

With low fps, the system captures fewer frames per second, so it misses intermediate positions of fast-moving objects. Since the system relies on an ArUco marker to determine the position of the robot, if it is travelling too fast, the ArUco marker will appear to be blurred, and the system will lose track of the robot. To mitigate this issue, we need to set the max speed of the robot to a value that can be detected by the system.

# Conclusion

## Project Aims

This project aimed to design and develop an alternative and affordable indoor positional system (IPS) capable of guiding an autonomous robot through a dynamically changing environment. Through systematic implementation and testing, this goal was successfully met. A modular environment has been constructed, allowing for rapid prototyping and efficient experimentation. ArUco markers were generated and used effectively for perspective warping and localisation as well as pose estimation. The robot was constructed to be compact, using low-cost parts with the ESP32 enabling Wi-Fi communication by utilising a local MQTT broker.

The camera calibration and perspective transformation steps allowed us to manipulate our camera into a top-down view, allowing us to map the environment onto a 2D grid, forming the foundation for accurate localisation and navigation. The UI was designed using Pygame, giving a clear and intuitive understanding and control of the environment and its desired path, while also checking for obstacle interference during pathfinding, allowing dynamic obstacle avoidance.

The A\* pathfinding algorithm, combined with a PID controller, enabled the robot to accurately reach the target waypoints. Furthermore, the implementation of B-Spline curves helped smooth the robot’s motion, and real-time control feedback was successfully achieved through the manual PID tuning process.

Each of the objectives – from environment setup and ArUco generation to localisation, pathfinding and control – was completed, creating a successful IPS prototype. The project demonstrated that a low-cost, computer vision-based IPS can achieve precise and dynamic autonomous movement in real time. This proof of concept sets a strong foundation for scaling the system to real-world applications in industrial, agricultural and healthcare scenarios.

## Project Requirements

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Category** | **ID** | **Requirement** | **Type** | **Priority** | **Achieved?** |
| **General** | G1 | Must be able to perform localising, pathfinding and obstacle avoidance seamlessly | Functional | High | Yes |
|  | G2 | One a new obstacle obstructs the robot’s current path, a new path will be generated. | Non-Functional | Medium | Yes |
|  | G3 | Use a 3d printed mount to orient the camera into a top-down position | Functional | High | Yes |
|  | G5 | 3d printed arena needs to be modular to enable rapid testing in a variety of environments | Non-Functional | Medium | Yes |
| **User** | U1 | The user should be able to identify the robot and the finishing point on the graph. | Functional | High | Yes |
|  | U2 | The user should be able to select the finishing node and identify any obstacles. | Functional | High | Yes |
|  | U3 | Users will be notified if an obstacle has crossed the path of the robot | Non-Functional | Medium | No |
|  | U4 | The user interface should be streamlined and easy to use, and understand | Non-Functional | Low | Yes |
|  | U5 | The user interface should run in parallel to the object detection and image processing algorithm. | Non-Functional | High | Yes |
|  | U6 | The user will be informed whether the connection with esp32 is established. | Non-Functional | Medium | Yes |
| **Robotic car** | R1 | Should be designed to be as compact as possible to ensure the area is a large enough environment | Non-Functional | High | Yes |
|  | R2 | Uses ESP32 to receive information from the software and control the wheels | Functional | High | Yes |
|  | R3 | Motor Drivers are needed to ensure bi-directional compatibility of the robot. | Non-Functional | High | Yes |
|  | R4 | The robot should have a long battery life for prolonged testing | Non-Functional | Low | Yes |
|  | R5 | A robot must have low latency when responding to commands from the system | Non-Functional | High | Yes |
|  | R6 | The battery source must be rechargeable to minimise environmental impact | Non-Functional | Low | Yes |
| **Software** | S1 | Must be written in Python to ensure rapid prototyping | Non-Functional | Medium | Yes |
|  | S2 | The system must be implemented using OpenCV, Pygame and Arduino IDE software. | Functional | High | Yes |
|  | S3 | The system should utilise multiprocessing for parallel development | Non-Functional | High | Yes |
|  | S | The system should run at least 30fps to ensure real-time operation | Non-Functional | Medium | No |
|  | S4 | All code must be annotated and easy to understand and follow | Non-Functional | Medium | Yes |
|  | S5 | The system should detect and track green objects in the arena and update the grid. | Functional | High | Yes |

## Research Questions

### Core Functionality and Design Evaluation

The vision-based system can effectively enable an autonomous ground vehicle to navigate and respond to dynamic environments using a 2D mapping approach

### Scalability

To scale this prototype up so it can be used in real-world applications such as industrial or agricultural settings, we would need to increase the FOV of the camera, This can be achieved in two ways. Using a greater ultrawide camera, the simplest approach, however, as the robot's ArUco marker size is relative to the camera's FOV, increasing the FOV will also require an increase in the ArUco marker size and subsequently a larger robot. The second approach is using a video processing method called image stitching, this is where multiple cameras are used in a top-down view to capture the whole area, I believe this is the optimal way as the FOV of the camera relative to the robot size will remain the same, meaning no changes will be made to the robots’ dimensions. However, image stitching is computationally expensive; therefore, a more powerful machine is necessary to run at a high FPS.

### Algorithm Efficiency

The A\* algorithm is extremely efficient; it features a low computation time, which allows us to quickly recalculate an alternative during pathfinding. As the left and right wheels of my robot have a speed imbalance between them, the PID controller helps counteract this and prevents drift when travelling in a straight line.

### Environmental Adaptation

What helps make my version of an IPS stand out from alternative IPS (UWB, Odometry) is how my system can adapt to changes in a dynamic environment, which makes it perfect in busy and constantly changing environments.

## Future Improvements

Several improvements could be made to enhance the system's functionality. While colour filtering is currently used for object detection, the same principle could be extended to identify areas of interest using a colour or even replaced with a neural network for more advanced object detection. This would make the robot truly autonomous, reducing the need for human input.

For example, a cleaning robot could identify dirty areas in real-time and navigate specifically to those spots.

Another potential improvement is scaling up the number of operational robots. Unlike other IPS methods, such as UWB or odometry-based systems, this approach can be easily expanded by simply assigning a new ArUco marker to each additional robot. This scalability could significantly increase productivity, especially in environments like hospitals where staff are often overburdened.

I would also include a way to auto-tune the PID values, as manual PID tuning is lengthy and not the most accurate method.

## Challenges

Throughout the project, I encountered several challenges, particularly due to my limited prior experience with localisation and pathfinding. To overcome these challenges, I invested time researching and studying different algorithms, such as the A\* algorithm and B-spline curves, to understand their function regarding my system. Additionally, I had to familiarise myself with new tools and libraries such as Pygame for visual simulation and MQTT for wireless communication.

The prototyping process for the robot also yielded some problems, in an attempt to miniaturize the robot as small as possible, multiple revisions had to be made to the perfboard and the chassis of the robot for it all to fit in such a compact space.

Perhaps my most challenging obstacle were the abstractions caused by varying light intensity during the obstacle detection. This forced me to utilize multiple post processing methods such as Gaussian blur, morphology effects and contour analysis to obtain an abstract free frame.

References

Alam, M. (2023). *Localisation in Robotics for Mobile Robots - Mansoor Alam - Medium*. [online] Medium. Available at: https://medium.com/@mansooralam129047/localization-in-robotics-for-mobile-robots-ec3ad31f99d4.

CFD, C. (2025). *An Introduction to B-Spline Curves*. [online] resources.system-analysis.cadence.com. Available at: https://resources.system-analysis.cadence.com/blog/msa2022-an-introduction-to-b-spline-curves.

docs.opencv.org. (2025). *OpenCV: Geometric Image Transformations*. [online] Available at: https://docs.opencv.org/4.x/da/d54/group\_\_imgproc\_\_transform.html.

Editorial (2023). *Mobile robot localization - Challenges and various methods*. [online] RoboticsBiz. Available at: https://roboticsbiz.com/mobile-robot-localization-challenges-and-various-methods [Accessed 5 May 2025].

GeeksforGeeks (2022). *Difference between Sequential and Parallel Computing*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/difference-between-sequential-and-parallel-computing [Accessed 5 May 2025].

GeeksforGeeks (2024). *Detecting ArUco markers with OpenCV and Python*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/detecting-aruco-markers-with-opencv-and-python-1/.

EMQX Team, E.T.C. (2025.). *MQTT Protocol Explained: The Basics and a Quick Tutorial*. [online] www.emqx.com. Available at: https://www.emqx.com/en/blog/the-easiest-guide-to-getting-started-with-mqtt.

Ni.com. (2025). *The PID Controller & Theory Explained*. [online] Available at: https://www.ni.com/en/shop/labview/pid-theory-explained.html?srsltid=AfmBOorswVYY\_NstgaM8pekuuaiujqRxueRwftvOGNbA2U1av2BRLTgi [Accessed 5 May 2025].

OpenCV (2018). *About OpenCV*. [online] OpenCV. Available at: https://opencv.org/about/.

OpenCV (2025). *Opencv: Camera Calibration*. [online] docs.opencv.org. Available at: https://docs.opencv.org/4.x/dc/dbb/tutorial\_py\_calibration.html.

Patel, A. (1997). *Introduction to A\**. [online] theory.stanford.edu. Available at: https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

Sadekar, K. (2020). *Understanding Lens Distortion | LearnOpenCV #*. [online] LearnOpenCV. Available at: https://learnopencv.com/understanding-lens-distortion/.

Staff, L.E. (2023). *Controlling DC Motors with DRV8833 Motor Driver and Arduino*. [online] Last Minute Engineers. Available at: https://lastminuteengineers.com/drv8833-arduino-tutorial/.

Appendix A: ArUco Generator

import cv2  
import numpy as np  
import matplotlib.pyplot as plt  
  
# Define the dictionary we want to use  
aruco\_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT\_6X6\_250)  
  
# Generate a marker  
marker\_id = 43  
marker\_size = 200 # Size in pixels  
marker\_image = cv2.aruco.generateImageMarker(aruco\_dict, marker\_id, marker\_size)  
  
cv2.imwrite('marker\_43.png', marker\_image)  
plt.imshow(marker\_image, cmap='gray', interpolation='nearest')  
plt.axis('off') # Hide axes  
plt.title(f'ArUco Marker {marker\_id}')  
plt.show()

Appendix B: Camera Matrix Code

import cv2  
import numpy as np  
import os  
import glob  
  
# Defining the dimensions of the checkerboard  
CHECKERBOARD = (6, 9)  
criteria = (cv2.TERM\_CRITERIA\_EPS + cv2.TERM\_CRITERIA\_MAX\_ITER, 30, 0.001)  
  
# Creating vector to store vectors of 3D points for each checkerboard image  
objpoints = []  
# Creating vector to store vectors of 2D points for each checkerboard image  
imgpoints = []  
  
# Defining the world coordinates for 3d points  
objp = np.zeros((1, CHECKERBOARD[0] \* CHECKERBOARD[1], 3), np.float32)  
objp[0, :, :2] = np.mgrid[0:CHECKERBOARD[0], 0:CHECKERBOARD[1]].T.reshape(-1, 2)  
prev\_img\_shape = None  
  
# Extracting path of individual image stored in a given directory  
images = glob.glob(r'C:\Users\dough\PycharmProjects\FINAL\_YEAR\_PROJECT\\*.jpg')  
for fname in images:  
 img = cv2.imread(fname)  
 gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  
 # Find the chess board corners  
 # If desired number of corners are found in the image then ret = true  
 ret, corners = cv2.findChessboardCorners(gray, CHECKERBOARD, cv2.CALIB\_CB\_ADAPTIVE\_THRESH + cv2.CALIB\_CB\_FAST\_CHECK + cv2.CALIB\_CB\_NORMALIZE\_IMAGE)  
  
 """  
 If desired number of corner are detected,  
 We refine the pixel coordinates and display   
 them on the images of a checker board  
 """  
 if ret == True:  
 objpoints.append(objp)  
 # refining pixel coordinates for given 2d points.  
 corners2 = cv2.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)  
  
 imgpoints.append(corners2)  
  
 # Draw and display the corners  
 img = cv2.drawChessboardCorners(img, CHECKERBOARD, corners2, ret)  
  
 cv2.imshow('img', img)  
 cv2.waitKey(0)  
  
cv2.destroyAllWindows()  
  
h, w = img.shape[:2]  
  
"""  
Performing camera calibration by   
passing the value of known 3d points (objpoints)  
and the corresponding pixel coordinates of the   
detected corners (imgpoints)  
"""  
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, None)  
  
print("Camera matrix : \n")  
print(mtx)  
print("dist : \n")  
print(dist)  
print("rvecs : \n")  
print(rvecs)  
print("tvecs : \n")  
print(tvecs)

Appendix C: Parallel Autonomous Robot Code

import multiprocessing  
from asyncio import timeout  
import cv2  
import numpy as np  
import time  
from PIL.ImageCms import Flags  
from numpy.ma.extras import average  
import math  
from multiprocessing import Process, Queue  
from queue import Empty  
import pygame  
from collections import deque  
import heapq  
import paho.mqtt.client as mqtt  
import json  
import matplotlib.pyplot as plt  
from scipy.interpolate import splprep, splev  
  
broker = ''  
port = 1883  
topic = "pathfinding/pwm"  
status\_topic = "esp32/status"  
control\_topic = "esp32/motors"  
  
CORNERS = False  
PATH = False  
START\_UP = True  
path\_list = None  
smoothed\_points = None  
esp32\_ready = False  
  
  
def control\_algorithm(queue1\_in):  
 import time  
 import math  
 from queue import Empty  
 import paho.mqtt.client as mqtt  
  
 pixels\_per\_cm = 6.46  
 control\_flag = False  
 current\_waypoints = []  
 new\_waypoints = []  
  
 prev\_time = time.time()  
  
 Kp\_distance = 0.315  
 Ki\_distance = 0.02  
 Kd\_distance = 0.03  
  
 Kp\_heading = 1.73  
 Ki\_heading = 0.01  
 Kd\_heading = 0.1

# Setup MQTT client  
 mqtt\_broker = ""  
 mqtt\_topic = "esp32/motors"  
 client = mqtt.Client()  
 client.connect(mqtt\_broker, 1883, 60)  
  
 while True:  
 distance\_integral = 0  
 heading\_integral = 0  
 prev\_distance\_error = 0  
 prev\_heading\_error = 0  
  
 try:  
 current\_waypoints, robot\_coords = queue1\_in.get(timeout=0.05)  
 if current\_waypoints == new\_waypoints:  
 pass  
 elif current\_waypoints is None:  
 control\_flag = False  
 else:  
 control\_flag = True  
 except Empty:  
 pass  
  
 # Send stop signal if no path  
 msg = {"left": 0, "right": 0}  
 client.publish(mqtt\_topic, str(msg))  
  
 while control\_flag:  
 if current\_waypoints is None:  
 break  
 first\_point = True  
 i = 0  
 while control\_flag and current\_waypoints is not None and i < len(current\_waypoints):  
  
 x\_goal, y\_goal = current\_waypoints[i]  
  
 try:  
 new\_waypoints, robot\_coords = queue1\_in.get(timeout=0.05)  
 if new\_waypoints == current\_waypoints:  
 pass  
 elif new\_waypoints is None:  
 control\_flag = False  
 break  
 else:  
 distance\_integral = 0  
 heading\_integral = 0  
 prev\_distance\_error = 0  
 prev\_heading\_error = 0  
 i = 0  
 first\_point = True  
 current\_waypoints = new\_waypoints  
 prev\_time = time.time()  
 continue  
 except Empty:  
 pass  
  
 if robot\_coords is None:  
 prev\_time = time.time()  
 msg = {"left": 0, "right": 0}  
 client.publish(mqtt\_topic, str(msg))  
 continue  
  
 current\_time = time.time()  
 dt = max(current\_time - prev\_time, 1e-6)  
 prev\_time = current\_time  
 x\_robot, y\_robot, theta = pose\_calculation(robot\_coords)  
 dx = x\_goal - x\_robot  
 dy = y\_goal - y\_robot  
 distance\_error = math.hypot(dx, dy)  
  
 desired\_heading = math.atan2(dy, dx)  
 heading\_error = desired\_heading - theta  
 heading\_error\_new = math.atan2(math.sin(heading\_error), math.cos(heading\_error))  
  
 wheel\_base\_pixels = 5 \* pixels\_per\_cm  
  
 # Distance PID  
 distance\_integral += distance\_error \* dt  
 distance\_derivative = (distance\_error - prev\_distance\_error) / dt  
 linear\_velocity = (  
 Kp\_distance \* distance\_error +  
 Ki\_distance \* distance\_integral +  
 Kd\_distance \* distance\_derivative  
 )  
 prev\_distance\_error = distance\_error  
  
 # Heading PID  
 heading\_integral += heading\_error\_new \* dt  
 heading\_derivative = (heading\_error\_new - prev\_heading\_error) / dt  
 angular\_velocity = (  
 Kp\_heading \* heading\_error\_new +  
 Ki\_heading \* heading\_integral +  
 Kd\_heading \* heading\_derivative  
 )  
 prev\_heading\_error = heading\_error\_new  
  
 if i == 0 and first\_point == True:  
 linear\_velocity = 0  
 if (-1 < heading\_error\_new < 1):  
 first\_point = False  
  
  
 left\_wheel = linear\_velocity + (angular\_velocity \* wheel\_base\_pixels / 2)  
 right\_wheel = linear\_velocity - (angular\_velocity \* wheel\_base\_pixels / 2)  
  
 left\_pwm = velocity\_to\_pwm(left\_wheel)  
 right\_pwm = velocity\_to\_pwm(right\_wheel)  
  
 # print(linear\_velocity, angular\_velocity)  
 # Publish PWM as MQTT message  
 msg = {"left": left\_pwm, "right": right\_pwm}  
 client.publish(mqtt\_topic, str(msg))  
  
 goal\_threshold = 30  
 if distance\_error < goal\_threshold:  
 print("yippe")  
 distance\_integral = 0  
 heading\_integral = 0  
 prev\_distance\_error = 0  
 prev\_heading\_error = 0  
 i = i + 1  
  
 control\_flag = False  
 new\_waypoints = current\_waypoints  
 break  
  
  
def velocity\_to\_pwm(velocity, max\_pwm=60, min\_pwm = 35, min\_velocity=0, max\_velocity=100):  
  
 if velocity > 0:  
 velocity = abs(velocity)  
 if velocity > 100:  
 velocity = 100  
 pwm = (velocity / 100.0) \* (max\_pwm - min\_pwm) + min\_pwm  
 return pwm  
  
 elif velocity == 0:  
  
 pwm = 0  
 return pwm  
  
 else:  
 velocity = abs(velocity)  
 if velocity > 100:  
 velocity = 100  
 pwm = -((velocity / 100.0) \* (max\_pwm - min\_pwm) + min\_pwm)  
 return pwm  
  
  
def pose\_calculation(robot\_coords):  
 delta\_x = robot\_coords[0][0] - robot\_coords[1][0]  
 delta\_y = robot\_coords[0][1] - robot\_coords[1][1]  
  
 x = int(((robot\_coords[0][0] + robot\_coords[2][0]) // 2))  
 y = int(((robot\_coords[0][1] + robot\_coords[2][1]) // 2))  
 theta = math.atan2(delta\_y, delta\_x)  
  
 return x, y, theta  
  
def insert\_midpoints(path\_list):  
 new\_path = []  
  
 for index, (y,x) in enumerate(path\_list):  
 if (y,x) == (0,3) or (y,x) == (0,4):  
 path\_list[index] = (y + 0.3, x)  
 elif (y,x) == (3,3) or (y,x) == (3,4):  
 path\_list[index] = (y - 0.3, x)  
  
  
  
 for i in range(len(path\_list) - 1):  
 p1 = path\_list[i]  
 p2 = path\_list[i + 1]  
  
 # Add original point  
  
 new\_path.append(p1)  
  
 # Calculate and insert midpoint  
 midpoint = ((p1[0] + p2[0]) / 2, (p1[1] + p2[1]) / 2)  
  
  
 new\_path.append(midpoint)  
  
 # Add the final point  
 new\_path.append(path\_list[-1])  
  
 return new\_path  
  
  
  
  
def spline\_line(path\_list):  
  
 print(len(path\_list))  
 from scipy.interpolate import splprep, splev  
 CELL\_SIZE = 80  
 points = []  
 path\_list.pop(0)  
 # Build points from the path list  
 for index, (x, y) in enumerate(path\_list):  
 point\_current\_y = (path\_list[index][0] \* CELL\_SIZE) + (0.5 \* CELL\_SIZE)  
 point\_current\_x = (path\_list[index][1] \* CELL\_SIZE) + (0.5 \* CELL\_SIZE)  
 points.append((point\_current\_x, point\_current\_y))  
  
 # Separate into x and y arrays  
 x = np.array([p[0] for p in points])  
 y = np.array([p[1] for p in points])  
  
 # Parameterize the points  
 tck, u = splprep([x, y], s=100.0) # <-- 's' controls the amount of smoothing; increase it for curvier results  
  
 # Create many points along the curve  
 u\_fine = np.linspace(0, 1, 25) # 100 points for a very smooth curve  
  
 # Evaluate the B-spline at the new points  
 x\_smooth, y\_smooth = splev(u\_fine, tck)  
  
 # Create final smoother points  
 smoother\_points = [(int(a), int(b)) for a, b in zip(x\_smooth, y\_smooth)]  
  
 return smoother\_points  
  
  
def astar(grid, start, end):  
 ROWS, COLS = grid.shape  
 open\_set = []  
  
 #insert start node with a cost of 0 into heap open\_set  
 heapq.heappush(open\_set, (0, start))  
 came\_from = {}  
 #g\_score dictionary measures distance travled from start node to current node  
 g\_score = {start: 0}  
 #f\_score dictionary predicts the total cost from start to end through this node  
 f\_score = {start: heuristic(start, end)}  
 #keeps track of explored nodes  
 visited = set()  
  
 #function checks out neighbors  
 def get\_neighbors(pos):  
 (y, x) = pos  
 directions = [(-1,0), (1,0), (0,-1), (0,1)]  
 for dy, dx in directions:  
 ny, nx = y + dy, x + dx  
 if 0 <= ny < ROWS and 0 <= nx < COLS and grid[ny, nx] != 1:  
 #yield allows you to return the value of the next neighbour without forgetting the others  
 yield (ny, nx)  
  
 while open\_set:  
 #returns the next best node to explore form heap (lowest f\_score  
 \_, current = heapq.heappop(open\_set)  
 if current == end:  
 path = []  
 #fills out path array with came\_from list  
 while current in came\_from:  
 path.append(current)  
 current = came\_from[current]  
  
 path = path[::-1]  
 path.insert(0, start)  
 return path  
 #adds current node into the visited set  
 visited.add(current)  
  
 #checks all walkable neighbors from def get\_neighbors  
 for neighbor in get\_neighbors(current):  
 #if we've checked with neighbor before skip  
 if neighbor in visited:  
 continue  
 #increase g score by 1  
 tentative\_g = g\_score[current] + 1  
 #get g\_score if not in dictionary return infinity  
 if tentative\_g < g\_score.get(neighbor, float('inf')):  
 #sets neighbour as current node  
 came\_from[neighbor] = current  
 g\_score[neighbor] = tentative\_g  
 #find f score  
 f\_score[neighbor] = tentative\_g + heuristic(neighbor, end)  
 #adds this neighbor to heap  
 heapq.heappush(open\_set, (f\_score[neighbor], neighbor))  
  
 return None # No path found  
  
def heuristic(a, b):  
 # Manhattan distance  
 return abs(a[0] - b[0]) + abs(a[1] - b[1])  
  
  
  
def pathfinding\_algorithm(queue\_in, queue\_out):  
 pygame.init()  
  
 #grab screen size dimensions from path\_array  
 SCREEN\_WIDTH = 800  
 SCREEN\_HEIGHT = 600  
 CELL\_SIZE = 80  
 WHITE = (255, 255, 255)  
 BLACK = (0, 0, 0)  
 BLUE = (0, 0, 255)  
 RED = (200, 0, 0)  
 GREEN = (0, 255, 0)  
 YELLOW = (255, 255, 0)  
  
  
 screen = pygame.display.set\_mode((SCREEN\_WIDTH, SCREEN\_HEIGHT))  
 start\_img = pygame.image.load('button.png').convert\_alpha()  
 clear\_img = pygame.image.load('button (1).png').convert\_alpha()  
  
 class Grid():  
 def \_\_init\_\_(self, obstacle\_array, ROWS, COLOUMNS):  
 self.obstacle\_array = obstacle\_array  
 self.ROWS = ROWS  
 self.COLOUMNS = COLOUMNS  
 self.path\_array = np.zeros((self.ROWS, self.COLOUMNS), dtype=int)  
 self.obstacle\_only\_array = np.zeros((self.ROWS, self.COLOUMNS), dtype=int)  
 self.destination = False  
 self.y\_start = None  
 self.x\_start = None  
 self.y\_final = None  
 self.x\_final = None  
 self.reset = False  
 self.path\_set = False  
 self.path\_reset = False  
 self.path\_computed = False  
 self.path\_list = []  
  
 def draw\_grid(self):  
 (x\_pos, y\_pos) = pygame.mouse.get\_pos()  
 if (x\_pos <= self.COLOUMNS \* CELL\_SIZE and y\_pos <= self.ROWS \* CELL\_SIZE) and not self.destination:  
 if pygame.mouse.get\_pressed()[0] == 1:  
 x\_grid = int(x\_pos // CELL\_SIZE)  
 y\_grid = int(y\_pos // CELL\_SIZE)  
 if 0 <= y\_grid < self.ROWS and 0 <= x\_grid < self.COLOUMNS:  
 if obstacle\_array[y\_grid, x\_grid] in [1, 2]:  
 print("error")  
 else:  
 self.y\_final = y\_grid  
 self.x\_final = x\_grid  
 self.destination = True  
  
 if self.reset:  
 self.destination = False  
 self.y\_final = None  
 self.x\_final = None  
 self.path\_computed = False  
 self.reset = False  
  
 if self.destination:  
 self.obstacle\_array[self.y\_final, self.x\_final] = 3  
  
 if self.path\_set:  
 for (y, x) in self.path\_list:  
 if self.obstacle\_array[y, x] == 1:  
 self.path\_set = False  
 self.path\_reset = True  
 elif self.obstacle\_array[y, x] == 0:  
 self.obstacle\_array[y, x] = 4 # Path marked in yellow  
  
 for row in range(self.ROWS):  
 for col in range(self.COLOUMNS):  
 if self.obstacle\_array[row, col] == 1:  
 color = RED  
 elif self.obstacle\_array[row, col] == 2:  
 color = GREEN  
 elif self.obstacle\_array[row, col] == 3:  
 color = BLUE  
 elif self.obstacle\_array[row, col] == 4:  
 color = YELLOW  
 else:  
 color = WHITE  
  
 pygame.draw.rect(screen, color, (col \* CELL\_SIZE, row \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))  
 pygame.draw.rect(screen, BLACK, (col \* CELL\_SIZE, row \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE), 1)  
  
 class Button():  
 def \_\_init\_\_(self, x, y, image, scale):  
 width = image.get\_width()  
 height = image.get\_height()  
 self.image = pygame.transform.scale(image, (int(width \* scale), int(height \* scale)))  
 self.rect = self.image.get\_rect()  
 self.rect.topleft = (x, y)  
 self.clicked = False  
  
 def draw(self):  
 action = False  
 pos\_1 = pygame.mouse.get\_pos()  
  
 if self.rect.collidepoint(pos\_1):  
 if pygame.mouse.get\_pressed()[0] == 1 and not self.clicked:  
 self.clicked = True  
 action = True  
  
 if pygame.mouse.get\_pressed()[0] == 0:  
 self.clicked = False  
  
 screen.blit(self.image, (self.rect.x, self.rect.y))  
 return action  
  
 # Main game loop  
 start\_button = Button(500, 400, start\_img, 1)  
 clear\_Button = Button(200, 400, clear\_img, 1)  
 obstacle\_array = None  
 run = True  
 initialize\_grid = False  
 initialize\_grid\_flag = False  
  
 while run:  
 screen.fill((WHITE))  
 try:  
 obstacle\_array = queue\_in.get(timeout=0.05)  
 ROWS, COLS = obstacle\_array.shape  
 initialize\_grid\_flag = True  
 except:  
 pass  
  
 start\_clicked = start\_button.draw()  
 clear\_clicked = clear\_Button.draw()  
  
 if not initialize\_grid and initialize\_grid\_flag == True:  
 grid = Grid(obstacle\_array, ROWS, COLS)  
 initialize\_grid = True  
 elif initialize\_grid\_flag == True:  
 grid.obstacle\_array = obstacle\_array  
 grid.draw\_grid()  
  
 if clear\_clicked:  
 grid.reset = True  
 grid.path\_set = False  
 grid.path\_reset = False  
 grid.path\_computed = False  
 queue\_out.put(('cleared', None))  
  
 if (start\_clicked and grid.destination and not grid.path\_computed) or grid.path\_reset:  
 grid.path\_reset = False  
 start = None  
 end = (grid.y\_final, grid.x\_final)  
  
 for y in range(grid.ROWS):  
 for x in range(grid.COLOUMNS):  
 if grid.obstacle\_array[y, x] == 2:  
 start = (y, x)  
 break  
  
 if start and end:  
 (grid.y\_start, grid.x\_start) = start  
 path = astar(grid.obstacle\_array, start, end)  
 if path:  
 grid.path\_set = True  
 grid.path\_list = path  
 grid.path\_computed = True  
 queue\_out.put(('path', path))  
 else:  
 queue\_out.put(('no\_path', []))  
  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 run = False  
  
 pygame.display.flip()  
  
 pygame.quit()  
  
def mark\_robot(obstacle\_array, topLeft, topRight, bottomRight, bottomLeft, coloumns, rows):  
  
 x\_cord = int((topLeft[0] + topRight[0] + bottomLeft[0] + bottomRight[0]) // 4)  
 y\_cord = int((topLeft[1] + topRight[1] + bottomLeft[1] + bottomRight[1]) // 4)  
  
 x\_grid = int(x\_cord // 80)  
 y\_grid = int(y\_cord // 80)  
  
 if 0 <= y\_grid < coloumns and 0 <= x\_grid < rows:  
 obstacle\_array[y\_grid, x\_grid] = 2  
  
 return obstacle\_array  
  
def robot\_detection(cropped\_frame):  
 cropped\_frame\_gray = cv2.cvtColor(cropped\_frame, cv2.COLOR\_BGR2GRAY)  
  
 aruco\_dict\_robot = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT\_6X6\_250)  
 parameters\_robot = cv2.aruco.DetectorParameters()  
  
 detector\_robot = cv2.aruco.ArucoDetector(aruco\_dict\_robot, parameters\_robot)  
  
 robot\_corners, robot\_ids, robot\_rejected = detector\_robot.detectMarkers(cropped\_frame\_gray)  
  
 if robot\_ids is not None and 43 in robot\_ids:  
 index = np.where(robot\_ids == 43)[0][0]  
 marker\_corners = robot\_corners[index]  
 true\_robot\_corners = marker\_corners.reshape((4, 2))  
 return true\_robot\_corners  
 else:  
 return None  
  
def object\_detection(cropped\_frame, coloumns, rows, history\_buffer):  
  
 # Convert to HSV  
 cropped\_frame\_HSV\_unblured = cv2.cvtColor(cropped\_frame, cv2.COLOR\_BGR2HSV)  
 cropped\_frame\_HSV = cv2.GaussianBlur(cropped\_frame\_HSV\_unblured, (5, 5), 0)  
  
# Define HSV ranges to capture all shades of green (light, medium, dark)  
 green\_lower\_1 = np.array([35, 20, 40]) # Adjusted to avoid over-detection & flickering  
 green\_upper\_1 = np.array([85, 255, 255])  
  
  
 # Create masks to filter out green color  
 green\_mask = cv2.inRange(cropped\_frame\_HSV, green\_lower\_1, green\_upper\_1)  
  
  
 kernel = np.ones((5, 5), np.uint8)  
 green\_mask = cv2.morphologyEx(green\_mask, cv2.MORPH\_OPEN, kernel)  
 green\_mask = cv2.morphologyEx(green\_mask, cv2.MORPH\_CLOSE, kernel)  
  
 num\_labels, labels, stats, \_ = cv2.connectedComponentsWithStats(green\_mask)  
  
 min\_area = 200  
  
 for i in range(1, num\_labels):  
 if stats[i, cv2.CC\_STAT\_AREA] < min\_area:  
 green\_mask[labels == i] = 0  
  
 green\_pixels = cv2.findNonZero(green\_mask)  
 obstacle\_array = np.zeros((coloumns, rows))  
 if green\_pixels is not None:  
 for point in green\_pixels:  
 x, y = point[0] # point[0] contains [x, y] from each non-zero pixel  
  
 grid\_x = x // 80  
 grid\_y = y // 80  
  
  
  
 if 0 <= grid\_y < coloumns and 0 <= grid\_x < rows:  
 obstacle\_array[grid\_y, grid\_x] = 1  
  
  
 true\_robot\_corners = robot\_detection(cropped\_frame)  
  
  
 if true\_robot\_corners is not None:  
 (topLeft, topRight, bottomRight, bottomLeft) = [tuple(map(int, pt)) for pt in true\_robot\_corners]  
 obstacle\_array = mark\_robot(obstacle\_array, topLeft, topRight, bottomRight, bottomLeft, coloumns, rows)  
  
  
 return obstacle\_array, true\_robot\_corners  
  
  
def perspective\_warp(ordered\_corners, new\_points, undistorted\_frame, average\_length, average\_width):  
 #apply perspective warp  
 matrix = cv2.getPerspectiveTransform(ordered\_corners, new\_points)  
 result = cv2.warpPerspective(undistorted\_frame, matrix, (average\_length, average\_width))  
  
 #crop result for rectangular grid image  
 start\_y = int((average\_width % 80) // 2)  
 start\_x = int((average\_length % 80) // 2)  
  
 end\_x = (average\_length - start\_x)  
 end\_y = (average\_width - start\_y)  
  
  
 cropped\_result = result[start\_y:end\_y, start\_x:end\_x]  
  
 return cropped\_result  
  
  
  
  
def rectangle(ordered\_corners, undistorted\_frame):  
 # find dimensions of undistored\_frame  
  
 max\_left = abs(ordered\_corners[0][1] - ordered\_corners[1][1])  
 max\_right = abs(ordered\_corners[2][1] - ordered\_corners[3][1])  
 average\_width = int((max\_left + max\_right) / 2)  
  
 max\_up = abs(ordered\_corners[1][0] - ordered\_corners[2][0])  
 max\_down = abs(ordered\_corners[0][0] - ordered\_corners[3][0])  
 average\_length = int((max\_up + max\_down) / 2)  
  
 columns = (int(average\_width // 80))  
 rows = (int(average\_length // 80))  
  
 start\_y = int((average\_width % 80) // 2)  
 start\_x = int((average\_length % 80) // 2)  
  
 end\_x = (average\_length - start\_x)  
 end\_y = (average\_width - start\_y)  
  
 cropped\_width = end\_x - start\_x  
 cropped\_height = end\_y - start\_y  
  
 return average\_width, average\_length, columns, rows, cropped\_width, cropped\_height  
  
  
def process\_markers(corners, ids, desired\_order):  
 ordered\_corners = np.zeros((len(desired\_order), 2), dtype=np.float32)  
 ids = ids.flatten()  
  
 # searches from ids and adds aruco-coordinate to ordered\_corners array  
 for i, marker\_id in enumerate(desired\_order):  
 if marker\_id in ids:  
 index = np.where(ids == marker\_id)[0][0]  
 top\_left = corners[index][0][0]  
 ordered\_corners[i] = top\_left  
 return ordered\_corners  
  
# Callback when MQTT message is received  
def on\_message(client, userdata, msg):  
 global esp32\_ready  
 message = msg.payload.decode("utf-8")  
 if message == "ready":  
 esp32\_ready = True  
 print("ESP32 is ready and connected!")  
  
# Callback when MQTT client is connected to broker  
def on\_connect(client, userdata, flags, rc):  
 print(f"Connected to MQTT broker with result code {rc}")  
 client.subscribe(status\_topic) # Subscribe to the status topic  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
 client = mqtt.Client()  
 client.on\_connect = on\_connect  
 client.on\_message = on\_message  
 client.connect(broker, port)  
  
 client.loop\_start()  
  
 while not esp32\_ready:  
 time.sleep(0.1)  
 print("hello by adele")  
  
 # Camera matrix and distortion coefficients  
 camera\_matrix = np.array([[650.07504615, 0, 624.17607378],  
 [0, 650.3368611, 366.05976318],  
 [0, 0, 1]])  
  
 dist\_coeffs = np.array([[-0.39029839, 0.188932, 0.00151121, -0.00146645, -0.04711856]])  
  
 old\_obstacle\_array = np.array([0])  
  
 HISTORY\_LENGTH = 5  
 history\_buffer = deque(maxlen=HISTORY\_LENGTH)  
  
 cap = cv2.VideoCapture(0)  
 cap.set(cv2.CAP\_PROP\_FRAME\_WIDTH, 1680)  
 cap.set(cv2.CAP\_PROP\_FRAME\_HEIGHT, 945)  
 #Queues for multiprocessing  
 queue1 = multiprocessing.Queue()  
 queue2 = multiprocessing.Queue()  
 queue3 = multiprocessing.Queue()  
  
  
 #index process p1  
 p1 = multiprocessing.Process(target=pathfinding\_algorithm, args=(queue1, queue2), daemon=True)  
 p2 = multiprocessing.Process(target=control\_algorithm, args=(queue3,), daemon=True)  
  
  
 p1.start()  
 p2.start()  
  
  
 time.sleep(1)  
 prev\_frame\_time = 0  
 while True:  
 new\_frame\_time = time.time()  
  
 # grab frame  
 ret, distorteted\_frame = cap.read()  
  
 if not ret:  
 print("Failed to grab frame")  
 break  
  
 # Undistort the frame  
 undistorted\_frame = cv2.undistort(distorteted\_frame, camera\_matrix, dist\_coeffs)  
 gray\_undistored\_frame = cv2.cvtColor(undistorted\_frame, cv2.COLOR\_BGR2GRAY)  
  
 # load in aruco markers  
 aruco\_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT\_6X6\_250)  
 parameters = cv2.aruco.DetectorParameters()  
  
 # detect aruco markers  
 detector = cv2.aruco.ArucoDetector(aruco\_dict, parameters)  
 corners, ids, rejected = detector.detectMarkers(gray\_undistored\_frame)  
  
 if ids is not None and not CORNERS:  
 # process aruco detected aruco codes into correct order  
 ordered\_corners = process\_markers(corners, ids, [60, 65, 64, 62])  
 average\_width, average\_length, coloumns, rows, cropped\_width, cropped\_height = rectangle(ordered\_corners, undistorted\_frame)  
 new\_points = np.float32([[0, average\_width], [0, 0], [average\_length, 0], [average\_length, average\_width]])  
 CORNERS = True  
  
 cropped\_frame = perspective\_warp(ordered\_corners, new\_points, undistorted\_frame, average\_length, average\_width)  
  
  
 #send information into queue1  
 obstacle\_array, true\_robot\_corners = object\_detection(cropped\_frame, coloumns, rows, history\_buffer)  
 #create path array  
 queue1.put(obstacle\_array)  
  
 try:  
 message, data = queue2.get\_nowait()  
 if message == 'path':  
 #print("Path found!")  
 data = insert\_midpoints(data)  
 path\_list = spline\_line(data)  
 elif message == 'no\_path':  
 #print("No path available.")  
 path\_list = None  
 elif message == 'cleared':  
 #print("Path cleared.")  
 path\_list = None  
  
 except Empty:  
 pass  
  
  
 queue3.put((path\_list, true\_robot\_corners))  
  
  
 fps = 1 / (new\_frame\_time - prev\_frame\_time) if (new\_frame\_time - prev\_frame\_time) > 0 else 0  
 prev\_frame\_time = new\_frame\_time  
 cv2.putText(cropped\_frame, f'FPS: {fps:.2f}', (10, 30), cv2.FONT\_HERSHEY\_SIMPLEX,  
 1, (0, 255, 0), 2, cv2.LINE\_AA)  
  
 if path\_list is None:  
 cv2.imshow("frame", cropped\_frame)  
 else:  
 for (x,y) in path\_list:  
 cv2.circle(cropped\_frame, (x,y), 5, (255,0,0), -1)  
 cv2.imshow("frame", cropped\_frame)  
  
  
 # Exit on pressing the 'q' key  
 if cv2.waitKey(1) & 0xFF == ord('q'):  
 break  
  
 # Release the capture object and close any open windows after the loop ends  
 cap.release()  
 p1.terminate()  
 p2.terminate()  
 cv2.destroyAllWindows()

Appendix C: Arduino Code

#include <Arduino.h>

#include <WiFi.h>

#include <PubSubClient.h>

#include <ArduinoJson.h>

// Motor control pins

#define MOTOR\_BL\_1 19

#define MOTOR\_BL\_2 21

#define MOTOR\_FL\_1 22

#define MOTOR\_FL\_2 23

#define MOTOR\_FR\_1 18

#define MOTOR\_FR\_2 17

#define MOTOR\_BR\_1 4

#define MOTOR\_BR\_2 16

// Wi-Fi credentials

const char\* ssid = "";

const char\* password = "";

// MQTT Broker

const char\* mqtt\_server = "";

const int mqtt\_port = 1883;

// Topics

const char\* status\_topic = "esp32/status";

const char\* control\_topic = "esp32/motors";

WiFiClient espClient;

PubSubClient client(espClient);

// WiFi setup

void setup\_wifi() {

  delay(10);

  Serial.begin(115200);

  Serial.println();

  Serial.print("Connecting to WiFi...");

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL\_CONNECTED) {

    delay(500);

    Serial.print(".");

  }

  Serial.println("Connected to WiFi");

}

// MQTT callback

void callback(char\* topic, byte\* payload, unsigned int length) {

  StaticJsonDocument<200> doc;

  DeserializationError error = deserializeJson(doc, payload, length);

  if (error) {

    Serial.print("JSON Parse Failed: ");

    Serial.println(error.c\_str());  //

    return;

  }

  int left\_pwm = doc["left"];

  int right\_pwm = doc["right"];

  Serial.print("Received left: ");

  Serial.print(left\_pwm);

  Serial.print(", right: ");

  Serial.println(right\_pwm);

  if (left\_pwm >= 0) {

  analogWrite(MOTOR\_FL\_1, 0);

  analogWrite(MOTOR\_FL\_2, left\_pwm);

  analogWrite(MOTOR\_BL\_1, 0);

  analogWrite(MOTOR\_BL\_2, left\_pwm);

} else {

  analogWrite(MOTOR\_FL\_1, -left\_pwm);

  analogWrite(MOTOR\_FL\_2, 0);

  analogWrite(MOTOR\_BL\_1, -left\_pwm);

  analogWrite(MOTOR\_BL\_2, 0);

}

// Right side motors (FR & BR)

if (right\_pwm >= 0) {

  analogWrite(MOTOR\_FR\_1, 0);

  analogWrite(MOTOR\_FR\_2, right\_pwm);

  analogWrite(MOTOR\_BR\_1, 0);

  analogWrite(MOTOR\_BR\_2, right\_pwm);

} else {

  analogWrite(MOTOR\_FR\_1, -right\_pwm);

  analogWrite(MOTOR\_FR\_2, 0);

  analogWrite(MOTOR\_BR\_1, -right\_pwm);

  analogWrite(MOTOR\_BR\_2, 0);

}

}

// MQTT reconnect

void reconnect() {

  while (!client.connected()) {

    Serial.print("Attempting MQTT connection...");

    if (client.connect("ESP32Client")) {

      Serial.println("connected");

      client.publish(status\_topic, "ready");

      client.subscribe(control\_topic);

    } else {

      Serial.print("failed, rc=");

      Serial.print(client.state());

      Serial.println(" try again in 5 seconds");

      delay(5000);

    }

  }

}

// Arduino setup

void setup() {

  setup\_wifi();

  client.setServer(mqtt\_server, mqtt\_port);

  client.setCallback(callback);  //

  // Motor pins

  pinMode(MOTOR\_FL\_1, OUTPUT); pinMode(MOTOR\_FL\_2, OUTPUT);

  pinMode(MOTOR\_BL\_1, OUTPUT); pinMode(MOTOR\_BL\_2, OUTPUT);

  pinMode(MOTOR\_FR\_1, OUTPUT); pinMode(MOTOR\_FR\_2, OUTPUT);

  pinMode(MOTOR\_BR\_1, OUTPUT); pinMode(MOTOR\_BR\_2, OUTPUT);

}

// Main loop

void loop() {

  if (!client.connected()) {

    reconnect();

  }

  client.loop();

}