# Computer Vision & Maze Navigation

EEE Lab Report 3

Daniel Marcovecchio
20471667 Due: 02/05/2023

University of Nottingham
UK | CHINA | MALAYSIA

# Abstract

This report attempts to inform on the construction and programming of a computer vision-based line following implementation for the EEEBot. This report details the design and implementation of a symbol recognition system using further computer vision techniques and aims to give background context to this achievement. Also, this report features the whole construction and programming of a Human Machine Interface implemented on the ESP32 onboard the EEEBot, to provide a menu interface for the entry of instructions to navigate a physical maze.

# Contents

# 1. Introduction & Project Context

## 1.1 Introduction

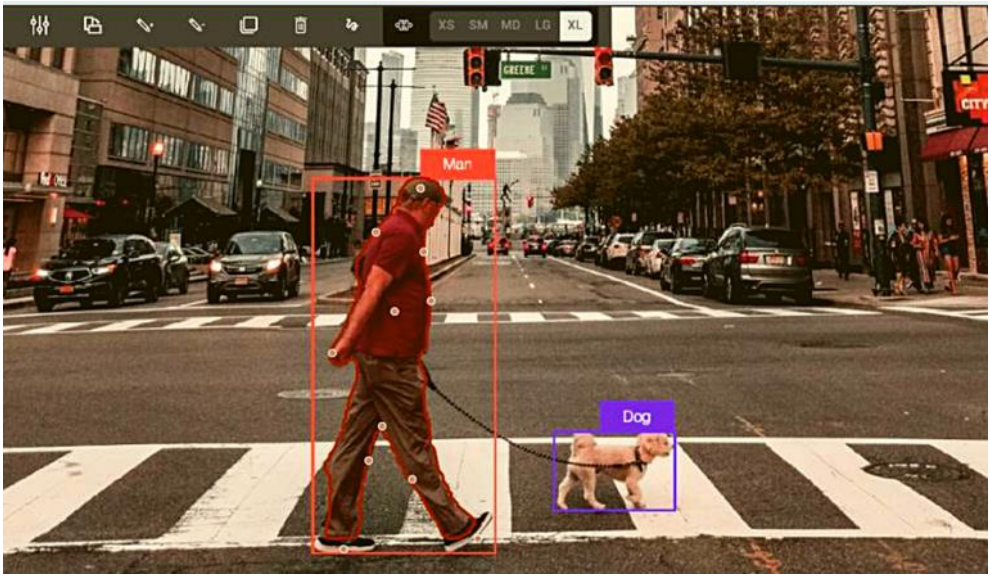
In this report, the design, construction and programming of a line following system using computer vision for the EEEBot is documented and explained. This report aims to inform on how a RGBYK line following algorithm was created and integrated with symbol recognition for decision making. Also, this report features the construction and programming of a full maze navigation system using a Human Machine Interface.

The systems documented in this report are in use in a global theatre from medical advancements to transportation via self-driving car technology [6]. An image recognition system was implemented in this report in order to realise the idea of switching the target colour of the line to follow. This was achieved by reading a purple-coloured symbol from the track as the EEEBot advances, performing many computer vision algorithms on it, and recognising it based off four symbols known to the EEEBot. Navigation of a mase required the use of a Human Machine Interface in order to input necessary commands to manoeuvre the EEEBot through a maze, whilst providing information to the User via an onboard LCD screen. Both systems feature heavily in real life applications as explained in section 1.2.

## 1.2 Project Context

### 1.2.1 Global Adoption of Image Recognition Technologies

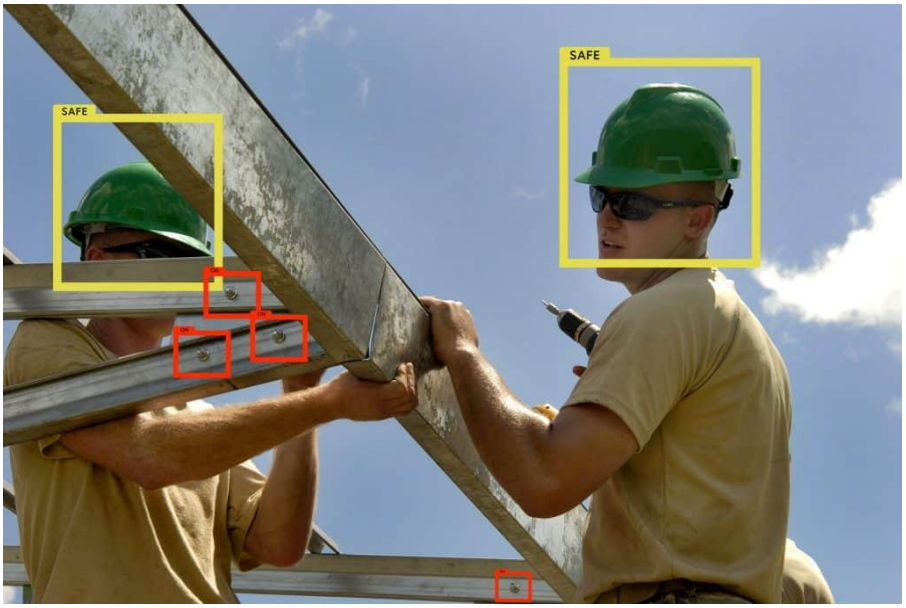
Image recognition is driven by neural networks that recognise image features to determine its likeness to others [7]. It has a very wide application in today's world in applications like Facial recognition in personal electronics, smartphones and surveillance systems to use in production lines for quality control [6]. In order to achieve a high degree of accuracy with these technologies, very large datasets of recognised, isolated, labelled images are used in training.




*Man walking a Dog detected by OpenCV [8]*

Say, for example, a construction site. In the modern-day era, health and safety is of paramount importance on construction and work sites. This is a task normally undertaken by an individual employed to ensure that the standards of health and safety are met. This can perhaps impede the

workers on the site slightly, and impact the overall length of the project, increasing costs and resources required. Computer vision using Image Recognition is a great fit for this issue. Cameras can be setup around the construction site (or existing surveillance cameras used) to monitor the construction crews and continually image them. Image recognition AI can be applied to each worker once identified from the image to identify what safety equipment is in use. For example, if the worker is missing a hard hat or a jacket, this can be flagged up quickly, and sent to a central control room or similar, and the necessary action can be taken to ensure the workers safety.




*Safety detection on workers [10]*

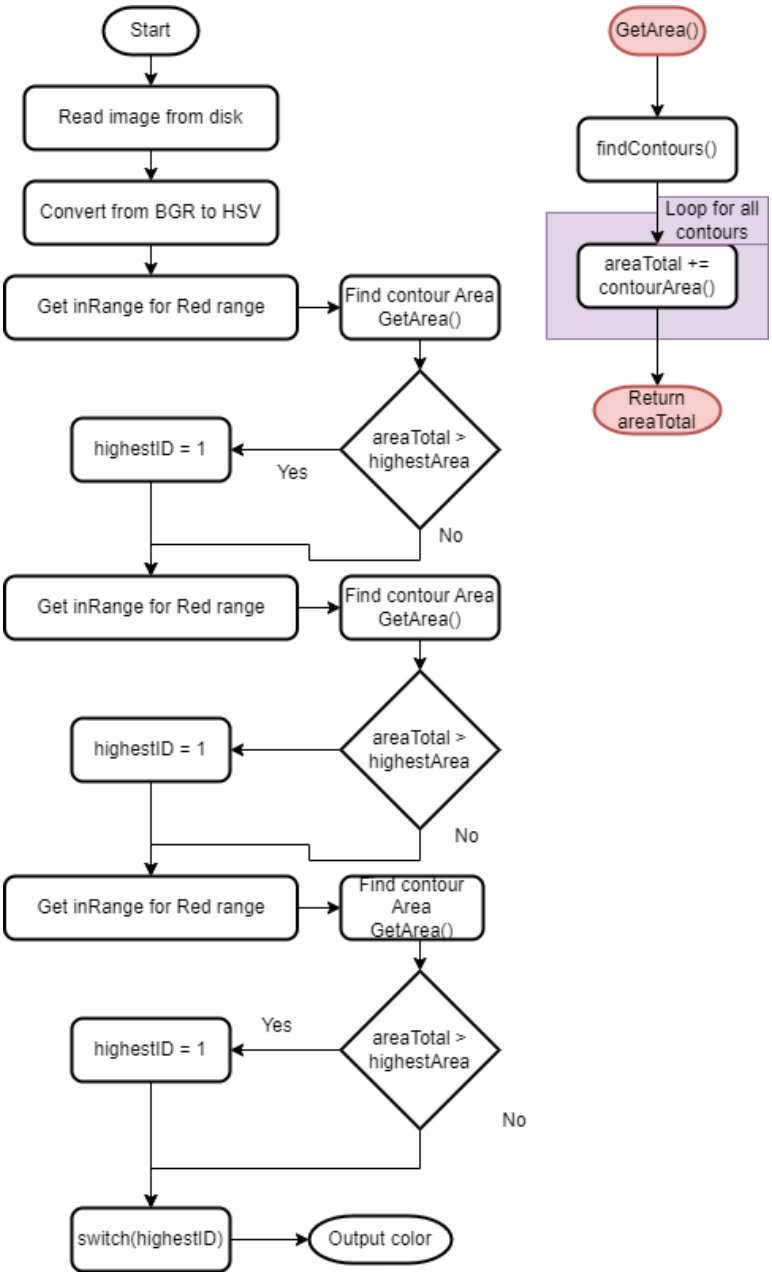
A similar use case is identifying rust and other defects in industrial sites, which without proper management could potentially lead to equipment/machinery failure and safety hazards. An AI can be trained to recognise what these faults look like and monitor a larger area and check points all of the time, rather than an inspection which may only take place daily or weekly.

### 1.2.2 Applications of Human Machine Interfaces

Human Machine Interfaces are normally known as Menu Interfaces and can be found in everywhere in modern day society. At a simple level they are implemented on personal appliances like microwaves and rice cookers; the way you press buttons on a TV remote to navigate the options on the TV; dialing a number on a landline phone/VOIP phone and interacting with the embedded LCD using the keypad to listen to voice messages. The list is pretty exhaustive.

# 2. OpenCV Introductory Task

## 2.1 Flowchart




## 2.2 Code Explanation

### 2.2.1 Reading from Disk and Colour Conversion

The solution devised for the introductory project allowed all images to be processed and the object colour detected at the same time. First, the image was read from the desk and stored in a matrix. Next, colours were converted from BGR to HSV colour space. This was necessary in order to perform the inRange function, as it must operate using the HSV range values that were selected for each colour.

```
Mat RedApple = imread("RedApple.bmp");

Mat RedAppleHSV;
```

```
cvtColor(RedApple, RedAppleHSV, COLOR_BGR2HSV);
```

### 2.2.2 The MostProminant() Function

Next, a function was created, named MostProminant() which was designed to calculate the most prominent colour in each picture. It took two inputs, the name of the image (to use when outputting), and the image itself. Next, the HSV ranges were defined as integer arrays, with the data worked out using the HSV tool provided.

```
int MostProminant(std::string name, Mat img) {
    int redRange[] = {0, 15, 168, 255, 91, 255};
    int greenRange[] = {30, 90, 20, 255, 45, 255};
    int blueRange[] = {95, 140, 80, 255, 60, 255};
```

In order to work out the prominence of each colour in the picture, a helper function was created called GetArea(), which would return the total area that the colour occupied. First, prior to the function call, inRange() was called for that range to check in order to isolate the object using the HSV range. The GetArea function would find all of the contours in the image using findContours(), and then loop for all of these contours, whilst calling contourArea(), to sum all of the contour areas.

```
inRange(img, Scalar(redRange[0], redRange[2], redRange[4]),
Scalar(redRange[1], redRange[3], redRange[5]), imgNew);
float area = GetArea(imgNew);
```

```
float GetArea(Mat img) {
    std::vector<std::vector<Point> > contours;
    findContours(img, contours, RETR_LIST, CHAIN_APPROX_SIMPLE);
    float totalArea = 0;
    for(int i=0; i<contours.size(); i++)
    {
        totalArea += contourArea(contours[i]);
```

This function call was repeated for each HSV range, and a series of if statements were used to store the highest value of the GetArea() and the respective colour of this.

```
if(area > maxArea) {
        maxArea = area;
        maxID = 1;
}
--------Repeats
```

Finally, a switch function was used to output the colour found for the image.

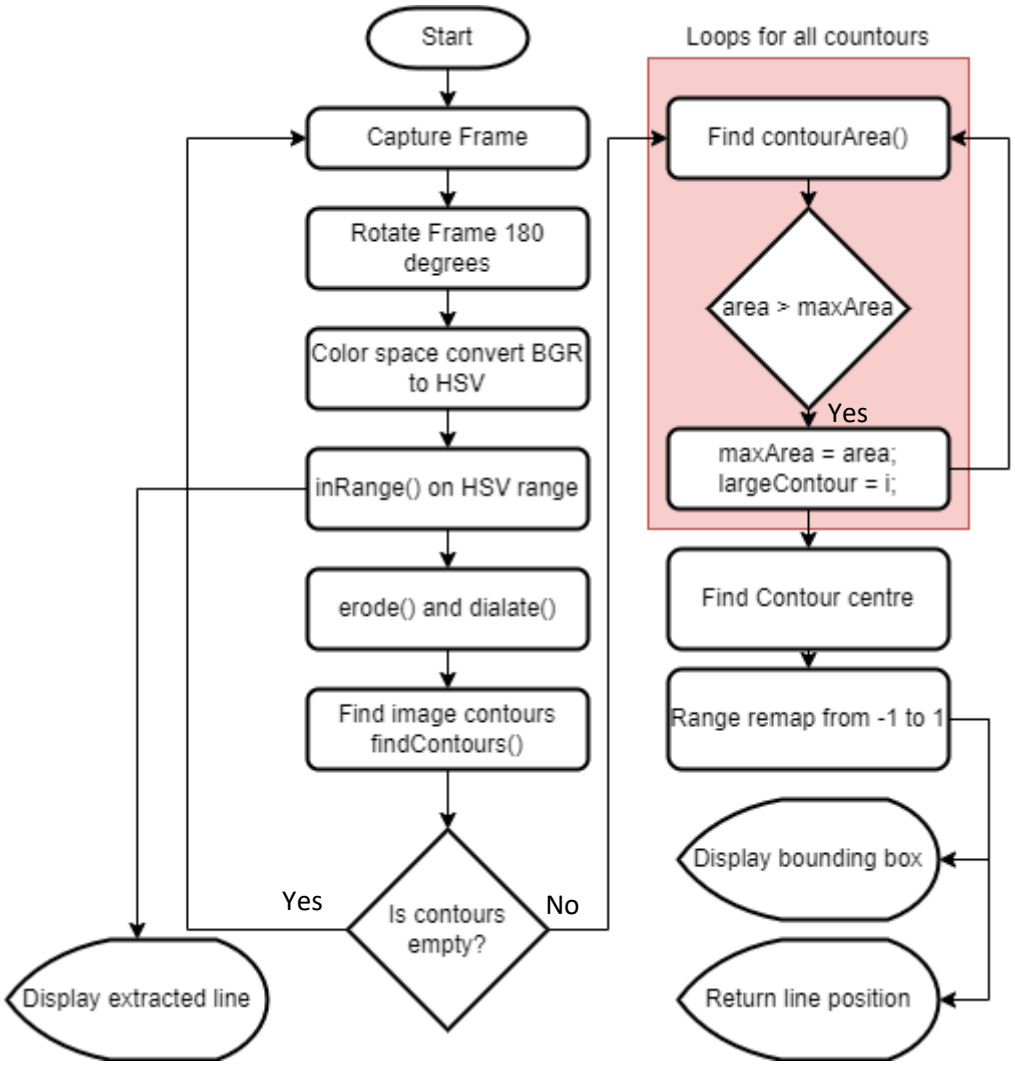
```
switch(maxID) {
        case 1: {
            std::cout << name << " is red!\n";
```

```
GreenApple is green!
GreenCar is green!
BlueApple is blue!
BlueCar is blue!
```

It should be noted that it was not possible to correctly work out the colour of the red pictures using this approach, due to there being more green in the red pictures than red due to the vegetation.

# 3. Line Following Approach with Computer Vision

This section of this report aims to cover how a Line Following solution was achieved using a camera onboard the EEEBot and the implementation of computer vision algorithms to process the image data and work out the line position. Then, as we have seen in Lab Report 2, line following logic had to be redesigned in order to consider multiple line colours, in combination with the Symbol Recognition system that is covered later.




## 3.1 Imaging of Black and Coloured Lines

### 3.1.1 The OpenCV approach

Code was written for the Raspberry Pi that would enable the camera to image as soon as it could, effectively taking a video, with OpenCV operating on every “frame” of this video. In the remainder of the report, the “frame” refers to the current image that is being processed using OpenCV. In order to create each frame, a loop was running in the main() method, which would continually capture a

frame using the captureFrame() method, provided in the OpenCV functions file. This was stored into a global variable called currentCapture.

```
Mat currentCapture;
```

```
while(currentCapture.empty())
	currentCapture = captureFrame();
```

### 3.1.2 The GetLinePosition() Function

In order to achieve the line position, it was first necessary to isolate the line that was to be followed, knowing its colour. It was expected that the EEEBot would encounter situations were multiple colours of lines would be visible in the camera frame, meaning that this system had to be reliable. It was given that the colours of the lines could have been: black, red, green, blue or yellow. Earlier in this report, it was shown how knowing the HSV range of the object could be used to extract it from an image/background. A similar approach was taken here, however it was now necessary to allow the input to vary every frame, depending on the colour of the line to follow.

First, a function was created called GetLinePosition() which accepted an integer array called hsvArr (HSV Array). The HSV Array was the range of HSV values to allow the line to be extracted from the background. It was 6 elements long; H – Low, H – High, S – Low, S – High, V – Low, V – High. The values for each colour were defined in the main method function, making up 6 total arrays (inclusive of the purple colour used for symbol recognition).

```
int redRanges[] = {340, 20, 60, 100, 60, 100};
int blueRanges[] = {190, 260, 60, 100, 60, 100};
int greenRanges[] = {58, 85, 101, 255, 0, 156};
int yellowRanges[] = {46, 83, 26, 255, 0, 161};
int blackRanges[] = {0, 179, 0, 0, 0, 55};
int purpleRanges[] = {117, 170, 23, 255, 0, 255};
```

Code was then written in the GetLineFunction which would copy the currentCapture frame to a new frame variable for operation on without disrupting the original frame. This was then rotated 180 degrees, so that the image was the right way up, as the camera was mounted upside down. Next, code was written to convert the colour of the frame from a BGR colour space to a HSV colour space, which allowed the operation using the HSV ranges. This was achieved with the OpenCV cvtColor() function. Then, the OpenCV function inRange() was used to extract the line object from the image using the range inputted into the function. This was saved to a new Matrix called finalImage. After this was achieved, the finalImage frame was outputted in order to confirm that the line had been correctly extracted.

```
void GetLinePosition(int hsvArr[])
{

    Mat frame;

    frame = currentCapture;

    cv::rotate(frame, frame, cv::ROTATE_180);
```

```
    Mat gray;
    Mat finalImage;
    Mat object;
    cv::cvtColor(frame, object, COLOR_BGR2HSV);
    cv::inRange(object, cv::Scalar(hsvArr[0], hsvArr[2], hsvArr[4]),
cv::Scalar(hsvArr[1], hsvArr[3], hsvArr[5]), finalImage);
```

During testing, it was noted that there was a lot of visible noise in the image of the extracted line (now a binary image). It was necessary to clean this noise up as to decrease the contour processing time (discussed in the next section), and to increase the reliability. To achieve this, a series of erode() and dilate() functions operated on the image, known as morphological transformation [1].

```
erode(finalImage, finalImage, cv::getStructuringElement(MORPH_ELLIPSE, Size(5, 5)));
dilate(finalImage, finalImage, cv::getStructuringElement(MORPH_ELLIPSE, Size(5, 5)));
dilate(finalImage, finalImage, cv::getStructuringElement(MORPH_ELLIPSE, Size(5, 5)));
erode(finalImage, finalImage, cv::getStructuringElement(MORPH_ELLIPSE, Size(5, 5)));
```

The output of this process is a binary image representing the extracted line.

## 3.2 Image Contours & Line Position

In order to achieve the goal of calculating the line position, the position of the extracted line object needed to be found in the frame, to determine how far right or left the line was in comparison to the robot. The most direct solution to achieving this was to use OpenCV's contouring system, which OpenCV describes as being a "curve joining all the continuous points (along a boundary), having same colour or intensity" [2]. They allow to detect shapes more easily (and useful in Symbol Recognition as covered later).

### 3.2.1 Finding the Image Contours

In order to find the image contours, the OpenCV findContours() function had to be implemented. This function takes an input matrix, and an array of arrays of points which was named contours. This array, after the function call was made, contained all of the contours in the image.

```
Mat contourOut = finalImage.clone();
std::vector<std::vector<cv::Point>> contours;
cv::findContours(contourOut, contours, RETR_LIST, CHAIN_APPROX_NONE);
```

### 3.2.2 Finding the Largest Contour

Providing that the line was extracted correctly using the HSV range, the largest object in the image of the inputted colour should have been the corresponding extracted line, due to all other colours being isolated, and the background of the track being white. This meant that the largest contour by area in the image would have been the area that the line occupied. It was a possibility that the there were other contours detected, not just the line, so code was designed to filter the largest contour detected, and therefore identify the line.

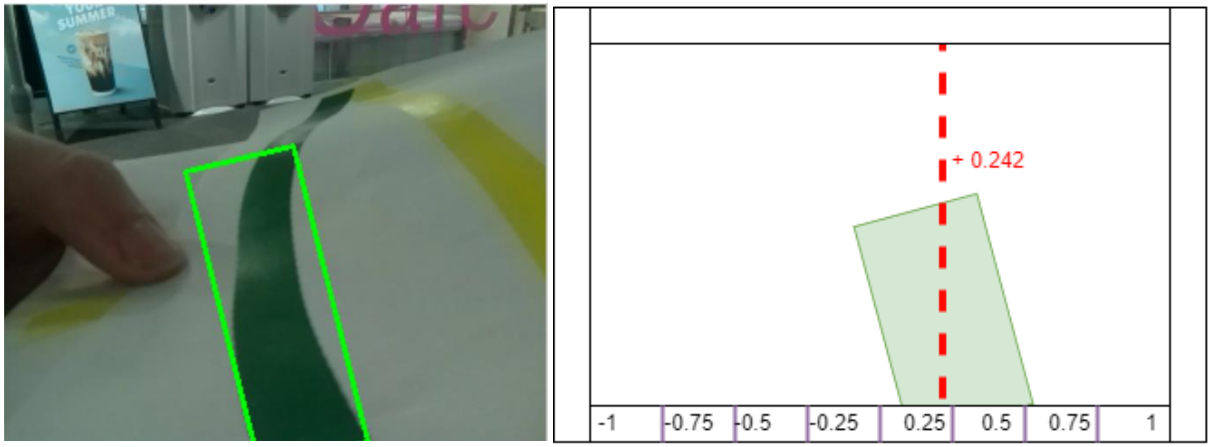
First, an if statement checked to see if any contours were detected at all, otherwise the rest of the code was skipped to increase performance for that frame. A process for finding the largest contour was then designed, which looped for each contour and the area of each contour calculated, and recorded if it was the highest. First, two variables were defined, being maxArea and largeContour, which respectively represented the largest area encountered and the array element ID of the corresponding contour. To achieve the loop, a for loop was implemented, which looped from 0 to the contours size. In order to calculate the area, an OpenCV contourArea function was used, which was type cast to a float and stored in the variable area. Finally, an if statement checked if the area calculated was greater than the current highest area (maxArea). If true, this would assign the largeContour and maxArea variables.

```
float maxArea = 0;
int largeContour = 0;
for(int i = 0; i < contours.size(); i++)
{
	cv::drawContours(contourImage, contours, i, color);
	float area = (float)cv::contourArea(contours[i]);
	if(area > maxArea)
	{
		largeContour = i;
		maxArea = area;
	}
```

The contours in the image look are displayed.

### 3.2.3 Calculating the Line Position

As discussed in the previous report, a weighted average determined how far left or right of the robot that the line was. This is equivalent to the line position, which could be calculated from the screen space position of the line in the frame. This is the X position of the centre of the line in the frame. Similar to the weighted average, the line position is most useful in the range of -1 to 1, where -1 is the line being the furthest to the left of the robot and vice versa for 1.

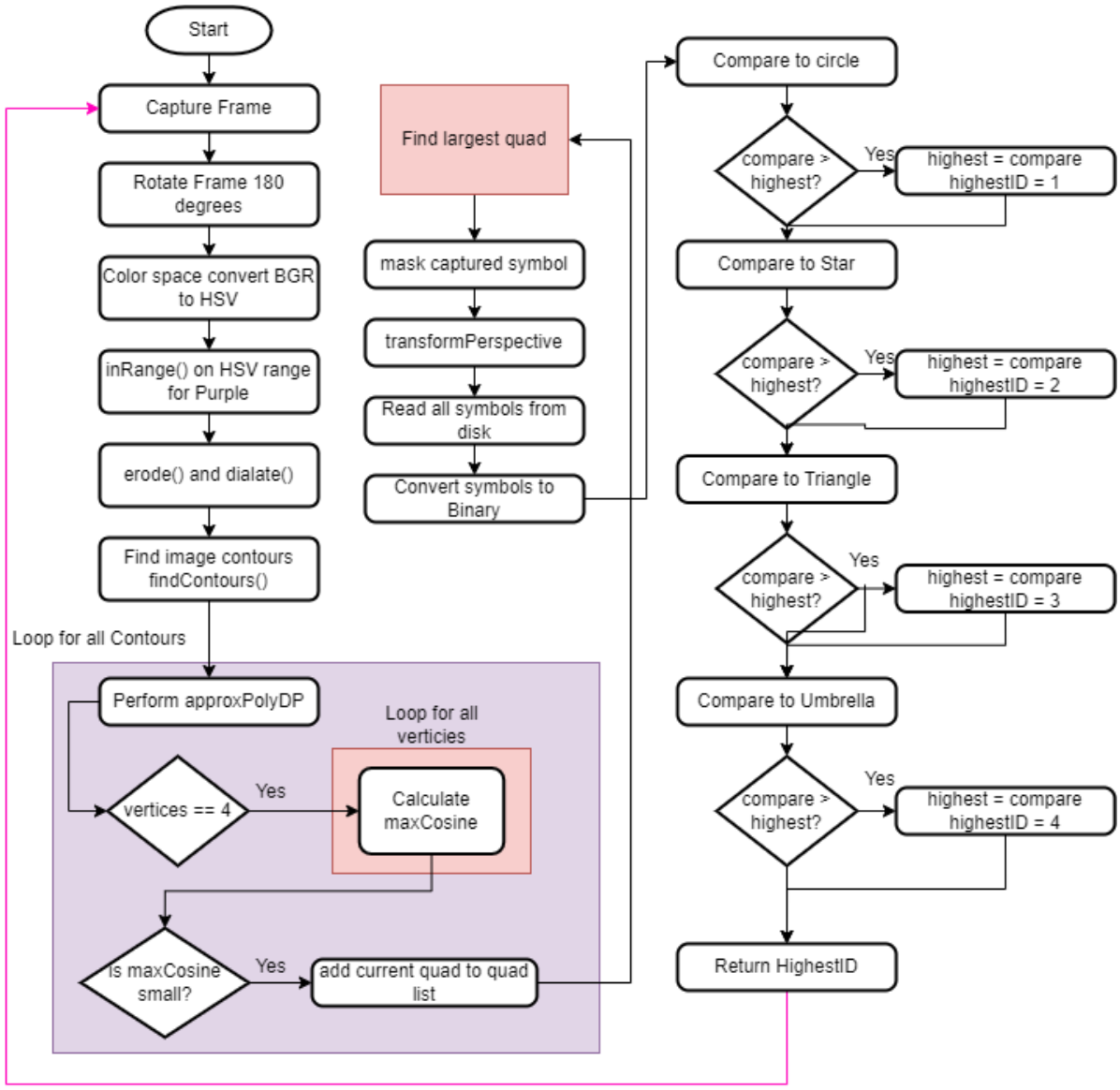



With the largest contour found, representing the current line, it was now necessary to find the centre position of this contour. In order to achieve this, a rotated bounding box had to be drawn around the contour. This was done using the OpenCV RotatedRect object. The minAreaRect function was used on the largest contour to return the smallest (by area) rectangle for the contour. Then, the centre of the contour was found using the given function findContourCentre on the largest contour. This returned a point which represented the contour centre in the frame. The X position of this point was then stored as another variable pointXPos. Finally, to calculate the Line Position, the range had to be remapped from the 360 wide (-160 to 160) frame size, to the -1 to 1 needed for the Line following code. This was achieved using a Map Function.

```
cv::RotatedRect bounding_rect = cv::minAreaRect(contours[largeContour]);
cv::Point contourCenter = findContourCentre(contours[largeContour]);
pointXPos = contourCenter.x;
//---
linePosition = MapFunction(pointXPos - 160, -160, 160, -1, 1);
```

# 4. Symbol Recognition Approach

This section of the report attempts to explain how symbol recognition for the 4 symbols (being Circle, Star, Triangle and Umbrella) was implemented successfully. Parts of this section will appear to be like the line following approach due to some methods used being the same.




## 4.1 Extracting the Symbol from the Image

### 4.1.1 The LookForSymbol Function

Each symbol was known to be coloured purple, so a HSV range was selected and a HSV range array was created for this. A function, LookForSymbol, was created which would return an integer representing the ID of the symbol found (1 = Circle, 2 = Star, 3 = Triangle, 4 = Umbrella) and 0 if no symbol was found. This function, similar to the GetLinePosition function, accepted an integer array for the HSV range.

```
int purpleRanges[] = {117, 170, 23, 255, 0, 255};
```

```
int LookForSymbol(int hsvArr[])
```

### 4.1.2 Symbol Isolation & Contour Finding

First, the frame image was rotated 180 degrees as to be the right way up again, due to the camera being mounted upside down. Then, the colour space was converted from BGR2HSV, using the OpenCV cvtColor() method covered previously. Then, the OpenCV inRange() function was used to filter the purple HSV range from the image, and store this result as a binary image in the finalImage matrix. This was necessary as to isolate just the symbol from the image, as it was required to find the symbol border.

```
cv::rotate(frame, frame, cv::ROTATE_180);
Mat gray;
Mat finalImage;
Mat object;
cv::cvtColor(frame, object, COLOR_BGR2HSV);
cv::inRange(object, cv::Scalar(hsvArr[0], hsvArr[2], hsvArr[4]),
cv::Scalar(hsvArr[1], hsvArr[3], hsvArr[5]), finalImage);
```

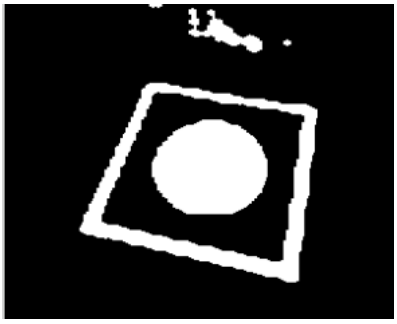
Then, the same erode and dilate functions [1] were applied to the image in order to eliminate any noise in the image. This was necessary as small bits of noise could have caused issues with the quad finding algorithm covered later, which could have made finding the border of the symbol harder.

```
erode(finalImage, finalImage, cv::getStructuringElement(MORPH_ELLIPSE, Size(5, 5)));
dilate(finalImage, finalImage, cv::getStructuringElement(MORPH_ELLIPSE, Size(5, 5)));
dilate(finalImage, finalImage, cv::getStructuringElement(MORPH_ELLIPSE, Size(5, 5)));
erode(finalImage, finalImage, cv::getStructuringElement(MORPH_ELLIPSE, Size(5, 5)));
```

Then, the OpenCV findContours() function was used to find and store the image contours into a vector of vectors of points called contours. This was done as to get the contour for the border of the symbol, in order to transform the symbol later.

```
Mat contourOut = finalImage.clone();
std::vector<std::vector<cv::Point>> contours;
cv::findContours(contourOut, contours, RETR_LIST, CHAIN_APPROX_SIMPLE);
```

### 4.1.3 Output of Symbol Isolation



## 4.2 Identifying the Symbol Border using Contour Approximation

### 4.2.1 Using approxPolyDP()

The Symbol Border for each symbol was a square, otherwise known as a quad. It was possible to detect quads in the frame by using contour approximation. Contour approximation takes a contour and reduces the number of vertices of the contour, whilst retaining the shape using a threshold value [4]. In order to achieve this, OpenCV's approxPolyDP function was used, which returns a

simplified contour based on the threshold value provided. The approxPolyDP() function was called for each contour in a for loop, using the arc length of the contour multiplied by 0.02 as the epsilion (threshold) value. If the returned approximated contour number of points was equal to 4, then a quad had been found, as quads have four vertices.

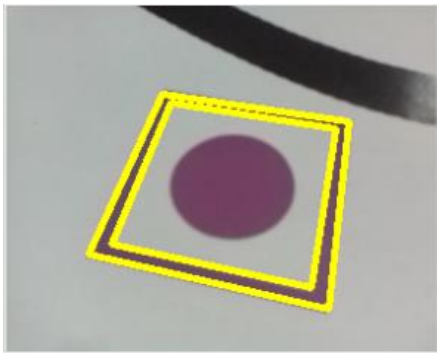
```
//Find all quads in the image
std::vector<std::vector<cv::Point>> quads;
cv::Mat new_final = frame.clone();
std::vector<cv::Point> point;
for(int i = 0; i < contours.size(); i++)
{

    cv::approxPolyDP(contours[i], point, cv::arcLength(contours[i], true) *
0.02, true);
    if(point.size() == 4 && cv::isContourConvex(point)) {
```

### 4.2.2 Joint Angle Analysis

The approximated contour was also checked to see if it was convex using the isContourConvex() method, as squares are convex polygons [5]. Then, the "maximum cosine of the angle between joint edges" [3] is computed by looping for all joint edges, calculating the angle between them (using an angle helper function credit of OpenCV documentation [3]), taking the absolute value of this angle and then finding the maximum out of all of the joint edges. If these cosines of these angles are very small, then the joints form a 90 degree angle with each other. This step is necessary as there were 4 sided polygons that were not square in shape, and so these had to be filtered out. A small cosine result means a quad has been confirmed, and this quad can be added to an array of quads.

```
double maxCosine = 0;

for(int j=2; j<5; j++)
{
      double cosine = std::fabs(angle(point[j%4], point[j-2], point[j-1]));
      maxCosine = MAX(maxCosine, cosine);
}

if(maxCosine < 0.3)
{
      quads.push_back(point);
      cv::drawContours(new_final, quads, -1, cv::Scalar(0, 255, 255), 3);
}
```

The detected quads were drawn onto an image and displayed.



### 4.2.3 Selecting the Largest Quad

The same method to select the largest contour by area mentioned in section 3.3.2 was used again to select the largest quad.

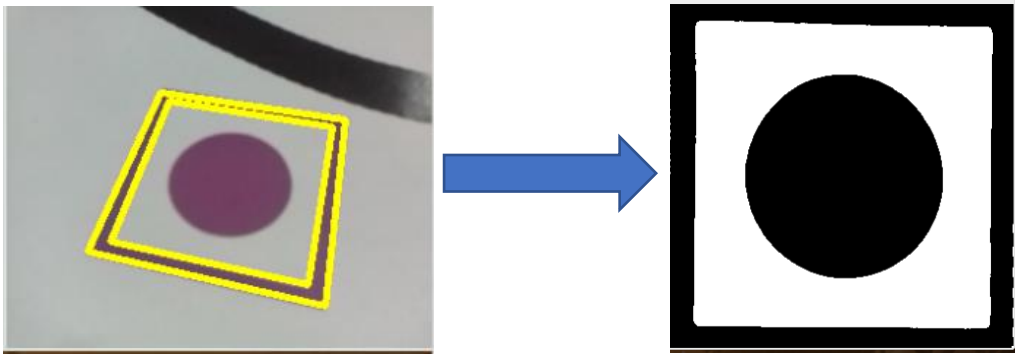
## 4.3 Symbol Comparison and Identification

### 4.3.1 Symbol Transformation

It was necessary to extract the symbol from the image, using a mask in order to transform it first. First, an if statement checked that a quad was detected, as to avoid trying to index an empty array when transforming the symbol. This was achieved by creating a mask from the largest quad by using the drawContours() method onto the mask. Then, this mask was applied to the original image. After, the provided transformPerspective function was used to transform the contour from the perspective to a flat image. This was essential to do as the captured symbol image would be compared to the four symbol images in the library, and this could only be done if they were both same perspectives, otherwise they would not match at all. Lastly, the captured symbol was converted to a binary image.

```
if(!quads.empty())
{
	cv::Mat mask(frame.size(), frame.type(), cv::Scalar(0));
	cv::drawContours(mask, quads, largeContour, Scalar(255, 255, 255), FILLED);
	cv::Mat symbolRaw;
	frame.copyTo(symbolRaw, mask);

	Mat symbolTransformed = transformPerspective(quads[largeContour],
symbolRaw, 320, 240);
	cv::cvtColor(symbolTransformed, symbolTransformed, cv::COLOR_BGR2GRAY);
	 cv::threshold(symbolTransformed, symbolTransformed, 128, 255,
cv::THRESH_BINARY | cv::THRESH_OTSU);
```

The output of this result transformed the image from the perspective of the robot to a flat image to the camera.



### 4.3.2 Reading & Colour Space Conversion of the Library Images

Each symbol was read from the disk using the imread function. Then, they were converted into grayscale before being converted to binary images. This is required in order for the compare Images function to work, as both input images must be binary. Each symbol was read and converted as follows:

```
Mat circle = cv::imread("/home/pi/Desktop/OpenCV-Template/Circle.png",
cv::IMREAD_GRAYSCALE);
cv::threshold(circle, circle, 128, 255, cv::THRESH_BINARY | cv::THRESH_OTSU);
-- -- -- -- -- Code repeats for each symbol
```

### 4.3.3 Image Comparison

In order to identify which symbol was imaged, the image had to be compared to every other symbol in the library for its likeness, to identify the highest match. This was achieved using the compareImages() function which takes the imaged symbol and the library reference symbol and compares the two, and returns a percentage likeness. In order to figure out the highest, a series of if statements were built to identify the largest likeness and which symbol achieved it. Before using the compareImages() function though, a resize() function had to be called to ensure that both the imaged symbol and the library symbol were the same size.

```
cv::resize(symbolTransformed, symbolTransformed, circle.size());
float highest = 0;
int highestID = 0;
float compare = compareImages(symbolTransformed, circle);
if(compare > highest)
{
      highest = compare;
      highestID = 1;
}
compare = compareImages(symbolTransformed, star);
if(compare > highest)
-- -- -- -- -- Code repeats for each symbol
```

Lastly, the likeness value was compared to a threshold value, to ensure that a symbol had actually been detected.

```
int threshold = 60;
if(highest > threshold)
{
      return highestID;
}
```

```
Detected Symbol: Circle
Detected Symbol: Circle
```

# 5. RGBYK Line Following Solution

## 5.1 System Overview

### 5.1.1 The Raspberry Pi

In order to perform line following with computer vision, a camera had to be mounted onto the EEEBot, with the accompanying processing power in order to perform computer vision algorithms using OpenCV on this image data. To achieve this, a Raspberry Pi was used in combination with a camera, with an output resolution of 320 x 240 pixels. The Raspberry Pi was mounted onto a 3D printed mount that would enable mounting to the vertical nylon dividers onboard the EEEBot (that were used to provide structure between the mainboard and expansion board), so that it would sit above the ESP32. The camera was mounted on a slot part of the 3D printed mount, angled ~20 degrees down as to allow the camera to view the track ahead.

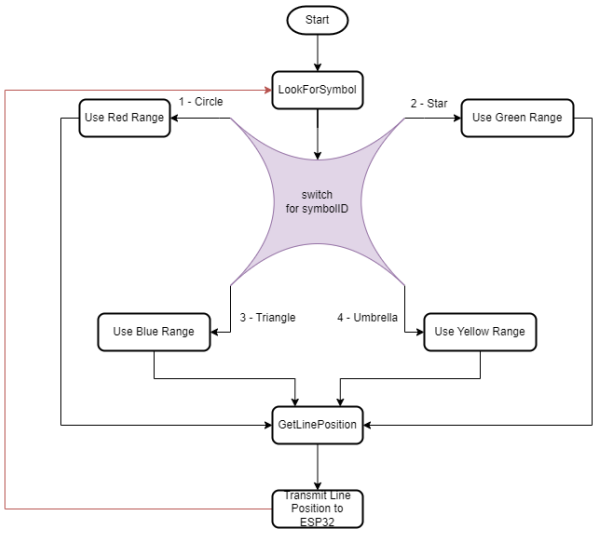
**[Insert picture of Raspberry pi mount atop EEEBot]**

### 5.1.2 I2C Connection

It was decided that the Raspberry Pi would connect to the ESP32 via I2C. This was achieved by wiring the 5V supply, Ground, SDA and SCL lines from the Raspberry Pi to the High sides of the Bi-Directional shifter. This would enable the Raspberry Pi to connect to the ESP32 (to transmit to the Arduino, similar to a repeater) if use of the other sensors (MPU6050 or HC-SR04) or enable direct connection with the Arduino in case a lower latency was required. The new I2C system would mean that the Raspberry Pi would act as a master, with the ESP32 acting as a "repeater" and the Arduino as the absolute slave. Due to software difficulties encountered (discussed later), it was only possible to transmit the line position from the Raspberry Pi to the slave, meaning that all line following logic had to be performed onboard the Raspberry Pi, which did raise some issues in development and testing.

## 5.2 Creating a United System

Earlier in this report, the creation of Symbol Recognition and Line following systems was explored, with each solution providing sensible data when tested individually. In order to achieve the full Line Following solution, it was necessary to unite these two systems with additional logic in order to aid in edge cases that the EEEBot would encounter.



### 5.2.1 Route Decision using the Detected Symbol

In order to determine the right route to take, a switch statement would operate on the returned symbol ID value returned from the LookForSymbol found. The corresponding HSV range would then be passed into the GetLinePosition function, switching the colour of the line being tracked. This would cause the EEEBot to change direction to follow the colour corresponding to the symbol.

```
      int ID = (int) LookForSymbol(purpleRanges);

        switch (ID){

            case 1: {
                std::cout << "Detected Symbol: Circle\n";
                GetLinePosition(redRanges);
                break;
            }
            case 2: {
                std::cout << "Detected Symbol: Star\n";
                GetLinePosition(blueRanges);
```

### 5.2.2 Transmitting the Line Position to the ESP32/Arduino

It was then necessary to transmit the weighted average to the EEEBot. At first, an attempt was made to perform all line following processing on board the raspberry pi, including the PID, steering and motor power controls, and transmit this information to the Arduino as a single large transmission. This was originally attempted by putting all of these variables into a char array containing all the data and using the Pi2C library [9] to transmit. However, it was discovered that this would result in no data being received at the Arduino, for reasons still unknown.

Instead, it was decided to keep the line following software (PID controller, wheel and steering control) onboard the ESP32/Arduino, and just transmit the Line Position. Furthermore, this could be expanded upon to use the unused out of range numbers (< -1 or > 1) to transmit status codes for edge case handling using the original optical line follower solution running on the ESP32/Arduino. To achieve this, the line position was made a public variable, and a transmit function was constructed to transmit every frame.

```
Pi2c car(4);
```

```
void Transmit()
{
    car.i2cWriteArduinoInt(linePosition);
}
```

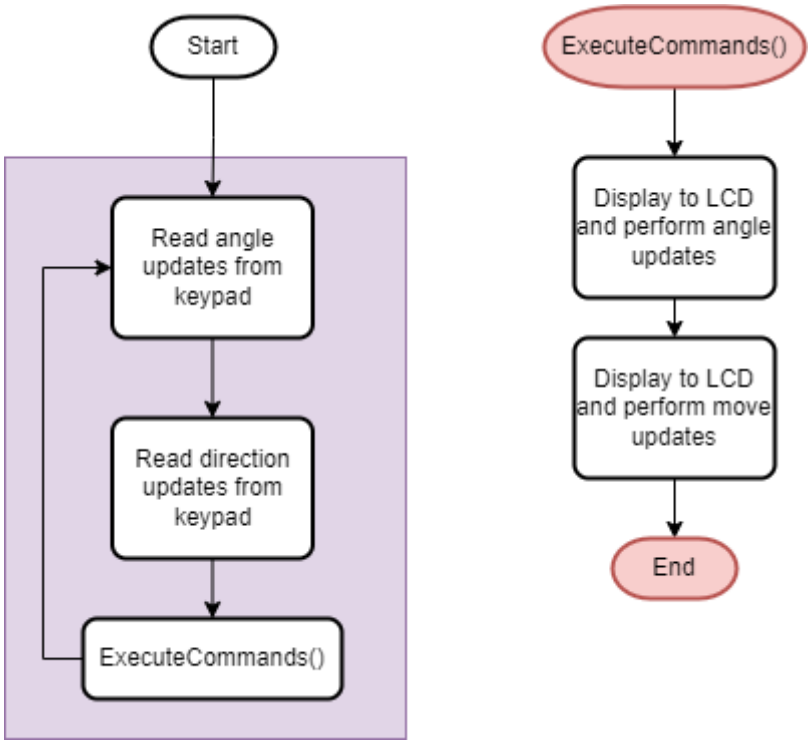
```
currentCapture.release();

int key = cv::waitKey(1);   // Wait 1ms for a keypress (required to update
windows)

Transmit();
```

# 6. Maze Navigation Approach & Implementation

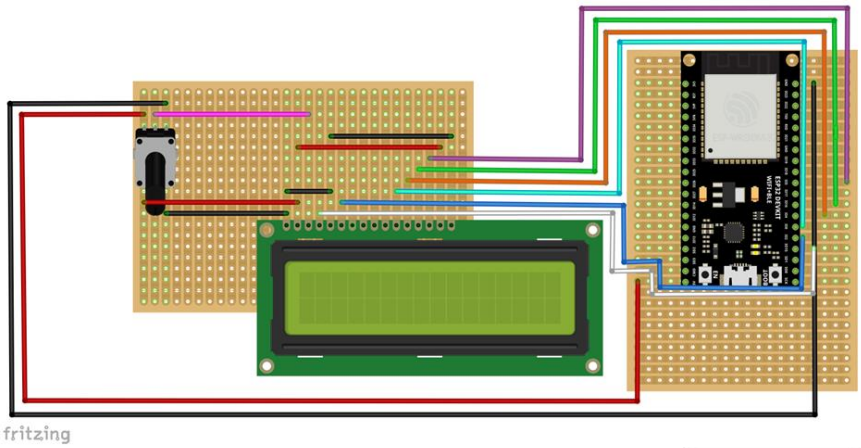
## 6.1 Section Overview

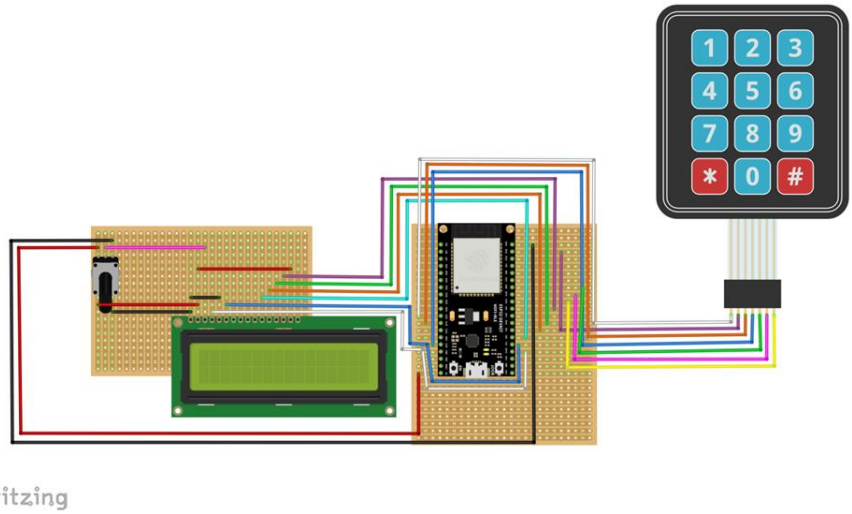
A human machine interface was required in order to enter the instructions to enable the EEEBot to navigate around the maze. This consisted of a keypad that would be used to enter the instructions and an LCD screen that would display the choice inputted, as well as the current instruction being executed forming the basis of a menu. These two systems together create the Human Machine Interface.

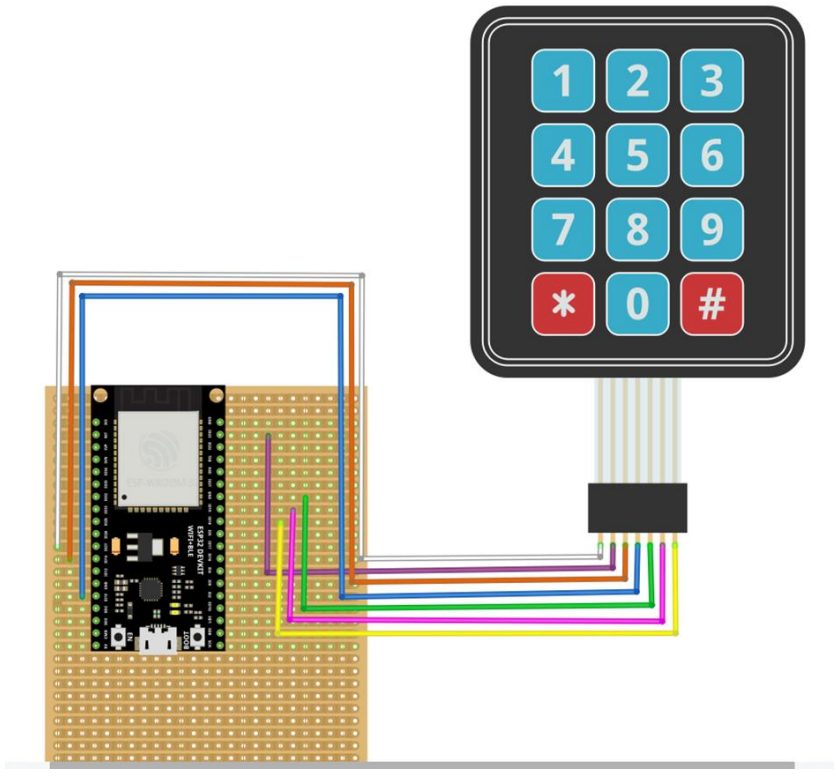
### 6.1.1 Flowchart



### 6.1.2 Wiring and Connection of the LCD and Keypad



## 6.2 Keypad Entry for Angle and Direction Movements

### 6.2.1 Angle Array Entry

First, a method called AngleArray() was created, which had parameters the current instruction "I" and the integer array buffer[] to store it to. Then, the user was prompted via the LCD to enter the angle change required.

```
void AngleArray(int i, int buffer[]) {
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Enter angle.");
  buffer[i] = 0;
  buffer[i + 1] = 0;
```

In order to take this input, a while loop was created which would only terminate if the '#' key was entered on the keypad. Keys 6 and 4 would represent a 90 degree turn clockwise and anticlockwise respectively. This instruction was then added to the buffer for the key that was pressed.

```
  char key = '0';
  while (key != '#') {
    key = keypad.getKey();
    if (key) {
      Serial.println(key);
      if (key == '6') {
        lcd.setCursor(0, 1);
        lcd.print("90 Clockwise             ");
        buffer[i] = 90;
        buffer[i + 1] = 1;
      }
      if (key == '4') {
        lcd.setCursor(0, 1);
        lcd.print("90 A-clockwise           ");
        buffer[i] = 90;
        buffer[i + 1] = 2;
```

### 6.2.2 Directions Array Entry

Similar to the entry for the Angle Array, a method was created called DirectionArray with the same parameters. The User was prompted to via the LCD to enter the Direction command. A while loop was created to loop until the '#' key was encountered, with the detection of the 2 and 0 keys representing Forwards and Reverse respectively. These instructions were then written to the buffer array.

```
while (key != '#') {
    key = keypad.getKey();
    if (key) {
      if (key == '2') {
        lcd.setCursor(0, 1);
        lcd.print("   ");
        lcd.setCursor(0, 1);
```

```
        lcd.print("Forwards");
        buffer[i] = 2;
      }
      if (key == '0') {
        lcd.setCursor(0, 1);
        lcd.print("Reverse");
        buffer[i] = 1;
      }
```

Next, the user was prompted to enter the distance for the EEEBot to move in the entered direction. To realise this, another while loop was created that looped until the '#' was created, with an if statement checking to see if the '*' key was entered, which would end the entry of the number digits. If '*' had not yet been encountered, the digits would continue to be appended, with this result outputted to the display after each key press. When the '*' key was pressed, the entered value was constrained to a positive integer.

```
lcd.print("Enter distance.");
  key = 'h';
  if (buffer[i] != 0) {
    while (key != '#') {
      key = keypad.getKey();
      if (key) {
        if (key == '*') {
          value = constrain(((value) / 10), 0, 10000000000000000);
          if (value < 0.01) {
            value = 0;
          }
        } else {
          int number = key - '0';
          value = ((10 * value) + (0.01 * number));
```

# 7. Conclusion

In this report, the design, construction and programming of a full RGBYK line following solution using a Symbol Recognition system for line decision has been documented. It has been shown how the use of computer vision can be used to take image data and effectively extract key features (knowing the colour) of a line in the image, and accurately track its position relative to the EEEBot. AI deep learning algorithms were used in conjunction with computer vision to identify instructional symbols on the track that the EEEBot was to traverse, extract and transform these symbols, before recognising them against a library of symbols stored on the EEEBot. It has been shown how the use of a Raspberry Pi with its powerful processors can be used with a digital camera for use with computer vision algorithms. Also, a full Human Machine Interface was created via the use of a keypad interfacing with an ESP32 that displayed to a LCD screen, to enable the user to program the EEEBot with a series of instructions to effectively navigate a maze.

## 7.1 Results and Findings

During testing of the symbol recognition system, it was noticed that the symbol recognition system had tremendous difficulty at detecting the symbol when it was far away, and thus appeared small in the camera frame. In the future, better colour filtering will be implemented to ensure that a crisper contour can be extracted, to avoid holes in said contours. If there were holes in any contours, which appears that there were, then this would have stopped a quad from being detected properly as there would not have been one extracted. Also, the symbol recognition system had great difficulty in processing the symbol when the EEEBot was moving, due to the motion blur and the low frame rate of the camera (due to processing delay between frames). In order to fix this in the future, a system could be created to allow the EEEBot to stop when it encounters a symbol and waits for it to image and identify the symbol, before advancing again.

# 8. References


[1] N/A, N. (2023) *Explain dilation of an image how to dilate an image in opencv*, *ProjectPro*. Available at: https://www.projectpro.io/recipes/what-is-dilation-of-image-dilate-image-opencv#:~:text=Dilation%20is%20usually%20performed%20after,in%20OpenCV%20using%20the%20cv2. (Accessed: 08 May 2023).

[2] N/A, N. (2023a) *Contour approximation method*, *OpenCV*. Available at: https://docs.opencv.org/4.x/d4/d73/tutorial_py_contours_begin.html (Accessed: 08 May 2023).

[3] N/A, N. (2023c) *Samples/CPP/squares.cpp*, *OpenCV*. Available at: https://docs.opencv.org/3.4/db/d00/samples_2cpp_2squares_8cpp-example.html (Accessed: 09 May 2023).

[4] Chakraborty, D. (2021) *OpenCV contour approximation*, *PyImageSearch*. Available at: https://pyimagesearch.com/2021/10/06/opencv-contour-approximation/ (Accessed: 09 May 2023).

[5] *Convex polygon - definition, formulas, properties, examples* (no date) *Cuemath*. Available at: https://www.cuemath.com/geometry/convex/ (Accessed: 09 May 2023).

[6] Kudan, N. (2023) *The evolution of AI and image recognition*. Available at: https://toloka.ai/blog/artificial-intelligence-image-recognition/#:~:text=Healthcare%2C%20marketing%2C%20transportation%2C%20and,based%20on%20AI%20image%20recognition. (Accessed: 09 May 2023).

[7] Ramachandran, S. (2023) *What is image recognition?*, *Nanonets AI & Machine Learning Blog*. Available at: https://nanonets.com/blog/image-recognition/ (Accessed: 09 May 2023).

[8] d'Archimbaud, E. (no date) *Image recognition with Machine Learning: How and why?*, *kili*. Available at: https://kili-technology.com/data-labeling/computer-vision/image-annotation/image-recognition-with-machine-learning-how-and-why (Accessed: 09 May 2023).

[9] Sheppard, J. (2014) *Johnnysheppard/PI2C: Raspberry pi C++ library for easy I2C communication to and from an Arduino, GitHub*. Available at: https://github.com/JohnnySheppard/Pi2c (Accessed: 09 May 2023).

[10] ?? (2019) *Deepomatic*. Available at: https://deepomatic.com/what-is-image-recognition (Accessed: 08 May 2023).

# Appendix

https://github.com/BlackHat0001/OpenCVLineFollower

https://pastebin.com/DrNESgfh