CG1111A Report AY 2024/2025

A-MAZE-ING RACE

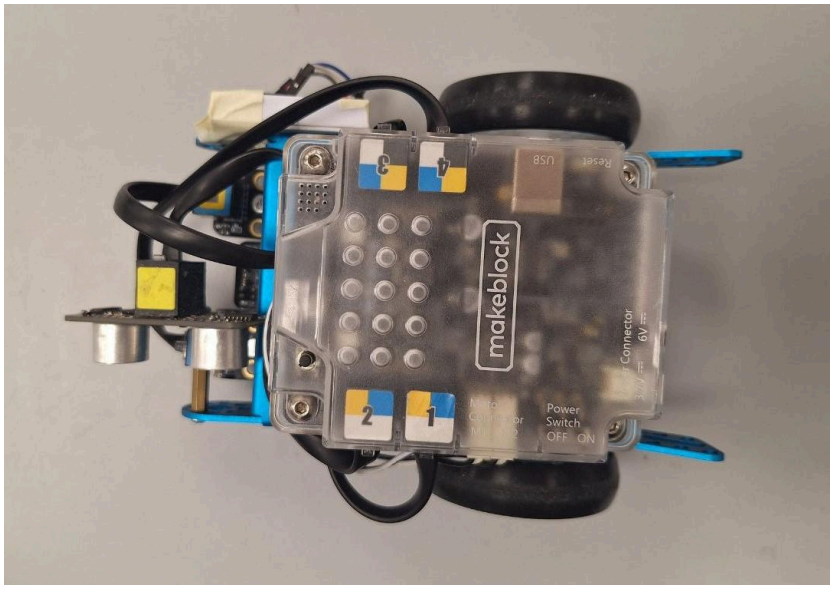
B03-S3-T2-Report

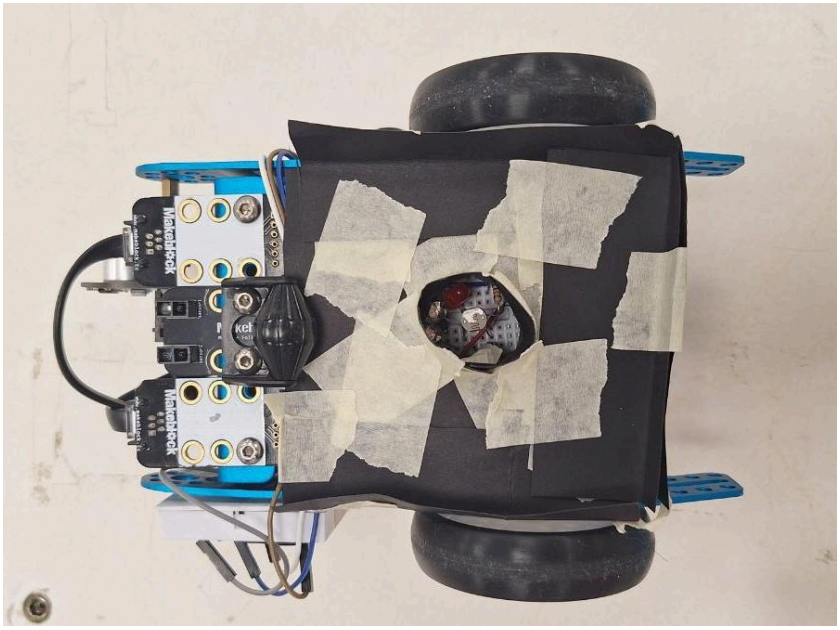
STUDIO NUMBER	B03
SECTION NUMBER	3
TEAM NUMBER	2
TEAM MEMBERS	PANG ANG SHENG ASHER / A0299871E POH WEI HAO / A0299839X PRANAV JANAKIRAMAN / A0306781B PREMIL ROSHAN / A0309256B

Table of Contents

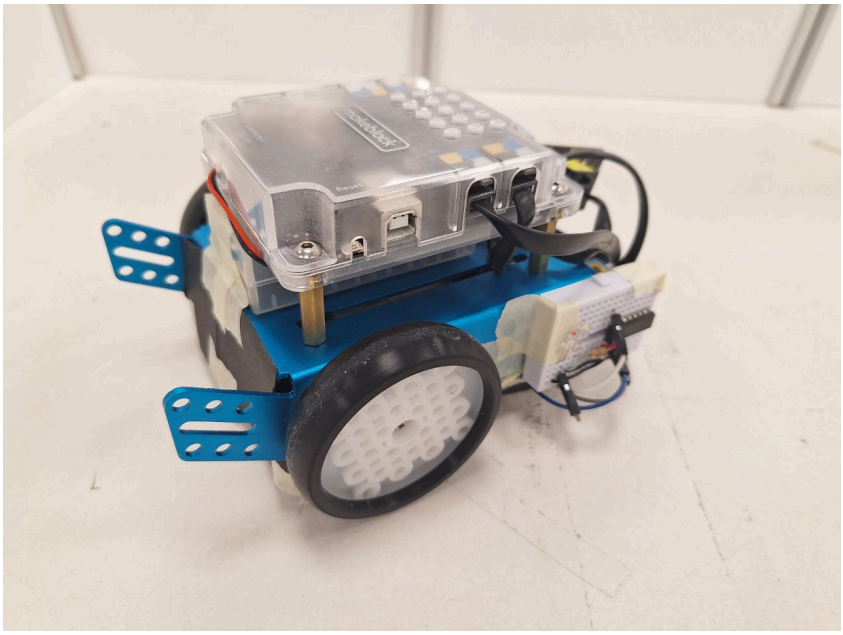
1. Pictures

1.1 Pictures of mBot

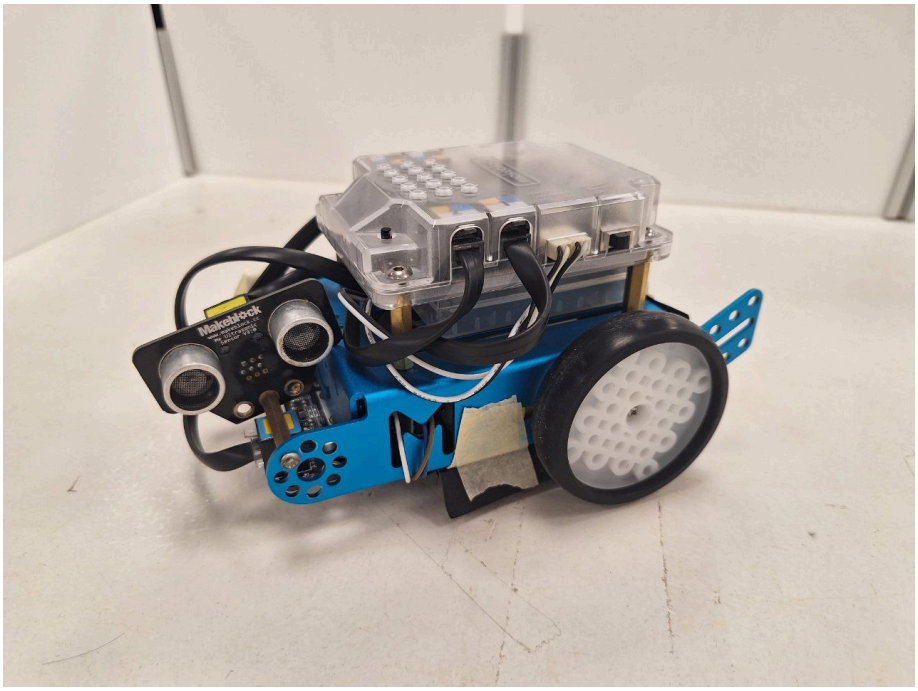
TOP VIEW


BOTTOM VIEW


RIGHT VIEW

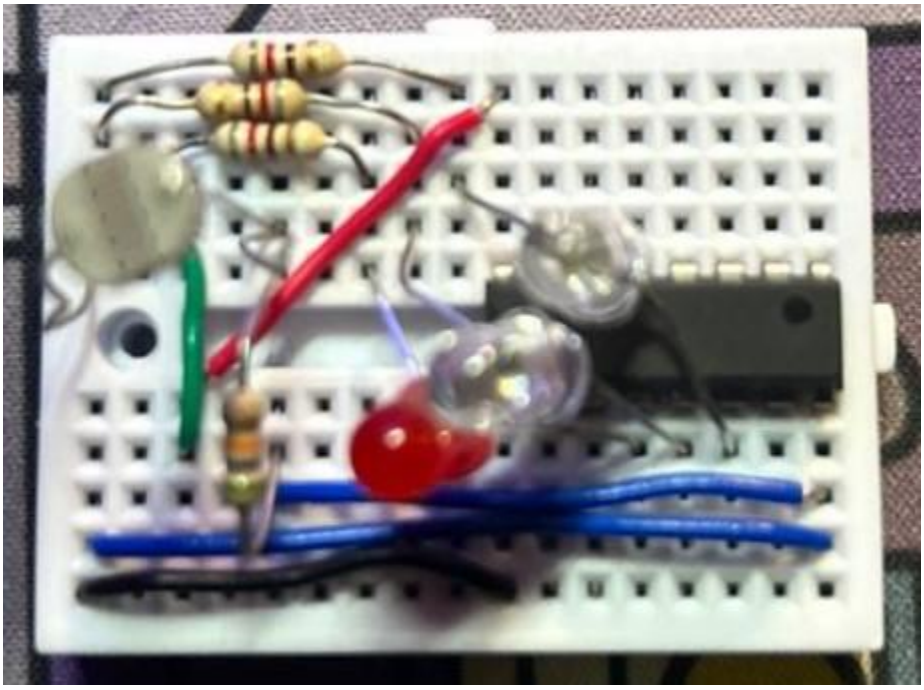
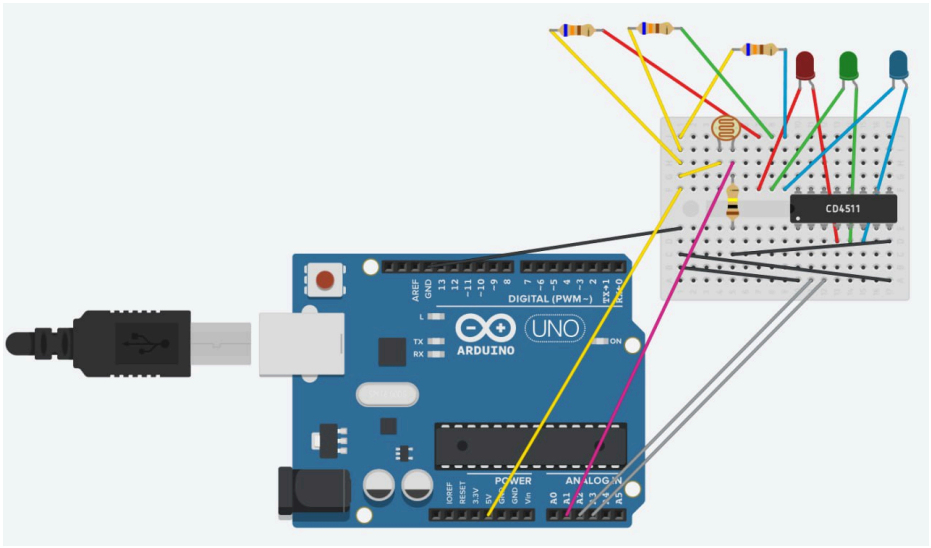


LEFT VIEW

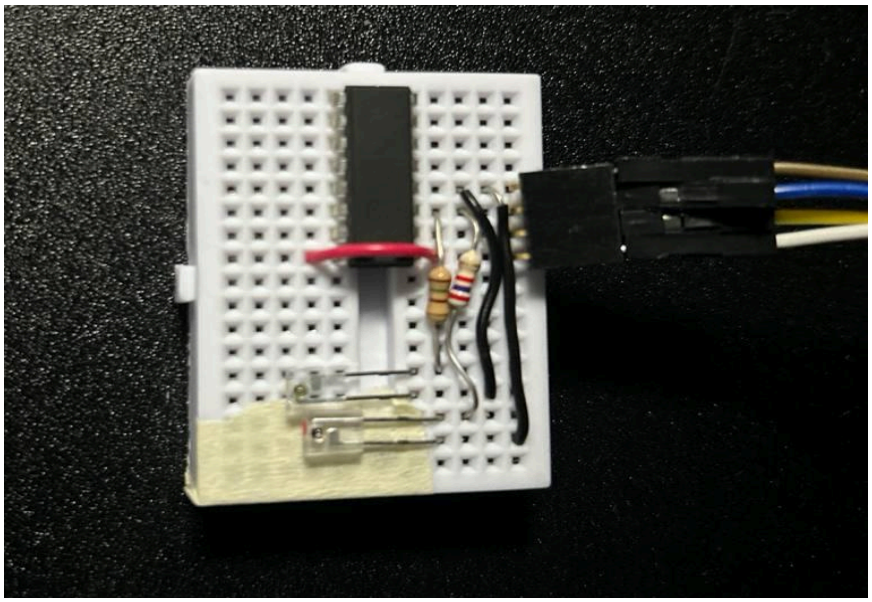
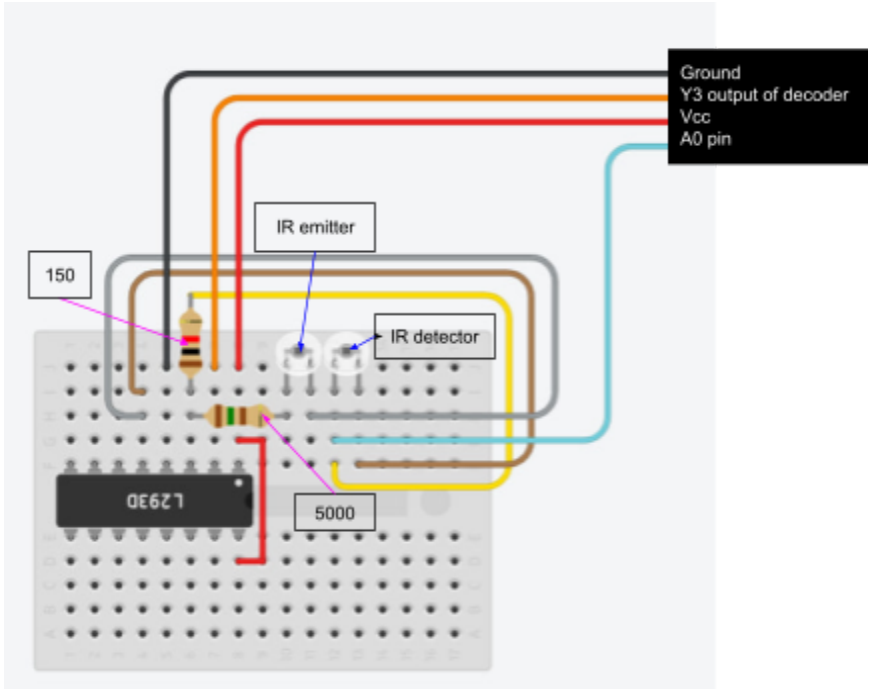


1.2 Pictures of Circuits

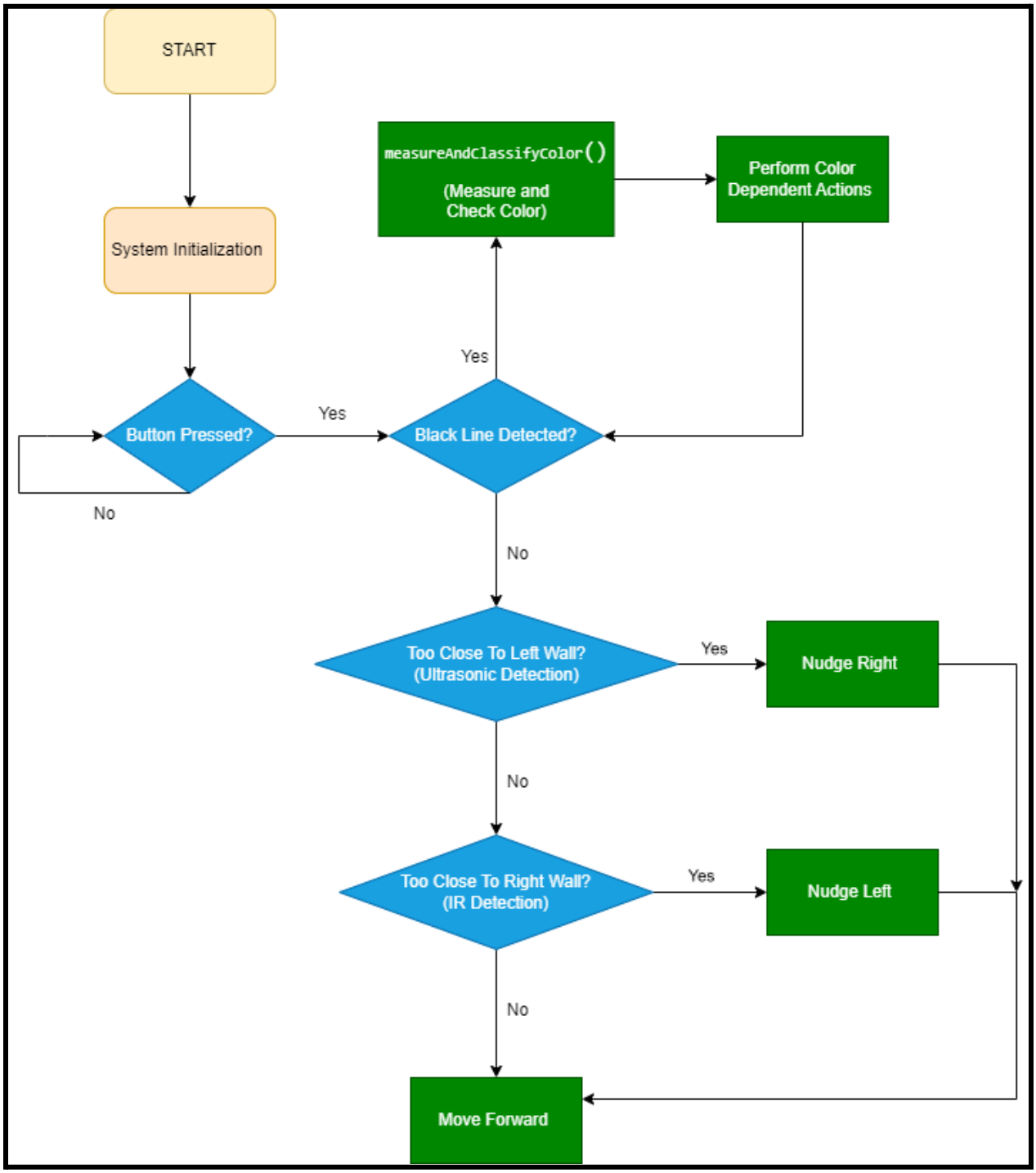
1.2.1 LDR Circuit

LDR CIRCUIT

 LDR Circuit's TinkerCAD

1.2.2 IR Sensor Circuit

IR SENSOR CIRCUIT

 IR Sensor Circuit's TinkerCAD

2. Algorithm and Coding



Overall Algorithm Flowchart

The mBot is designed to autonomously navigate a maze while responding to various color and wall conditions. Equipped with light-dependent resistors (LDRs) for color detection and ultrasonic and IR sensors for sensing wall proximity, the robot can recognize and adapt to different scenarios it encounters along the path. The algorithm to achieve this is as follows:

After initialization, the mBot waits for the button press to begin the main maze-solving loop.

Maze-Solving Loop:

- While the mBot is not on a black line:
    - It first checks if both left and right walls are present (indicating it is in a grid cell flanked by walls).
    - If either wall is absent, the mBot continues moving forward.
    - If both walls are present and the mBot gets too close to a wall, it nudges in the opposite direction to maintain balance.
- When the mBot detects a black line:
    - It stops and performs the color challenge.
    - After completing the color challenge, it resumes navigating the maze.

Color Challenge:

- Upon encountering a **white** color challenge:
    - The mBot stops and enters an infinite loop, playing a celebratory tone.

For the complete mBot algorithm and implementation details, including additional explanations, refer to the source code in the provided zip file and the included Doxygen comments.

3. Implementation of mBot Components

3.1 Ensuring the mBot Moves Straight

3.1.1 Ultrasonic Sensor

We implemented a PID (Proportional-Integral-Derivative) wall following algorithm that uses an ultrasonic sensor to ensure that our mBot stays at a fixed distance from the wall while traveling in a straight line. Using the function `distanceCm()`, provided by the `MeUltrasonicSensor` library, we measure the distance to the wall with an ultrasonic sensor. With this reading, we can assess whether the mBot needs to adjust its trajectory to maintain the desired distance.

The core logic of the PID controller involves comparing the measured distance to a predefined desired distance, set at 10 cm. The PID controller continuously calculates an error term, defined as the difference between the desired distance and the actual measured distance. The PID algorithm is given by:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,d\tau + K_d \frac{de(t)}{dt}$$

The proportional, integral, and derivative components are computed using their respective gains (Kp, Ki, and Kd, set at 5.0, 0.0, and 0.5, respectively). The error terms are then determined by incorporating the current error and the difference between the current and previous errors, as shown in the following code implementation:

```
// Calculate PID
float error = DESIRED_DISTANCE - distance;
integral += error * (sampleTime / 1000.0);
float derivative = (error - previousError) / (sampleTime /
1000.0);
float output = Kp * error + Ki * integral + Kd *
derivative;
previousError = error;
```

The output of the PID controller adjusts the motor speeds, ensuring that any deviation from the target distance triggers a corrective action to keep the mBot aligned with the wall. Specifically, the left and right motor speeds are adjusted based on the output, as shown:

```
int leftSpeed = -(baseSpeed + output);
int rightSpeed = baseSpeed - output;
motorLeft.run(leftSpeed);
motorRight.run(rightSpeed);
```

This adjustment ensures that if the mBot is too close or too far from the wall, it corrects its path by slowing down or speeding up either motor, thereby maintaining a smooth, straight-line trajectory while following the wall.

3.1.2 IR Sensor

To enhance the mBot's ability to detect walls and obstacles in both left and right directions, an IR sensor is positioned opposite the ultrasonic sensor, adding an extra layer of safety during wall-following. The IR sensor complements the PID-controlled ultrasonic sensor by helping detect proximity to the right wall, thus enhancing the mBot's navigation accuracy and preventing potential collisions. As recommended by the project briefing, the IR sensor circuit was powered using the 2-to-4 decoder's $1Y_3$ output pin, which allowed for greater efficiency in logic as the IR sensor would never be used concurrently with the color sensor, preventing interference between the two circuits. The IR sensor operates by pulsing the IR emitter on and off to account for the variance in ambient lighting. This detection is achieved through the following code segment:

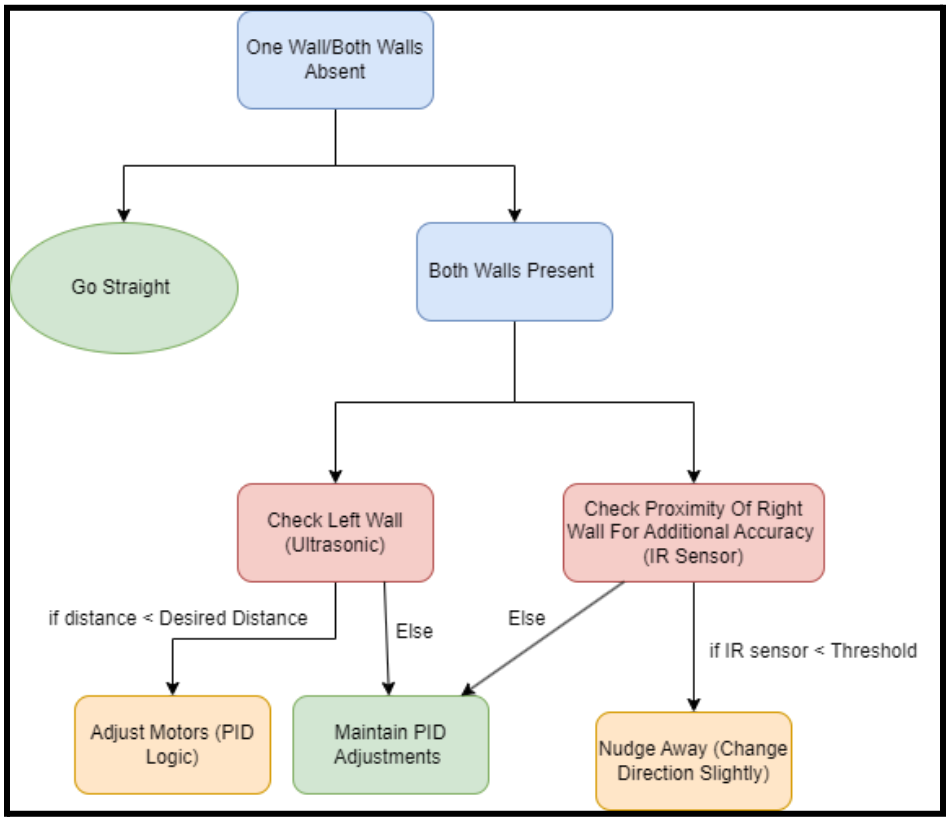
```
int sensorValue = analogRead(irDetectorPin); // Read the analog value
from the IR detector
if (sensorValue < detectionThreshold) { // If sensor value is below
the threshold, an object is detected
    motorLeft.run(120);
    motorRight.run(180);
}
```

In this implementation, the `analogRead(irDetectorPin)` function reads the IR sensor's value, which is then compared to a predefined threshold `(detectionThreshold = 100)`. If the detected value falls below this threshold, it indicates that an object or wall is too close, prompting the mBot to adjust its motion.

This IR logic operates independently of the primary PID algorithm used for ultrasonic distance control. While the ultrasonic sensor primarily handles precise distance measurements and smooth wall-following adjustments, the IR sensor acts as a secondary safeguard. By detecting sudden changes or proximity issues beyond the capability of the ultrasonic sensor, the IR sensor ensures the mBot responds quickly to unexpected obstacles, enhancing navigational accuracy.

For choosing the appropriate resistance values for our IR circuit, we referred to the relevant datasheets, hints from the project brief, and performed trial and error tests. The IR emitter uses a 150 Ω resistor to limit the current and prevent damage, while the IR detector is stabilized with a 5 kΩ pull-down resistor to ensure consistent and reliable analog readings. These values were selected to optimize the IR sensor's performance in detecting reflected infrared signals for wall-following behavior.

3.1.3 Integrating the Functionality of Both Sensors



The flowchart describes a wall-following logic for a robot using sensors and motor adjustments to navigate:

The process starts by determining whether one or both walls are absent or if both walls are present.

1. If one or both walls are absent, the robot moves straight ahead without further adjustments.
2. If both walls are detected, the robot takes additional steps for accuracy:

The left wall is checked using an ultrasonic sensor, and motor speed and direction are adjusted using PID (Proportional-Integral-Derivative) logic to maintain a consistent distance.

The proximity to the right wall is checked with an IR sensor for added accuracy. If the right wall is too close, the robot nudges slightly away to correct its course.

Adjustments are made as follows:

- Motor speed is adjusted through PID logic.
- PID adjustments continue to keep the robot stable.
- If too close to the right wall, the robot nudges away to maintain the correct path.

This combination of sensors and PID adjustments ensures the robot can handle varying paths and obstacles, maintaining stability and reliable wall-following in varied environments.

3.2 Decoding Colour Challenges

3.2.1 Black Line Detector

Our black line detector in the mBot code uses the lineFollower sensor connected to PORT_1, represented by the MeLineFollower object:

```
#define LINE_FOLLOWER_PIN PORT_1
MeLineFollower lineFollower(LINE_FOLLOWER_PIN);
```

The mBot utilizes two infrared (IR) sensors to detect a black line by emitting and detecting IR light. Dark surfaces, such as the black line, absorb IR light and reflect very little, resulting in the sensors reading a value of 0. Within the loop function, the mBot library function `readSensors()` captures the sensor values, stored as `lineSensorValue`. Detection of the black line occurs when both sensors register a value of 0, prompting the robot to halt by invoking the `stopMotors()` function. Subsequently, the robot proceeds with the color challenge by calling `measureAndClassifyColor()`. The code implementation is shown below:

```
int lineSensorValue = lineFollower.readSensors();
    // Continuously monitor the line sensor until it detects a value of 0
    if (lineSensorValue == 0) {
      // Stop the robot when black line is detected
      stopMotors();
      // Turn on lights and perform color measurement
      measureAndClassifyColor();
    }
```

3.2.2 Colour Sensor

Our implementation of the mBot's color sensor functionality involves using three LEDs, red, green, and blue controlled via pins A2 and A3. The sensor works by consecutively shining each LED onto the target paper to determine its color, measuring the intensity of the reflected light for each wavelength. The reflected light intensity is captured using a light-dependent resistor (LDR), which provides a corresponding potential difference between its terminals.
Since different colors of paper absorb and reflect varying amounts of light from each LED, the system effectively captures these variations, allowing accurate classification of the paper's color based on the total reflected intensity for each of the red, green, and blue LEDs. This process enables precise color identification through sequential LED activation and reflective light analysis.

```
//Initialise control pins for LEDs
pinMode(A2, OUTPUT);
pinMode(A3, OUTPUT);
```

Light Dependent Resistor (LDR): The LDR controlled by pin A1, measures the intensity of the light that is reflected off the colour paper.

```
pinMode(A1, INPUT);
```

k-Nearest Neighbours (k-NN) Algorithm: We utilized the k-Nearest Neighbors (k-NN) algorithm to classify six predefined color values based on the measured RGB intensities captured by the LDR. The k-NN algorithm functions by comparing new data points to a set of pre-labeled data, assigning the classification based on the closest "k" neighbors in the feature space.

In our implementation, we calibrated each color by taking ten separate RGB readings and calculating the average, which provides a reliable representation of each color's characteristic values. These averaged RGB values are stored in the `colorDataset[6]` array. When new RGB values are measured, they are compared against this dataset using the k-NN algorithm to determine the closest match and classify the color accurately. This process enables effective and consistent color classification through systematic comparison and similarity-based categorization. Additionally, it simplifies troubleshooting and recalibration by allowing easy updates to the dataset as needed.

```
// Dataset with 6 predefined colors
Color colorDataset[6] = {
  {695, 784, 739, "Red"},     // RGB values for Red
  {720, 838, 761, "Orange"},  // RGB values for Orange
  {604, 880, 785, "Green"},   // RGB values for Green
  {595, 873, 854, "Blue"},    // RGB values for Blue
  {767, 899, 867, "Pink"},    // RGB values for Pink
  {763, 927, 893, "White"}    // RGB values for White
};
```

The k-NN approach used focuses on computing the Euclidean distance between the measured RGB values and the predefined colors in the dataset to determine the closest match.

The `calculateDistance` function computes the Euclidean distance between two sets of RGB values. This distance metric is crucial for identifying the color that is most similar to the measured one. Given two sets of RGB values (e.g., r1, g1, b1) and r2, g2, b2), the function calculates the Euclidean distance, which quantifies how close the colors are in a three-dimensional RGB space. The Euclidean distance is computed as:

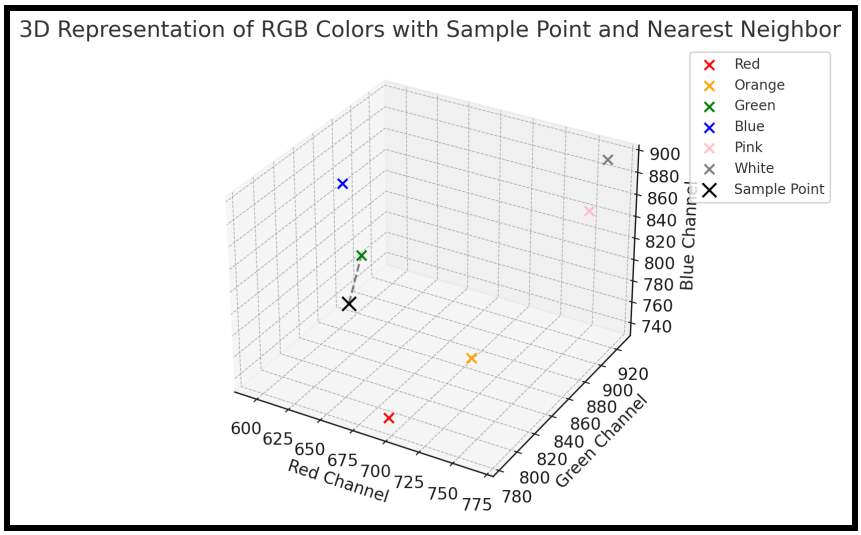
$$\text{Distance} = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$$

The `classifyColor` function then takes measured RGB values and classifies the color using the k-NN method. It uses the Euclidean distance from the measured color to each of the six predefined colors in the dataset, which are stored in `colorDataset`. Since k = 1, the algorithm finds the color with the **minimum distance**, indicating the closest match.

To achieve this, the function iterates through the dataset, calculates the distance to each color, and keeps track of the minimum distance and its corresponding index. Finally, it returns the label of the color that has the smallest distance, effectively classifying the measured color.

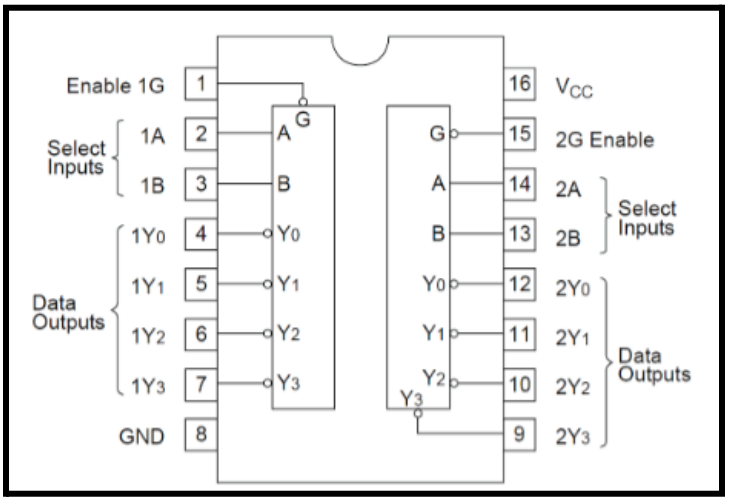
This implementation of k-NN is straightforward and computationally efficient given that k = 1 and the dataset contains only six color entries. The simplicity of the code also aids in **troubleshooting and recalibration** since updating the dataset or modifying parameters is easy and intuitive. The code implementation is detailed below:

```
const int k = 1;

// Function to calculate Euclidean distance between two colors
float calculateDistance(int r1, int g1, int b1, int r2, int g2, int b2) {
  return sqrt(pow(r1 - r2, 2) + pow(g1 - g2, 2) + pow(b1 - b2, 2));
}

// Function to classify the measured color using k-NN
String classifyColor(int red, int green, int blue) {
  float distances[6];

  // Calculate distance from the measured color to each color in the dataset
  for (int i = 0; i < 6; i++) {
    distances[i] = calculateDistance(red, green, blue, colorDataset[i].red,
colorDataset[i].green, colorDataset[i].blue);
  }

  // Find the nearest neighbor (since k = 1, just find the minimum distance)
  int nearestIndex = 0;
  float minDistance = distances[0];
  for (int i = 1; i < 6; i++) {
    if (distances[i] < minDistance) {
      minDistance = distances[i];
      nearestIndex = i;
    }
  }

  // Return the label of the nearest color
  return colorDataset[nearestIndex].label;
}
```

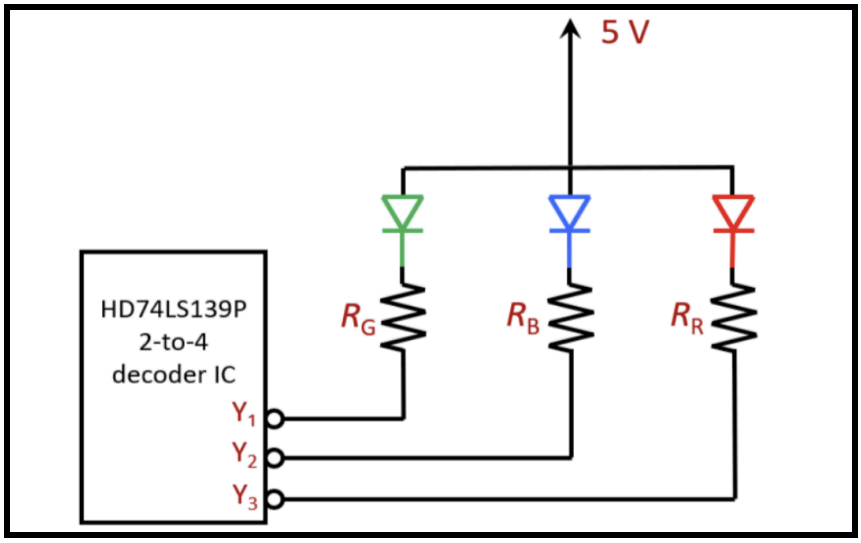


Visualization of k-NN Algorithm

The design of the LED circuit (Figure 3.2) is based on adaptations from "The A-maze-ing Race Project 2024," which provides an efficient approach to LED control by utilizing a 2-to-4 decoder (Figure 3.1). The 2-to-4 decoder allows two digital output pins from the mBot to manage multiple components, specifically, the three LEDs and an IR emitter via the Y0, Y1, Y2, and Y3 data output pins respectively. This configuration optimizes the use of limited output pins by enabling selective control of each component, contributing to a more streamlined and resource-efficient circuit design.



(Fig. 3.1) HD74LS139P Top View from HD74LS139 Datasheet



(Fig. 3.2) LED Circuit Design from "The A-maze-ing race Project 2024"

Following the recommendations from "The A-maze-ing Race Project 2024," we ensured that the 2-to-4 decoder output pins did not exceed the 8 mA maximum current limit. Using the "HD74LS139P Datasheet," along with the datasheets for the red, blue, and green LEDs, we calculated the minimum resistance required for the three LEDs, as represented in Figures 3.3, 3.4, and 3.5.

For the red LED, the resistance was determined using the formula:

$$R = \frac{(5V - 0.5V - 1.85V)}{8mA} = 331.25\Omega$$

For the blue and green LED resistance calculation:

$$R = \frac{(5V - 0.5V - 3V)}{8mA} = 187.5\Omega$$

However, after testing and optimization through trial and error, we ultimately selected a resistance value of 625 Ω for the LEDs, as it provided the most consistent sensor readings and operational reliability. The higher resistance value helped ensure the circuit was stable and that current remained well within safe limits, enhancing the reliability of the readings.

Recommended Operating Conditions

Item	Symbol	Min	Typ	Max	Unit
Supply voltage	V_{CC}	4.75	5.00	5.25	V
Output current	I_{OH}	—	—	–400	µA
	I_{OL}	—	—	8	mA
Operating temperature	Topr	–20	25	75	°C

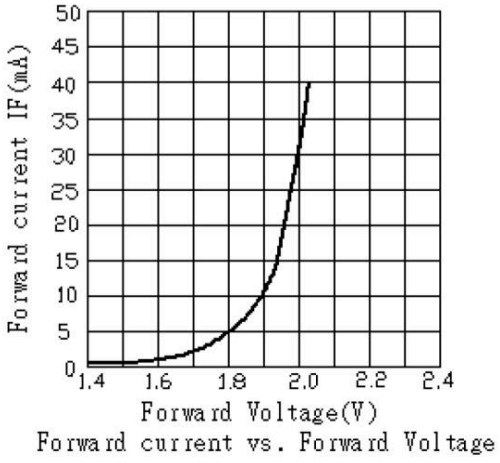
Electrical Characteristics

(Ta = –20 to +75 °C)

Item	Symbol	min.	typ.*	max.	Unit	Condition
Input voltage	V_{IH}	2.0	—	—	V	
	V_{IL}	—	—	0.8	V	
Output voltage	V_{OH}	2.7	—	—	V	V_{CC} = 4.75 V, V_{IH} = 2 V, V_{IL} = 0.8 V, I_{OH} = –400 µA
	V_{OL}	—	—	0.4	V	I_{OL} = 4 mA; V_{CC} = 4.75 V, V_{IH} = 2 V, V_{IL} = 0.8 V
		—	—	0.5		I_{OL} = 8 mA
Input current	I_{IH}	—	—	20	µA	V_{CC} = 5.25 V, V_I = 2.7 V
	I_{IL}	—	—	–0.4	mA	V_{CC} = 5.25 V, V_I = 0.4 V
	I_I	—	—	0.1	mA	V_{CC} = 5.25 V, V_I = 7 V
Short-circuit output current	I_{OS}	–5	—	–42	mA	V_{CC} = 5.25 V
Supply current	I_{CC}	—	6.8	11	mA	V_{CC} = 5.25 V, Outputs enabled and open
Input clamp voltage	V_{IK}	—	—	–1.5	V	V_{CC} = 4.75 V, I_{IN} = –18 mA

Note: * V_{CC} = 5 V, Ta = 25°C

(Fig. 3.3) Max Output Current (8 mA) and Output Voltage (0.5V) from HD74LS139 Datasheet



(Fig. 3.4) Datasheet for Red LED

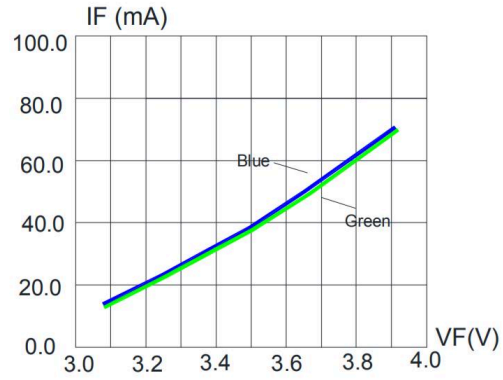


FIG.1 FORWARD CURRENT VS. FORWARD VOLTAGE.

(Fig. 3.5) Datasheet for Blue & Green LED

The complete process of measuring and classifying a color occurs within the `measureAndClassifyColor()` function. This function systematically activates the red, green, and blue LEDs, which is achieved by setting specific combinations of the A2 and A3 pins:

- The red LED is turned on by setting both A2 and A3 to LOW.
- The green LED is activated by setting A2 to HIGH and A3 to LOW.
- The blue LED is turned on by setting A2 to LOW and A3 to HIGH.

To turn off all the LEDs, both A2 and A3 are set to HIGH.

As each LED is activated sequentially, the light-dependent resistor (LDR) measures the intensity of light reflected off the colored paper, capturing the different components of the reflected light. These values are recorded and stored as `red_reading`, `green_reading`, and `blue_reading`, respectively, which form the basis for classifying the color of the target object.

```
// Function to measure red light intensity
int measure_red() {
  digitalWrite(A2, LOW);  // Turn on red LED (A2 LOW, A3 LOW)
  digitalWrite(A3, LOW);
  delay(10); // Ensure stable light output
  int red_reading = analogRead(LDRbrightness);
  return red_reading;
}

// Function to measure green light intensity
int measure_green() {
  digitalWrite(A2, HIGH);   // Turn on green LED (A2 HIGH, A3 LOW)
  digitalWrite(A3, LOW);
  delay(10); // Ensure stable light output
  int green_reading = analogRead(LDRbrightness);
  return green_reading;
}

// Function to measure blue light intensity
int measure_blue() {
  digitalWrite(A2, LOW);   // Turn on blue LED (A2 LOW, A3 HIGH)
  digitalWrite(A3, HIGH);
  delay(10); // Ensure stable light output
  int blue_reading = analogRead(LDRbrightness);
  return blue_reading;
}

// Function to turn off both LEDs
```

```
void off_lights() {
  digitalWrite(A2, HIGH);
  digitalWrite(A3, HIGH);
}
```

After classifying the color, the mBot uses the detected color information to perform an appropriate response. The LED lights up using the color detected, executed through the `rgbLed.setColor()` and `rgbLed.show()` functions, followed by an action corresponding to that color. The specific actions for each detected color are as follows:

- **White**: The mBot lights up white and plays a celebratory tune.
- **Red**: The mBot lights up red and turns left.
- **Green**: The mBot lights up green and turns right.
- **Blue**: The mBot lights up blue, turns right, moves forward, and turns right again.
- **Pink**: The mBot lights up pink, turns left, moves forward, and then turns left again.
- **Orange**: The mBot lights up orange and performs a U-turn.

This approach enables the mBot to not only visually indicate the detected color but also perform specific actions in response, enhancing interaction and demonstrating different navigational behaviors.

```
if (classifiedColor == "White") {
    rgbLed.setColor(0, 255, 255, 255);
    rgbLed.show();
    johnCenaIntro();
  }
  else if (classifiedColor == "Red") {
    rgbLed.setColor(0, 255, 0, 0);
    rgbLed.show();
    turnLeft();
  }
  else if (classifiedColor == "Green") {
    rgbLed.setColor(0, 0, 255, 0);
    rgbLed.show();
    turnRight();
  }
  else if (classifiedColor == "Blue") {
    rgbLed.setColor(0, 0, 0, 255);
    rgbLed.show();
    turnRight();
    delay(10); // Move forward a bit
```

```
    moveForward();
    delay(1000); // Adjust delay as needed to move forward a bit
    turnRight();
  }
  else if (classifiedColor == "Pink") {
    rgbLed.setColor(0, 255, 105, 180);
    rgbLed.show();
    turnLeft();
    delay(10); // Move forward a bit
    moveForward();
    delay(1000); // Adjust delay as needed to move forward a bit
    turnLeft();
  }
  else if (classifiedColor == "Orange") {
    rgbLed.setColor(0, 255, 165, 0);
    rgbLed.show();
    ReverseTurn();
  }
}
```

4. Celebratory Music

Upon detecting a white color, the mBot was programmed to play a celebratory tune. For this occasion, we chose John Cena's iconic theme music. To ensure an accurate recreation of the theme, we converted the BPM (Beats Per Minute) of the song into a usable millisecond per note value, incorporating these calculations into the variable `int beatDuration`. This allowed us to manipulate the timing for each note independently, making adjustments quick and troubleshooting efficient, all while paying close attention to the timing and rhythm to match the original as faithfully as possible.

This attention to detail in timing and meticulous approach paid off, resulting in a well-executed recreation of John Cena's theme music that brought the celebratory moment to life with precision and excitement. The code for the music is shown below:

```
void johnCenaIntro() {
  // Set BPM to 90 (each beat is 666.67 ms)
  const int beatDuration = 600;  // in milliseconds

  // Frequencies and durations inspired by John Cena's intro theme
  buzzer.tone(392, beatDuration); // G for 1 beat
  delay(int(beatDuration * 0.05 * 0.9));
  buzzer.tone(440, int(beatDuration * 0.5 * 0.9)); // A for half a
beat
  delay(int(beatDuration * 0.005 * 0.9));  // Shortened delay
  buzzer.tone(349, int(beatDuration * 0.5 * 0.9)); // F for half a
beat
  delay(int(beatDuration * 0.1 * 0.9));

  buzzer.noTone();
  delay(int(beatDuration * 0.1 * 0.9));

  buzzer.tone(392, int(beatDuration * 2 * 0.9)); // G for 2 beats
  delay(int(beatDuration * 0.3 * 0.9));

  buzzer.noTone();
  delay(int(beatDuration * 0.3 * 0.9));

  buzzer.tone(466, beatDuration); // Bb for 1 beat
  delay(int(beatDuration * 0.1 * 0.9));
```

```
  buzzer.tone(440, int(beatDuration * 0.5 * 0.9)); // A for half a
beat
  delay(int(beatDuration * 0.005 * 0.9));  // Shortened delay
  buzzer.tone(349, int(beatDuration * 0.5 * 0.9)); // F for half a
beat
  delay(int(beatDuration * 0.1 * 0.9));

  buzzer.noTone();
  delay(int(beatDuration * 0.1 * 0.9));

  buzzer.tone(392, int(beatDuration * 2 * 0.9)); // G for 2 beats
  delay(int(beatDuration * 0.3 * 0.9));

  buzzer.noTone();
}
```

5. Improving the IR and LDR Sensors

5.1 IR Sensor

Prototype 1: Suboptimal Placement and Design Of IR Circuit

In the initial prototype, the infrared (IR) sensor was positioned on top of the robot. However, the limited range of the IR sensor, compared to the ultrasonic sensor, resulted in suboptimal detection performance. Additionally, the initial circuit design included long wires, which often interfered with the functionality of the sensors. Specifically, due to the natural movements of the mBot, these wires occasionally obstructed the IR sensor, leading to inconsistent readings. Furthermore, the components were positioned very close together in the circuit, making it cluttered and increasing the chances of physical interference.

The placement of the circuit on the top of the mBot also contributed to challenges in sensor reliability. This configuration increased the distance between the sensors and the wall, which reduced the accuracy of detection and led to erratic behavior. The combination of suboptimal sensor placement and physical interference from loose wiring undermined the overall effectiveness of the system during testing.

Similarly, our initial code for the IR sensor circuit proved to be problematic. Our first iteration of the code used a dedicated output pin for the IR emitter, and also continuously read inputs from the IR detector, even when the robot was stationary or detecting a color. This approach resulted in several issues: we lacked sufficient analog pins for the simultaneous operation of both the color sensor and the IR sensor, and the continuous reading from the IR detector created inefficiencies in the system. By constantly providing a feedback loop to the mBot, the IR sensor consumed processing resources even when its data was not necessary, thereby slowing down the overall efficiency of the robot's operations.

These challenges underscored the need for a redesigned layout, more robust code and improved shielding to ensure accurate and reliable readings across all distance ranges.

Final Improved Implementations to Increase Robustness

To resolve these issues, we decided to rearrange the electronics of the mBot to allow for the IR circuit to be mounted on the right-hand side of the mBot. This modification allowed the IR components to be positioned closer to the wall, enabling more accurate detections and better overall sensor performance. By moving the IR circuit to a more optimal position, we were able to reduce the distance between the sensor and the walls of the maze, which contributed to more consistent and reliable readings during operation.

In addition to repositioning the IR circuit, we also reorganized the circuit components to improve the layout. We cut down the wire length to the bare minimum required, which helped in isolating the IR components more effectively. This reduction in wire length not only minimized physical interference but also improved the overall neatness and organization of the circuit. By addressing the clutter and ensuring that components were better spaced, we reduced the potential for cross-interference among different elements of the system.

These collective changes significantly mitigated the majority of the physical interference issues that had previously affected the IR sensor's performance. As a result, we observed a substantial improvement in the accuracy and consistency of the IR sensor, which greatly enhanced the overall effectiveness and reliability of the mBot during its operation.

Additionally, we made adjustments to the code to address some of the issues we faced in our first prototype. Following the hints provided in the project briefing, we made use of the 2-to-4 decoder, connecting the IR emitter to the 1Y3 output pin of the decoder. Modifying our code accordingly, we set the activation of the IR emitter to correspond to HIGH outputs from pins A2 and A3. This allowed for the operation of the 3 LEDs using the other 3 possible configurations of signals from the decoder. We also ensured that the IR detector’s value was only taken when the mBot was in motion, specifically only before and after the mBot had detected a line and not during the color sensing loop, which would avoid the unnecessary slowing of the mBot’s color detection.

These improvements not only allowed the IR sensor to work more consistently independent of environmental conditions, but also enabled efficient use of the available resources, resulting in a smoother and less complicated operation of the mBot.

5.2 LDR Sensor

Prototype 1: Inadequate Shielding From Ambient Light

In our first prototype, we used a black paper covering on the underside of the bot to reduce ambient light interference. However, this design had notable shortcomings. The hole cut for the LDR and three LEDs was significantly larger than needed, leaving the sensor to still be overly exposed to ambient light. This led to stray light interfering with the LDR, resulting in inconsistent and unreliable readings. Additionally, the lack of proper shielding around the edges of the bot allowed light to enter from multiple angles, further distorting the LDR values. These challenges highlighted the critical need for improved isolation and protection of the sensor area. As a result, we focused on developing a more effective shielding solution in later iterations to ensure accurate color detection and reliable sensor performance.

Final Improved Implementations to Increase Robustness

To address the shortcomings of our previous implementation, we sought to improve the underside covering of the LDR. We more accurately measured the size of the color sensor components that needed to be exposed and precisely cut a hole that was only as large as necessary. Additionally, we extended the covering into a cylindrical enclosure that reached as close to the floor as possible, further minimizing ambient light interference.

In addition to the black covering, we introduced a skirt-like design extending from the edges of the mBot. The skirt protruded downwards towards the floor, effectively shielding the sensor and LEDs from stray light entering from the sides. This innovative approach ensured that the LDR received light primarily from the LEDs and the surface directly beneath it, thereby improving the consistency and accuracy of our color readings.

The combination of the black paper and the skirt design worked synergistically to isolate the sensor region, reducing noise from external light sources and enhancing the bot's overall performance in detecting color. This design consideration was critical in maintaining the reliability of the readings, even in varying lighting conditions.

Calibration Improvement for LDR Circuit

In the initial stages of our design, during the calibration process for the LDR circuit, we only measured the RGB values of each color once. While this method provided a basic understanding of the color readings, it lacked the precision and reliability needed for consistent performance. Small fluctuations in sensor readings due to environmental factors, such as lighting conditions or slight inconsistencies in the surface, were not accounted for, leading to inaccuracies during the practice run.

For the final prototype, we improved the calibration process by taking multiple readings for each color. Specifically, we measured the RGB values ten times for each color and calculated the average of these readings. Additionally, we took readings from multiple mazes around the lab to accommodate varying conditions, which allowed us to achieve a more robust data set to calibrate our mBot. This adjustment significantly reduced the influence of outlier values and provided a more accurate and consistent representation of each color's characteristics. By averaging the readings, we were able to program the mBot with more reliable thresholds, which enhanced its efficiency in distinguishing between colors during operation.

This enhanced calibration process was critical in ensuring the accuracy and robustness of the mBot's color sensing capabilities, especially in varied or unpredictable environments.

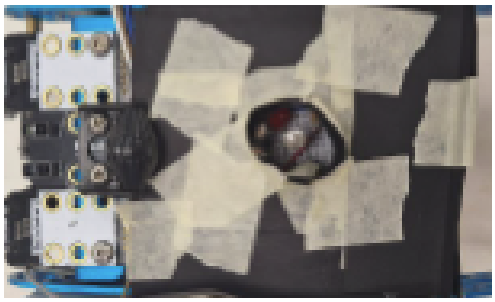
6. Roles and Responsibilities

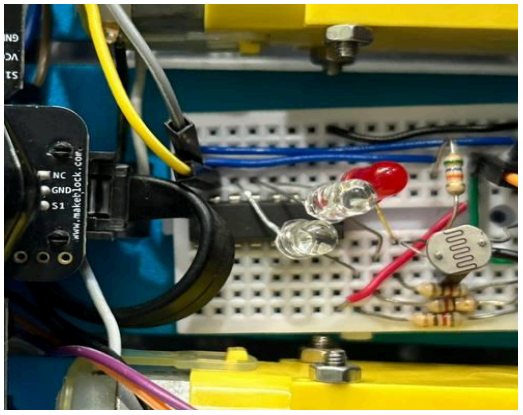
To streamline the workload and ensure an efficient and productive development process, our group delegated specific responsibilities for different components of the robot. This division of tasks allowed us to work in parallel, leveraging individual strengths while maintaining consistent progress.

The following table outlines the responsibilities assigned to each group member and the corresponding tasks undertaken:

Pranav	- Contributed to the report writing. - Assisted with the code for robot movement and Wall Detection. - Assisted in designing the LDR casing. - Assisted with the implementation of the color sensing logic in code.
Roshan	- Contributed to the report writing. - Assisted with the circuitry design of the LDR. - Assisted with the IR sensor circuitry. - Assisted with the implementation of the color sensing logic in code.
Wei Hao	- Contributed to the report writing. - Assisted with the LDR circuitry. - Assisted with the IR sensor circuitry. - Assisted in the design of the LDR casing.
Asher	- Contributed to the report writing. - Assisted with the IR sensor circuitry. - Assisted with the LDR circuitry. - Assisted with the implementation of the IR sensor logic in code.

7. Challenges Encountered and their Solutions

No.	Challenge	Solution	Relevant Images
1.	Inconsistent color sensing readings occurred due to variations in ambient lighting, which influenced the LDR measurements and led to fluctuations in detected values and inaccuracies.	To address the issue of inconsistent color readings, we implemented a housing for the LDR circuit. The housing was constructed using black paper, designed to enclose the entire circuit as closely to the ground as possible. This enclosure effectively blocked the majority of ambient light, preventing it from being detected by the LDR. As a result, the LDR readings were isolated, ensuring they primarily reflected light from the RGB LEDs, thereby improving the accuracy and consistency of color detection.	

2.	While the PID algorithm improved the accuracy of wall-following, its corrections were overly aggressive, resulting in erratic, jerky movements. Frequent, abrupt adjustments negatively impacted the stability and smoothness of the mBot's navigation.	To address the issue of overly aggressive corrections, we conducted thorough testing of the PID parameters, particularly focusing on the Kp value. We observed that decreasing Kp resulted in smoother corrections; however, reducing it excessively led to insufficient corrective action. It was crucial to find a balanced "sweet spot" for Kp to ensure the mBot maintained stable and accurate wall-following behavior.	
3.	The RGB values for different colors were often too similar, resulting in frequent incorrect color detection.	The issue was traced to the initial position of the LDR, which was placed too far from the LEDs, causing excessive light diffusion before detection. To resolve this, we repositioned the LDR closer to the LEDs, minimizing diffusion and improving color differentiation accuracy.	

4.	Inconsistent line detection by the mBot often led to it failing to stop in time, resulting in collisions with walls.	The inconsistent line detection was traced to inefficiencies in the code logic, including unnecessary checks and nested statements, which increased the number of operations the mBot had to execute. This caused delays, preventing the mBot from stopping promptly upon detecting the line. To resolve this, we removed unnecessary functions and loops and streamlined the algorithm, resulting in faster and more efficient code execution, thereby improving the mBot's responsiveness and accuracy in detecting lines.	