CG1111A Engineering Principles and Practice 1

AY25/26

A-maze-ing Race Project Report

Studio Group No.	B01
Section No.	S2
Team No.	T1
Team Members	Eethan Zhang You Jie (A0324607H) Ernest Goh Chin Wee (A0323962B) Eugene Quek (A0324685W) Gabriel Yu Yao Soon (A0324071R)

Table of Contents

1. Project Overview

This report provides an overview of the “A-maze-ing” Race project, a robotics challenge that entails the design, programming, and testing of an mBot — a motorised robot engineered to autonomously navigate a maze while avoiding obstacles and completing designated waypoint tasks in the shortest possible time. The project serves as a simulation of real-world autonomous navigation systems and incorporates ultrasonic, infrared, and custom colour-sensing technologies to fulfil the specified design objectives.

The project requirements encompass the specification of the navigation program, waypoint interaction, and the execution of turns based on colour-coded instructions. Consequently, this project serves as a comprehensive assessment of the knowledge and skills acquired throughout the course. These include:

1. DC Circuit Principles
2. Arduino Programming and Debugging
3. Datasheet Interpretation
4. Circuit Building and Analysis
5. Sensor Principles

By applying the knowledge and skills acquired throughout the course to both circuit design and programming, the team successfully met the project requirements and achieved complete navigation of the maze. The A-maze-ing Race project requirements are as follows:

1. Collision Avoidance
2. Waypoint Challenge: Colour Sensing
3. Waypoint Challenge: Turn Decision-Making
4. Controlled Turning
5. End of Maze Chime

This report will further elaborate on the methods and strategies employed to enable the mBot to successfully navigate the maze and fulfil all the specified requirements of the challenge.

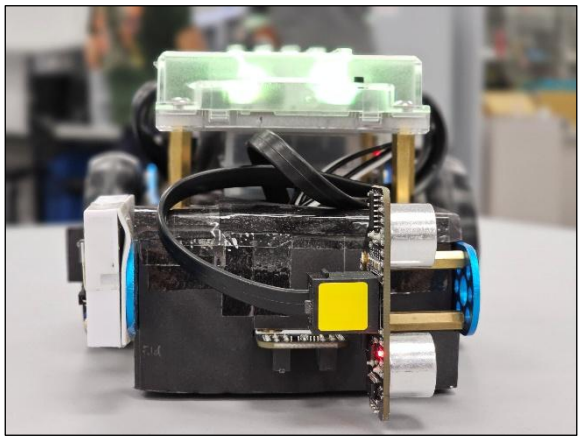
2. Design of mBot

Together with the mBot, we received an Arduino Uno microcontroller board, two 5V motors for the wheels, a line sensor and an ultrasonic sensor. We decided to place the ultrasonic sensor on the left side of the mBot, connected via Port 1. The line sensor installed at its designed location, was connected to the mBot via Port 2. We installed the infrared (IR) proximity sensor on the right side of the mBot, connected via Port 3. The colour sensor was positioned beneath the mBot, the circuit consisting of mainly the Light Dependent Resistor (LDR), and 3 LEDs for the red, green and blue colours respectively. The colour sensor was connected to the mBot via Port 4.

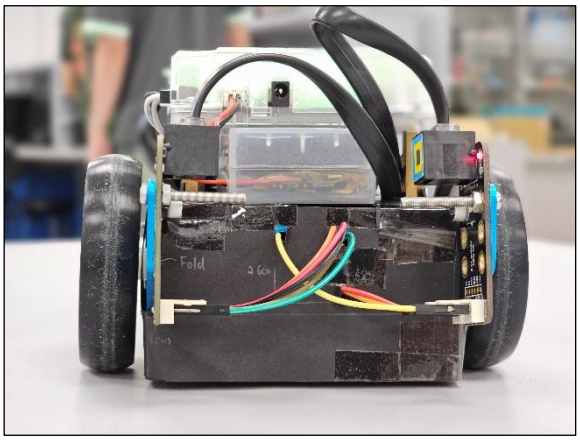
We then surrounded the LDR, the LEDs and the bottom portion of the mBot with black paper respectively to prevent light from the external surroundings to interfere with the LDR readings, hence increasing the colour sensor's accuracy.

To avoid brushing against the maze walls and to keep the circuits neat, the wires and end connectors of components (e.g. resistors) were cut short and kept compact. This also prevented the wiring and components underneath the mBot from being dragged across the floor and hence steering the mBot in an unintended direction.

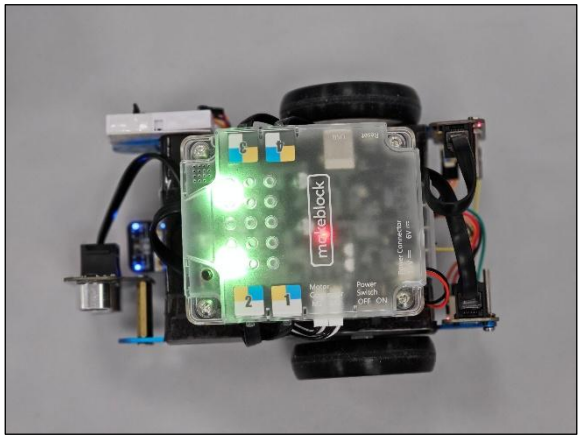
Table 1 below shows images of our mBot, and Figure 1 displays the overall circuit system:



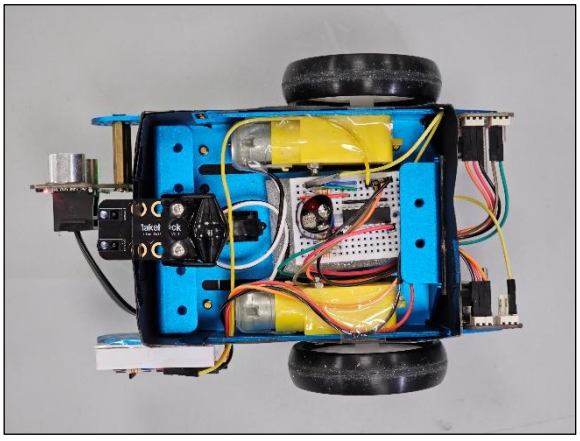
Front View



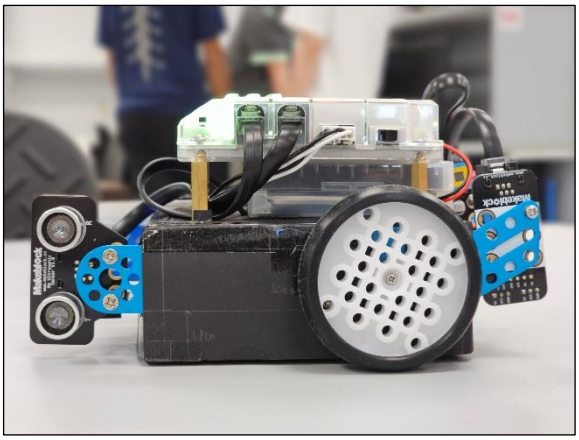
Back View



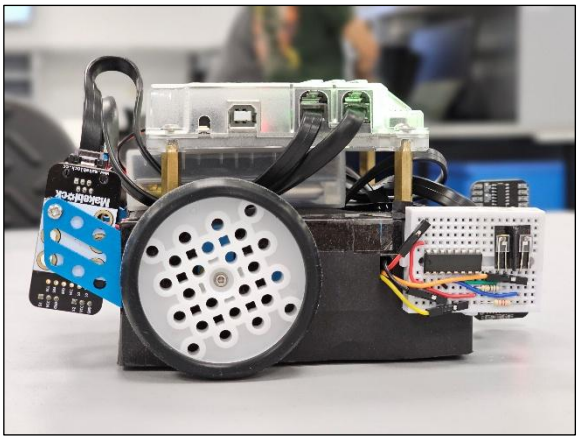
Top View



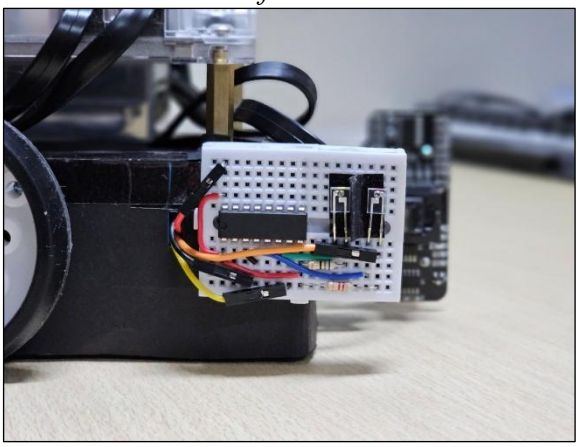
Bottom View



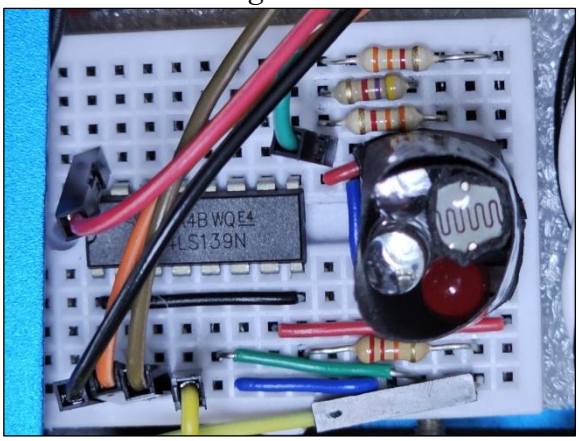
Left View



Right View



IR Sensor Circuit



Colour Sensor Circuit

Table 1: View of our mBot from Different Perspectives

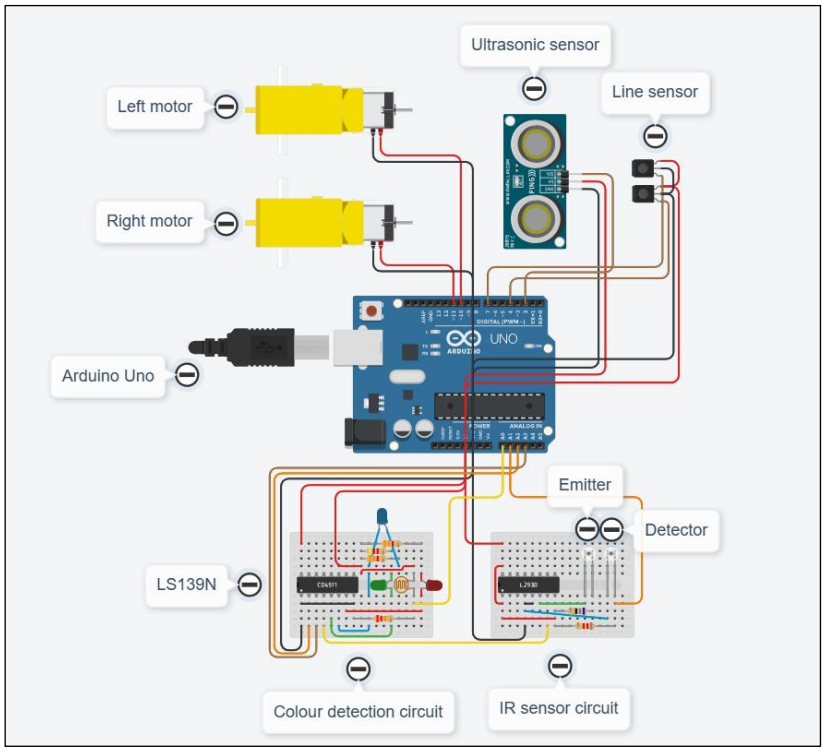


Figure 1: Entire Circuit of our mBot

3. Program Algorithm

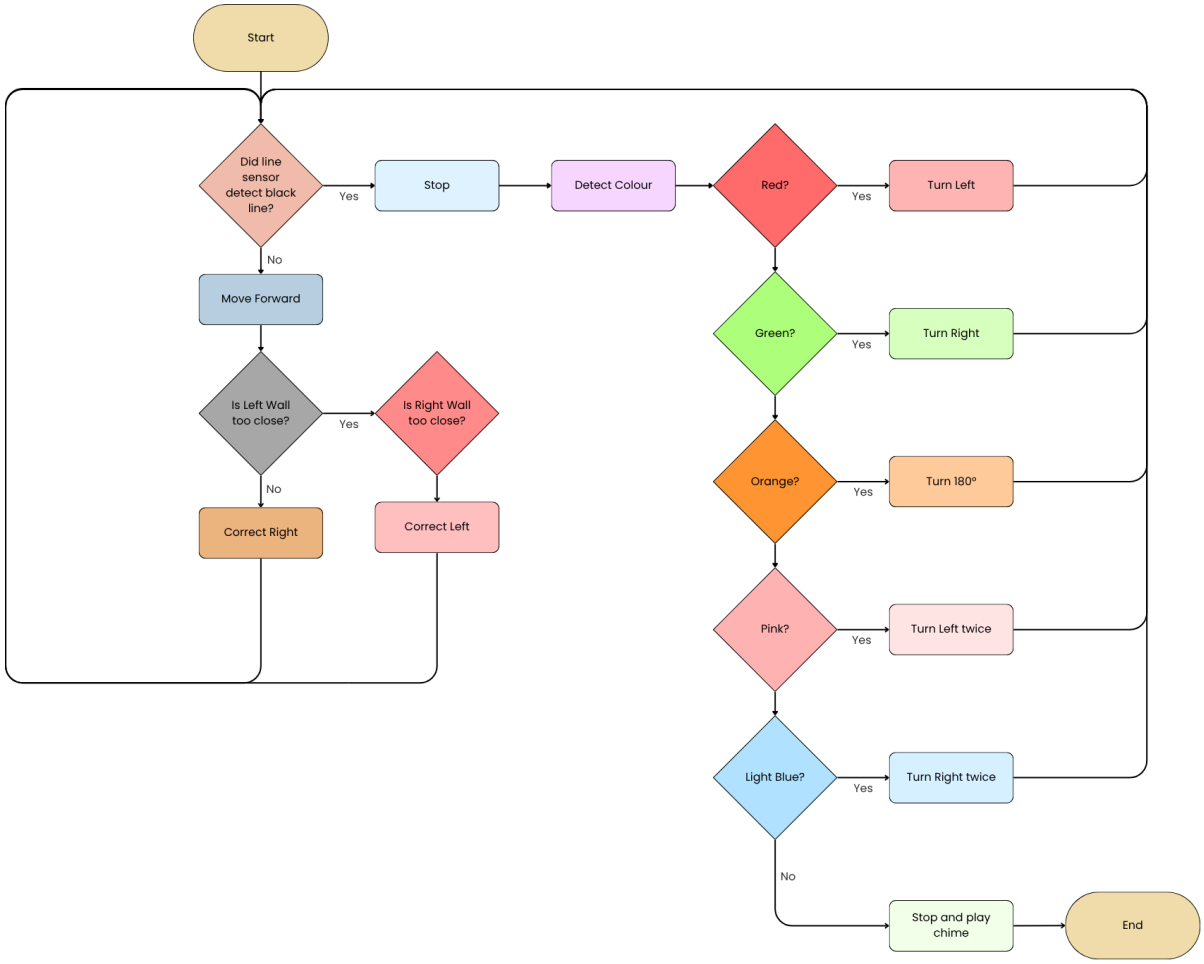


Figure 2: Algorithm Decision Flowchart

The figure above illustrates the overall algorithm governing the operation of the mBot. The primary functions of the robot are executed within a continuous loop, during which the mBot advances forward if the line sensor does not detect a black line. When this condition is no longer satisfied, the mBot activates its colour sensor to identify the colour of the surface beneath it and responds accordingly. If the detected colour is white, the mBot plays a victory chime, indicating the successful completion of the maze, and after which the program terminates. The detailed functions, decision-making processes, and implementation of individual program components will be further elaborated in the subsequent sections of this report.

4. Implementation of Subsystems

4.1 mBot Movement

This portion will cover the movement and motion algorithm of the mBot, including the forward movement, direction correction, and stopping at the end of each straight. These functions as the robot's navigation system and ensure smooth and reliable traversal of the maze.

4.1.1 Moving Straight

```
/**
 * This function is a general movement function used to move robot forward.
 */
void move(int L_spd, int R_spd) {
  if (stop == false) {
    leftWheel.run(-L_spd);
    rightWheel.run(R_spd);
  }
  else {
    stopMove();
  }
}
```

Figure 3: Move Function

Firstly, we tested the two DC motors and discovered that even though they are identical, there are slight variations in the torque output. As such, it was difficult to match the speed of both motors for the mBot to move in a completely straight line. This is most likely due to manufacturing tolerances in the motors, leading to minute differences in components such as the magnets, brush and commutator quality and bearing friction. This leads to the motors each having slightly different speed-voltage characteristics at the same duty cycle.

To solve this problem, we had to find the deviations between the speed of both motors. In our case, the deviation value required for straight movement was around **–15** for the right motor. The constant is factored into the functions for the forward movement of the mBot, as shown below.

```
// run mBot only if status is 1
if (status) {
  ultrasound();             // update global variable dist
  if (!on_line()) {         // check if on black line
    if (dist != OUT_OF_RANGE)  {
      // wall present, run pd_control
      led.setColor(255, 255, 255);
      led.show();
      pd_control();
    }
    else {
      // Re-initialise previous error to zero to prevent past interference if ultrasonic sensor goes out of range
      prev_error = 0; // no wall present on leftside, call IR to check right side (nudge left if we determine robot is too close to wall on right)
      bool adjustmentMade = checkRight();
      //Only move forward if no adjustment was made
      if (!adjustmentMade) {
        led.setColor(0, 255, 255);
        led.show();
        // Move forward
        move(MOTORSPEED, 240);
      }
    }
  }
```

Figure 4: Difference in motor speed

If this system fails, other measures are implemented to ensure that the mBot moves in a straight line, which we will discuss in a following section.

4.1.2 Infrared Sensor

4.1.2.1 IR Circuit Design

The Infrared Sensor is mounted on the right side of the mBot, it consists of 2 main components – the IR emitter which emits IR rays that bounce off surfaces and the IR detector that detects the IR rays to determine the distance between the wall surface and the mBot. The main purpose of this circuit is to prevent the mBot from colliding with the maze walls on the right.

ABSOLUTE MAXIMUM RATINGS AT TA=25℃

PARAMETER	MAXIMUM RATING	UNIT
Power Dissipation	75	mW
Peak Forward Current (300pps, 10 μs pulse)	1	A
Continuous Forward Current	50	mA
Reverse Voltage	5	V
Operating Temperature Range	-40℃ to + 85℃	
Storage Temperature Range	-55℃ to + 100℃	
Lead Soldering Temperature [1.6mm(.063") From Body]	260℃ for 5 Seconds	

Table 2: LTE-302 Absolute Maximum Ratings

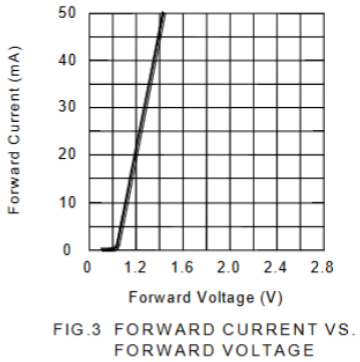


Figure 5: Forward Current vs Forward Voltage, LTE-302

According to the **LTR-302 datasheet**, as shown in the Table and Figure above, the maximum continuous forward current is specified as **50 mA**, with a corresponding forward voltage of approximately **1.6 V**. Based on these specifications, the minimum series resistor required is calculated to be **68 Ω**, as shown below:

Finding the voltage across V_R,

$$V_R = (5 - 1.6) = 3.4\,V$$

Using $V = IR$,

$$R_{emitter} > \frac{V_R}{I} > \frac{3.4}{50 \times 10^{-3}} > 68\Omega$$

After several tests, the final resistor value for the emitter was determined to be **68 Ω**, as it provided the optimal balance between brightness and sensor responsiveness. As for the resistor value of the IR detector, we decided to go with **8.2 kΩ**, as it provided the best balance between sensitivity and detection range.

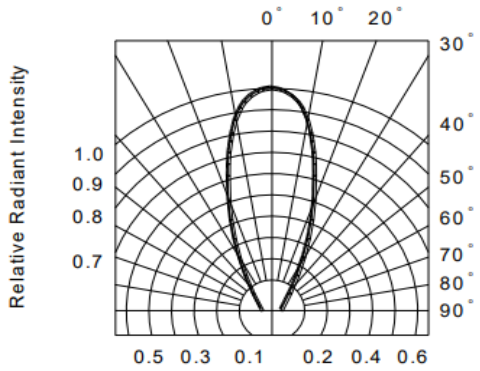


Figure 6: Radiation Diagram, LTE-302

The figure above shows the radiation diagram of the IR emitter. As seen in the diagram, the power is heavily concentrated in a narrow, forward-facing beam. To maximize the intensity of the light reflected from the maze walls back to the detector and minimize interference from ambient light, we mounted the emitter and detector close to each other with a black spacer in between. The black spacer is made using a piece of black paper folded to create a protruded wall which could be simply slotted behind the IR emitter and detector, a simple yet effective device.

The images below show the final IR circuit design:

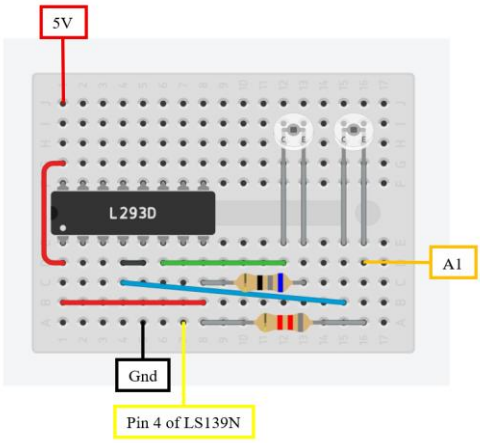


Figure 7: Final IR Circuit Design (Digital)

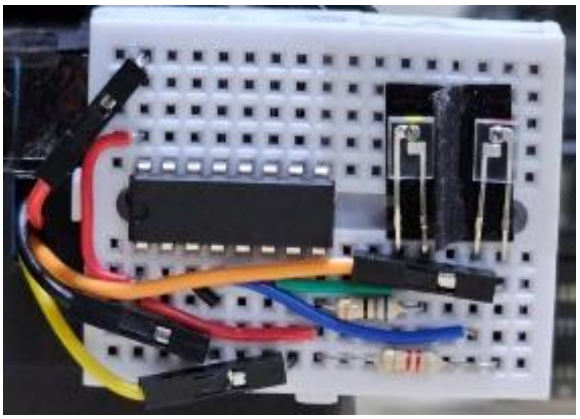


Figure 8: Final IR Circuit Design (Physical)

4.1.2.2 IR Reading Principles

In this section, we will discuss the working principles of the IR sensor. The IR sensor in this circuit operates on the principle of ambient light subtraction to ensure reliable readings, even in different lighting conditions.

Our custom IR function `getRightCorrection()` is shown below.

```
int getRightCorrection() {
  int localAmbientIR;
  int irVolt;
  int difference;

  // 1. Turn OFF IR Emitter to read ambient light
  // The OFF state is {1, 1} according to your comments
  for(int i = 0; i < 2; i++) {
    digitalWrite(ledArray[i], 1);
  }
  delay(IRWait); //Wait for sensor to settle
  localAmbientIR = analogRead(IR);

  // 2. Turn ON IR Emitter to read reflected light
  // The ON state must be {0, 0}
  for(int i = 0; i < 2; i++) {
    digitalWrite(ledArray[i], 0);
  }
  delay(IRWait);
  irVolt = analogRead(IR);

  // 3. Turn OFF IR Emitter (Good practice)
  for (int i = 0; i < 2; i++) {
    digitalWrite(ledArray[i], 1);
  }

  // 4. Calculate the difference
  difference = irVolt - localAmbientIR;
```

Figure 9: getRightCorrection() Function

Firstly, an analogue reading is taken with the IR detector without any IR emission. This value represents the ambient IR reading from the surroundings with sources such as sunlight or artificial lights.

A `delay(IRWait)` function is then executed immediately after toggling the emitter. This delay is calibrated to **20ms** and is critical for ensuring measurement accuracy. It provides the necessary sensor settling time for the IR detector's circuit to stabilize at its new voltage level after the IR light changes. Reading the analogue pin before stabilizing would result in inaccurate readings for both ambient and reflected light.

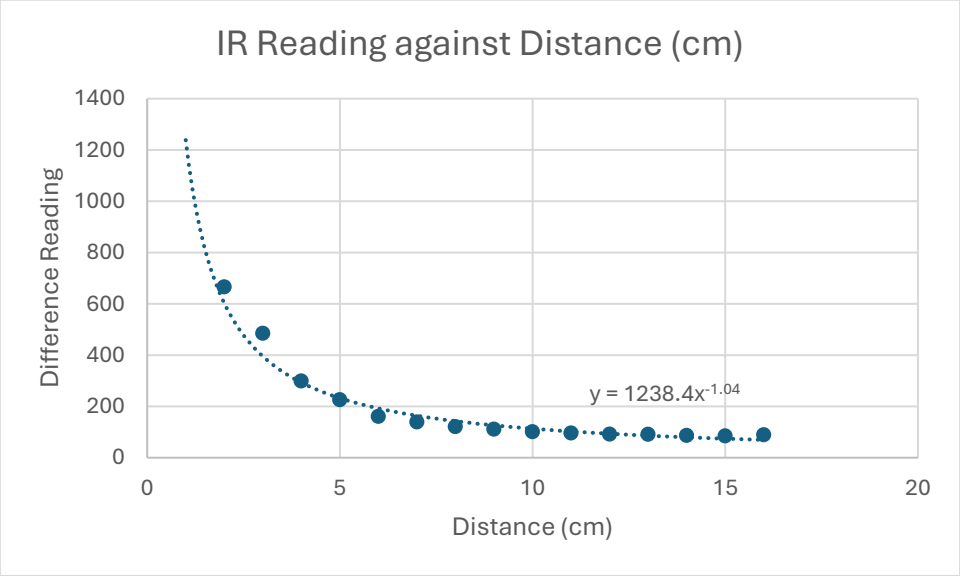
After attaining the ambient reading, the IR emitter immediately emits, sending out a strong IR pulse. The same reading process is used here to obtain a second reading (`irVolt`). This value is the combined value of the signal + noise (ambient noise and IR light from the IR emitter). The final reading can then be calculated using the formula:

$$difference = irVolt - localAmbientIR$$

This calculation subtracts the noise from the signal + noise, leaving only the pure signal from the IR emitter that was reflected back to the mBot, hence representing the distance between the wall and IR sensor.

4.1.2.3 IR Distance Calibration

To determine the relationship between IR readings and distance, we characterised our sensor using the same methodology as Studio 12. Due to our specific sensor configuration, a higher intensity of IR light reflected results in a higher analogue reading. The results can be seen in the graph as follows:



Graph 1: IR Reading against Distance

Based on the graph, when there is no wall, the IR reading is low, similar to the `localAmbientIR`. By analysing the sensor's response curve, we can see that the IR reading increases exponentially when below a certain distance. Above 6cm, the readings are all relatively similar, hovering around **85 and 121** based on our data. However, below 6cm, the reading magnitude increases sharply, from **161 at 5cm to 666 at 1cm**.

The elbow at the **6cm** mark (magnitude **139**) makes it an extremely reliable threshold. By setting our `IR_THRESHOLD` to **140**, we can confidently distinguish between a distant wall and a nearby obstacle, eliminating false triggers from distant objects. 6cm would also act as an optimal safety buffer, as it is far enough to allow the mBot to react effectively before a collision.

4.1.3 Ultrasonic Sensor

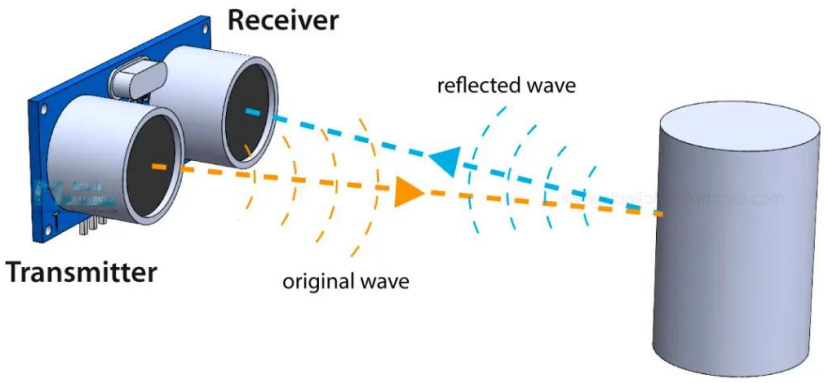


Figure 10: Ultrasonic Sensor Working Principle

The ultrasonic sensor is a primary component of our obstacle avoidance system. It is mounted on the left side of the mBot to continuously measure the distance to the left wall, which the mBot uses to navigate and straighten itself.

The sensor operates on the principle – time-of-flight. It emits a high-frequency sound pulse (ultrasonic) using its **TRIG pin** and measures the time taken for the pulse's echo to return using its **ECHO pin**.

$$Distance\ (cm) = \frac{Time\ of\ Flight\ (\mu s)\ \times\ Speed\ of\ Sound\ \left(\frac{cm}{\mu s}\right)}{2}$$

The speed of sound is approximately **0.0343 cm/µs**. The total time recorded reflects the time taken for the sound pulse to travel to and from the obstacle, hence to the calculated value is divided by 2 as shown in the formula.

Our `ultrasound()` function uses a preexisting library from the MeMCore library, developed for the mBot to control the ultrasonic sensor. The function calls `ultraSensor.distanceCm()` to get a raw distance reading. The reading is then compared with `SIDE_MAX`, which we defined as **18cm** as the threshold in which the wall is counted as not present. If the distance measured is greater than this value, the `dist` variable will be set to the `OUT_OF_RANGE` flag value (100, but any value will work). This filtered `dist` variable will be passed to the wall-avoidance algorithm, which we will cover in a later portion.

```
/********** Functions (Ultrasound & Line Sensor) **********/

/**
 * This function updates the global variable dist to current distance between the ultrasonic sensor and the closest object (wall) to it.
 */
void ultrasound() {
  dist = ultraSensor.distanceCm();
  if (dist > SIDE_MAX)
  {
    dist = OUT_OF_RANGE;
  }
}
```

Figure 11: ultrasound() Function

4.1.4 Wall Avoidance

To ensure that the mBot continues to move through the course without hitting any obstacles, a wall avoidance algorithm is implemented using a combination of an IR sensor and ultrasonic sensor. As shown in the Design of mBot section, the IR sensor is positioned on the right side of the mBot, while the ultrasonic sensor is positioned on the left side. Through our testing, we found that the ultrasonic sensor provided a cleaner, and more stable distance reading than the IR sensor, hence being more reliable. Based on this test, we prioritize its use in our control logic.

We explored two different versions of the algorithm, both utilising a **Proportional-Derivative (PD)** algorithm for steering control.

4.1.4.1 PD Control Algorithm

Before we get into the different iterations of the wall avoidance algorithm, we will first talk about the core of the algorithm, the Proportional-Derivative (PD) controller. This is a closed-loop feedback mechanism that allows the mBot to steer smoothly, rather than using a simple hardcoded swerve which can lead to unstable and jerky movement.

The PD algorithm calculates a `totalCorrection` value based on two components, the Proportional, referring to how far we are from the target, and the Derivative, referring to how fast we are approaching or leaving the target. This final correction value is then used to adjust the motor speeds.

Proportional (P) Component

The P-component provides the primary steering force. It reacts proportionally to the current error: how far the mBot is from its target distance.

$$Error = Desired\ Distance - Current\ Distance$$

$$P = k_p \times Error$$

The higher the kp value, the more aggressive the mBot reacts to any deviation. For example, if the kp value = **40**, and the mBot drifts **2cm** away from the wall **(Error = -2.0)**, the P-component would provide a strong correction of **-80**, forcing a sharp turn back towards the wall.

Derivative (D) Component

The D-component provides damping and stability. It reacts to the rate of change of the error, effectively predicting if the mBot will overshoot its target.

$$Error\ Delta = Error - Previous\ Error$$

$$D = k_d \times Error\ Delta$$

If the mBot turns too quickly towards the wall, its error changes rapidly. This results in a large Error Delta, and the D-component provides a counter-correction to slow the turn, preventing overshooting and the oscillating behaviour from the mBot. A lower kd value would provide higher dampening force to stabilize the aggressive turns from a high kp value.

Final Correction and Motor Control

The final steering command is the sum of the two components:

$$Total\ Correction = P \times D = \left(k_p \times Error\right) + (k_d \times Error\ Delta)$$

This totalCorrection value is then applied to the motors using differential steering. A positive correction steers the bot right, and a negative correction steers it left.

$$L\ Motorspeed = MOTORSPEED + totalCorrection$$

$$R\ Motorspeed = MOTORSPEED - totalCorrection$$

where MOTORSPEED is **[255]**. This PD system will form the basis of the following implementations of our wall-avoidance algorithm.

4.1.4.2 Version 1: Ultrasonic Main with IR Fallback

The flowchart for Version 1 of the wall avoidance algorithm is shown below in the following figure.

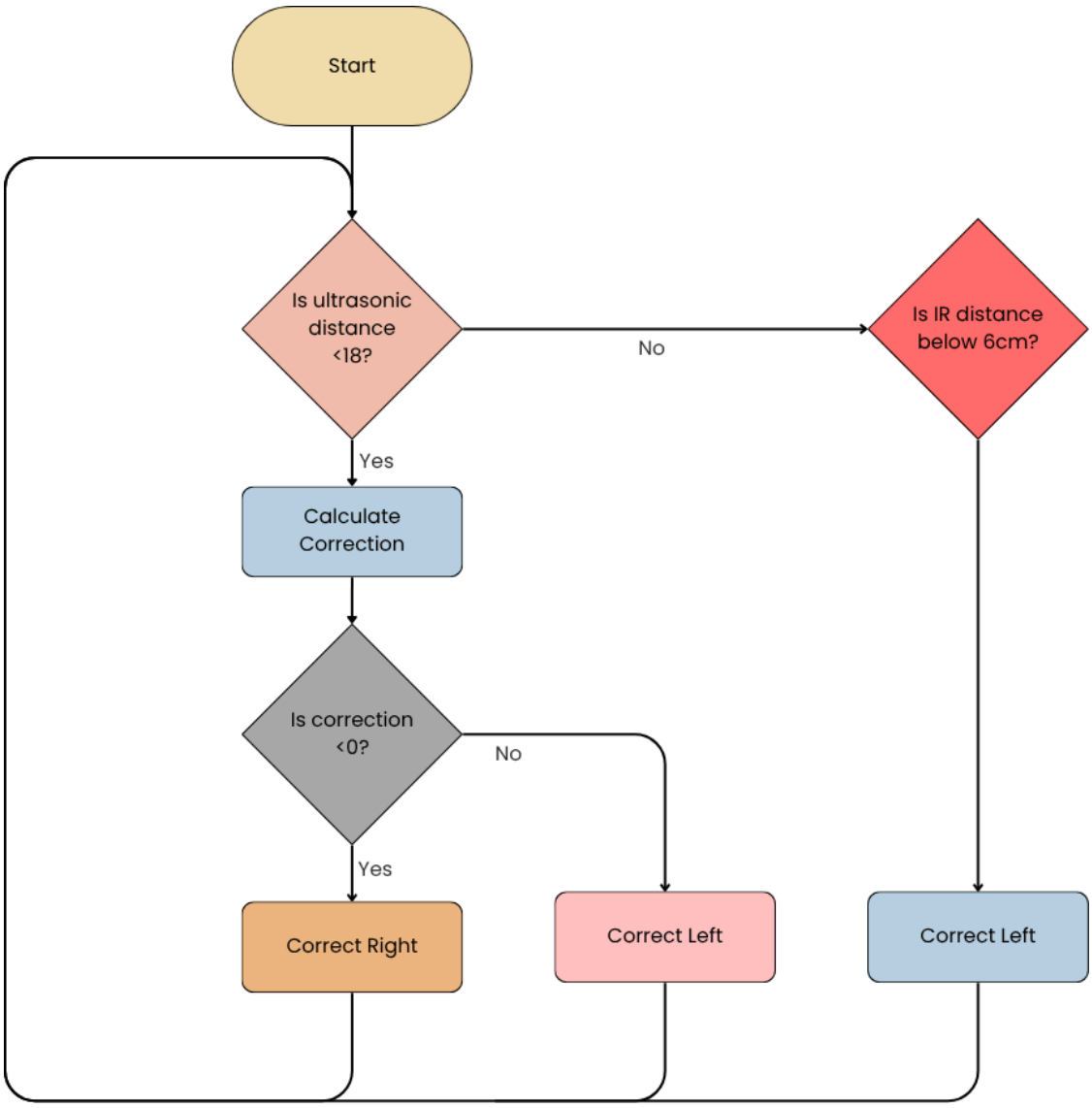


Figure 12: Version 1 Algorithm Flowchart

The first version of the algorithm established a clear sensor hierarchy as follows:

```
// run mBot only if status is 1
if (status) {
  ultrasound();             // update global variable dist
  if (!on_line()) {         // check if on black line
    if (dist != OUT_OF_RANGE)  {
      // wall present, run pd_control
      led.setColor(255, 255, 255);
      led.show();
      pd_control();
    }
    else {
      // Re-initialise previous error to zero to prevent past interference if ultrasonic sensor goes out of range
      prev_error = 0; // no wall present on leftside, call IR to check right side (nudge left if we determine robot is too close to wall on right)
      bool adjustmentMade = checkRight();
      //Only move forward if no adjustment was made
      if (!adjustmentMade) {
        led.setColor(0, 255, 255);
        led.show();
        // Move forward
        move(MOTORSPEED, 240);
      }
    }
  }
```

Figure 13: Sensor Hierarchy

1. **Primary Mode**: The code first checks the ultrasonic sensor's distance reading. If the reading is within the working range of **<=18cm**, the system enters the primary wall following mode. In this mode, the Proportional-Derivative (PD) algorithm is executed. Where, The PD algorithm tries to maintain the constant desired distance of **12.5cm** from the left wall by applying differential speed corrections, using the formulas shown below.

$$L\_Motorspeed = Base + Correction$$
$$R\_Motorspeed = Base - Correction$$

```
/*
 * Function used P and D components of PID to calculate the new PWM values for left motor and right motor.
 * Error is the difference between the robot's current position and the desired position of the robot.
 * After finding the error, we multiply the error by the proportional gain which helps us minimise this error.
 * Change of error (error_delta) helps us prevent oscillation by adding "damping" effect to our correction.
 */
void pd_control() {
  error = desired_dist - dist;                    // P Component of PID
  error_delta = error - prev_error;               // D Component of PID
  correction_dble = (kp * error) + (kd * error_delta);
  correction = (int)correction_dble;

  // Determine direction of correction and execute movement
  if (correction < 0) {
    L_motorSpeed = 255 + correction;
    R_motorSpeed = 255;
  } else {
    L_motorSpeed = 255;
    R_motorSpeed = 255 - correction;
  }
  move(L_motorSpeed, R_motorSpeed);

  //Initialise current error as new previous error (D Component of PID)
  prev_error = error;
}
```

Figure 14: PD Algorithm Using Ultrasonic Sensor Readings

2. **IR Fallback Mode**: If the mBot is detected more than **18cm** away from the left wall (ultrasonic side), the algorithm automatically falls back to utilising the IR sensor. In this mode, the IR sensor operates as a simple safety check. If the IR sensor detects a nearby wall or obstacle (indicated by a difference reading of less than threshold of **–140**, which is approximately **6cm**), it applies a fixed nudge value of **–40** to the left wheel while maintaining the speed of the right wheel.

```
bool checkRight() {
  int localAmbientIR;
  int irVolt;
  int difference;

  for(int i = 0; i < 2; i++) { //Turn off IR emitter to read ambient light
    digitalWrite(ledArray[i], 0);
  }
  delay(IRWait); //Wait for sensor to settle
  localAmbientIR = analogRead(IR);
  int irVolt = analogRead(IR); //Measure voltage across IR Detector

  for(int i = 0; i < 2; i++) { //Turn on IR emitter to read reflected light
    digitalWrite(ledArray[i], 1);
  }
  delay(IRWait);
  irVolt = analogRead(IR);

  for (int i = 0; i < 2; i++) { //Turn off IR emitter (Good practice)
    digitalWrite(ledArray[i], 1);
  }

  difference = irVolt - localAmbientIR; //int difference = ambientIR - irVolt;
  // --- Debugging ---
  Serial.print("Ambient: ");
  Serial.print(localAmbientIR);
  Serial.print(", irVolt: ");
  Serial.print(irVolt);
  Serial.print(", Difference: ");
  Serial.println(difference);
  // -----------------

  if (difference > IR_THRESHOLD)
  {
    // Nudge left
    led.setColor(0, 255, 255);
    led.show();
    move(215,255);
    Serial.print("Nudging left...");
    delay(150);
    return true;
  }
  return false;
}
```

Figure 15: Fallback Mechanism with Left Nudge

However, although this version offered a smoother trajectory when following a continuous wall due to the dedicated ultrasonic PD control and faster response time due to only using one sensor at a time, it suffered from a crucial flaw: **unresponsiveness to obstacles on the right** (IR side). As the IR sensor was relegated to a fallback mechanism, the mBot had no immediate reaction to any unforeseen obstacle on the right side while actively tracking the left wall (ultrasonic side). This was deemed unacceptable as it did not meet the project requirements.

4.1.4.3 Version 2: Integrated Control with IR Override

The second and final version resolves the drawbacks of the first by employing a control system that preserves the original PD steering precision while adding the continuous right-side obstacle avoidance.

The flowchart for Version 2 is as shown below.

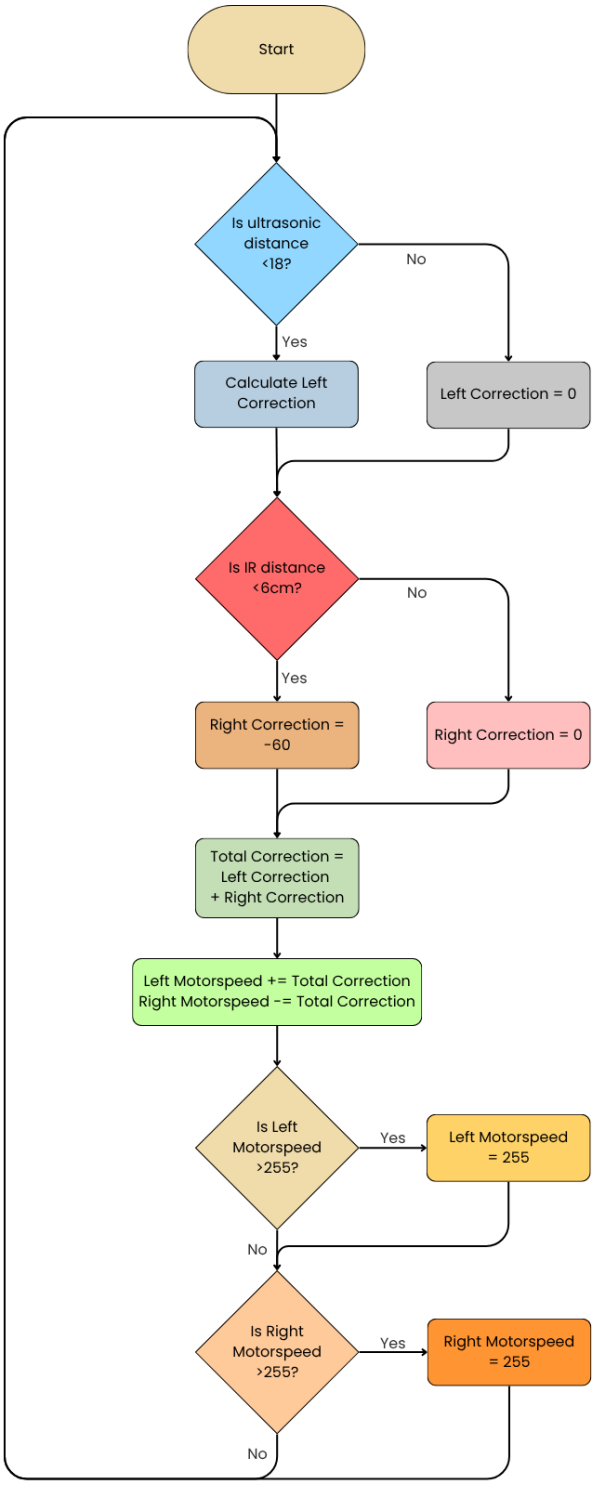


Figure 16: Version 2 Algorithm Flowchart

The new structure removes the conditional sensor hierarchy from the original version and instead consolidates the sensor inputs before calculating the motor output.

1. **PD Correction (Primary Steering)**: The `pd_control_wall_follow()` function is called every loop iteration, regardless of the ultrasonic and IR readings. It calculates the precise steering value based on the ultrasonic sensor's distance error. This provides the continuous, high precision steering needed to maintain the desired distance of **12.5cm** from the left wall.

```
void loop()
{
  // run mBot only if status is 1
  if (status) {
    if (!on_line()) {       // check if on black line

      // --- NEW LOGIC ---
      // 1. Get PD steering correction from the left wall (Ultrasonic)
      int pdCorrection = pd_control_wall_follow();

      // 2. Get avoidance correction from the right wall (IR)
      // This function returns RIGHT_WALL_NUDGE (negative) if too close, or 0 otherwise.
      int irCorrection = getRightCorrection();

      // 3. Combine corrections
      // The IR correction is directly added to the PD correction.
      // If IR nudges left (e.g., -60), it overrides or strengthens the turn.
      int totalCorrection = pdCorrection + irCorrection;

      // 4. Apply the total correction
      // Base speed is MOTORSPEED (e.g., 255)
      L_motorSpeed = MOTORSPEED + totalCorrection;
      R_motorSpeed = MOTORSPEED - totalCorrection;

      // 5. Clamp speeds
      if (L_motorSpeed < 0) L_motorSpeed = 0;
      if (R_motorSpeed < 0) R_motorSpeed = 0;
      if (L_motorSpeed > 255) L_motorSpeed = 255;
      if (R_motorSpeed > 255) R_motorSpeed = 255;
      // 6. Final move command
      move(L_motorSpeed, R_motorSpeed);
      // --- END OF NEW LOGIC ---
```

Figure 17: Final Version of PD Algorithm

2. **IR Correction (Override)**: The `getRightCorrection()` function is also called every loop iteration which acts as a strong override switch. If the IR sensor detects an obstacle close by (difference reading of less than **–140**), it returns a strong negative correction value of **–60** (increased from **-40** before), instantly forcing a left turn. If no obstacle is present, **it returns the value [0]**.

```
int getRightCorrection() {
  int localAmbientIR;
  int irVolt;
  int difference;

  // 1. Turn OFF IR Emitter to read ambient light
  // The OFF state is {1, 1} according to your comments
  for(int i = 0; i < 2; i++) {
    digitalWrite(ledArray[i], 1);
  }
  delay(IRWait); //Wait for sensor to settle
  localAmbientIR = analogRead(IR);

  // 2. Turn ON IR Emitter to read reflected light
  // The ON state must be {0, 0}
  for(int i = 0; i < 2; i++) {
    digitalWrite(ledArray[i], 0);
  }
  delay(IRWait);
  irVolt = analogRead(IR);

  // 3. Turn OFF IR Emitter (Good practice)
  for (int i = 0; i < 2; i++) {
    digitalWrite(ledArray[i], 1);
  }

  // 4. Calculate the difference
  difference = irVolt - localAmbientIR;
  // 5. Check for a wall
  // (because irVolt goes DOWN near a wall)
  // You may need to adjust this "-100" threshold.
  if (difference < -140)
  {
    // Wall detected! Return the nudge value.
    // Serial.print("Nudging left...");
    return RIGHT_WALL_NUDGE;
  }

  // No wall detected
  return 0;
}
```

Figure 18: IR Sensor Algorithm

3. Consolidated Motor Control: The final motor speeds are calculated by directly summing the two corrections:

$$totalCorrection = pdCorrection + irCorrection$$

If the `pdCorrection` is positive (turn right) but the `irCorrection` is strongly negative (turn left), the negative IR value dominates, ensuring the robot turns away from the immediate threat on the right. This maintains the responsiveness needed for unforeseen obstacles while running the PD algorithm.

The combination of the continuous ultrasonic PD steering with an immediate IR safety override results in a much more robust and reliable robot. This design achieves the required precision of wall-following while guaranteeing instantaneous obstacle avoidance on the opposite side, addressing the critical flaw in Version 1.

While Version 2 achieves our goal of continuous obstacle avoidance, it introduces a performance trade-off in the form of increased processing overhead. The `loop()` must now sequentially execute `pd_control_wall_follow()` and `getRightCorrection()`, each containing its own sensor read operations which block code execution for a short time (`delay(IRWait)`).

This constant polling of both sensors in every cycle, as opposed to the conditional polling in Version 1, adds latency to the system's control loop. Although this theoretically could cause issues in the responsiveness of the mBot, our testing confirmed that the total loop time remains well within acceptable limits.

4.2 mBot Waypoint Challenge

4.2.1 Line Sensor

Figure 19: MakeBlock Line Sensor

A Makeblock line sensor is mounted beneath the mBot, positioned in front of its mini caster wheels. The sensor consists of two pairs of infrared (IR) transmitters and receivers. Each pair provides a Boolean output indicating whether it is “IN” (on the black line) or “OUT” (off the black line). This functionality is achieved through the continuous emission of infrared light. Since black surfaces absorb infrared radiation, the absence of reflected IR detected by the receiver signifies that the mBot is positioned over a black line, prompting the sensor to return an “IN” signal.

4.2.2 Colour Sensor

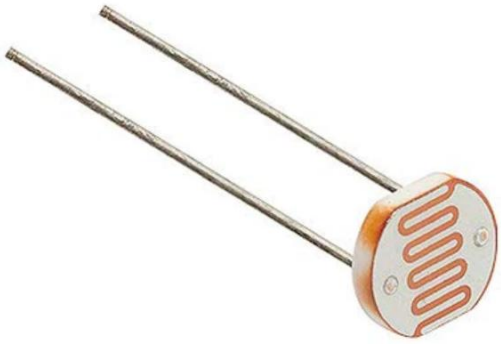


Figure 20: LDR Used for the Colour Sensor

The colour sensor is mounted on the underside of the mBot to detect the colours of waypoints, allowing it to execute the corresponding operations necessary for navigating and completing the maze. The sensor operates by sequentially flashing its three primary colour LEDs – red, blue and green, and utilising a light-dependent resistor (LDR) to measure the amount of reflected light from each colour channel. By comparing the reflected intensities of these primary colours, the sensor determines the colour of the surface beneath it.

4.2.2.1 Colour Sensor Circuit

According to the HD74LS139 Dual 2-Line-to-4-Line Decoder datasheet, the maximum allowable output current is specified as **8 mA**. Based on this specification, the corresponding resistor values for each of the RGB LEDs can be calculated. With reference to the LED datasheets, the corresponding forward voltages are **1.88 V** for the red LED and estimated value of **1.7 V** for the blue and green LEDs, respectively.

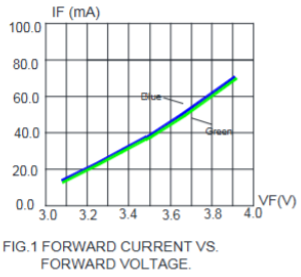


Figure 21: Forward Current vs Forward Voltage, Blue and Green RGB

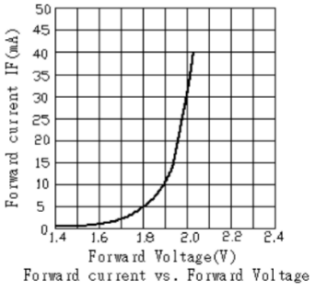


Figure 22: Forward Current vs Forward Voltage, Red RGB

$$V_R = 5 - 1.88 = 3.12\,V$$

$$V_R = 5 - 1.7 = 3.3\,V$$

Using $V = IR$,

$$R_{red} = \frac{3.12}{8 \times 10^{-3}} = 390\Omega$$

$$R_{Blue\&Green} = \frac{3.3}{8 \times 10^{-3}} = 412.5\Omega$$

After multiple tests, the final resistor values were set to **3.3 kΩ** for the Red and Blue LEDs, and **4.7 kΩ** for the Green LED, as this combination provided the best balance between brightness and sensor responsiveness.

In a well-lit environment, the LDR exhibits a resistance of approximately **30 kΩ**. To achieve an optimal voltage divider output for Arduino input, the series resistor was selected as **22 kΩ**, replacing the initial **100 kΩ** used in previous setups. This configuration ensures the potential difference across the LDR falls within the Arduino's analogue input range, improving sensor responsiveness.

The images below show the final colour sensor circuit design:

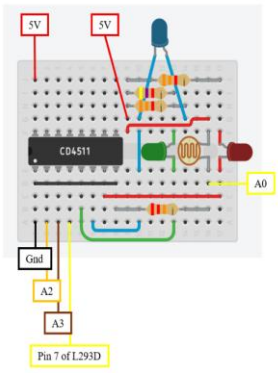


Figure 23: Final Colour Sensor Circuit Design (Digital)

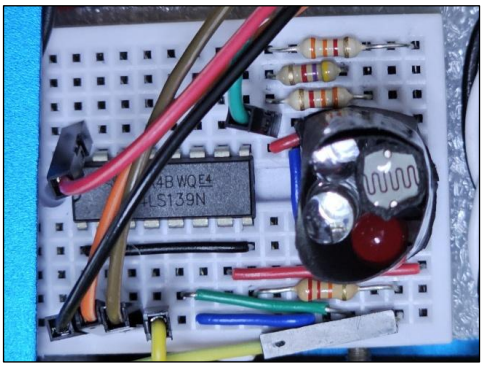


Figure 24: Final Colour Sensor Circuit Design (Physical)

4.2.2.2 Colour Sensor Physical Design

To ensure accurate and repeatable waypoint detection, the LDR must only measure light from the dedicated LEDs reflecting off the surface of the coloured waypoint. Any ambient or direct light from the surroundings would contaminate the readings and make them unreliable resulting in incorrect colour recognition.

Hence. to achieve reliable colour recognition and optimal light isolation, we implemented a multi-layer shielding solution.

Component Shielding

The LDR and three LEDs are placed within close proximity with each other. To prevent light travelling directly from the LED to the LDR, we surrounded the LDR with a narrow black tube. This shielding ensures that only light reflected off the surface of the waypoint reaches the LDR.

A second, wider and taller black tube is also placed around the entire sensor array (LDR + LEDs). This tube extends all the way down as close to the ground as possible, creating a dark, enclosed chamber where little external light can enter, ensuring only the light reflected off the surface is measured.

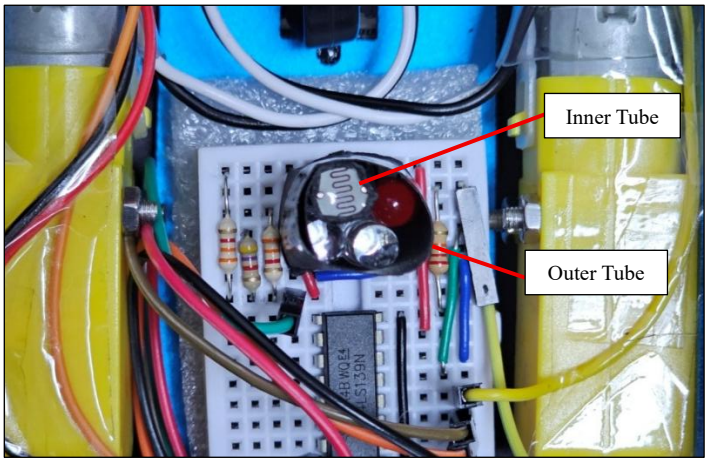


Figure 23: Component Shielding of the LDR and LEDs

Chassis-Level Shielding

To further prevent general light leakage from contaminating the LDR reading from above or through the multiple holes in the mBot chassis, we constructed a "skirt" from the same black material. This skirt encircles the entire chassis, similar to a car's side skirt, and reaches close to the ground. The skirt also extends up and covers all exposed areas on the mBot chassis, leaving only tiny holes for the wiring to pass through. This final layer blocks almost all remaining light from entering the system, ensuring that the colour sensor works regardless of external lighting conditions.

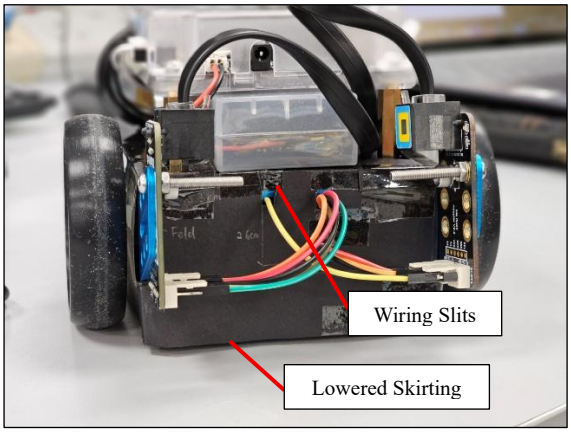


Figure 24: Skirting Extended to Ground and Wiring

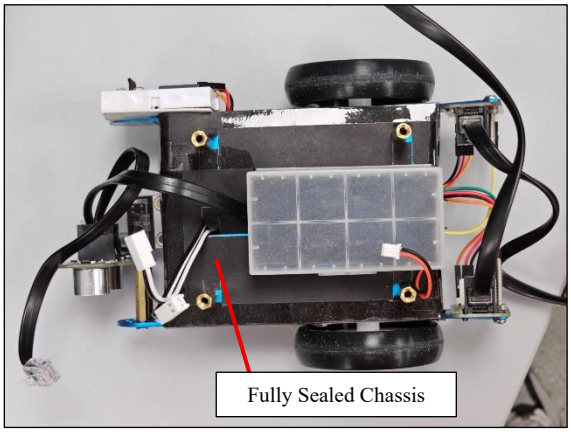


Figure 25: Covering All Potential Gaps in Chassis

This comprehensive, three-layered shielding design was critical to the reliability of our colour classification algorithm, as it provided the clean and consistent readings necessary for normalization.

4.2.2.3 Colour Sensor Algorithm

Below is the flowchart for the colour classification.

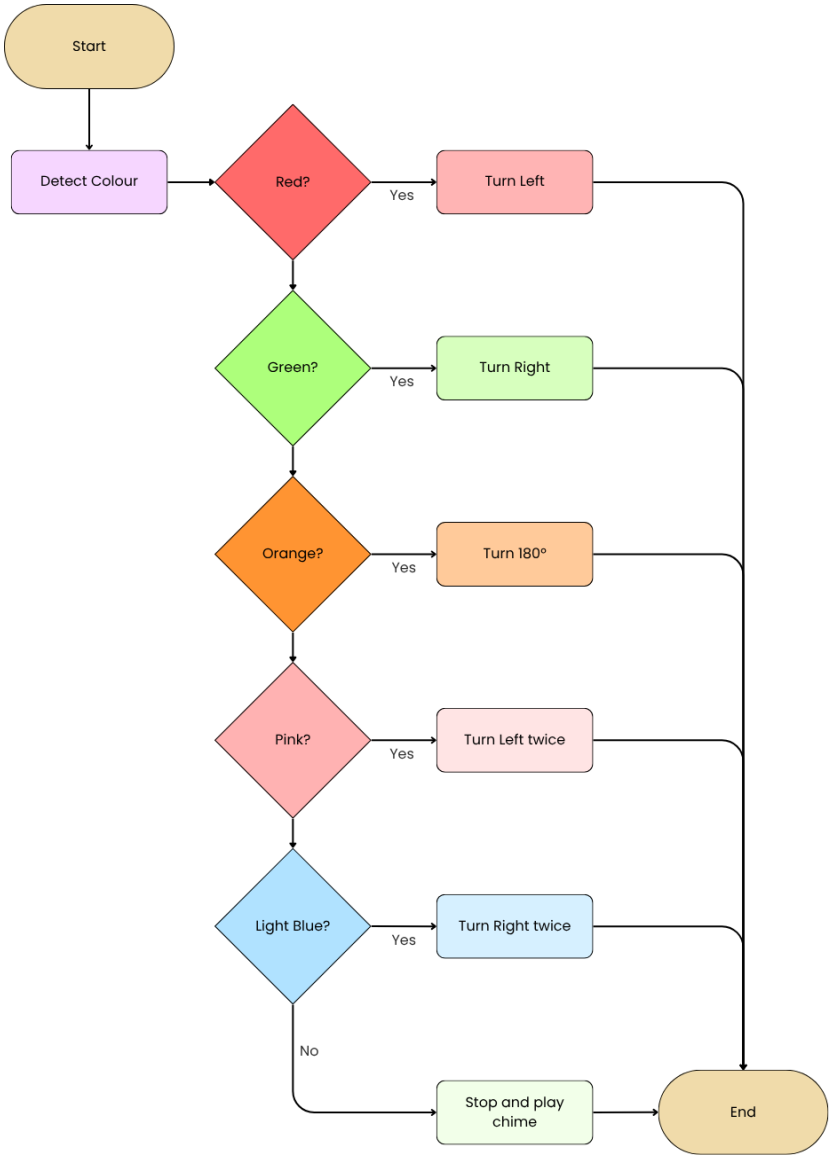


Figure 26: Colour Classification Flowchart

The colour sensor algorithm consists of mainly 2 functions: `read_color()` & `classify_color()`.

```
/********** Functions (Color Detection) **********/

/**
 * Function turns on one colour at a time and measure LDR voltage for each colour to estimate respective R/G/B values.
 */
void read_color() {
  for(int c = 0; c <= 2; c++){
    // Serial.print(colourStr[c]);
    // TURN ON LIGHT
    for (int i = 0; i < 2; i++) {
      digitalWrite(ledArray[i], truth[c][i]);
    }
    delay(RGBWait);

    colourArray[c] = getAvgReading(3);

    int result = (colourArray[c] - blackArray[c])/(greyDiff[c])*255;
    if (result > 255) {
      result = 255;
    }
    else if (result < 0) {
      result = 0;
    }
    colourArray[c] = result;

    // TURN OFF LIGHT
     for (int i = 0; i < 2; i++) {
      digitalWrite(ledArray[i], 0);
    }
    delay(RGBWait);
    //Serial.begin(9600);
    // Serial.println(int(colourArray[c]));
  }

}
```

Figure 27: read_colour() Function

The `read_color()` function measures RGB intensity by sequentially enabling each LED through a 2-to-4 decoder according to the truth table. Reflected light readings from the LDR are normalized against calibrated white and black values and stored in `colourArray[]`. To improve accuracy, `getAvgReading()` averages three consecutive analogue readings, reducing noise and ensuring more stable results.

```
/**
 * This function finds the average reading of LDR for greater accuracy of LDR readings.
 */
int getAvgReading(int times) {
  int reading;
  int total = 0;
  for (int i = 0; i < times; i++) {
    reading = analogRead(LDR);
    total = reading + total;
    delay(LDRWait);
  }
  return total / times;
}
```

Figure 28: getAvgReading() Function

4.2.2.4 Colour Calibration

The mBot colour detection and classification relies on accurately detecting and classifying coloured patches on the maze floor. To achieve this, we must first normalize the raw readings to a standard 0-255 scale with pure black and white values.

The raw analogue readings from the LDR are highly dependent on the ambient light, LED brightness and mBot skirting. Using the same `read_colour()` and `getAvgReading()` functions from the previous section, we obtained the average raw reading for pure white paper and black paper, which represent the maximum and minimum light levels the sensor can detect. From these values, we calculated the sensor's dynamic range with the formula below, which we store as `greydiff[]`.

$$greydiff[c] = whiteArray[c] - blackArray[c]$$

When the mBot is running, the `read_colour()` function measures the current raw LDR value for each channel (RGB) and uses the formula below to normalize the reading, normalizing it and scaling it from its arbitrary range (e.g. 204-813) to a standard 8-bit RGB scale (0-255).

$$NormalizedValue = \frac{currentReading - blackArray[c]}{greydiff[c]} \times 255$$

Using this formula, we can measure the RGB readings for each of the tested colours as shown the table below.

Colour	**R Value**	**G Value**	**B Value**
Black	208	255	24
White	811	850	433
Red	255	114	104
Blue	71	222	246
Green	85	224	178
Orange	255	177	103
Pink	255	229	225

Table 3: Measured RGB Values for Each Colour

The RGB values for each colour will differ depending on the LDR's configuration and skirting setup, hence the recorded values may not necessarily translate to the actual colour in the RGB colour space.

4.2.2.5 Classify Colour

The normalized values are passed to the `classify_color()` function.

```
/**
 * Function looks at RGB values stored in colourArray[] and compare it with defined points for each color stored in colors[].
 */
int classify_color() {
  int classified = 6;
  measured_blue = colourArray[0];
  measured_green = colourArray[1];
  measured_red = colourArray[2];

  double minDistance = 9999;  // Initialize with a large value

  for (int i = 0; i < 6; i++) { // 6 is the number of known colors
    double distance = sqrt(pow(colors[i].red - measured_red, 2) + pow(colors[i].green - measured_green, 2) + pow(colors[i].blue - measured_blue, 2));
    if (distance < minDistance) {
      minDistance = distance;
      classified = colors[i].id;
    }
  }
  return classified;
}
```

Figure 29: classify_colour() Function

The `classify_color()` function determines the colour detected by the colour sensor by comparing the measured RGB values stored in `colourArray[]` with the predefined set of reference colour readings from before.

To quantify the similarity between the measured colour and each reference colour, the function applies the **Euclidean distance formula**:

$$d = \sqrt{(R_m - R_c)^2 + (G_m - G_c)^2 + (B_m - B_c)^2}$$

The function calculates this distance for each reference colour and identifies the one with the **smallest distance**, which represents the closest match to the colour detected. The corresponding colour ID is then returned as the final classification result. By using this method, the mBot can reliably recognize colours even when small variations in lighting or sensor readings occur, ensuring accurate responses during operation.

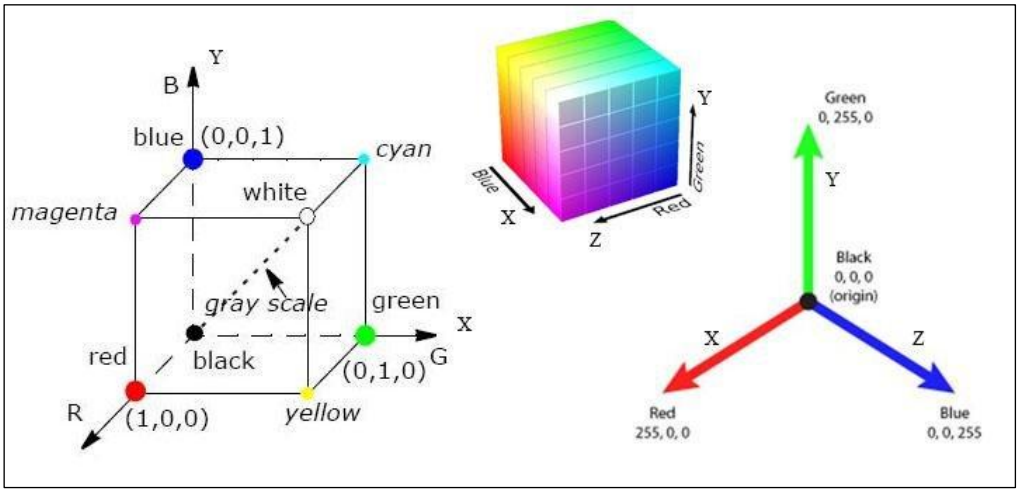


Figure 30: Colour Distribution of 3D RGB Colour Space

4.2.3 Waypoint Navigation and Execution

While the PD control system handles the navigation between waypoints, a separate system handles the execution of manoeuvres (turning) at the waypoints (coloured papers). This logic is triggered when the `on_line()` function detects a black line, at which point the mBot stops its PD wall following and enters a waypoint state.

The `execute_waypoint()` function acts as a simple state machine, using a switch statement to select the correct manoeuvre based on the colour id. Each case (0-5) corresponds to a specific colour and calls a unique movement function.

Colour	**ID**
White	0
Red	1
Blue	2
Green	3
Orange	4
Pink	5

Table 4: Colour ID Table

```
/**
 * Function takes in the color of waypoint classified by mBot and calls respective
 functions to move mBot to complete waypoint objective.
 */
void execute_waypoint(const int color)
{
  switch(color) {
  case 0:
    // code block for white
    // Serial.println("White Colour!");
    led.setColor(255, 255, 255); // set both LED to WHITE
    led.show();
    stop = true;
    status = false;
    break;
  case 1:
    // code block for red
    // Serial.println("Red Colour!");
    led.setColor(255, 0, 0); // set both LED to RED
    led.show();
    turnLeft(TIME_FOR_LEFT_TURN);
    break;
  case 2:
    // code block for blue
    // Serial.println("Blue Colour!");
    led.setColor(0, 0, 255); // set both LED to BLUE
    led.show();
    doubleRight(TIME_FOR_1_GRID_BLUE);
    break;
  case 3:
    // code block for green
    // Serial.println("Green Colour!");
    led.setColor(0, 255, 0); // set both LED to GREEN
    led.show();
    turnRight(TIME_FOR_RIGHT_TURN);
    break;
  case 4:
    // code block for orange
    // Serial.println("Orange Colour!");
    led.setColor(153, 76, 0); // set both LED to ORANGE
    led.show();
    uTurn();
    break;
  case 5:
    // code block for pink
    // Serial.println("Pink Colour!");
    led.setColor(255, 192, 203); // set both LED to PINK
    led.show();
    doubleLeft(TIME_FOR_1_GRID_PINK);
    break;
  default:
    // code block for no color classified
    // Serial.println("No Colour!");
    led.setColor(0, 0, 0); // set both LED to OFF
    led.show();
    break;
  }
}
```

Figure 31: execute_waypoint() Function

All waypoint-based manoeuvres are performed using a method called **dead reckoning**, which means they rely on timed movements rather than sensor feedback. The core manoeuvres are shown in the table below:

Function	Purpose
`turnLeft(time)/turnRight(time)`	Executes a point turn by running both wheels at the same speed, in opposite directions
`forwardGrid(time)`	Moves the mBot straight forward by running the wheels at the same speed in the same direction.
`doubleLeft(time)/doubleRight(time)`	Combines the normal turn and forwardGrid functions, in the following sequence: 1. Turn left/right 2. Move forward 3. Turn left/right (same direction as before)
`uTurn(time)`	Turn 180° by running both wheels at the same speed, in opposite directions

Table 5: Manoeuvring Functions

```
void forwardGrid(int time) {
  leftWheel.run(-MOTORSPEED);
  rightWheel.run(MOTORSPEED);
  delay(time);
  stopMove();
}

/**
 * Function allows mBot to make a 90 degrees clockwise turn.
 */
void turnRight(int time) {
  leftWheel.run(-(MOTORSPEED - 15));
  rightWheel.run(-(MOTORSPEED - 15));
  delay(time);
  stopMove();
}

/**
 * Function allows mBot to make a 90 degrees counter-clockwise turn.
 */
void turnLeft(int time) {
  leftWheel.run(MOTORSPEED - 15);
  rightWheel.run(MOTORSPEED - 15);
  delay(time);
  stopMove();
}
void doubleRight(int time) {
  turnRight(TIME_FOR_SECOND_RIGHT_TURN);
  delay(10);
  forwardGrid(time);
  delay(100);
  turnRight(TIME_FOR_SECOND_RIGHT_TURN);
}

/**
 * Function allows mBot to make a 90 degrees counter-clockwise turn, followed by moving forward by 1 tile, lastly following with another 90 degrees c
 */
void doubleLeft(int time) {
  turnLeft(TIME_FOR_LEFT_TURN);
  delay(10);
  forwardGrid(time);
  delay(100);
  turnLeft(TIME_FOR_SECOND_LEFT_TURN);
}

/**
 * Function allows mBot to make a 180 degrees clockwise turn.
 */
void uTurn() {
  turnRight(TIME_FOR_UTURN);
}
```

Figure 32: Turning Functions

For all turns, we use a motor speed of `MOTORSPEED - 15 = 240` (`MOTORSPEED` is set to **255**). The reason why we do this is because turning at full speed (255) may lead to wheels slippage, especially on the smooth surface of the coloured papers. To reduce wheel slippage, we reduced the motor speed by 15, as we found that it offered a good balance of grip and speed.

Every turn also uses different timings, as shown in the table below:

Constant	Purpose	Value
`TIME_FOR_LEFT_TURN`	Duration for left turn	360
`TIME_FOR_RIGHT_TURN`	Duration for right turn	360
`TIME_FOR_1_GRID_PINK`	Duration for forward movement in `doubleLeft()` function	855
`TIME_FOR_1_GRID_BLUE`	Duration for forward movement in `doubleRight()` function	855
`TIME_FOR_SECOND_LEFT_TURN`	Duration for second left turn in `doubleLeft()` function	370
`TIME_FOR_SECOND_RIGHT_TURN`	Duration for second right turn in `doubleRight()` function	380
`TIME_FOR_UTURN`	Duration for U-turn	730

Table 6: Duration Constants of Turns

Some of the differences in durations can be explained by the slight differences in the motor output of both right and left motors. The timings have been fine-tuned to achieve the highest level of accuracy in the turns, but there are still instances in which the turns may not be completely accurate. In those cases, we rely on the wall avoidance system to correct and straighten the mBot.

4.2.4 Celebratory Tune

For our celebratory tune, we decided to use the “Samsung washing machine” chime. Using the frequency table below, we coded the values and durations of each note manually. This function is activated upon the detection of the colour “white”, to signify the end of the maze.

	Octave 0	Octave 1	Octave 2	Octave 3	Octave 4	Octave 5	Octave 6	Octave 7	Octave 8
C	16.35	32.70	65.41	130.8	261.6	523.3	1047	2093	4186
C#	17.32	34.65	69.30	138.6	277.2	554.4	1109	2217	4435
D	18.35	36.71	73.42	146.8	293.7	587.3	1175	2349	4699
D#	19.45	38.89	77.78	155.6	311.1	622.3	1245	2489	4978
E	20.60	41.20	82.41	164.8	329.6	659.3	1319	2637	5274
F	21.83	43.65	87.31	174.6	349.2	698.5	1397	2794	5588
F#	23.12	46.25	92.50	185.0	370.0	740.0	1480	2960	5920
G	24.50	49.00	98.00	196.0	392.0	784.0	1568	3136	6272
G#	25.96	51.91	103.8	207.7	415.3	830.6	1661	3322	6645
A	27.50	55.00	110.0	220.0	440.0	880.0	1760	3520	7040
A#	29.14	58.27	116.5	233.1	466.2	932.3	1865	3729	7459
B	30.87	61.74	123.5	246.9	493.9	987.8	1976	3951	7902

Figure 33: Frequency Table for Musical Notes

```
/**
 * Samsung Washing Machine Chime (first 10 notes)
 * Source: Schubert's "Die Forelle" (The Trout), transposed to E major
 */

void finishMaze() {
  const int BPM = 210;            // close to real tempo
  const int Q = 60000 / BPM;      // quarter note in ms
  const int E = Q / 2;            // eighth note
  const int R = E / 1.5;
  const int F = E / 2;            // 1/16 note

  // Frequencies (Hz)
  const int DSH5 = 622;
  const int E5 = 659;
  const int FSH5 = 740;
  const int GSH5 = 831;
  const int A5 = 880;
  const int B5 = 988;
  const int CSH6 = 1109;
  const int D6 = 1175;

  int melody[] = { E5, A5, CSH6, A5, E5, B5, A5, GSH5, FSH5, E5, E5, A5, CSH6, A5, E5, A5,
   GSH5, FSH5, GSH5, A5, DSH5, E5, E5, GSH5, A5, GSH5, FSH5, GSH5, A5, E5, A5, GSH5, D6, B5,
   GSH5, A5, FSH5, A5, E5, B5, GSH5, A5, A5, GSH5, FSH5, A5, GSH5, B5, A5, E5, B5, GSH5, A5};
  int dur[]    = { E, Q, Q, Q, Q, F, F, F, F, Q, E, Q, Q, Q, E, E, E, F, F, E, E, Q, E, Q, F,
  F, F, F, Q, E, E, Q, F, F, F, Q, Q, Q, Q, E, E, Q, F, F, Q, F, F, F, Q, Q, E, E, Q};

  for (int i = 0; i < 53; i++) {
    buzzer.tone(MUSIC_PIN, melody[i], dur[i]);
    delay((int)(dur[i] * 1.05));
    buzzer.noTone(MUSIC_PIN);
    delay(15);
    if (i == 31) delay(235);
    if (i == 41) delay(135);
    if (i == 4 || i == 9 || i == 21 || i == 35 || i == 38 || i == 49) delay(335);
  }
}
```

Figure 34: Celebratory Tune (Samsung Washing Machine Tune)

5. Work Division

Individual contributions are listed in the table below.

Name	Contributions
Eethan	Built IR sensor circuit, assisted in calculation of theoretical resisor values, fixed circuitry for colour sensor, conducted testing of mBot, wrote report
Ernest	Wrote the first version of the main wall avoidance algorithm, handled coding calibration for the colour sensor, worked on the circuitry for the colour sensor, wrote report
Eugene	Calculated theoretical values for IR circuit and LED resistor values for colour sensor, researched on colour sensor working principles, fine-tuned and calibrated the mBot's turning, wrote report
Gabriel	Wrote second version of the main algorithm, worked on fine tuning of mBot behaviour, cleaned up wiring and skirting for the mBot, created the celebratory tune, wrote report

6. Difficulties faced and solutions

No.	Difficulties	Solutions
1	The IR Sensor was unable to work concurrently with the ultrasonic sensor.	Utilize a formula to determine error distance.
2	Initially, the colour sensor was unable to successfully detect any of the colours. The colour sensor's output would either be zero or the maximum intensity for each of the RBG colours differing from the actual colour.	To correct this, we had to change the values of the resistor connected to the LEDs and modify our code several times. Eventually, we found resistor values which kept the LEDs from being too bright or dim and fixed our code to accurately detect colours underneath the colour sensor.
3	Due to the PD control, the mBot jitters as it travels straight.	To improve speed, adjusted both KP and KD values.
4	Inconsistent turning radius, causing the mBot to turn too much or not enough at waypoints inconsistently.	Conducted multiple tests runs and calibrations to determine the best and least error-prone radius.