CG1111A

ENGINEERING PRINCIPLES AND PRACTICE I

TAN ZHE HUI (A0273087X)

TAY GUANG SHENG (A0273178W)

TEH ZE XUE (A0269848A)

TEJAS ANANTH KUMAR (A0281316J)

TABLE OF CONTENT

1. mBot Pictorial Overview

1.1 Main Body

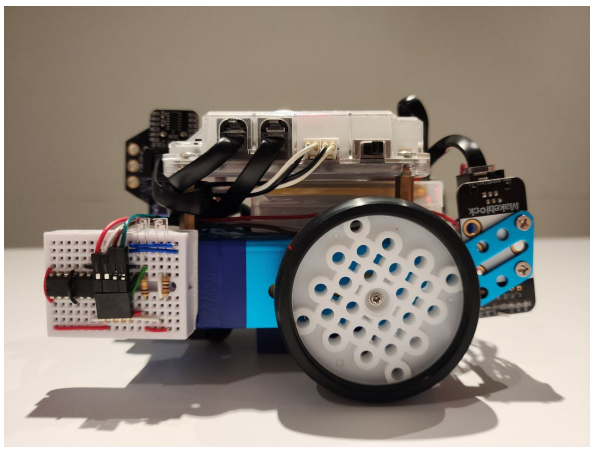

Figure 1: Left View

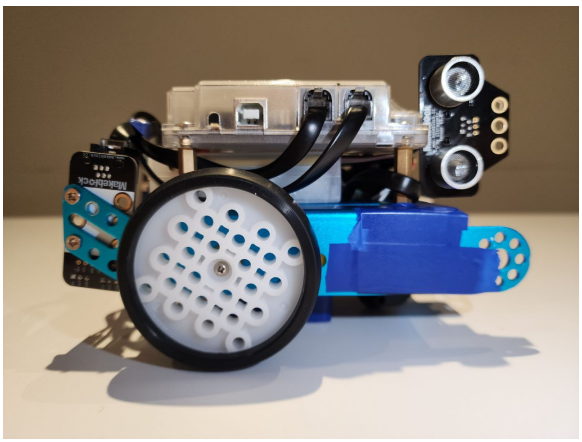

Figure 2: Right View

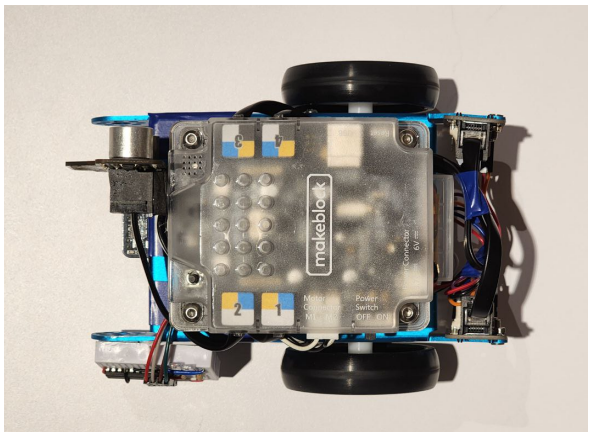

Figure 3: Top View

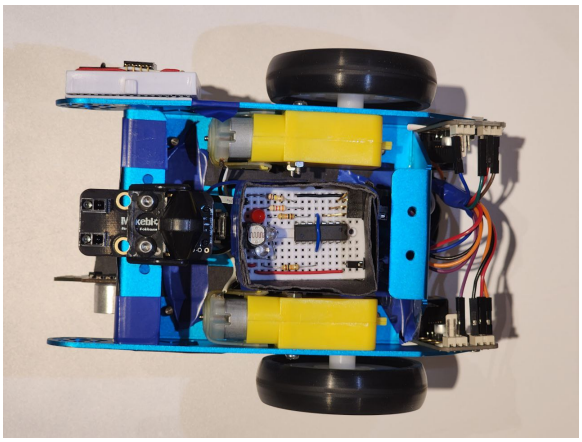

Figure 4: Bottom View

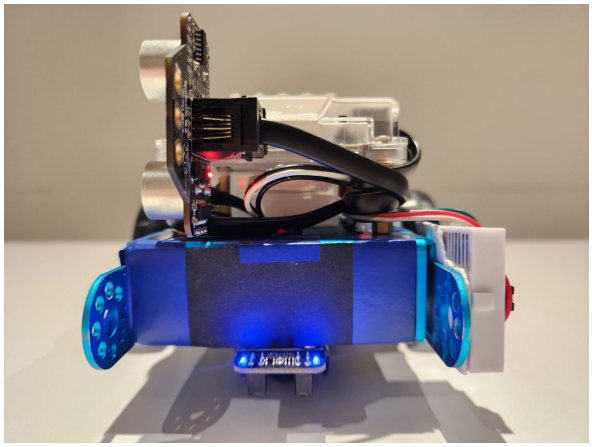

Figure 5: Front View

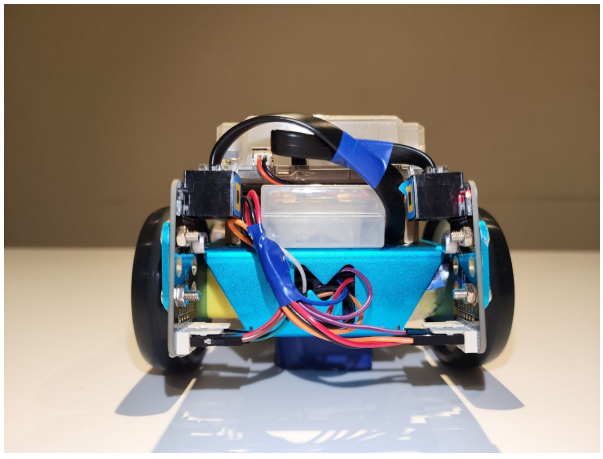

Figure 6: Back View

1.2 LDR Sensor Circuit

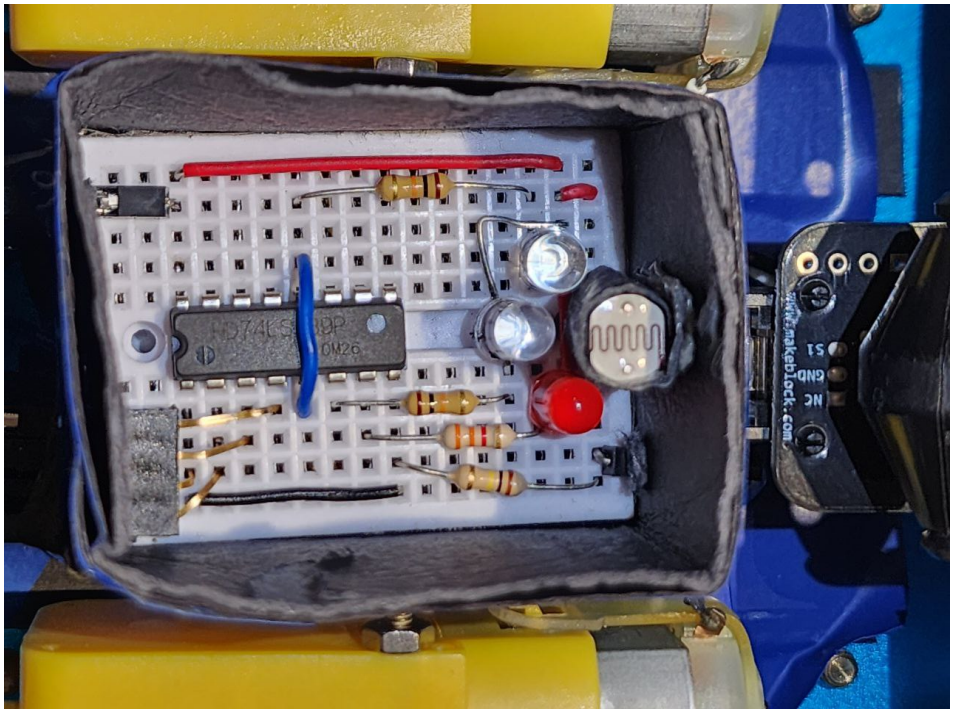

Figure 7: LDR Sensor Circuit

1.3 Infrared Sensor Circuit

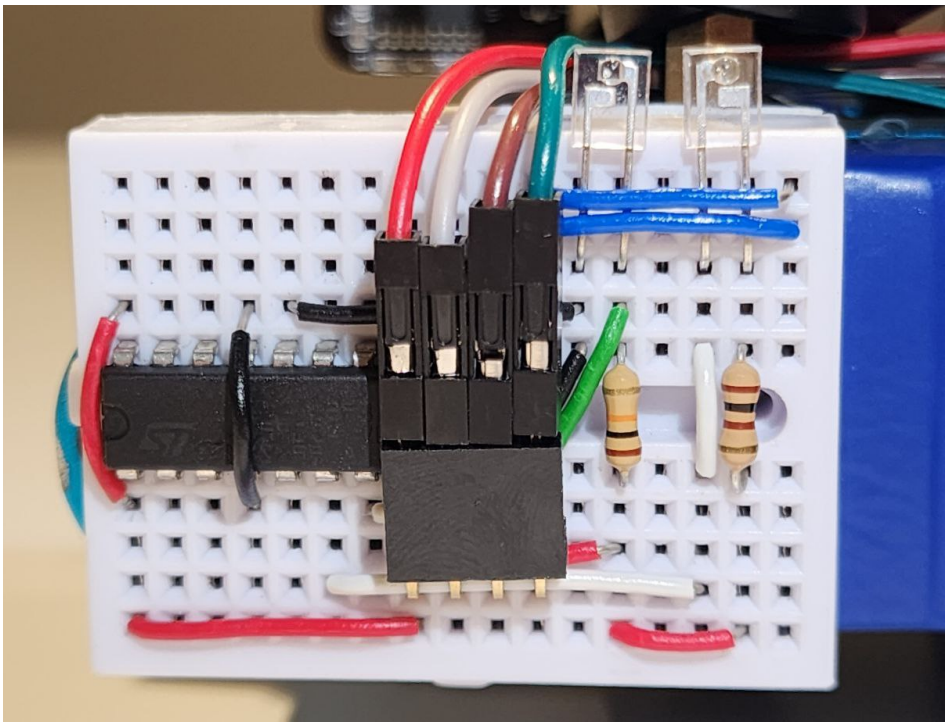

Figure 8: Infrared Sensor Circuit

2. Overall Algorithm

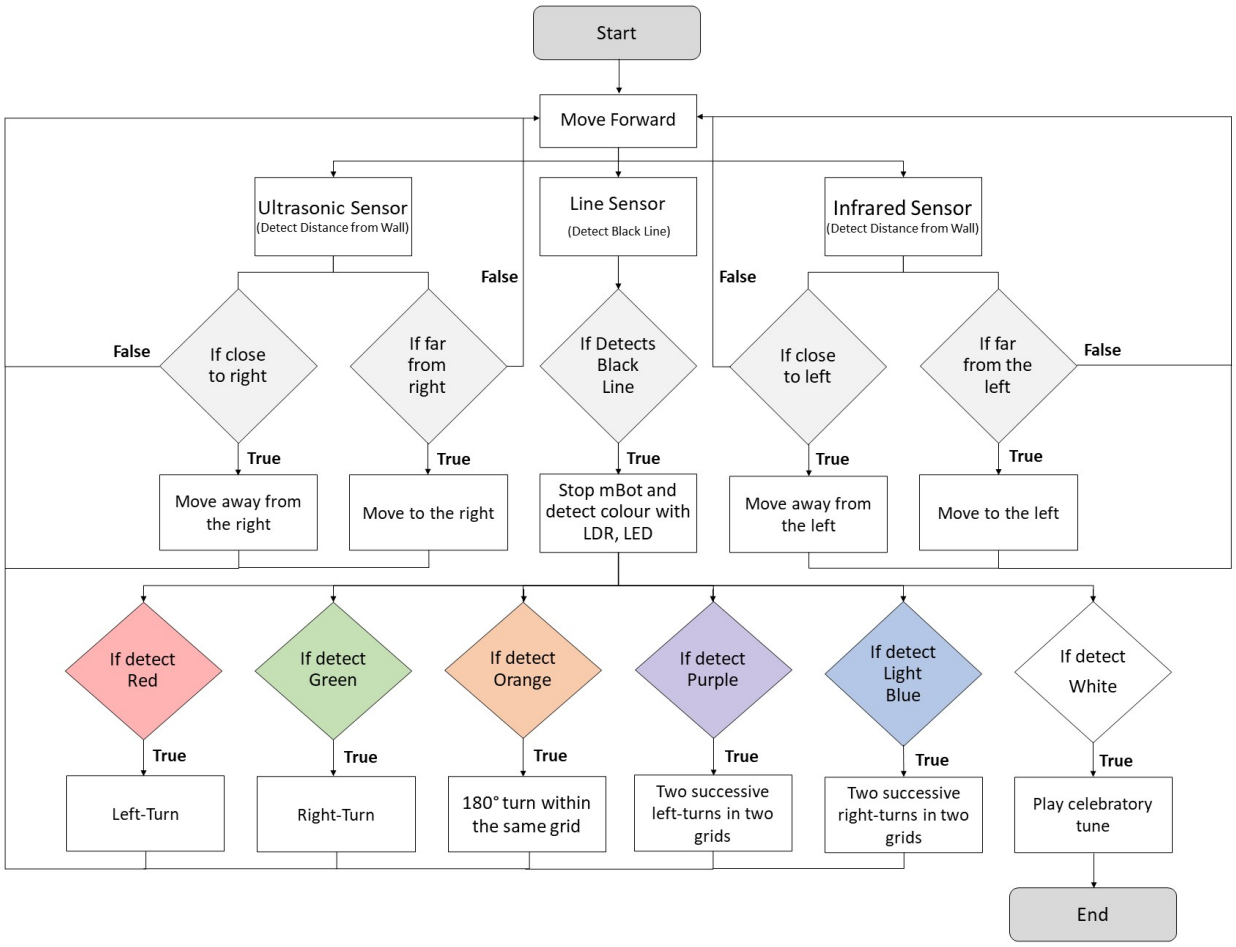

Figure 9: Overall Algorithm

2.1 Move Forward

When the mBot is placed in the maze, it is programmed to move forward. While moving forward, three sensors come into play: the Ultrasonic Sensor and Infrared Sensor which continuously monitor for the distance between the wall and the mBot, and the Line Follower which detects the black line on the floor.

2.2 Ultrasonic And Infrared Sensor

Before the maze, the Ultrasonic and Infrared Sensors undergo calibration to ensure accurate distance detection. If the distance between the wall and the robot is too close, it will adjust by deviating in the opposite direction, equating the distances on both sides. On the other hand, if the robot is too far from the wall, the robot will deviate in the opposite direction to maintain consistent distances. For example, if the ultrasonic sensor

detects that it is too close to the right, it will command the mBot to move away from the right to maintain even distance. Afterwards, the mBot will continue to move forward.

2.3 Line Detector And Colour Sensor

The Line Detector continuously scans for a black line. Upon detecting the line, it instructs the mBot to halt and activate the Colour Sensor. Before entering the maze, the Colour Sensor undergoes calibration using white and black paper to recognise colours. It emits red, green, and blue light, while the LDR measures the reflected light from the paper with each light source. The combination of these values determines the colour, and the mBot responds accordingly (Refer to **Table 1**). After completing the specified actions (except for detecting the colour white), the mBot resumes the forward movement and the loop repeats.

Table 1: Turns

Colour	Action
Red	Left-turn
Green	Right-turn
Orange	180 turn within the same grid
Purple	Two successive left turns in two grids
Blue	Two successive right turns in two grids
White	Celebratory Tune

2.4 Termination

Termination of the code happens when the mBot exits the loop. This happens when the Colour Sensor detects white resulting in it playing the celebratory tune indicating the end of the algorithm.

3. Navigating Forward

3.1 Overview

While the mBot is moving forward, it is essential to have mechanisms to help it maintain a straight-line trajectory if not it may collide with the wall. For example, when the mBot starts travelling, it might be perfectly aligned to the centre of the path. However, as time passes, factors such as the following will affect the movement of the mBot.

- Battery: The power supplied to the wheels is influenced by the level of charge in the battery, and this, in turn, affects how much each wheel turns, even if the TURN_SPEED constant in the code remains unchanged. Hence, both wheels may not have the same rate of wheel turns, resulting in deviation.
- Wheel: The accumulation of dust and debris on the wheel surfaces affects the wheels' traction.
- Surface Conditions: Some parts of the maze have a rougher surface compared to the rest, and this affects the speed and turning of the mBot, especially since the shielding of the colour sensor drags over the ground.

These deviations may seem slight but can accumulate over time, leading to significant deviations from the intended path. Especially in this maze where the travelling space is not huge, any slight deviation will result in the mBot colliding with the wall. To address this issue, we have implemented both Ultrasonic and Infrared (IR) sensors to help the mBot maintain a straight path. This section will explore how these sensors contribute to the mBot's ability to navigate accurately and avoid collisions.

3.2 Ultrasonic Sensor

3.2.1 Calculations

The ultrasonic sensor comprises one emitter and one receiver. The emitter will emit high-frequency sound waves(ultrasonic waves) and when the emitted sound waves encounter an object, they bounce off the surface and return back to the receiver. By measuring the time it takes for the sound waves to travel to the wall and back, we are able to measure the distance by applying the following formula. The ‘2’ is in the formula because the sound has to travel back and forth hence travelling double the distance.

$$Distance = (Time\ x\ Speed\ of\ Sound) / 2,\ where\ Speed\ of\ Sound = 340m/s$$

3.2.2 Arduino Code

The following is the Arduino code for the calculation of distance. As the time measured by the sensor is in milliseconds and we want the distance to be in cm, we multiply the product by 0.0001.

```
long duration = pulseIn(ULTRASONIC, HIGH, TIMEOUT);
float distanceCm = 0.5 * duration * SPEED_OF_SOUND * (0.0001);
```

3.2.3 Mounting of Sensor

The Ultrasonic Sensor is positioned in a vertical orientation at the front of the mBot (Refer to **Figure 10)**. The reasons are as follows

1. Our team has considered the possibility of missing walls within the maze, which could make it challenging for the robot to move in a straight path. Hence, we have opted for a vertical sensor orientation instead of a horizontal one as it is more effective at detecting pillars within the maze, which, in turn, aids the mBot in navigating in a straight line. Hence, this means that our robot is capable enough to travel straight even with just a small surface area.

2. Mounting vertically minimises the robot's overall length as opposed to having parts to protrude out . This configuration becomes especially important as it helps prevent potential collisions when there is a delay in detecting nearby walls or black lines. With this configuration, we can allow our mBot to travel at higher speed as the stopping distance required is reduced as compared to mounting it horizontally, where parts of the sensor protrude out at the front.

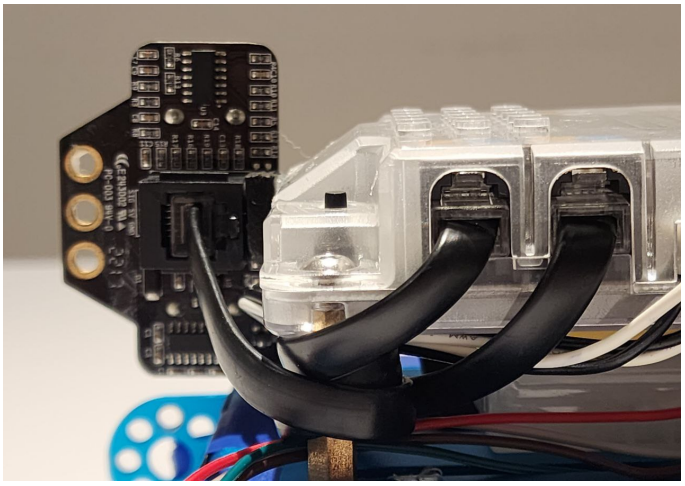

Figure 10: Mounting of Ultrasonic Sensor

3.3 Infrared Sensor

3.3.1 Circuit

The Infrared Sensor consisting of the circuit's ground and power supply is connected to Port 3 on the mCore. (Refer to **Figure 11**) The power supply is connected to several components: the VCCs for the H-bridge, and enable pins 1 and 2 on the H-bridge. It also supplies voltage to both the IR emitter and the potential divider circuit containing the IR detector.

The Pin 7 (Input 2) on the H-bridge is linked to Y0 on the Decoder Chip. During the movement of the robot, Y0 outputs a LOW voltage, creating a potential difference, allowing current to flow through the IR emitter. Since the negative terminal is connected to Pin 3 on the H-bridge (Output 2), this flow induces a fluctuation in the potential difference across the IR detector.

The potential divider circuit is created by having an output wire connected between the resistor and the positive terminal of the IR detector. This ensures that the output voltage varies based on the intensity of reflected infrared. This variation can be translated into the distance between the robot from the maze walls.

Resistance Value

From the project brief, it is understood that the resistance value of 8.2kΩ (that was used in the studio) may not be the most appropriate for the IR detector. A larger detector resistance makes the receiver voltage more responsive. But a value that is too large may lead to saturation easily even with minimal IR.

Regarding the resistance for the IR emitter, since the IR emitter produces a forward voltage, it will result in the actual current being lower than the calculated one. Thus, we decided to use the resistance value of 100Ω which gives an actual value of less than 50mA. Therefore, we came to the conclusion of using the 100Ω and 10kΩ resistors for the IR Emitter and IR detector respectively.

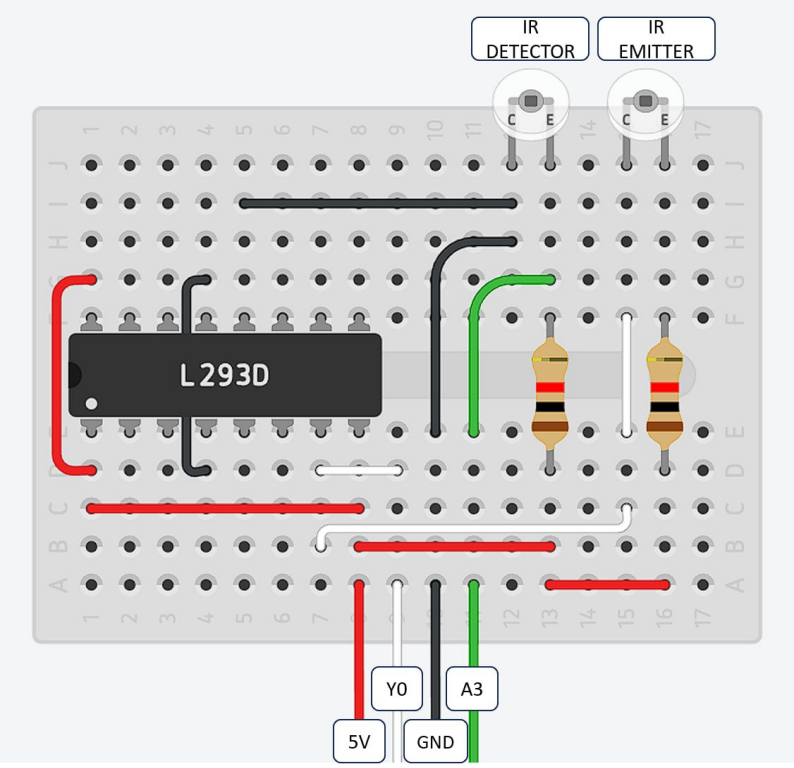

Figure 11: TinkerCAD Circuit for Infrared Sensor

3.3.2 Arduino Code

Firstly, turnOnIR and turnOffIR are functions to activate and deactivate the IR emitte. They function similarly to the shineRGB functions providing input to the 2-4 decoder to activate/deactivate decoder output pin Y0.

```
void turnOnIR() {
  digitalWrite(A0, LOW);
  digitalWrite(A1, LOW);
}

void turnOffIR() {
  digitalWrite(A0, HIGH);
  digitalWrite(A1, HIGH);
}
```

Next, the infraredSensorDistance measures the distance between emitter and wall and returns a float output. It is based on the following equation:

$$d_{IR} = \sqrt{\frac{IR\_FACTOR}{V_i - V_f}}$$

Where:

- V_i is the initial IR detector reading, measuring ambient light
- V_f is the final reading after IR emitter has been activated
- IR_FACTOR is a calibrated proportionality constant to convert inverse root voltage to distance

This takes into account for the ambient light variations since the distance calculated is based on the difference in IR detector readings after the IR emitter is activated. The IR_WAIT constant is used as a delay to allow for the IR detector resistance to stabilize, similar to RGB_WAIT constant.

```
float infraredSensorDistance() {
  // Activate the IR emitter
  turnOnIR();
    delayMicroseconds(IR_WAIT);  //  Allow  time  for  the  IR  emitter  to
stabilize

  // Measure the distance using the IR receiver
  int initialReading = analogRead(IR_PIN);

  turnOffIR();
  delayMicroseconds(IR_WAIT);
  int finalReading = analogRead(IR_PIN);

  // Calculate the difference in sensor readings
  float voltageDifference = initialReading - finalReading;

  // Calculate and return the distance
  return calculateDistance(voltageDifference);
}

float calculateDistance(float voltageDifference) {
  if (voltageDifference == OFFSET) { // Avoid division by zero
    return -1; // Indicate an error or invalid distance
  }

   float  distance  =  sqrt(IR_CALIBRATION_FACTOR  /  (voltageDifference  -
OFFSET));
  return distance;
}
```

Within the computePID() function that will be explained in **Section 3.4** (Proportional Integral Derivative), the following two lines of code address the usage of the IR sensor. Specifically, these lines gauge the distance between the robot and the maze wall in situations where the ultrasonic sensor fails to detect a wall. While the infrared sensor may be inherently less precise, it serves as a contingency that enables the robot to still navigate accurately when the wall on the ultrasonic sensor side of the robot is absent.

```
if (currentDistance > SET_POINT * 1.5) {
    currentDistance == 27 - (filteredDistance() + 9);
}
```

3.3.3 Calibration

To find the IR_CALIBRATION_FACTOR constant, we followed the following steps.

1. Rearrange the formula to make factor subject of equation
2. Calculate factor using data points. At known distances ranging from 0 - 15cm, data point was collected for V_i and V_f. This will give multiple values of IR_CALIBRATION_FACTOR
4. Average thecalculated factors. Since IR_CALIBRATION_FACTOR is a constant, taking the average across n measurements reduces the error in derivation.

By calculating the IR_CALIBRATION_FACTOR for different areas, we are able to tune the IR sensor to detect distance while mitigating ambient light variation.

3.3.4 Mounting of Sensor

We have mounted the sensor on the left side of the mBot. Similar to our approach to mounting the ultrasonic sensor, we hope to have the sensor within the mBot's framework and not protruding out of the mBot to prevent potential collision with the walls. To prevent wire entanglement with the wheels or inadvertent contact with walls, we have used electrical tape to secure the wires to the bot. The above measures ensure a tidy and secure arrangement, enhancing the overall functionality of the IR sensor.

3.4 Proportional Integral Derivative (PID)

3.4.1 Background

In order to ensure the mBot travels in a straight line, we are required to use an Ultrasonic Sensor and an IR sensor to detect the distance the robot is from the wall to adjust the trajectory of the robot. However, to avoid the robot zig-zagging as it travelled, our group decided to adopt the PID function instead. Compared to using a nudge function that alters the path of the robot by a set amount, this algorithm reduces the magnitude of correction as the mBot gets closer to moving in a straight line. Thus, smoothing out the path of travel of the robot and fulfilling the requirements of the robot travelling as straight as possible.

The PID controller is a closed-loop feedback controller consisting of proportional, integral and derivative terms. **Table 2** contains the definition of the common terms used in PID.

Table 2: Definition of Terms

Term	Definition
Setpoint	The desired value that the system aims to maintain. In our context, the setpoint would be the ideal distance between the wall and the mBot. It's the target distance that the PID controller will try to achieve and maintain.
Error	The difference between the setpoint and the current value. In our context, it is the difference between the actual distance from the wall and the setpoint distance.
Output	The corrective action taken by the PID controller. It's the actual command sent to the robot's motors or actuators, based on the combined calculations of the P, I, and D terms. In our context, it is the output that adjusts the robot's right and left motor speed, to help it stay in a straight line.
Proportional (P)	The proportional term attempts to drive the position error to zero by contributing to the control signal proportionally to the current position error. A higher proportion gain leads to a faster response to changes. However, it may result in overshooting the desired outcome.
Integral (I)	The integral term considers the accumulated error over time. It helps eliminate any steady-state error that may exist and brings the

	system to the setpoint. The integral gain is crucial for handling situations where the proportional control alone might not be sufficient.
Derivative (D)	The derivative term accounts for the rate of change of the error. It helps in damping the system's response, reducing overshooting, and enhancing stability. The derivative gain is beneficial in preventing the system from oscillating excessively.

The PID formula is typically expressed as:

$$u(t) = Kp \cdot e(t) + Ki \cdot \int_0^t e(\tau)d\tau + Kd \cdot \frac{d}{dt} e(t)$$

Where:

- $u(t)$ is the control output at time t.
- $e(t)$ is the error or the difference between the setpoint and the measured value.
- Kp, Ki and Kd are the proportional, integral and derivative gains, respectively

The key to effective PID control is to find the right balance of these three gains. The proportional term deals with present error, the integral term with past error, and the derivative term with future error.

3.4.2 Arduino Code

The function, computePID() uses the current distance detected either from the ultrasonic sensor or the IR sensor to compute the amount the output required for the mBot motors such that it can travel straight at the centre of the path. Should this distance be zero or excessively beyond the predefined setpoint which is defined as exceeding 150% of the setpoint, meaning that there's no wall, the function zeroes out the key variable of previously recorded error. In these scenarios, it also returns -1, which typically indicates an out-of-range and no actions will be carried out.

At the heart of the function is the error computation, defined as the difference between the current distance and the setpoint. Based on this error, the function calculates the control output. However, unlike a traditional PID controller, this code only factors in the proportional and derivative elements. The proportional part multiplies the error by a predefined proportional gain, while the derivative part is the product of the rate of change in error (current minus previous error) and the derivative gain. After this calculation, the previous error is updated for use in the next iteration.

The Integral part of a PID controller mainly addresses steady-state errors, or those ongoing discrepancies between the desired setpoint and the actual readings when the system is stable. In our project, the PID's job is simply to ensure the robot maintains a straight path. This objective usually requires the system to react quickly to immediate changes, which the Proportional and Derivative elements are well-equipped to handle. Additionally, given that our mBot spends only a brief amount of time navigating around obstacles, the likelihood of significant error accumulation is quite low. With this in mind, we decided that incorporating the Integral aspect of the PID wouldn’t be necessary for our specific application.

Additionally, to ensure that the output signal stays within a manageable limit, we have placed a limit to ensure that our mBot does not go over the maximum control output. This is a precaution to prevent the mBot from executing overly aggressive manoeuvres. The function then proceeds to return this adjusted control output, which ideally adjusts the robot's movement to travel in a straight line at the centre of the maze path.

```
float computePID() {
    float currentDistance = ultrasonicSensorDistance();

    if (currentDistance > SET_POINT * 1.5) {
       currentDistance == 27 - (filteredDistance() + 9);
    }

    if (currentDistance == 0.0 || currentDistance > SET_POINT * 1.5) {
        previousError = 0.0;
        return -1;
    }

    float error = currentDistance - SET_POINT;

    float adjustedOutput = PROPORTIONAL_GAIN * error + DERIVATIVE_GAIN *
(error - previousError);
    previousError = error;

    if (adjustedOutput > MAX_CONTROL_OUTPUT) {
        adjustedOutput = MAX_CONTROL_OUTPUT;
    } else if (adjustedOutput < -MAX_CONTROL_OUTPUT) {
        adjustedOutput = -MAX_CONTROL_OUTPUT;
    }

    return adjustedOutput;
}
```

3.4.3 Calibration

As mentioned previously, the key to effective PID control is to find the right balance of the proportional gain and derivative gain. In order to find out the most ideal value for each of the gains, we conducted experiments with varying Kp and Kd values. During the experiment, we displaced the mBot with respect to a setpoint and recorded the distance against the duration graph to see how well the PID works to reach and maintain the desired distance over time.

Proportional Gain

In this case, we vary the proportional gain to see how it affects the PID output. The graph shown in **Figure 14** shows three different Kp values: 15, 33 and 50 plotted against time and ultrasonic sensor distance.

From the graph, we can infer that a lower Kp results in a more gradual approach to the set point, seen with Kp = 15, where the line is smoother and less reactive to changes. A higher Kp, such as 50, shows a more aggressive approach, where the system overshoots and oscillates around the set point as it tries to stabilise. The Kp = 33 line shows a balance between the two, with moderate oscillations and a steadier approach to the set point.

Therefore, we have chosen 33 as our Kp value as it shows the most steady approach to the set point compared to other values we have tested.

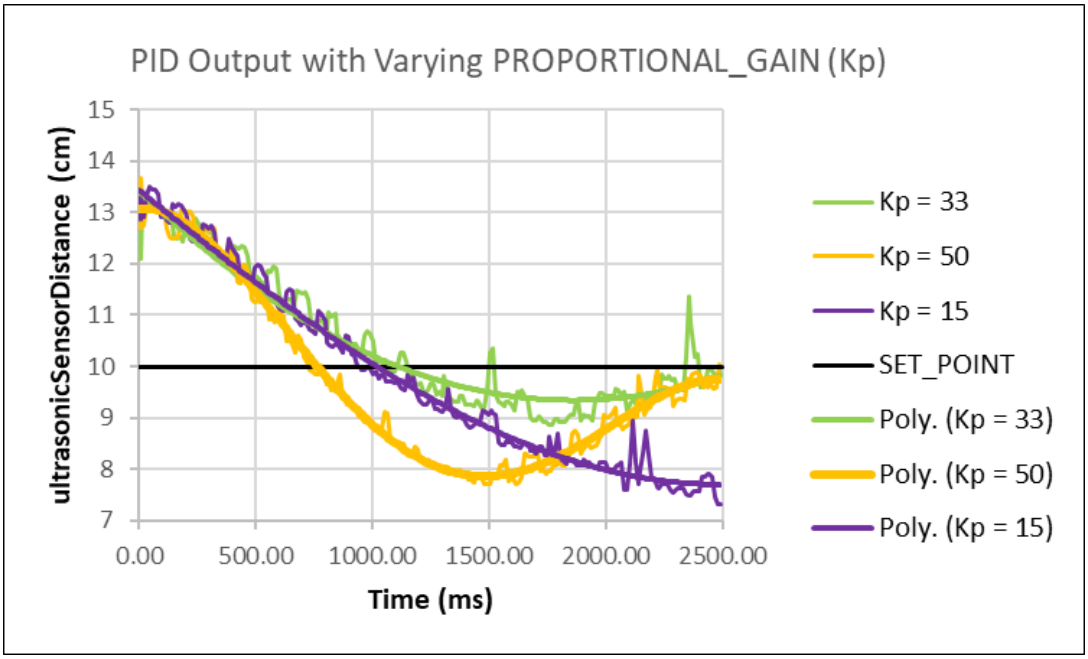

Figure 14: PID Output with Varying Proportional Gain(Kp)

Derivative Gain

In this case, we vary the derivative gain to see how it affects the PID output. The graph shown in **Figure 15** shows three different Kd values: 800, 200 and 1400 plotted against time and ultrasonic sensor distance.

From the graph, we can see that a lower Kd value demonstrates a more gradual convergence towards SET_POINT. When Kd = 200 (too low), there is insufficient dampening of proportional gain, causing the bot to travel in a path with low period. However, when Kd = 1400 (too high), proportional gain is offset by derivative gain leading to greater deviance from SET_POINT and low responsiveness.

Therefore, by choosing a median Kd = 800, it ensures that the proportional gain is dampened to a balanced extent.

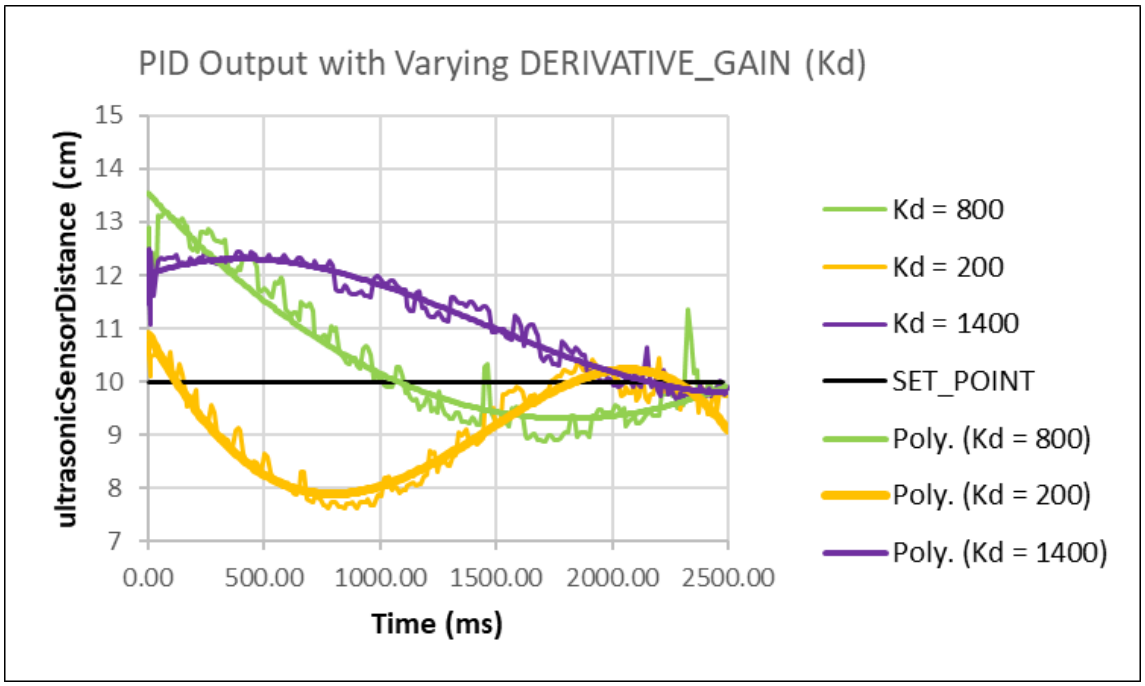

Figure 15: PID Output with Varying Proportional Gain(Kp)

3.5 Smoothing Algorithms

3.5.1 Background

The idea of adopting a smoothing algorithm was from the chapter on Filters. During the testing of the sensors, we realised that occasionally there were a lot of fluctuations in the readings and it is important for us to filter out noise by removing erratic or rapid fluctuations in the data. This smoothing process helps reduce the impact of transient noise or small fluctuations in the sensor readings, leading to a more stable and reliable measurement of distance. As we have observed that the infrared sensor reading fluctuates a lot, we decided to implement the smoothing algorithm for the IR.

3.5.2 Arduino Code

Initially, the algorithm maintains a variable previousFilteredDistance, set to 0.0. This variable holds the last stable or 'filtered' distance value, which is updated over time as new readings are obtained and processed.

In the filteredDistance() function, it first acquires a distance measurement from the infrared sensor. Once the new distance is obtained, the algorithm checks if the difference between this new reading and the previous filtered distance exceeds a predefined THRESHOLD. This threshold is a crucial component of the algorithm as it determines the sensitivity of the filter to changes in distance measurements. If the difference is greater than the threshold, it implies a significant change in distance, and in such cases, the algorithm directly returns the new distance measurement. This approach allows the system to respond promptly to substantial changes in distance, ensuring that important shifts are not ignored.

A SMOOTHING_FACTOR closer to 1 makes the filter more responsive to recent measurements, allowing more noise to pass through while a factor closer to 0 increases the influence of the previousFilteredDistance, therefore smoothing out the noise more effective, but also making the filter slower to respond to changes in the measured distance.

However, if the change in distance is within the threshold, suggesting a minor variation, the algorithm applies a smoothing formula. This formula is a weighted average that combines the new distance measurement with the previous filtered distance. The weights are determined by the SMOOTHING_FACTOR, a constant between 0 and 1. The term (1 - SMOOTHING_FACTOR) * previousFilteredDistance represents the portion of the old value in the new filtered result, while SMOOTHING_FACTOR *

distance represents the contribution of the new reading. A higher smoothing factor gives more weight to new readings, making the filter more responsive but less stable, while a lower factor does the opposite, enhancing stability at the cost of responsiveness.

```
float previousFilteredDistance = 0.0;
float filteredDistance() {
  float distance = infraredSensorDistance();
  if (abs(distance - previousFilteredDistance) > THRESHOLD) {
    return distance;
  } else {
    return (1 - SMOOTHING_FACTOR) * previousFilteredDistance +
(SMOOTHING_FACTOR * distance);
  }
}
```

4. Waypoint Challenge

4.1 Overview

At each waypoint challenge grid, there will be a black strip and colour paper underneath the mBot. The mBot is required to stop when the Black Line Detector detects the black strip and activate the colour sensor which will detect the colour underneath the mBot. The mBot will perform the turns based on the detected colour, following the corresponding actions outlined in **Table 1.**

4.2 Black Line Detector

The Black Line Detector is used to detect the black line when the mBot crosses it, utilising Infrared(IR) rays. The Makeblock Line Sensor consists of 2 sets of IR receivers and transmitters as shown in **Figure 16**. Given that black surfaces absorb a wide range of wavelengths across the electromagnetic spectrum, including infrared, black paper absorbs more light than reflecting it. This will result in the IR Receiver being unable to receive its reflected IR.

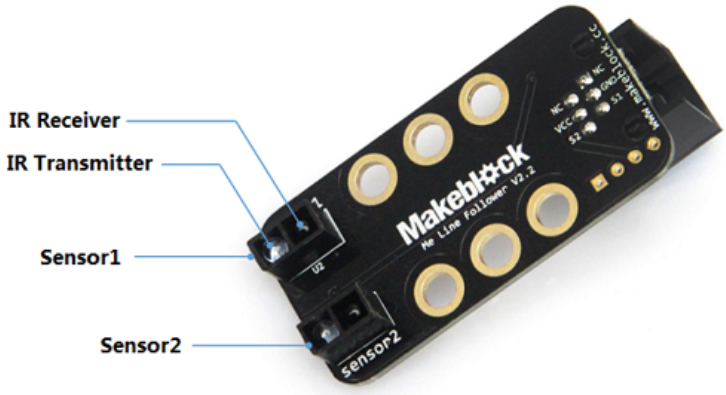

Figure 16: Makeblock Line Sensor

In our code, we use a boolean variable to yield a “true” or “false” outcome when the line detector registers the presence of a black line. We used an if statement to ascertain the existence of a black line, examining the signals from both sensors. We opted to return true only when both sensors detect the black line to prevent any false detections by either sensor. This helps to improve the overall reliability of the outcome.

```
bool reachWayPoint() {
  int sensorState = lineFinder.readSensors();
```

```
  if (sensorState == S1_IN_S2_IN) {
    return true;
  } else {
    return false;
  }
}
```

4.3 Colour Sensors

4.3.1 Overview

All colours can be derived from a combination of the three primary colours, red, green and blue (RGB). Surfaces absorb and reflect different amounts of light from each of the three primary colours, and it is this combination of the reflected light that is perceived as different colours to our eyes. Utilising this concept with a Light-Dependent Resistor (LDR) in a potential divider circuit, and reading the output voltage, the values of reflected red, green, and blue light can be determined by sequentially activating the red, green, and blue Light Emitting Diodes (LED). The reflected colour can then be calculated.

4.3.2 Circuit

Overview

The colour sensor circuit performs two functions: receiving input from the mBot core to shine the RGB LEDs and returning output voltage across the LDR to the mBot core. The circuit consists of a HD74LS139P 2-to-4 decoder IC chip, one 5-100kΩ LDR), Red/Blue/Green Light Emitting Diodes (LED) and three corresponding resistors. Refer to **Figure 17** for the circuit diagram.

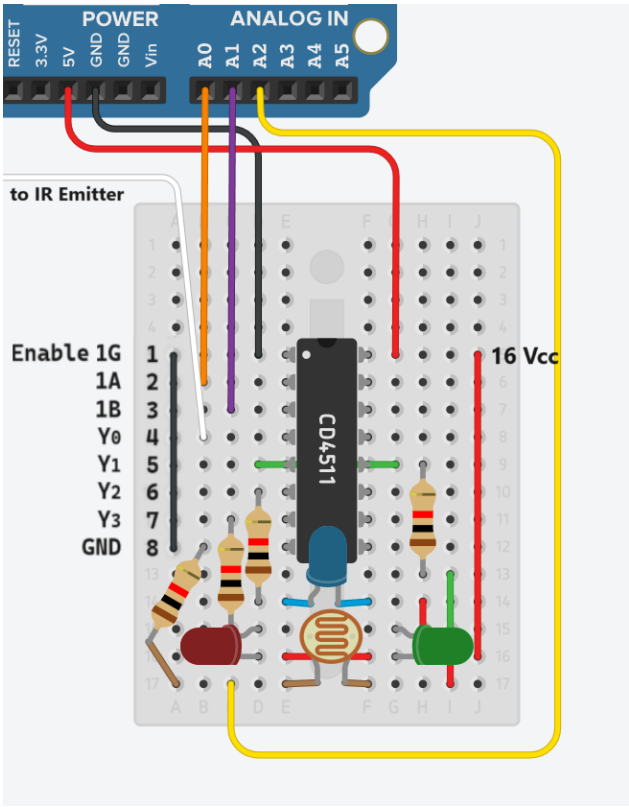

Figure 17: Colour Sensor Circuit

RGB LED Control

RGB LED control is achieved via the 2-4 decoder IC chip (Refer to **Table 3**). It takes a 2-bit binary input and returns one of 4 possible output lines. It is connected to the analog output (A0, A1), 5V power supply (Vcc) and ground (GND) pins via an RJ25 adapter connected to Port 4 of the mBot Core. Firstly, the decoder Vcc (pin 16) is connected to a 5V power supply to power the chip. Next, the Enable 1G (pin 1) is connected to GND since it is an active LOW decoder. When each of the 4 decoder output pins Y0 - Y3 (pins 4-7) are activated, they are set to logic LOW. Lastly, inputs 1A and 1B (pins 2-3) are connected to analog output pins A0 and A1 from mBot Core.

The 3 RGB LEDs are mounted equidistant from the LDR. RGB LEDs are connected in parallel, with positive terminals connected to Vcc and negative terminals of Red, Blue and Green LEDs are connected to a resistor and Y3, Y2 and Y1 (pins 7, 6, 5) respectively.

LDR Voltage Reading

LDR reading is achieved via a potential divider circuit. LDR is connected in series to a 100kΩ resistor. This resistance value was chosen to match the 5-100kΩ variable

resistance of LDR. Analog input pin A2 is connected across the LDR to measure voltage.

RGB LEDs are controlled with the following When one of the pins, Y1, Y2 or Y3 is activated, it will output logic LOW, thus allowing current to flow through one of the three LEDs, lighting it up. The combination of inputs is controlled by the 1A and 1B input pins which are connected to S1 and S2 on Port 4 of the mCore which corresponds to analog output A0 and A1 on the Arduino board. The reflected light is then read through the voltage output from the potential divider circuit with the LDR. This voltage is read through the S1 pin on Port 3 of the mBot Core corresponding to analog input A2 on the Arduino board.

Table 3: Function Table Summarising 2-4 Decoder Inputs/Outputs

Inputs			Outputs			
Enable	Select					
G	B	A	Y_0	Y_1	Y_2	Y_3
H	X	X	H	H	H	H
L	L	L	L	H	H	H
L	L	H	H	L	H	H
L	H	L	H	H	L	H
L	H	H	H	H	H	L

H ; high level, L ; low level, X ; irrelevant

4.3.3 Code

We followed a bottom-up approach in writing colour detection code. At the lowest level, the shineRed, shineGreen and shineBlue functions set analog output pins A0, A1 to activate decoder output pins Y1, Y2, Y3 to light up their respective LEDs.

```
void shineRed() {
  digitalWrite(A0, HIGH);
  digitalWrite(A1, HIGH);
}
void shineGreen() {
  digitalWrite(A0, HIGH);
  digitalWrite(A1, LOW);
}
void shineBlue() {
  digitalWrite(A0, LOW);
  digitalWrite(A1, HIGH);
}
```

On the next level, these functions were consolidated to a shineRGB function taking integer parameter i. Switching based on this argument, lower-level functions are called, simulating a RGB LED.

```
void shineRGB(int colour) {
  switch (colour) {
    case 0:
      shineRed();
      break;
    case 1:
      shineGreen();
      break;
    case 2:
      shineBlue();
      break;
  }
}
```

Next, the getAverageReading function reads the voltage across LDR via analog pin A2. It takes in integer parameter times, measuring an averaged reading based on times. Taking multiple readings of LDR voltage reduces error and returns a more stable output.

```
int getAvgReading(int times) {
  int reading;
  int total = 0;

  for (int i = 0; i < times; i++) {
    reading = analogRead(LDR);
    total += reading;
    delay(LDR_WAIT);
  }
  return total / times;
}
```

Finally, the detectColour() function differentiates between coloured paper and returns integer input from -1 to 4 (Refer to figure X). The function has 3 main components.

1. Reading LDR Values and updating colourArray
   Using the shineRGB function to activate red, green and blue LEDs in sequence. Light is shined onto the coloured paper, with some wavelengths being absorbed while others are reflected. This affects the intensity of reflected light, in turn, affecting LDR resistance. The change in resistance is measured by the potential divider circuit, with the averaged voltage read with the averageReading function.

```
int detectColour() {
  // 1. Shine RGB, read LDR after some delay
  int det = -1;
  int min_dev = MIN_DEV;
  long curr_dev;
  for (int i = 0; i <= 2; i++) {
    shineRGB(i);
    delay(RGB_WAIT);
    colourArray[i] = getAvgReading(DET_NO);
    colourArray[i] = abs((colourArray[i] - blackArray[i]) /
(greyDiff[i]) * 255);
    shineNone();
  }
```

Next, the averagedReading is normalised to be within the range of 0 - 255 with the following formula.

$$R_{normalised} = \frac{R - min(R)}{max(R) - min(R)} \times 255$$

Where:

- R is the averageReading output
- $min(R)$ and $max(R)$ are the minimum and maximum intensities reflected for each colour RGB LED shone.

By deducting the minimum reading from the averaged reading and multiplying it by the range, all variations fall between zero and one. Next, this is multiplied by the 8-bit integer limit 255 to match 8 bit RGB values.

2. <u>Colour classification using sum of squared difference</u>

$$SSD = \sum_{c=0}^{2} (colourArray[c] - calibratedColour[c])$$

Where:

- c is an iterator through Red, Green and Blue
- $colourArray[c]$ is the normalised reading for each colour LED shone
- $calibratedColour[c]$ is calibrated reading corresponding to a particular colour paper.

By taking the calibrated value as a mean, the sum of squared difference effectively calculates how much each measured value deviates from it. The calibrated colour which deviates least from the detected colour is the closest match.

```
  // 2. Run algorithm for colour classification (Sum of Squares)
```

```
  for (int i = 0; i < 5; i++) {
    curr_dev = 0;
    for (int j = 0; j < 3; j++) {
      curr_dev += (coloursArray[i][j] - colourArray[j]) *
(coloursArray[i][j] - colourArray[j]);
    }
    if (curr_dev < min_dev && curr_dev > 0) {
      det = i;
      min_dev = curr_dev;
    }
  }
```

3. Displaying classified colour via mCore onboard LED
   Based on the detected value, the onboard LEDs are set to values from displayArray. This is because the actual readings may differ in intensity. Displaying the detected colour helps in debugging the sensor. Finally, the detected value integer is returned.

```
  // 3. Set mBot onboard LED to display array colours
  if (det > -1) {
    led.setColor(displayArray[det][0], displayArray[det][1],
displayArray[det][2]);
    led.show();
  }
  return det;
}
```

4.3.4 Calibration

Calibration of RGB_WAIT and LDR_WAIT constants

RGB_WAIT and LDR_WAIT are integer constants that determine delay times. LDR_WAIT is the delay between each reading in the LDR reading averaging function. Its value was set to an arbitrarily low value at 10ms to ensure sufficient change in LDR reading between each reading. RGB_WAIT is the delay between shining the LED and taking the average reading across LDR. This delay allows the resistance across the LDR to stabilise, resulting in a more accurate reading of LDR voltage. **Figure 18, 19, 20** shows stabilisation duration when detecting red, green and blue paper respectively. It is observed that for each primary colour being detected, it takes around 200ms for the LDR resistance to stabilise. Furthermore, the detectColour function takes DET_NO*LDR_WAIT ms to get averaged LDR readings, which also gives LDR more time to stabilise. Therefore RGB_WAIT was set to 100ms.

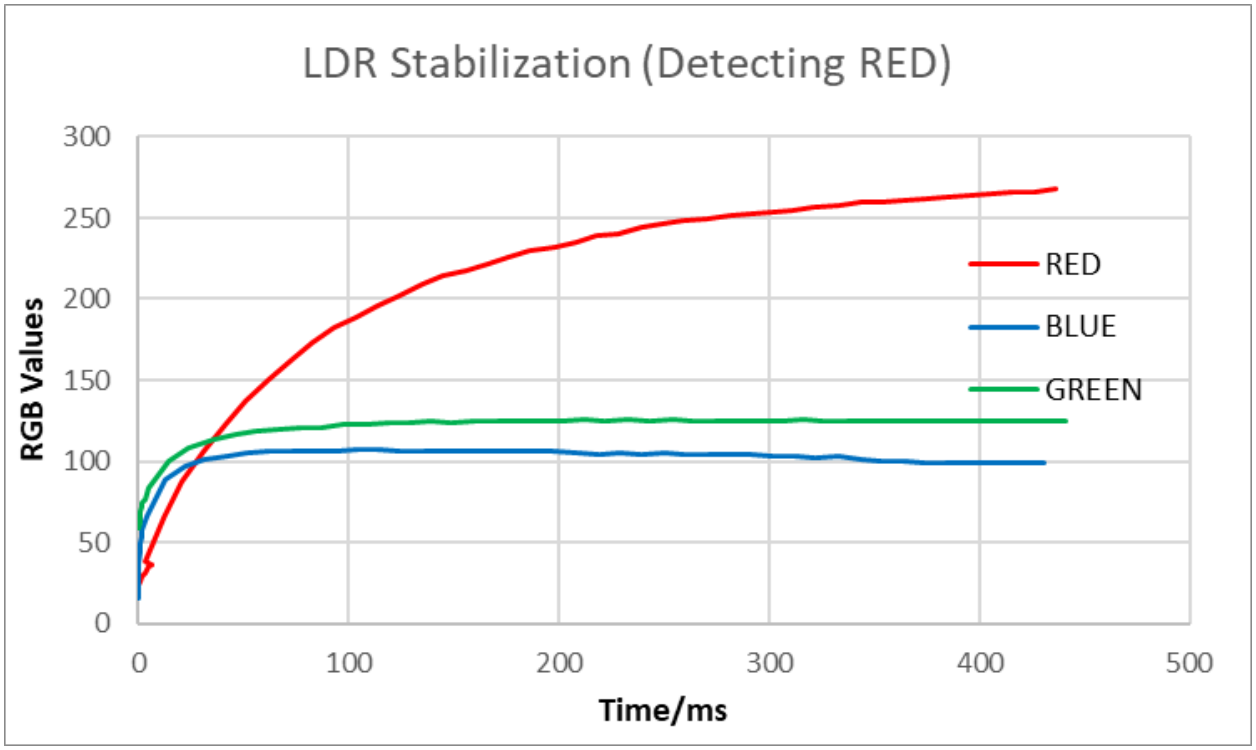

Figure 18: Stabilisation duration when detecting RED

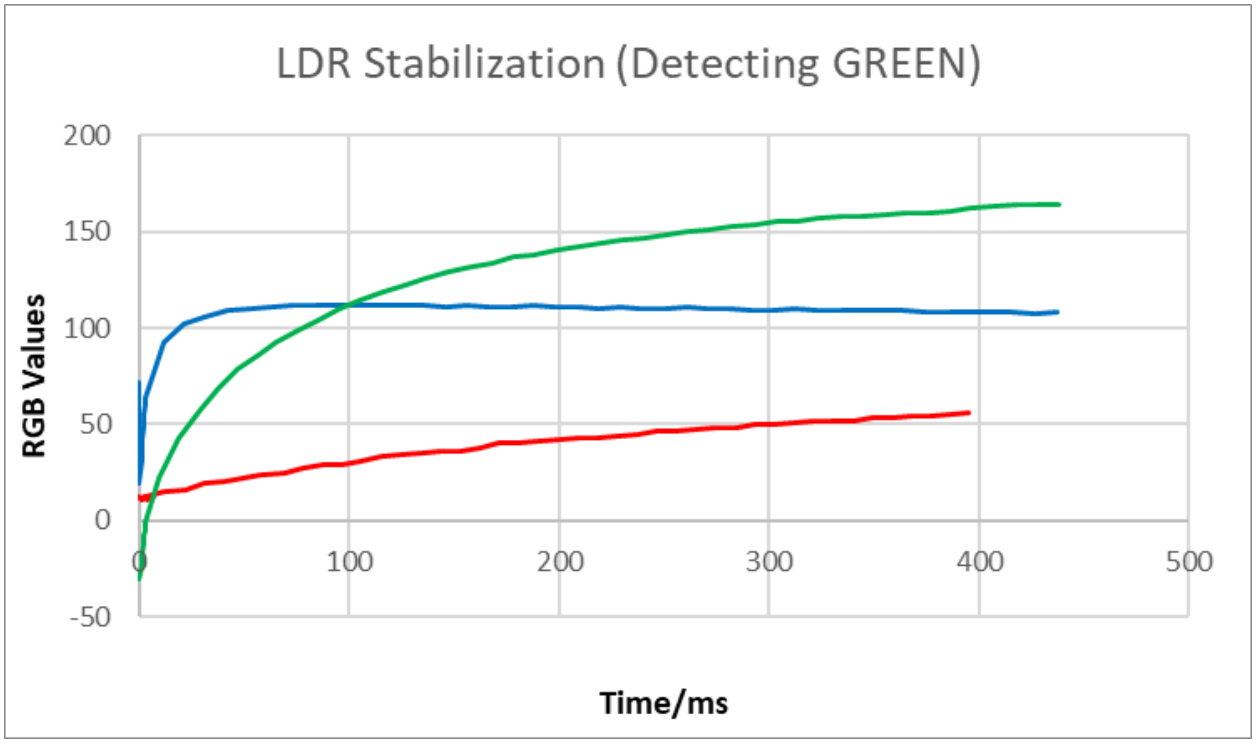

Figure 19: Stabilisation duration when detecting BLUE

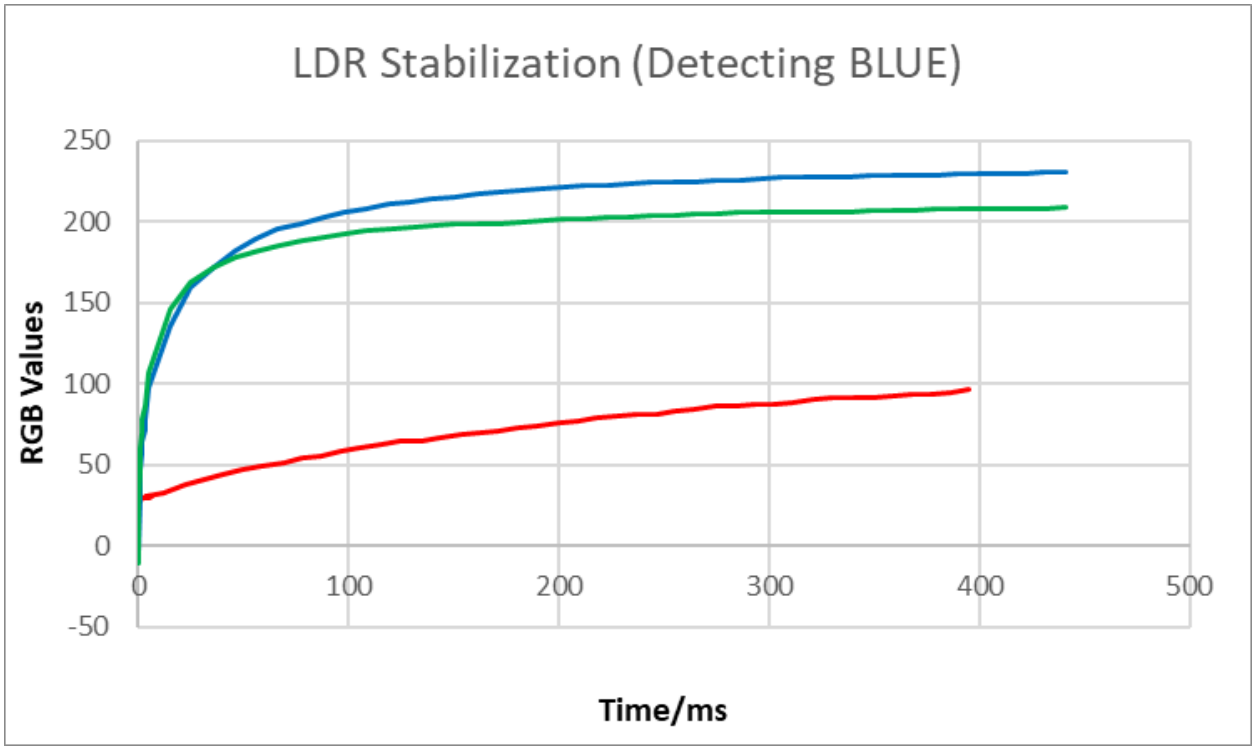

Figure 20: Stabilisation duration when detecting GREEN

Calibration of Black/White/GreyDiff Arrays

The colour detector underwent calibration using the setBalance code. This function executes a dual calibration process for both white and black papers, capturing RGB readings for each colour and subsequently computing the differences between black and white readings. The resulting values, referred to as the 'grey difference', are stored in an array. This array establishes a spectrum between black and white, serving as a reference range for all other colours. By analysing the combination of red, green, and blue light reflected, the calibrated colour detector can effectively determine and identify various colours within this established range.

```
void setBalance() {
 // Prompting user to place white, black paper and taking calibration
readings
  Serial.println("BEGIN WHITE/BLACK CALIBRATION");
  for (int i = 0; i < 2; i++) {
    (i == 0) ? Serial.println("WHITE PAPER") : Serial.println("BLACK
PAPER");
    delay(5000);
    for (int j = 0; j <= 2; j++) {
      shineRGB(j);
      delay(RGB_WAIT);
      if (i == 0) {
        whiteArray[j] = getAvgReading(CAL_NO);
```

```
      } else {
        blackArray[j] = getAvgReading(CAL_NO);
        greyDiff[j] = blackArray[j] - whiteArray[j];
      }
      shineNone();
      delay(RGB_WAIT);
    }
  }
  Serial.println("CALIBRATION COMPLETE");

  // Printing out array values
  for (int i = 0; i < 3; i++) {
    Serial.print("int ");
    Serial.print(calibrationStr[i]);
    Serial.print("[] = {");
    for (int j = 0; j < 3; j++) {
      if (i == 0) {
        Serial.print(whiteArray[j]);
      } else if (i == 1) {
        Serial.print(blackArray[j]);
      } else {
        Serial.print(greyDiff[j]);
      }
      (j < 2) ? Serial.print(", ") : Serial.println("};");
    }
  }
}
```

Serial Monitor Output

```
BEGIN WHITE/BLACK CALIBRATION
WHITE PAPER
BLACK PAPER
CALIBRATION COMPLETE
int whiteArray[] = {591, 810, 802};
int blackArray[] = {108, 241, 294};
int greyDiff[] = {-483, -569, -508};
```

4.3.5 Mounting of Sensors

We mounted the colour sensors in the empty space underneath our mBot. To minimise ambient light interference, we constructed a black skirting around the breadboard. Recognizing the need for enhanced light isolation, we subsequently augmented the skirting with a second layer. Additionally, we repositioned the colour sensor to maximise its proximity to the ground. This adjustment ensures that the reflected light is of ample intensity for detection, and predominantly captures light reflected off the ground rather than from other surfaces. The details to the process of mounting of the sensor is elaborated in **Section 6.1** Skirting.

4.4 Movement of mBot

In the context of the mBot's movement, we implemented the concept of wishful thinking. Wishful thinking is a top-down approach, assuming that prerequisite functions have already been implemented, then proceeding to implement said functions. We implemented this concept for each of the turns our robot had to complete on the maze. For instance, our application of "wishful thinking" involved the assumption that the code and calibration for simpler movements could be compounded to perform more complex manoeuvres. As such, we were able to simplify the rest of the turns into the following:

Table 4: Wishful Thinking for Orange, Purple and Blue

Colour	Action
Orange	2 left turns within the same grid
Purple	1 left turn followed by moving forward for one grid and another left turn
Blue	1 right turn followed by moving forward for one grid and another right turn

The provided code controls the movements of the mBot, with movement constants dictating the power output to the motors, thereby regulating speed, turning speed, and degree of turns. The forward movement function incorporates additional code to facilitate path straightening, mitigating the risk of collisions with walls. This is achieved through the implementation of the PID function above, which dynamically adjusts the robot's orientation to the right or left based on the deviation from the centre of the path. By leveraging PID control, the robot can autonomously maintain a more accurate and consistent trajectory, enhancing its overall navigation capabilities and avoiding potential obstacles. The oneGridForward() function has been specifically calibrated to move the robot precisely one grid forward. This functionality is used for waypoints designated by Light Blue and Purple. By fine-tuning the movement to cover a single grid, the robot can efficiently traverse the specified distance, before executing its second turn.

```
// Movement Constants
#define FORWARD_SPEED 255
#define TURN_SPEED 255
#define TURN_TIME 310
#define UTURN_TIME 580

void moveForward() {
  float deviation = computePID();
  //Serial.println(deviation);
```

```
  if (deviation == -1) {
    leftMotor.run(-FORWARD_SPEED);
    rightMotor.run(FORWARD_SPEED);
  }
  else {
    leftMotor.run(-FORWARD_SPEED - deviation);
    rightMotor.run(FORWARD_SPEED - deviation);
  }
}

void stopMotor() {
  leftMotor.stop();
  rightMotor.stop();
  delay(1000);
}

void oneGridForward() {
  leftMotor.run(-FORWARD_SPEED);
  rightMotor.run(FORWARD_SPEED);
  delay(750);
  stopMotor();
}

void leftTurn() {
  leftMotor.run(TURN_SPEED);
  rightMotor.run(TURN_SPEED);
  delay(TURN_TIME);
  stopMotor();
}

void rightTurn() {
  leftMotor.run(-TURN_SPEED);
  rightMotor.run(-TURN_SPEED);
  delay(TURN_TIME);
  stopMotor();
}

void uTurn() {
  leftMotor.run(TURN_SPEED);
  rightMotor.run(TURN_SPEED);
  delay(UTURN_TIME);
  stopMotor();
}

void doubleLeftTurn() {
  leftTurn();
  oneGridForward();
  leftTurn();
}

void doubleRightTurn() {
  rightTurn();
  oneGridForward();
```

```
  rightTurn();
}
```

4.5 Celebratory Tune

The celebratory tune is triggered when the robot identifies a black line, halts its movement and subsequently detects the underlying surface colour as white. This is executed through the "celebrate()" function, which processes two arrays in tandem. These arrays house musical notes and their corresponding durations. The celebrate() function concurrently traverses both arrays, extracting note and duration pairs. These extracted values are then passed as arguments to the buzzer.tone() function, resulting in the playback of the music.

```
void celebrate() {
for (int i = 0; i < 31; i++) {
  int noteDuration = 1000 / CrazyFrog_duration[i];
  //Plays each note/duration pair
  buzzer.tone(CrazyFrog_note[i], noteDuration);
  //Here 1.05 is tempo, increase to play it slower
  int pauseBetweenNotes = noteDuration * 1.05;
  delay(pauseBetweenNotes);
  }
buzzer.noTone();
}
```

5. Work Distribution

Table 5: Work Distribution

Name	Work
Tan Zhe Hui	• Worked on the colour sensor • Wrote Part of the Code for the mBot • Contributed to the group report
Tay Guang Sheng	• Coded the PID of the mBot • Worked on the PID • Contributed to the group report
Tejas Ananth Kumar	• Worked on the IR sensor and its circuit • Worked on the ultrasonic sensors and its wiring • Contributed to the group report
Teh Ze Xue	• Worked on the colour sensors and also with the calibration • Coded the IR sensor of the mBot • Worked on the IR sensor and its circuit • Contributed to the group report.

6. Overcoming Challenges

6.1 Colour Calibration Error

Location for Calibration

Initially, we conducted our calibrations on the side tables of the laboratory. However, we noticed that the colour values calibrated at these benches did not consistently detect the correct colour in the maze. This inconsistency arose because the values detected at both locations slightly differed. The maze's white walls reflected ambient light, creating a brighter environment compared to the side tables. Consequently, to account for these variations in ambient light conditions during actual runs, we performed the calibrations for both the IR and colour sensors directly within the actual maze.

Choice of Resistance

In order to stay within the 8 mA current limit of the 2-to-4 Decoder, we initially calculated that a minimum resistance of 625Ωwas necessary for each LED, based on Ohm's Law. However, when we applied these values in our colour sensor circuit, we faced a challenge. The RGB readings for white and black—critical for determining the grey difference—were too subtle. This issue was traced back to the excessive brightness of all three LEDs, leading to unreliable sensor readings. To resolve this, we adjusted the resistance values through a process of careful experimentation. We found that the sensor's accuracy improved significantly when the green and blue LEDs were each connected with a 10k ohm resistor in series. Meanwhile, for the red LED, a 3.2k ohm resistor in series proved to be more effective. This specific resistance for the red LED was chosen to enhance its brightness, considering that red light has a longer wavelength than blue and green light.

Choice of Algorithm

In the initial stages, our team implemented a method based on logic statements to interpret the data from the colour sensor, as detailed in **Annex 7.1**. This approach involved defining specific logic conditions for each Colour, based on the range of RGB values detected by the sensor. For instance, if the sensor reads a certain range of red, green, and blue values, the algorithm classifies the colour accordingly in a range. However, this technique, while straightforward, was found to be prone to errors. The colour detector frequently misinterpreted colours, especially when encountering shades that fell on the boundaries of our predefined logic ranges. This is especially so for colours red and orange, blue and purple.

Recognizing the need for a more reliable method, we transitioned to an alternative approach using the sum of squares method. This revised strategy involved comparing

the detected colour values with an array of pre-calibrated colours. For each calibrated colour, we calculated the sum of the squared differences from the detected Colour values. We then identified the calibrated colour with the smallest difference from the detected colour. This method proved far more robust than our initial logic statement-based approach. It allowed for a better distinction between colours, particularly in cases where the detected Colour values were close. As a result of this significant improvement in accuracy, we decided to adopt the sum of squares method for our colour sensor.

Skirting

The continuous testing of our mBot has led to wear and tear on the skirting, which in turn can let in unwanted light, adversely affecting the accuracy of colour detection. To counteract this issue, we have intentionally designed the outer layer to be adjustable, ensuring an optimal height that effectively blocks out all ambient light (Refer to **Figure 21**). This design element is crucial, as it allows us to manually lower the skirting before each test run. By doing so, we ensure that the shield is as close to the ground as possible, which is key to maintaining high accuracy levels.

This ability to adjust the skirting has proven to be a significant enhancement to our mBot's functionality. When the skirting is positioned in close proximity to the ground, it effectively minimises any external light interference. This adjustment has proven to be effective, contributing to an enhanced level of accuracy for our mBot. The close proximity of the skirting to the ground minimises external interference and improves the reliability of colour detection, ultimately optimising the performance of our robotic system.

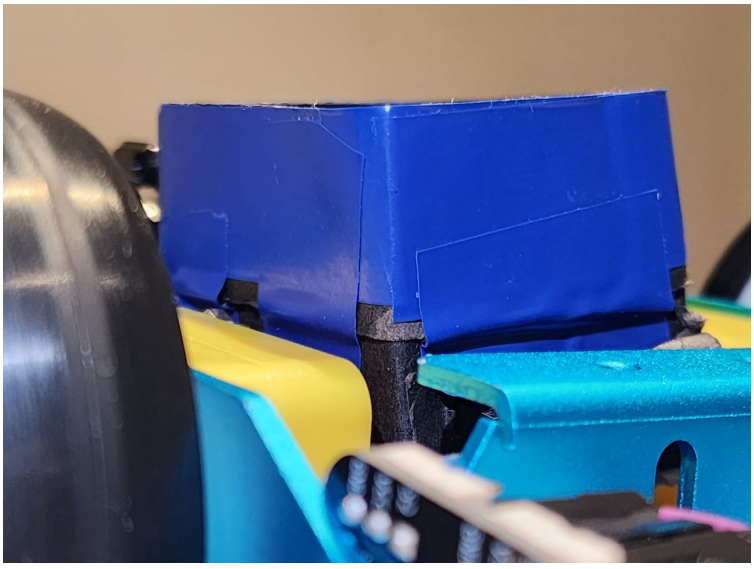

Figure 21: Adjustable Skirting

Position of LDR

We encountered a persistent challenge where the robot's colour detection was error-prone. Despite redoing the skirting and undergoing multiple recalibrations, the issue persisted. Upon closer investigation, we pinpointed the problem to the Light Dependent Resistor (LDR), which was shifting during the robot's movement, intermittently obstructing some of the LEDs and causing inconsistent readings. To address this issue, we implemented a solution by securing the LDR in place using black paper and tape (Refer to **Figure 22**). This measure prevented the LDR from moving and obstructing the LEDs, thereby ensuring stable and accurate colour readings.

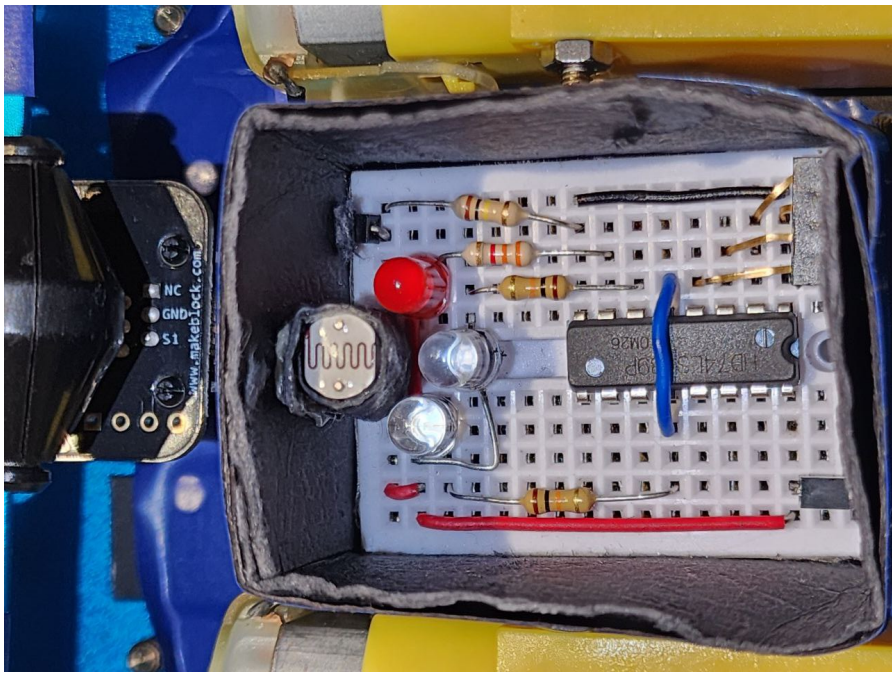

Figure 22: Fixing the Position of LDR

6.2 PID Correction

Before implementing the PID function, our team used the 'nudgeLeft' and 'nudgeRight' methods (Refer to **Annex 7.2**) to correct the robot's path deviations. This strategy, however, led to complications such as overcorrection and undercorrection, resulting in either maze wall collisions or a zigzagging robot trajectory. Recognizing these limitations, we shifted to a PID-based approach. This transition not only reduced the risk of collisions but also ensured a smoother and more accurate path for the robot.

6.3 Hardware Challenges

Wire Management

We encountered an unexpected challenge with cable management, where loose cables posed issues for both the ultrasonic and IR sensors. These cables not only occasionally interfered with the sensors but also dragged along the floor as they had to navigate around the skirting to connect to the breadboard of the colour sensor. To address this issue, we implemented a solution: taping down the loose wires to the sides of the robot and creating small holes in the skirting to feed the wires through to the colour sensor. This not only secured the cables in place but also ensured a cleaner and more organised setup, minimising potential disruptions caused by the cables.

Components not Secured

We faced a recurring issue where our robot consistently veered to the left, despite the proper functioning of the PID function. After thorough debugging, we pinpointed the cause: the left motor's screws were loose, subtly affecting the robot's trajectory. To address this, we tightened the left motor's screws and subsequently checked and secured all other components. Additionally, we established a routine practice of inspecting all screws and ensuring equipment stability before each maze run. This proactive approach not only resolved the current problem but also aimed to prevent similar issues in the future, maintaining the overall integrity of our robot's performance

7. Annexes

7.1 Annex A: Alternative Code for Colour Sensor

First iteration of colour sensor code, using logical statements to determine the detected colour.

```
//red
     if (colourArray[0] > 245 && colourArray[1] < 135 &&
colourArray[2] < 115) {
    det = 0;
  }
  //green
     if (colourArray[0] < 150 && colourArray[1] > 160 &&
colourArray[1] < 200 && colourArray[2] < 150 && colourArray[2] >
110) {
    det = 1;
  }
  //orange
     if (colourArray[0] > 245 && colourArray[1] > 140 &&
colourArray[1] < 180 && colourArray[2] < 125) {
    det = 2;
  }
  //purple
       if (colourArray[0] < 135 && colourArray[0] > 170 &&
colourArray[1] > 130 && colourArray[1] < 170 && colourArray[2] <
220 && colourArray[2] > 180) {
    det = 3;
  }
  //blue
       if (colourArray[0] > 150 && colourArray[0] < 190 &&
colourArray[1] > 220 && colourArray[2] > 220) {
    det = 0;
  }
```

7.2 Annex B: Code for Nudging

First iteration of path adjustment code, utilising nudgeLeft and nudgeRight functions

```
void nudgeLeft() {
  leftMotor.run(NUDGE_SPEED);
  rightMotor.run(NUDGE_SPEED);
  //delay(10);
  stopMotor();
}

void nudgeRight() {
  leftMotor.run(-NUDGE_SPEED);
  rightMotor.run(-NUDGE_SPEED);
  //delay(TURN_TIME);
  stopMotor();
}

void adjustPath() {
    float currentDistance = ultrasonicSensorDistance();
    if (currentDistance > TARGET_DISTANCE * 2) {
      currentDistance == 27 - (filteredDistance() + 9);
    }
    if (currentDistance > TARGET_DISTANCE && currentDistance <
TARGET_DISTANCE * 2) {
      nudgeRight();
    } else if (currentDistance < TARGET_DISTANCE) {
      nudgeLeft();
      return;
    }
}
```