CG1111A Engineering Principles & Practice I

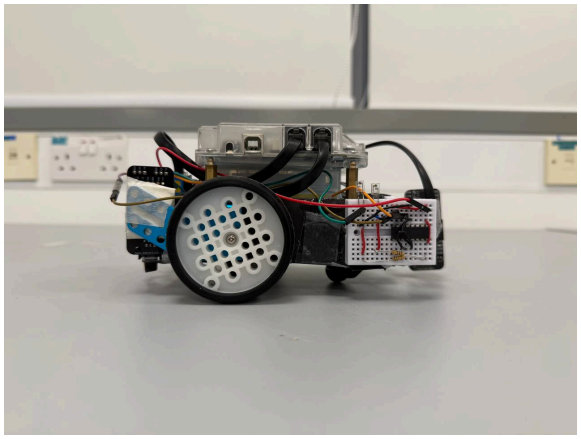
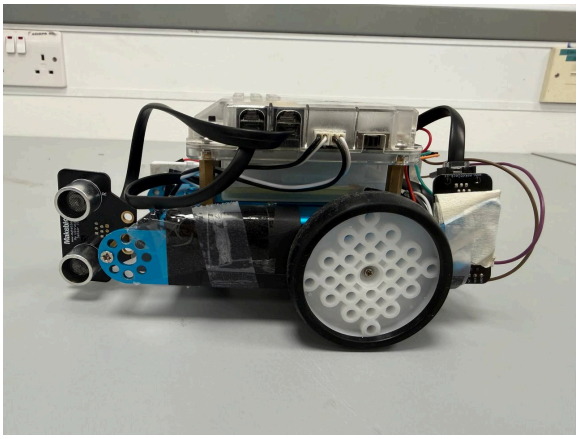
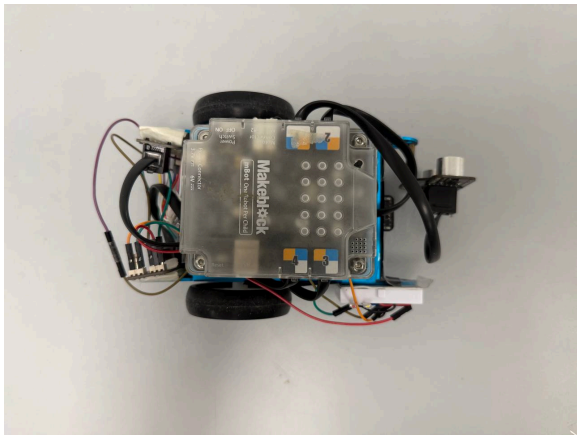
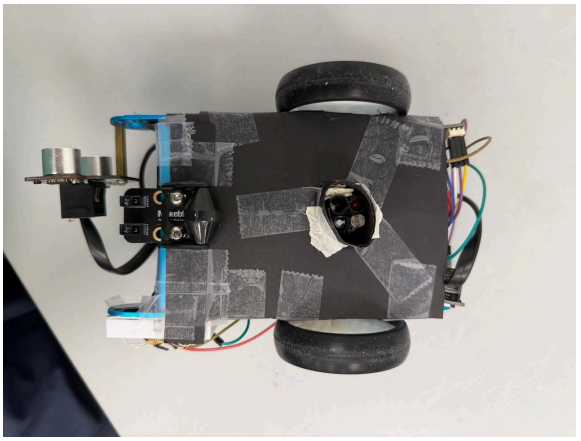
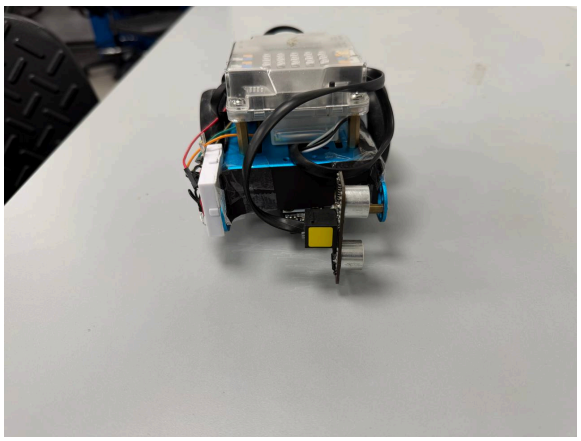
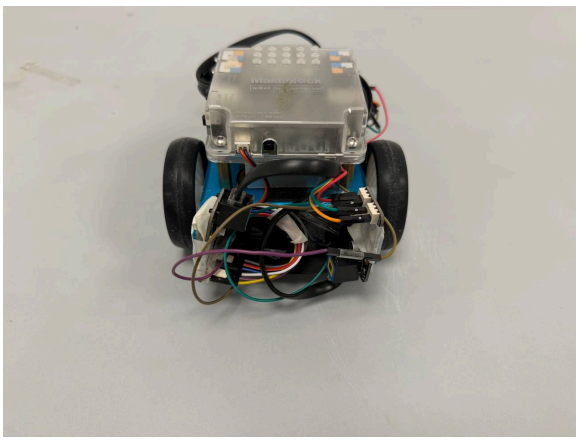
The A-maze-ing Race Project 2025

Studio Group Number	B04
Section Number	S1
Team Number	T2
Team Members	Christopher Widyadi Duong Nghiep Hoang Gangadhar Nandi Gautham Garg Divyansh

Table of Contents

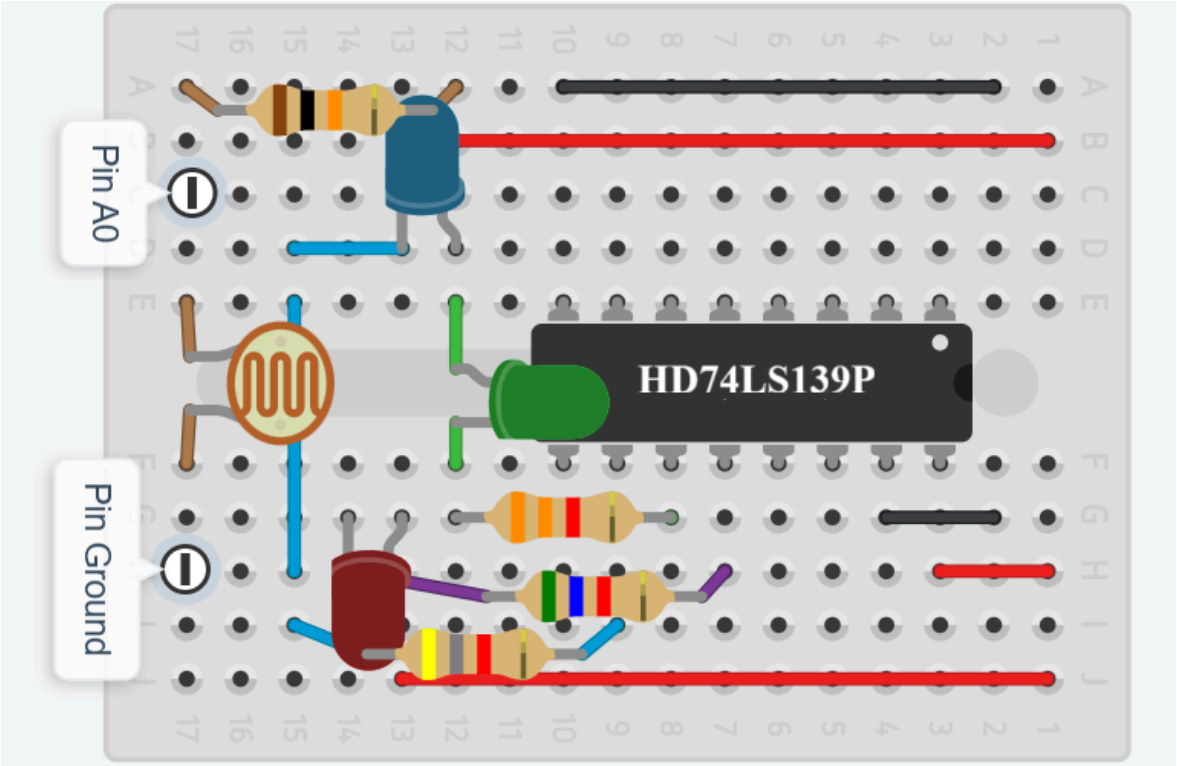
1 Overview

1.1 Pictures of the mBot

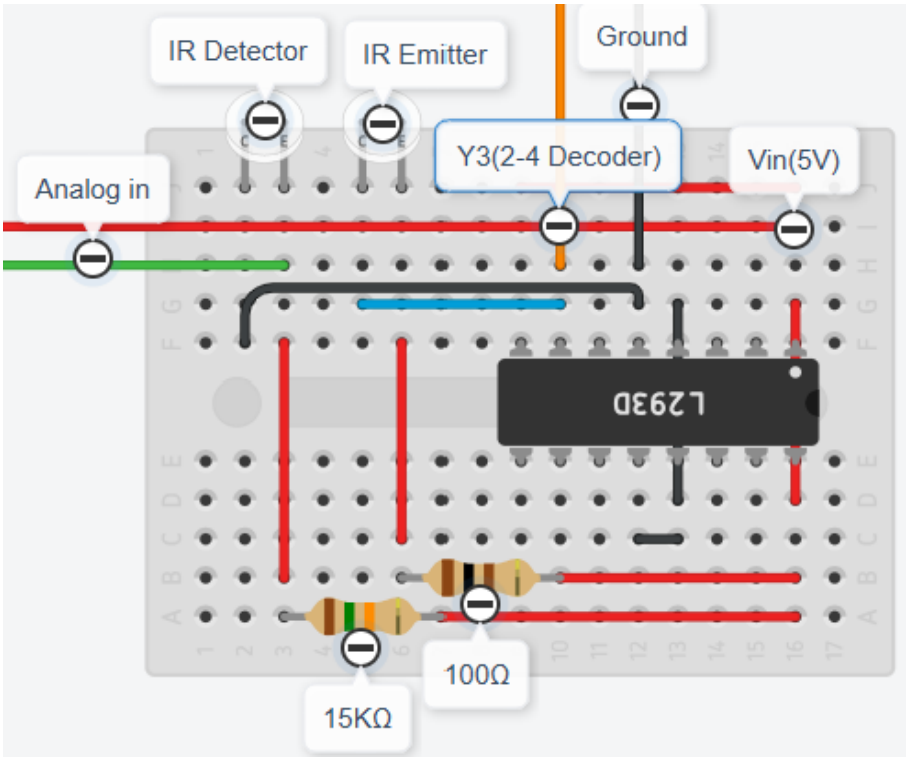
Right View	Left View
	
Top View	**Bottom View**
	
Front View	**Back View**
	

1.2 TinkerCAD Circuit Model

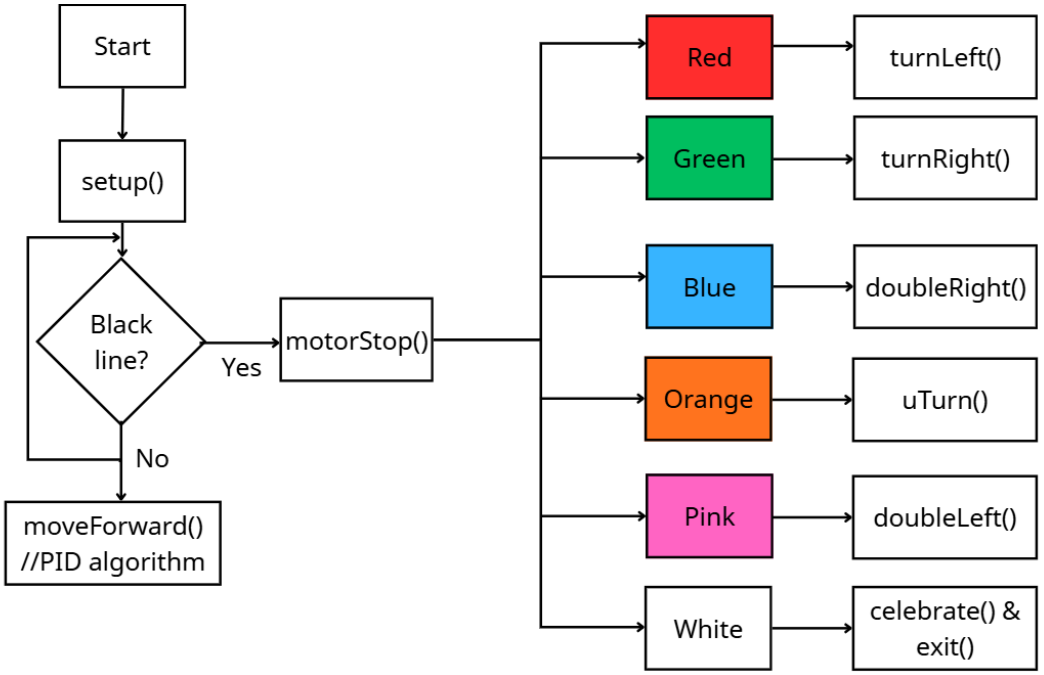
Colour Sensor



IR Sensor



1.3 Overview of the Algorithm



The algorithm begins with the setup() function, which initializes the system by configuring pinMode() settings and declaring necessary variables. Subsequently, our mBOT continuously monitors for the presence of a black line while the robot moves forward, utilizing a PID control algorithm for path tracking. Upon detecting a black line, the motors halt, and the system proceeds to identify the color beneath the sensor. Based on the detected color, the robot executes the corresponding predefined action.

2 Work Distribution

Name	Contribution
Christopher Widyadi	● Wrote the code for navigation, ultrasonic sensor, IR sensor, detecting colours and the main loop ● Worked on project report
Duong Nghiep Hoang	● Wrote the systematic code for the color-sensing circuit and IR sensor. ● Worked on the project report. ● Contributed to testing and fixing the functionality of the mBot.
Gangadhar Nandi Gautham	● Designed the IR sensor circuit ● Helped write code for the IR detection and navigation ● Worked on project report
Garg Divyansh	● Designed the color-sensing circuit and tidied the wires for the same. ● Made the skirting for the color-sensing circuit. ● Calibrated the white and black arrays, along with all the colours. ● Worked on the project report.

3 Colour Sensor

3.1 Overview

All colours are made up of varying proportions of three primary colours - red, green, and blue. White light is made up of all colours; when white light strikes an object, some colours are absorbed by the material and the others and reflected back. The colour of an object can be determined by the amount of red, green and blue light reflected from its surface. For example, a red object will absorb green and blue light and reflect most red light which will cause our eyes to detect the object as red. This is the basic working principle of our colour sensing circuit, which will shine red, green and blue light on the coloured tiles and, depending on the intensity of red, green and blue light reflected from the surface, determine its colour based on the known characteristic RGB components of the light reflected from each of the coloured surfaces (Red, Green, Blue, Pink, Orange, White).

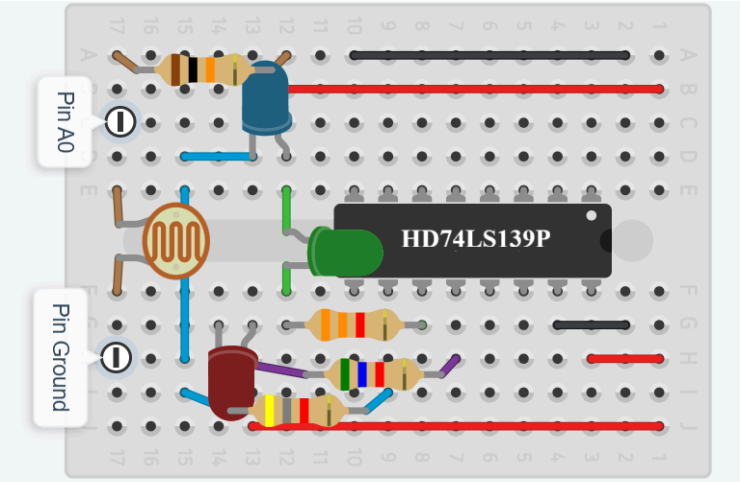


Figure 5.1: Colour-Sensing Circuit

3.2 Working

3.2.1 Overview

We have used three LEDs (Red, Green, Blue), a Light-Dependent Resistor (LDR) and 2-to-4 decoder to build the colour-sensing circuit. The mBot has a total of 8 available pins, 4 of which are already being used by the ultrasonic sensor and the line sensor (each of them use 2 each) and 1 is being used by the IR receiver. Out of the 3 remaining pins, we have to use 1 pin (A0) to measure the LDR reading. Therefore, we are using the 2-to-4 decoder so that we can control the 3 LEDs using 2 mBot pins (A2 and A3).

When the line sensor detects the black strip of paper, the mBot stops. Thereafter, we repeatedly toggle the states of pins A2 and A3 to turn on the Red, Green and Blue LEDs. For each LED, we take the voltage-divider reading of the LDR from pin A0. Each time while taking an LDR reading, we take an average of 5 readings, taken at intervals of 10 milliseconds, from the LDR for each LED using the *getAvgReading(int times)* function. Thereafter, we modify the LDR readings so that it matches the RGB colour code, where each reading is in the range 0-255.

3.2.2 Workflow

***void input(int code)*:**

We referred to the following table while toggling the states (HIGH or LOW) of pins A2 and A3 to switch on Red, Green and Blue LEDs via the 2-to-4 decoder. We ensured a delay of 200 milliseconds between 2 successive LEDs so that the LDR readings for one LED does not interfere with the LDR readings for another LED. This is accomplished using the *input(int code)* function, where *code* corresponds to the number of the output which has to be tuned LOW.

Table I: Function Table Summarizing the Behaviour of the HD74LS139P 2-to-4 decoder

Inputs			**Outputs**				
Enable	**Select**						
G	**B**	**A**	**Y0**	**Y1**	**Y2**	**Y3**	**LED switched on**
H	X	X	H	H	H	H	None
L	L	L	L	H	H	H	Red
L	L	H	H	L	H	H	Green
L	H	L	H	H	L	H	Blue
L	H	H	H	H	H	L	IR sensor

***void* setBalance()*:**

We start by calibrating the colour sensor. We did this before the actual program execution using the *setBalance()* function. To calibrate the colour sensor, we took the LDR readings for each LED using the above method, once while placing the mBot on white paper and then once for black paper. We manually took the LDR readings for white and black fields under different lighting conditions and then hard-coded the average of those raw values in two single-dimensional arrays - *whiteArray[]* and *blackArray[]*. Thereafter, we calculated the greyscale by subtracting the respective values of the *blackArray[]* from the *whiteArray[]* and stored the resultant greyscale in *greyDiff[]*.

***char* color_sensing()*:**

The *color_sensing()* function uses the same approach as the *setBalance()* function to obtain the RGB LDR readings for every colour. We need to modify these readings because the analog pins of the arduino output values in the range 0-1023 (0V-5V) and the range of the values in the RGB colour code is 0-255 (e.g. the RGB colour code for Red is 255, 0, 0). Since we already have the RGB values for white and black surfaces, we now have the upper and lower limits of the LDR's readings. We can now use these values to calculate the RGB values for each colour which is consistent with the RGB colour code format. This can be obtained by using the following formula.

$$\frac{Avg\ LDR\ Readings - Black\ Values}{Grey\ Values} * 255$$

This formula is used for the LDR readings obtained for every LED and then the resulting value is stored in the single-dimensional array *colourArray[]*. In code, it looks like this.

```
colourArray[c] = (colourArray[c] - blackArray[c]) / (greyDiff[c]) *
255.0
```

Thus, we have now obtained the RGB colour code for each waypoint challenge. Thereafter, this function calls the *classifyColour(float Red, float Green, float Blue)* to ascertain and then return the colour of the corresponding tile.

P.S. *colourArray[]*, *whiteArray[]*, *blackArray[]* and *greyDiff[]* store values in the following format.

```
(<RED VALUE>, <GREEN VALUE>, <BLUE VALUE>)
```

***float euclidian_distance(float r, float g, float b, float r2, float g2, float b2)*:**

Before talking about the *classifyColour(float Red, float Green, float Blue)* we must talk about the Euclidian Distance algorithm which we have employed while classifying the colour. The Euclidian Distance algorithm is used to calculate the shortest distance between two points by taking the square root of the sum of the squared differences for each dimension. We are using this algorithm to return the shortest distance between two arrays. To reduce complexity, we are not taking a square root for the sum of the squared differences. Instead, we are just returning the sum of the squares of the differences between r and r2, g and g2, b and b2. In code, it looks like this.

```
float dr = r - r2, dg = g - g2, db = b - b2;
return dr * dr + dg * dg + db * db;
```

We need the Euclidian Distance formula because before the actual program execution, we just invoked the *color_sensing()* from *void loop()* so that we can obtain the RGB values our colour-sensing circuit was returning for each coloured tile (Red, Green, Blue, Pink, Orange, White). Thereafter, we stored these RGB colour codes for each colour in the double-dimensional array *colours_values[7][3]*. In code, it looks like this.

```
int colour_values[7][3] = { {286, 405, 230},
                  {0, 0, 0},
                  {259, 163, 105},
                  {271, 207, 130},
                  {115, 212, 105},
                  {252, 338, 205},
                  {95, 193, 145},
                 };
char* possible_colours[7] = {"White", "Black", "Red", "Orange", "Green",
"Pink", "Blue"};
```

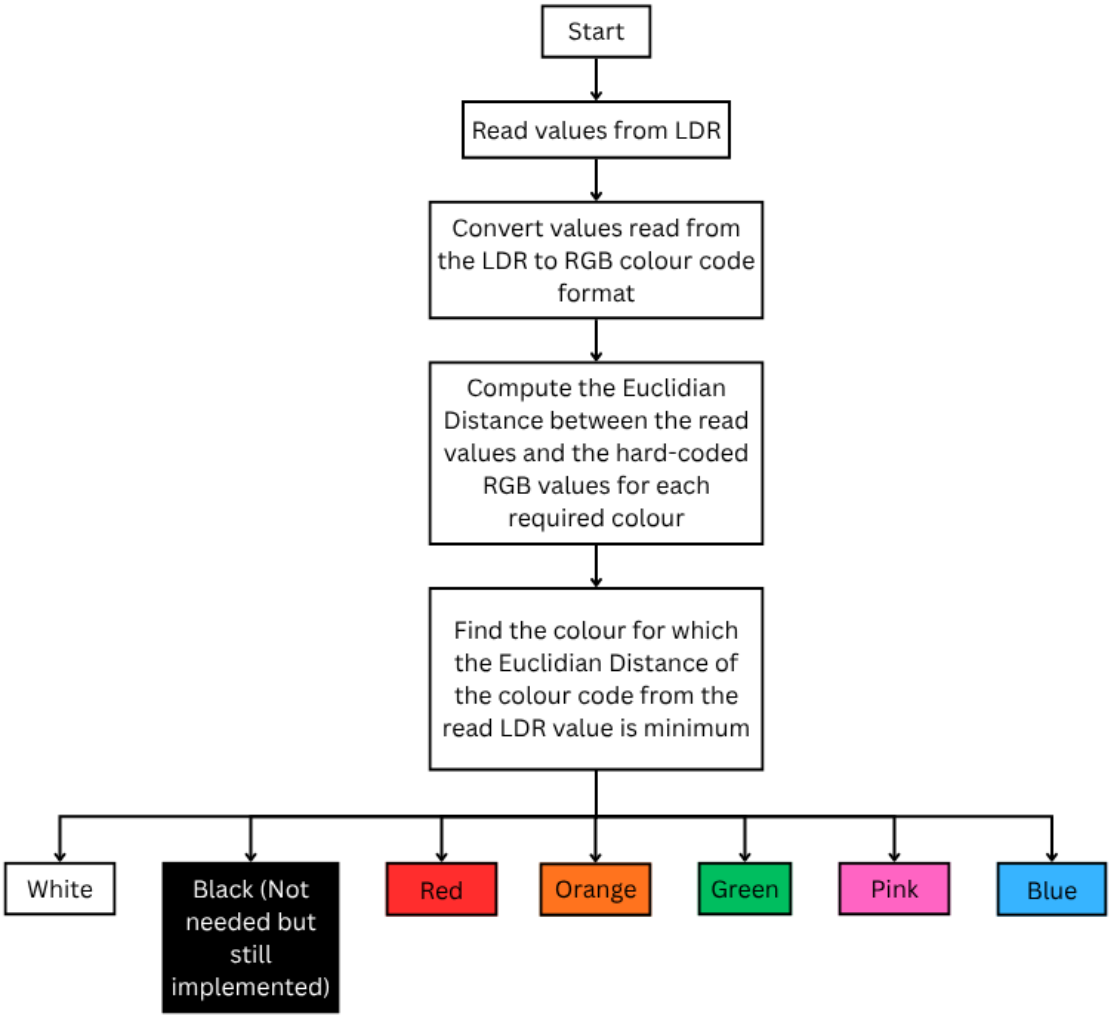
The *possible_colours[]* array stores the colour of the different tile, in order such that the RGB colour code at each index of *colour_values[7][3]* corresponds to the colour at the same index of

possible_colours[]. Since the LDR readings will not be exactly the same, they still will be fairly close to the hard-coded values. Thus, we use the Euclidian distance formula to return the 'distance' between the RGB colour code returned from the LDR and the hard-coded RGB colour code of each colour in *colour_values[7][3]*.

***char* classifyColour(float Red, float Green, float Blue)*:**

Finally, we use the *classifyColour(float Red, float Green, float Blue)* to ascertain the colour corresponding to the RGB colour code obtained from the LDR. It loops through the colour codes stored in *colour_values[7][3]* and calculates the Euclidian Distance between the obtained LDR reading in RGB format and each of the colour codes. Thereafter, it stores the colour for which the Euclidian Distance is the least in *return_colour*. Finally, the function returns *return_colour*.

3.3 Colour Sensor Flowchart



4 Navigation

4.1 Ultrasonic Sensor

4.1.1 Overview

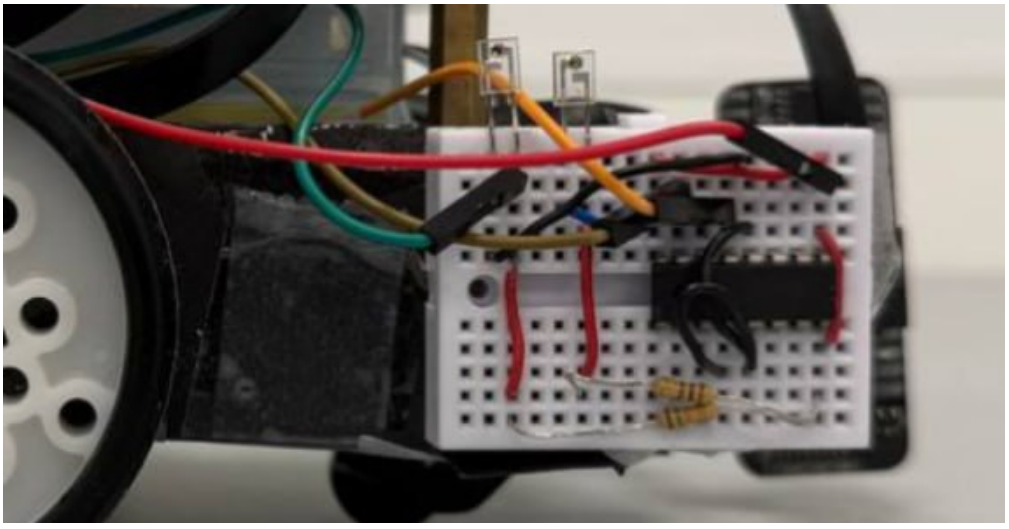
An ultrasonic sensor serves to measure the distance between it and an object by emitting an ultrasonic wave and measuring the time taken for the wave to travel to a surface and reflect back to its receiver. The distance is then computed by multiplying half of this travel time by the speed of sound (approximately 340 m/s). This sensor is used with our proportional control algorithm to maintain a constant distance from the wall.

4.1.2 Mounting the Ultrasonic sensor

For the ultrasonic sensor, we mounted it on the front of the mBot, protruding out of its chassis at a 90 degree angle. It was previously mounted at a lower angle, but the problem was that it kept detecting the chassis instead of the walls, so we decided on the 90 degree angle.

Furthermore, another problem we encountered was that it sometimes detected wooden side walls of the playfield, treating them as main walls. This created jitter and unpredictable behavior when no main wall was present. However, after testing it with the PID system, we found that it did not change the results in any significant way, hence we decided that it was not a great concern.

4.2 IR Sensor



4.2.1 Overview

As the robot approaches the wall, the intensity of reflected infrared (IR) light increases. This results in a higher base current generated by the phototransistor. An increase in base current leads to a greater collector current, causing more current to flow through the load resistor. Consequently, the voltage at the input pin drops, indicating the robot’s proximity to the wall.

4.2.2 Reducing ambient light

As the emitter turns on, the analog reading between the two terminals of the IR receiver reflects a combination of ambient light and IR emission. To reduce the impact of ambient light, the program first measures the value before turning on the IR emitter. The final reading is calculated as the difference between the values obtained with the IR emitter turned on and off, isolating the contribution of the emitter from ambient interference.

4.2.3 Implementation of the IR Sensor

read_ir_distance() is designed to get a stable reading from the IR sensor.

The code takes two readings (*val_before*, *val_after*) separated by a delay. We turn the IR emitter ON and OFF in a timeframe of 100 microseconds by calling *input*(3) and then turning it off by calling *input*(0). *val_after* - *val_before* isolates the strength of the reflected signal by subtracting the ambient noise, which is then converted into *voltage*.

```
float read_ir_distance() {
    input(0);
    float val_before = analogRead(IR_RECIEVER_PIN);
    delayMicroseconds(100);
    input(3);
    float val_after = analogRead(IR_RECIEVER_PIN);
    delayMicroseconds(100);

    float voltage = (val_after - val_before) * 5.0 / 1023.0;
    float distance = 20.495 * pow(voltage, -1.1904);

    return distance;
}
```

The final line returns a *distance* that maps the sensor's non-linear voltage output to a calculated distance in centimeters, using an equation derived from our experiments.

4.3 PID Navigation

4.3.1 Finding distance

This code implements a sensor-fallback system to get a reliable distance measurement. It prioritizes the ultrasonic sensor and only uses the IR sensor as a backup when the ultrasonic sensor's readings are considered erroneous. This is due to the unreliability of the IR sensor.

```
long find_distance() {
    long ultrasonic_dist = ultrasonic_distance();

    if (ultrasonic_dist == 0 || ultrasonic_dist > 20) {
        use_ir_sensor = true;
        return read_ir_distance();
    }
    return ultrasonic_dist;
}
```

The function first attempts to get a distance from the **ultrasonic_distance**() function. It then checks if the ultrasonic reading is 0 (failure or timeout) or > 20 (outside the reliable range for PID). If the ultrasonic reading fails this check, it uses the IR sensor. It sets the *use_ir_sensor* flag to true and returns the value from **read_ir_distance**().

4.3.2 Motivation for a (P)ID System

While navigating the maze, the mBot must travel in a straight path without hitting walls. Simply setting both motors to the same speed does not guarantee a straight line. Due to manufacturing inaccuracies and environmental factors, motors rarely spin at identical speeds. In our case, the mBot consistently drifted to the right. To correct this, we implemented a Proportional (P) control algorithm in the PID control system that uses the ultrasonic sensor, and the IR sensor to maintain a set distance from the right wall.

4.3.3 Proportional Control

A P-controller is a type of closed-loop feedback algorithm that aims to maintain a desired state by making corrections based on feedback. Its core components are:

1. **Setpoint**: The desired state for the system. In our case, the setpoint is 8 cm away from the right wall.
2. **Error**: The difference between the current state and the setpoint. (Error = Current Distance - 8 cm).
3. **Output**: The correction applied to the system. For a P-controller, this output is directly proportional to the error.

The algorithm takes the distance from the ultrasonic sensor as its input. It then calculates an error and outputs a motor speed correction that is proportional to that error. This process steers the mBot back to the 10 cm setpoint.

4.3.4 Implementation of a Proportional Control

Since P-control is a feedback loop, it requires constant updates. The ultrasonic sensor must be polled in every iteration of the main loop while the mBot is moving. This allows the algorithm to compute a new correction and apply it to the motors at a high frequency.

Our control algorithm is implemented in two parts: **calculate_PID**(), and **moveForward**(). Firstly, we initialised values of *kp* (constant for the P-term), which was set to 18.0 based on our testing.

The **calculate_PID**() function calculates the proportional component. It calculates the *error* by comparing the desired *setpoint* (8.0 cm) to the distance measured by the ultrasonic sensor. This *error* is then multiplied by the proportional gain *kp* to get the final *pid* output. The result is clamped to +/- 255 to prevent the correction value from exceeding the motor's maximum speed range. Hence, this makes it so that if the mBot is too far, it will return a negative *pid*, and positive *pid* vice versa.

```
double calculate_PID() {
    double kp = 18.0;
    double setpoint = 8.0;
    double distance = find_distance();
    double error = setpoint - distance;

    double pid = kp * error;

    if (pid > 255) pid = 255;
    if (pid < -255) pid = -255;
    return pid;
}
```

The **moveForward**() function takes the *pid* output and applies it to the motors. If the mBot is too far, the *leftMotor* will speed up, and the *rightMotor* will slow down due to a negative *pid*, leading to the mBot turning right, moving closer to the wall. Vice versa, if the mBot is too close, the *rightMotor* will speed up, and the *leftMotor* will slow down due to a positive *pid*, leading to the mBot turning left, moving farther away from the wall.

moveForward() also changes based on whether the distance was from the IR sensor, or from the ultrasonic sensor, as they are facing opposite sides of the mBot. Hence for the IR sensor, we will negate the *speed_PID*.

```
void moveForward() {// Code for moving forward for some short interval}
    speed_PID = calculate_PID();

    if (use_ir_sensor) {
        speed_PID = -speed_PID
        use_ir_sensor = false;
    }

    leftMotor.run(MOTOR_SPEED - speed_PID);
    rightMotor.run(-MOTOR_SPEED - speed_PID);}
```

4.3.5 Why only a P-system was used

A full PID algorithm has three components: Proportional (P), Integral (I), and Derivative (D). After testing, we implemented a P-only controller as it was the simplest, and best performance for our mBot.

1. **Reason for excluding the Integral (I) controller**: The I-term sums errors over time to eliminate steady-state error. In our testing, the mBot did not suffer from significant steady-state error, hence the P-controller was able to bring it near the setpoint. Adding an I-term was unnecessary and would have added complexity.

2. **Reason for excluding the Derivative (D) controller:** The D-term predicts future errors by looking at the rate of change of the error, which helps dampen oscillations. However, the D-term is highly sensitive to sensor noise. Our ultrasonic sensor, like any real-world sensor, produces readings with slight, rapid fluctuations. The D-term amplified this noise, causing the mBot's motors to jitter unpredictably. Hence, we found that it was better for the D-term to be excluded.

4.4 Navigation Logic

To navigate the Maze, we defined some constants that were used to perform the rotations and movement.

Constant	Description
MOTOR_SPEED	Set to **225**. The standard speed for the mBot when moving in a straight line. This value is calibrated with the PID controller and other timings, so it is kept constant to maintain precision.
TURNING_SPEED	Set to **160**. The speed at which the mBot executes turns.
TURN_TIME	Set to **500** ms. The time in milliseconds required for the mBot to complete a 90 degree left or right turn (for red or green tiles).
DOUBLE_LEFT_STRAIGHT_TIME	Set to **1100**. The time in milliseconds the mBot moves straight between the two turns of a double left turn.
DOUBLE_RIGHT_STRAIGHT_TIME	Set to **850**. The time in milliseconds the mBot moves straight between the two turns of a double right turn.

5 Challenges Faced

5.1 IR Sensor Sentivity

Problem Faced

Inconsistency in readings was one of the issues we faced when dealing with our IR circuit. One moment we were getting accurate readings and suddenly it would go haywire. The ambient lighting conditions also affected our readings a lot.

Solution Implemented

We originally chose the resistor values in our circuit to be the same as the ones from a previous studio (150Ω and 8.2kΩ) but it soon proved to be the wrong choice for our use case.

A higher resistance of 15kΩ was used for the IR detector since it provides a very small current that requires high resistance to produce a voltage that is readable and usable.

The emitter resistor was decreased from 150 Ω to 100 Ω to increase the IR LED's output intensity. This stronger IR signal improved the sensor's ability to operate under bright ambient lighting conditions.

5.2 Colour Sensor

Problem Faced

1. External lighting conditions and erroneous calibrations were the main problems we ran into with the Colour Sensor.
2. The different LEDs had different brightnesses, which affected the overall LDR readings, making them inaccurate.

Solution Implemented

1. To decrease the effects of external lighting conditions on our sensor our solution was to isolate the LEDs and LDR on the bottom of the robot using black paper as shielding. A layer of black paper was put on the underside of the robot to stop any light from reflecting onto it and a cutout was made for the sensor around which further skirting was made. We made it long enough so as to not allow a gap between the ground and the skirting which might have allowed external light to enter through. Individual skirting amongst the individual parts of our sensor also led to a drastic increase in accuracy when reading colours, as it prevented light from the LED to directly enter the LDR without reflection.

2. We experimented with different resistance values for the different RGB LEDs until we found the suitable resistances for all the LEDs to be of the same brightness.

6 Appendix

6.1 Accessory Functions

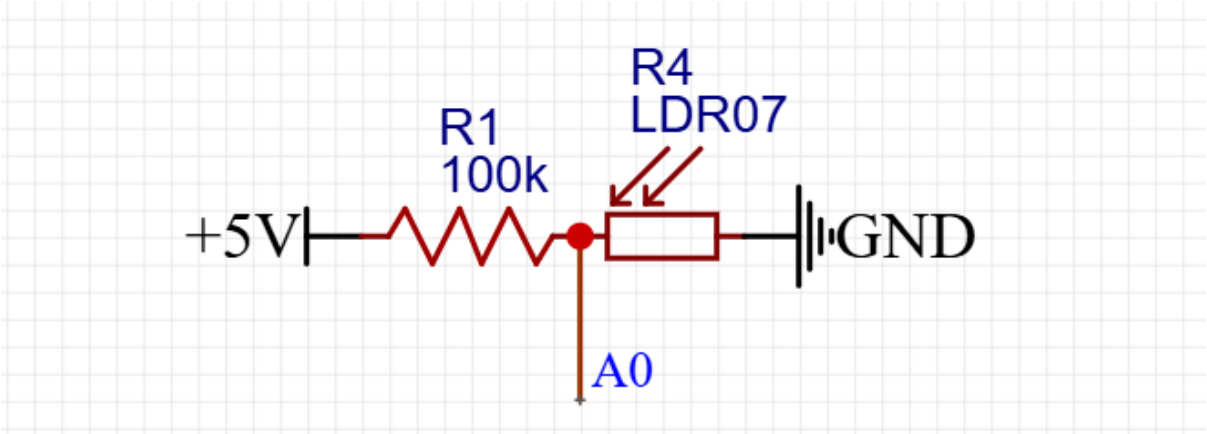
***void turnLedOn(char* s)*:**

This function is used to display the colour detected by the colour-sensing circuit via the in-built LEDs of the mBot - useful for debugging.

***void celebrate()*:**

This function is used to play the short celebratory tune at the end of the maze, when the colour sensor detects white.

6.2 LDR



6.3 Color-sensing, 2-4 Decoder and IR sensor