CG1111A Engineering Principles and Practice I

The A-Maze-ing Race Written Report

Studio Group Number: B01
Section Number: 2
Team Number: 1

Team Members:

Names	Student ID
Beal Ng Zhi Jie	A0272435E
Benjamin Thomas Komatt	A0277469H
Brandon Kang	A0272866N
Bu Yin Shuang	A0282186X

Table of Contents

1. Introduction

In this project, we were tasked to build a robot to find its way through a maze in the shortest time possible. The robot will face several colour challenges at intermediate waypoints while attempting to complete the maze. To complete the maze, the robot has to successfully decipher all the colours accurately at these waypoints that direct the robot to the end of the maze while bumping the walls as little as possible.

2. Overall Algorithm and Design

Overview of Design

Table 1 below shows the overall design of our mBot. The ultrasonic sensor is mounted on the left of the mBot and the IR sensor is mounted on the right. The wirings were kept as compact as possible to have little exposed wires that might brush against the maze walls. The wires on the breadboard were also cut to an appropriate length.

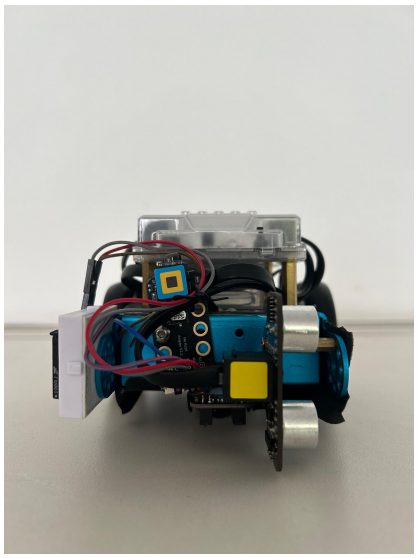
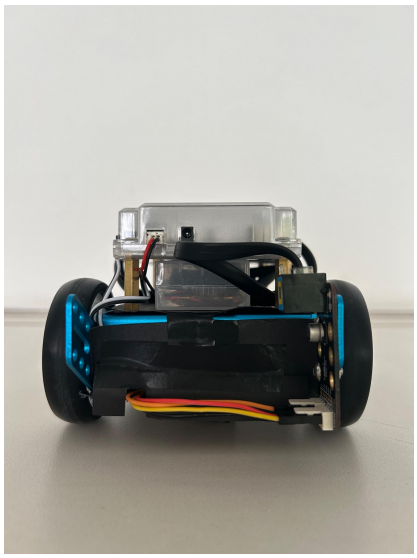
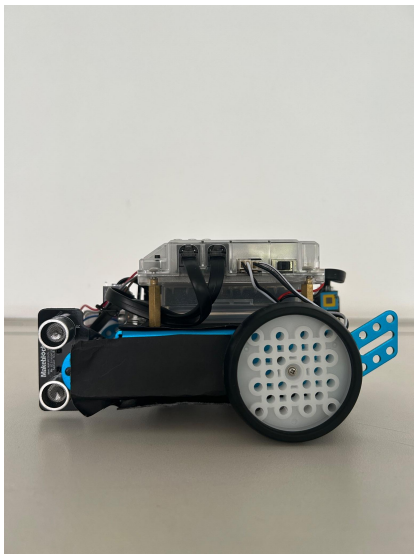
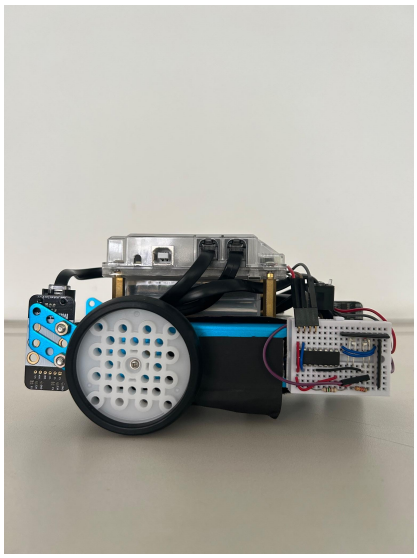
Front View	Back View
	
Left View	**Right View**
	

Table 1: Overview of the mBot design

Overview of Algorithm

The algorithm loops through and looks for a black line. If a black line is detected, it uses the colour sensor to decipher the colour and moves accordingly in the direction the colour corresponds to. If no black line is detected, the mBot continues to move straight and uses the Ultrasonic sensor and IR detector to guide its movements such that it does not bump

into the wall and moves in a straight path. Once the colour white has been detected, the mBot will play a celebratory tune and stop. The overall algorithm can be seen in Figure 1.

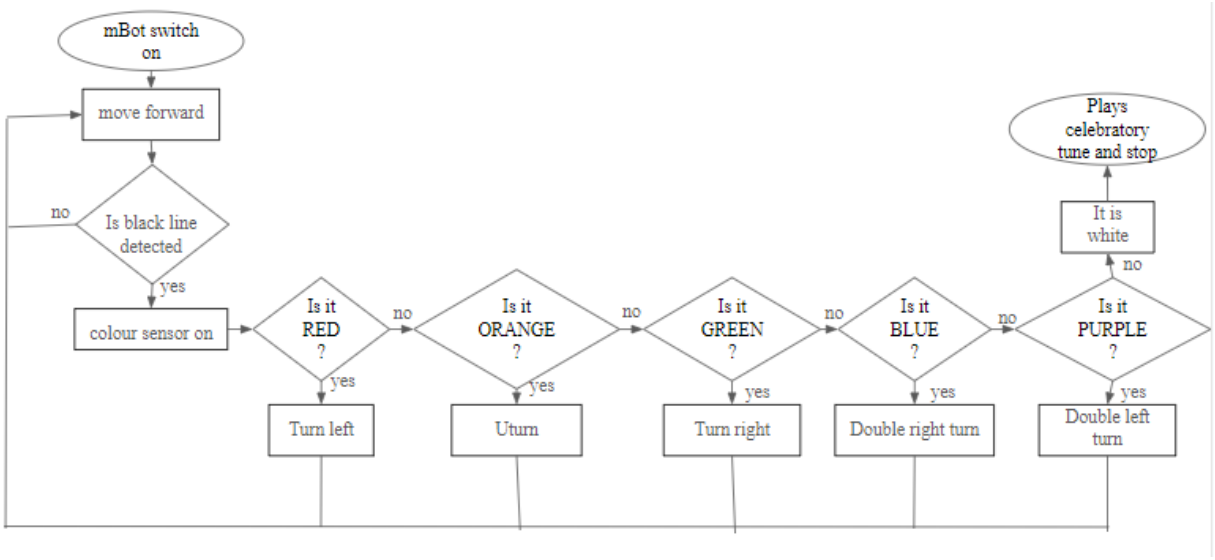


Figure 1: Flowchart of Overall Algorithm

3. Implementation of Various Subsystems

The robot is required to perform multiple functions through its various subsystems. The robot will be relying on two side proximity sensors for it to align itself with the maze, as well as a colour sensing circuit and an IR circuit built on mini-breadboards.

3.1. Movement

3.1.1. Overview

The main movements were moving forward, and stopping. As the mBot moves forward in a straight line, the Ultrasonic Sensor and the IR Sensor are the two side proximity sensors that guide the mBot in a straight line to align itself (Refer to 3.3). The other movements of the mBot are executed depending on the colours detected at the waypoints.

3.1.2. Code

Moving forward and stopping

The speed of the motors was initialised to a variable "motorSpeed" which was set to 250. Since both wheels moved in opposite directions when moving forward (right wheel clockwise and left wheel anti-clockwise), the right motor would have a positive value while the left motor would have a negative value. The functions for moving forward and stopping are shown in Figure 2.

```
void stopMotor() {  // Code for stopping motor
  leftMotor.stop(); // Stop left motor
  rightMotor.stop(); // Stop right motor
}
void moveForward() {  // Code for moving forward for some short interval
  leftMotor.run(-motorSpeed); // Negative: wheel turns anti-clockwise
  rightMotor.run(motorSpeed); // Positive: wheel turns clockwise
}
```

Figure 2: Functions for moving forward and stopping

The Makeblock Line Sensor is used to detect whether there is a black line. It is mounted underneath the mBot and protruding out from the front. It uses two pairs of smaller IR sensors to detect the black line. Thus, the code for the Line Sensor consists of checking whether both of these IR sensors detect black for it to count as a black line detected. If a black line is detected, the mBot will stop and start scanning for the colour.

```
int sensorState = lineFinder.readSensors(); // read the line sensor's state
  if (sensorState == S1_IN_S2_IN) {   //Checks if both sensors detect black
    //Colour detection code
  }
  else {
    //Code to continue moving forward
  }
```

Figure 3: Code for Line Sensor to detect black line

Turning

Depending on the colour detected, the mBot needs to execute one of the 5 types of turns. The 5 types of turns corresponding to their respective colours are shown in Table 2 below.

Colour	Interpretation
Red	Left-turn
Green	Right turn
Orange	180° turn within the same grid
Purple	Two successive left-turns in two grids
Light Blue	Two successive right-turns in two grids

Table 2: Colour interpretation and their corresponding turns

For turning left and right, both motors need to be positive or negative (both clockwise or anti-clockwise) for it to execute a left or right turn. The function for turning left and right accepts a parameter "time" which is the duration to turn left or right. This can be adjusted accordingly depending on the angle the mBot turns (Refer to 3.1.3 for more details on why this was used). The uTurn() function is similar to the function for turning right, however, the duration of the turn is longer. Figure 4 shows the code for turning left, turning right, and U-turn. TURNING_TIME_MS is initialised to 340 at the start of the code. The delay for the uTurn() function was set to (2*TURNING_TIME_MS - 80) as it was the most consistent turning duration for the mBot to make a 180° turn.

```
void turnRight(int time) {
  leftMotor.run(-motorSpeed);
  rightMotor.run(-motorSpeed);
  delay(time);
}
void turnLeft(int time) {
  leftMotor.run(motorSpeed);
  rightMotor.run(motorSpeed);
  delay(time);
}
void uTurn() {
  leftMotor.run(-motorSpeed);
  rightMotor.run(-motorSpeed);
  delay(2*TURNING_TIME_MS - 80);
}
```

Figure 4: Code for turning left, turning right, and U-turn

For two consecutive left or right turns, we are executing the same code for a single left or right turn twice. After the first turn, the mBot will move forward for a short period equivalent to traversing the distance of one grid on the maze, and stop, before executing the second turn.

```
void doubleLeftTurn() {   // Code for double left turn
  turnLeft(TURNING_TIME_MS-15);
  stopMotor();
  moveForward();
  delay(650);
  stopMotor();
  delay(100);
  turnLeft(TURNING_TIME_MS+15);
  stopMotor();
  delay(10);
}
void doubleRightTurn() {  // Code for double right turn
  turnRight(TURNING_TIME_MS-10);
  stopMotor();
  moveForward();
  delay(690);
  stopMotor();
  delay(100);
  turnRight(TURNING_TIME_MS);
  stopMotor();
  delay(10);
}
```

Figure 5: Code for double left turn and double right turn

3.1.3. Difficulties Faced

Initially, we noticed that although the double left turn and double right turn were using the same turnLeft() and turnRight() functions respectively twice, each of the turns were still not turning at the same angle. This issue can be illustrated in Figure 6, which were screenshots from a video taken during testing. The first left turn in Figure 6 when the mBot reaches purple can be seen to be larger than 90°, it then proceeds to move forward by one grid, and the second left turn was much smaller than 90°, even though the same turnLeft() function was called in the doubleLeftTurn() function. This could sometimes lead to bumping against the wall during turning and causing us to take more time to complete the maze.

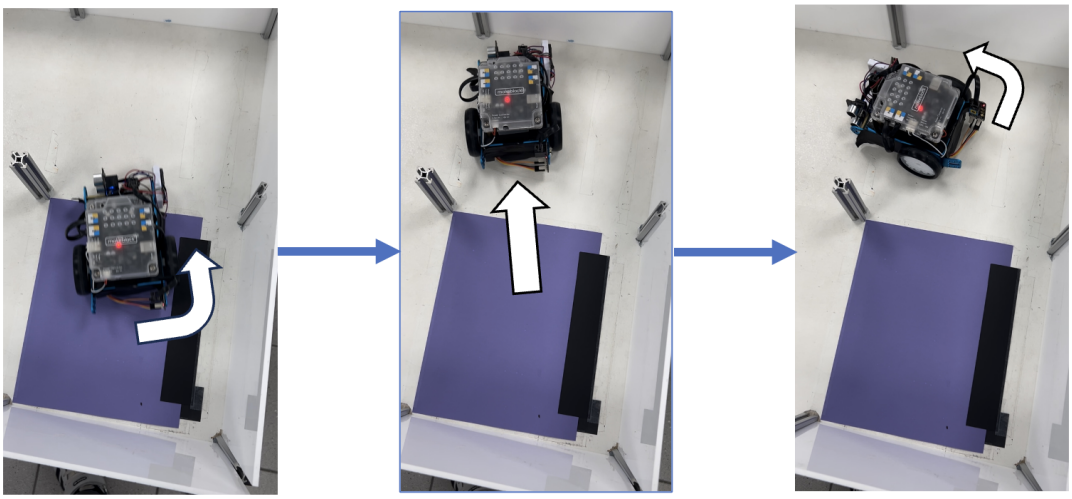


Figure 6: Example of an issue with turning during a double left turn

3.1.4. Solutions

Our code for turning left and right initially did not accept any parameters and fixed the delay time to TURNING_TIME_MS. To resolve this turning issue, we decided to pass in a "time" argument for our turnLeft() and turnRight() functions so that we could manually set the turning time to be longer or shorter depending on whether the turning angle was too much or too little. After incorporating this into our double turn functions and calibrating the turning time for the first and second turn to turn 90°, the double turn functions were more accurate in executing the double turns.

Before

```
void turnLeft() {
  leftMotor.run(motorSpeed);
  rightMotor.run(motorSpeed);
  delay(TURNING_TIME_MS);
}
void doubleLeftTurn() {   // Code for double left turn
  turnLeft();
  stopMotor();
  moveForward();
  delay(650);
  stopMotor();
  delay(100);
  turnLeft();
  stopMotor();
  delay(10);
}
```

After

```
void turnLeft(int time) {
  leftMotor.run(motorSpeed);
  rightMotor.run(motorSpeed);
  delay(time);
}
void doubleLeftTurn() {   // Code for double left turn
  turnLeft(TURNING_TIME_MS-15);
  stopMotor();
  moveForward();
  delay(650);
  stopMotor();
  delay(100);
  turnLeft(TURNING_TIME_MS+15);
  stopMotor();
  delay(10);
}
```

Figure 7: Before vs After Code snippet of changes made for turning left

Figure 7 above shows an example of the changes made to our turnLeft() and doubleLeftTurn() functions, and how we manually adjusted the turning time so that both turns in the double left turn will make the mBot turn 90°. The same solution was also applied to our turnRight() and doubleRightTurn() functions.

3.2. Colour Recognition

3.2.1. Overview

The colour sensor was built using individual red, green, and blue LEDs and we will be using the HD74LS139P 2-to-4 decoder to control which LED to turn on at any given point in time. The LDR is used along with this LED set up in a potential divider to recognize the colour at the waypoint from the voltage values when different LEDs are shone on the colour paper. The code records the R, G, and B values of the unknown colour and determines which colour it is.

3.2.2. Circuit

The hardware portion for colour sensing can be divided into 2 parts, the circuit controlling the LEDs and the LDR circuit for determining the light intensity.

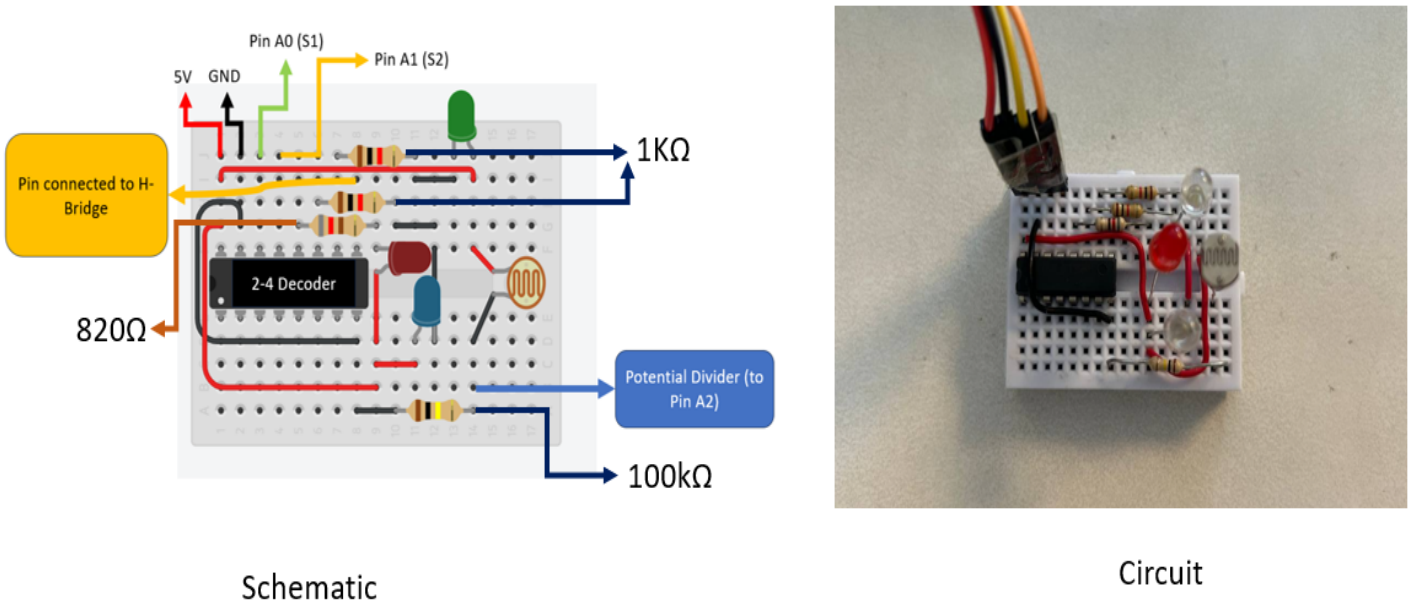


Figure 8: LDR colour sensor design and circuitry.

For the resistors chosen, the values were decided if the amount of current entering the 2-4 decoders were less than 8mA and can provide the right brightness for the LDR to detect the changes.

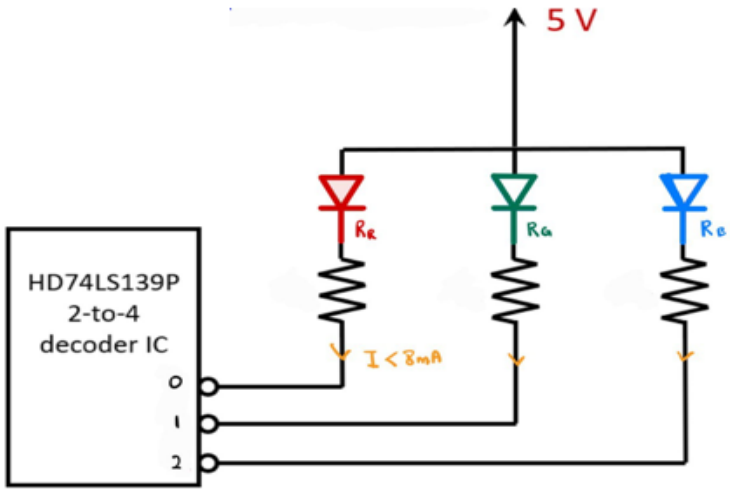


Figure 9: Circuit diagram for LEDs and 2-to-4 decoder

As the current values have to be less than 8mA, we tested with the resistor values of 560Ω for each LED. We calculated the current values using Ohm's law, V = IR. Since there is a constant 5 Volt supply, the current (I) flowing through each LED would be around 8.2mA which indicates that we need a higher resistance value. For our second choice, we picked all the resistors to be 820Ω which gave us the current flowing in each LED around 6.1mA which was in the required operating region. We also realized that the Red LED was in the suitable range of brightness, however, the green and blue were too bright for the LDR. Hence we decided to increase their resistance value by 1kΩ, which was then able to produce an accurate reading for the colour.

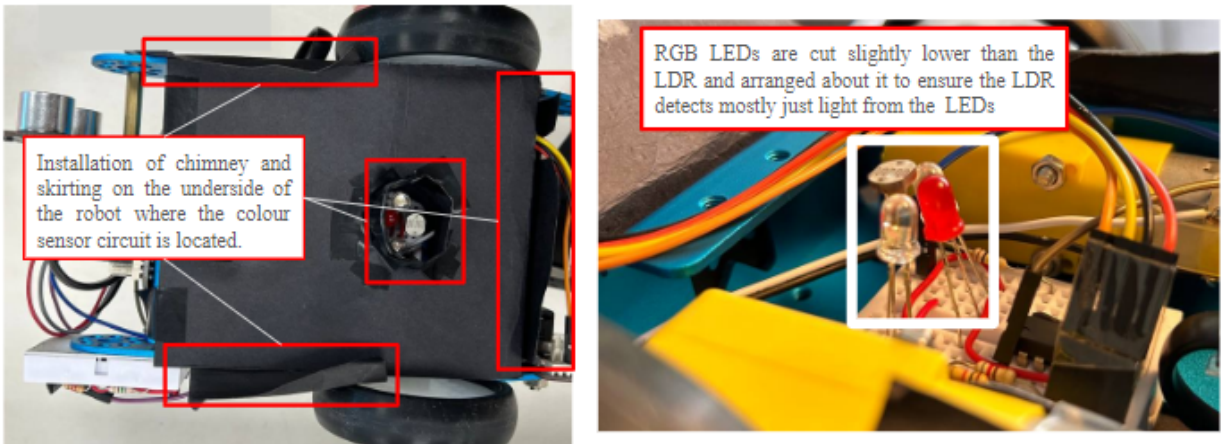


Figure 10: Skirting and chimney on the underside of the robot (left) and heights of the LEDs and LDR (right)

To prevent inaccurate and unreliable readings produced by the LDR, skirting and chimney (refer to Figure 10 above) were added to the robot. Additionally, the legs of the LEDs and LDR are deliberately cut at a height where the LDR is as close to the ground surface as possible. All of the LEDs, in return, are cut slightly lower than the LDR (refer to Figure 10). Since the LDR is protected within the skirting and chimney, as well as brought closest to the ground possible, it prevents ambient light from being detected and ensures that any light detected would come from the LEDs reflected off the coloured papers.

3.2.3. Code

For recognizing the different colours, our code looks at the shortest distance the unknown colour is from the 6 colours of the waypoints, red, orange, green, blue, purple, and white, by comparing the RGB values of the unknown colour with these 6 colours.

Control of LEDs

For the software portion for controlling the LEDs, we represent 4 states of the decoder as a number from 0 – 3. States 0, 1, 2, and 3 represent shining red LED, shining green LED, shining blue LED, and activating the IR emitter respectively. The two signal pins A0 and A1 from the mCore are used to control the inputs to the 2-to-4 decoder. This is implemented in a function decoder_state() as shown in Figure 11 below.

```
#define INPUT_A A0
#define INPUT_B A1

void shineIR() {   // Code for turning on the IR emitter only
  analogWrite(INPUT_A, 255);
  analogWrite(INPUT_B, 255);
}
void shineRed() {   // Code for turning on the red LED only
  analogWrite(INPUT_A, 0);
  analogWrite(INPUT_B, 0);
}
void shineGreen() {   // Code for turning on the green LED only
  analogWrite(INPUT_A, 255);
  analogWrite(INPUT_B, 0);
}
void shineBlue() {  // Code for turning on the blue LED only
  analogWrite(INPUT_A, 0);
  analogWrite(INPUT_B, 255);
}

void decoder_state(int state) {   //controls the 2-to-4 decoder
  if (state == 0) {  //turn on RED LED
    shineRed();
  }
  else if (state == 1) {  //turn on GREEN LED
    shineGreen();
  }
  else if (state == 2) {  //turn on BLUE LED
    shineBlue();
  }
  else if (state == 3) {  //turn off all LEDs and turn on IR emitter
    shineIR();
  }
}
```

Figure 11: Code for controlling the 2-to-4 decoder

Therefore the state number passed into the decoder_state() function will determine the inputs from the signal pins which allows us to control which LED will light up and when to activate the IR emitter.

Calibration

To obtain RGB values for the colours, we would have to first obtain the RGB values for black and white, as well as their difference, and store them in arrays as shown in Figure 12. This was done using the setBalance() function where each of the 3 LEDs was shone on the black and white samples for a short time to obtain their RGB values respectively.

```
float colourArray[] = {0,0,0};
float whiteArray[] = {979,991,978};
float blackArray[] = {947,936,892};
float greyDiff[] = {32,55,86};

void setBalance()   //for calibration of white and black
{
  //set white balance
  Serial.println("Put White Sample For Calibration ...");
  delay(5000);           //delay for five seconds for getting sample ready
  //scan the white sample.
  //go through one colour at a time, set the maximum reading for each colour -- red, green and blue to the white array
  for(int i = 0; i <= 2; i++) {
    decoder_state(i);
    delay(RGBWait);
    whiteArray[i] = getAvgReading(5);         //scan 5 times and return the average,
    Serial.println(whiteArray[i]);
    decoder_state(3);
    delay(RGBWait);
  }
  //done scanning white, time for the black sample.
  //set black balance
  Serial.println("Put Black Sample For Calibration ...");
  delay(5000);     //delay for five seconds for getting sample ready
  //go through one colour at a time, set the minimum reading for red, green and blue to the black array
  for(int i = 0; i <= 2; i++){
    decoder_state(i);
    delay(RGBWait);
    blackArray[i] = getAvgReading(5);
    Serial.println(blackArray[i]);
    decoder_state(3);
    delay(RGBWait);
  //the differnce between the maximum and the minimum gives the range
    greyDiff[i] = whiteArray[i] - blackArray[i];
  }

  //delay another 5 seconds for getting ready colour objects
  Serial.println("Colour Sensor Is Ready.");
  delay(5000);
}
```

Figure 12: Code for calibration

The values in whiteArray[], blackArray[] and greyDiff[] shown in Figure 12 are the values we obtained after calibration. The decoder_state() function is used here to control the sequence of LEDs to light up to obtain the respective R, G, and B values for the black and white samples. We use the getAvgReading() function here to obtain an average reading from 5 readings for each of the LEDs shone.

```
#define LDR_RECEIVER A2

int getAvgReading(int times)  //get average readings
{
  //find the average reading for the requested number of times of scanning LDR
  int reading;
  int total = 0;
  //take the reading as many times as requested and add them up
  for(int i = 0;i < times;i++){
    reading = analogRead(LDR_RECEIVER);
    total = reading + total;
    delay(LDRWait);
  }
  //calculate the average and return it
  return total/times;
}
```

Figure 13: Code for obtaining average reading

For the reading of values from the LDR, the analog sensing pin A2 from the mCore is used to measure the voltage from the LDR circuit.

After this we would need to calibrate the RGB values for red, orange, green, blue, purple and white respectively. We can use the code from Figure 15 to obtain these RGB values. These values are then stored in a 2D array which will be used later for sensing the colours (More details under the next section on Colour Sensing). Figure 14 shows the calibrated values we used. The 2D array is sorted in the order of red, orange, green, blue, purple and white.

```
float colours[6][3] = {{220, 122, 139}, {223, 176, 148}, {79, 180, 127}, {127, 215, 237}, {151, 171, 198}, {231, 255, 255}};
```

Figure 14: Calibrated RGB values for the 6 colours

Colour Sensing

The first step is similar to the calibration. We shine the three LEDs for a short period and obtain the readings from the LDR. The readings are then stored in an array colourArray[] whereby they are adjusted and updated to be values between 0 - 255.

```
for (int c = 0; c <= 2; c++) {
  decoder_state(c); //turn ON the LED, red, green or blue, one colour at a time.
  delay(RGBWait);
  //get the average of 5 consecutive readings for the current colour and return an average
  colourArray[c] = getAvgReading(5);
  /*the average reading returned minus the lowest value divided by the maximum possible range,
  multiplied by 255 will give a value between 0-255, representing the value for the current reflectivity (i.e. the colour LDR is exposed to)*/
  colourArray[c] = ((colourArray[c] - blackArray[c])/(greyDiff[c]))*255;
  decoder_state(3);  //turn off the current LED colour
  delay(RGBWait);
}
```

Figure 15: Code for obtaining RGB values for unknown colour

After obtaining the RGB values for the unknown colour, we compare these RGB values with the RGB values of the 6 colours we calibrated and stored in the colours[6][3] array. We can make a comparison by comparing the Euclidean distance of the unknown colour with each of the 6 colours, red, orange, green, blue, purple, and white. The colour that has the shortest distance to the unknown colour, will be considered as the identity of the unknown colour. The Euclidean distance, d, between two colours, (r_1, g_1, b_1) and (r_2, g_2, b_2) can be calculated by the given equation:

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

Figure 16 below shows an example of a 3D representation of the euclidean distance between two colours.

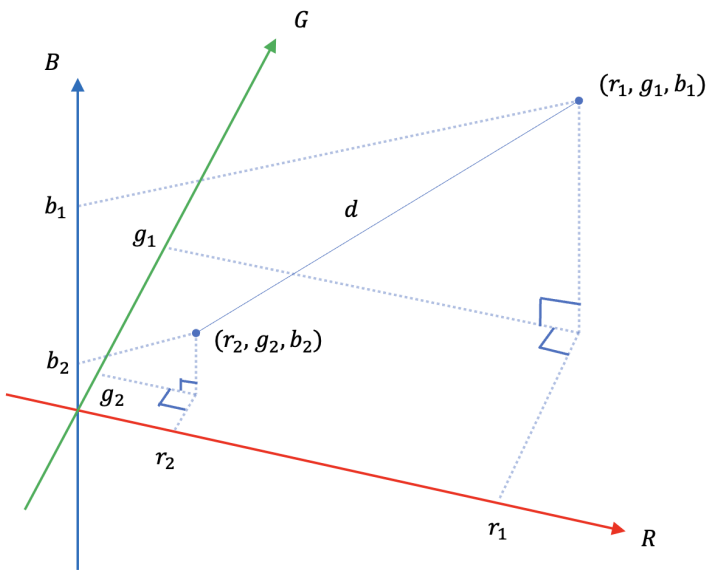


Figure 16: 3D example representation of the Euclidean distance between 2 points

This method can be represented in code by iterating through the RGB values for all 6 colours in the colours[6][3] array and calculating the Euclidean distance to each colour, while keeping track of the colour with the minimum distance using the variables “min” and “minDist”. The resulting code is shown in Figure 17.

```
int min = 0;
long minDist = 195075; //255^2 + 255^2 + 255^2
for (int i = 0; i < 6; i++) {
  long distance = square(colours[i][0]-colourArray[0]) + square(colours[i][1]-colourArray[1]) + square(colours[i][2]-colourArray[2]);
  //Serial.println(distance);
  if (distance < minDist) {
    minDist = distance;
    min = i;
  }
}
```

Figure 17: Code for finding the shortest Euclidean distance

After the code in Figure 17 is executed, the identity of the unknown colour is stored in the variable "min" in the form of an integer. The integers 0, 1, 2, 3, 4, and 5 represent red, orange, green, blue, purple, and white respectively. The next portion is just to simply execute the corresponding turns depending on the colour that was identified, which can be written in code as shown in Figure 18.

```
if (min == 0) {
  //Serial.println("red");
  stopMotor();
  turnLeft(TURNING_TIME_MS-5);
  stopMotor();
  delay(50);
}
else if (min == 1) {
  //Serial.println("orange");
  stopMotor();
  uTurn();
  stopMotor();
  delay(200);
}
else if (min == 2) {
  //Serial.println("green");
  turnRight(TURNING_TIME_MS-5);
  stopMotor();
  delay(50);
}
else if (min == 3) {
  //Serial.println("blue");
  doubleRightTurn();
  delay(50);
}
else if (min == 4) {
  //Serial.println("purple");
  doubleLeftTurn();
  delay(50);
}
else {
  //Serial.println("white");
  stopMotor();
  celebrate();
}
```

Figure 18: Code for executing movements after interpreting the colour

For the case of white, we simply stop the motors and play the celebratory tune. The celebrate() function is shown below in Figure 19.

```
void celebrate() {  // Code for playing celebratory tune
  buzzer.tone(392, 200);
  buzzer.tone(523, 200);
  buzzer.tone(659, 200);
  buzzer.tone(784, 200);
  buzzer.tone(659, 150);
  buzzer.tone(784, 400);
  buzzer.noTone();
}
```

Figure 19: Code for celebratory tune

3.2.4. Difficulties Faced

Issues with Chimney

Building the very first chimney, we went with a square configuration shown in Figure 20 below.

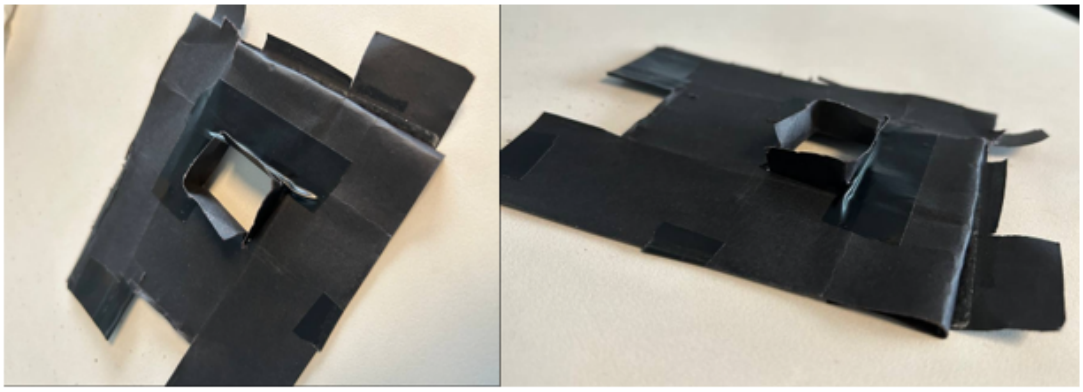


Figure 20: Square configuration of the undercarriage chimney

After using this configuration and testing, it was able to provide shielding for the colour sensor. However after subsequent test runs, we realised the structural integrity of the chimney's undercarriage was failing, which led to inconsistencies in the LDR colour detection circuit as there was less shielding.

To improve the overall undercarriage chimney, we replaced the paper with thicker black paper. Furthermore, we decided to reduce the size of the chimney to reduce the amount of ambient light from reaching the sensor. Refer to Figure 21 below for the new chimney.

Figure 21: New smaller chimney made of harder black paper and black taped to ensure stronger structural integrity

After running several test runs with the new chimney, we noticed that it was too compact, resulting in some of the LEDs being under the LDR or being blocked by the black paper. Another problem that we unknowingly introduced was the tape. By taping the inner walls of the chimney, the reflective nature of the tape caused the colour sensor to have slightly inconsistent readings.

To fix these issues, we decided to increase the chimney size for more space for the LEDs to be arranged around the LDR (but small enough to prevent excessive ambient light from being picked up). As we still desired the chimney to be sturdy, we still taped the inner surface, but we included another piece of black paper in the chimney to block the reflective tape. Refer to Figure 22 below for the final chimney.

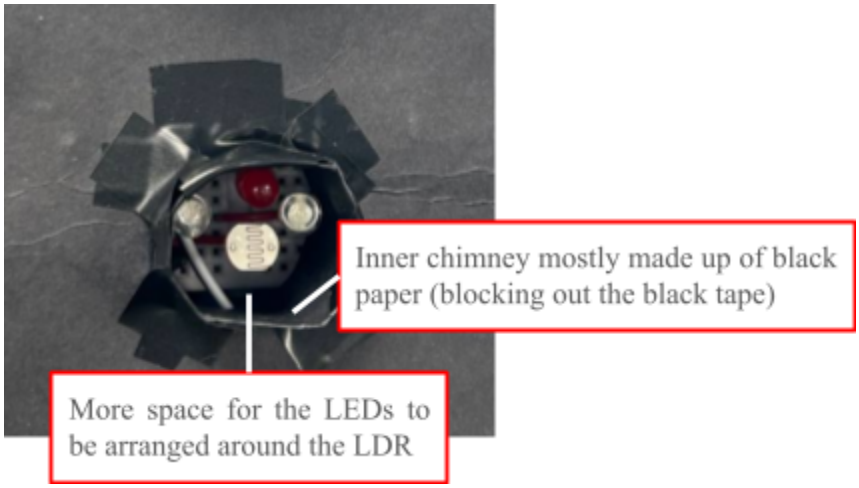


Figure 22: Final chimney

With this new chimney, it was able to provide appropriate shielding for the components and have enough support to prevent it from falling apart.

Inaccuracy of previous colour detection method

Initially, our method for determining colour was much more simpler, but it led to more inaccurate results. We compared the RGB values with one another. For example, if the G value for the unknown colour is larger than the R and B values, then the unknown colour is green. For colours like red and orange, we compare the G value against a fixed number, such as 140. If the G value is greater than 140, the unknown colour is interpreted as orange, and if the value is less than 140, then the unknown colour is determined to be red. The value 140 was obtained after testing on red and orange samples a few times and an observation was made that orange generally had green values that were more than 140. A similar method was used to differentiate blue from purple. Figure 23 shows the previous code which uses this method for determining colours.

```
if (colourArray[0] > colourArray[1] && colourArray[0] > colourArray[2]) {  //red is the prominant colour detected
  if (colourArray[1] > 140) {
    //Serial.println("orange");
    stopMotor();
    uTurn();
    stopMotor();
    delay(200);
  }
  else {
    //Serial.println("red");
    stopMotor();
    turnLeft();
    stopMotor();
    delay(50);
  }
}
else if (colourArray[1] > colourArray[0] && colourArray[1] > colourArray[2]) {  //green is the prominant colour detected
  turnRight();
  stopMotor();
  delay(50);
  //Serial.println("green");
}
else if (colourArray[2] > colourArray[1]) {  //blue is the prominant colour detected
  if (colourArray[0] < 150) {
    //Serial.println("blue");
    doubleRightTurn();
    delay(50);
  }
  else {
    //Serial.println("purple");
    doubleLeftTurn();
    delay(50);
  }
}
else { //white detected
  //Serial.println("white");
  stopMotor();
  celebrate();
}
```

Figure 23: Previous code that was used to determine colour

It was noticed that using this method led to inaccurate results as the readings are not always consistent due to factors such as lighting conditions. For example, the threshold G value of 140 to differentiate orange from red may keep changing. We noticed that sometimes both red and orange will give a G value of more than 140 and sometimes both red and orange to give a G value of less than 140. This would lead to inaccurate recognition of colours.

3.2.5. Solutions

To overcome the issue of the inaccuracy of our previous way of sensing colours in 3.2.4, we decided to change our method to find the Euclidean distance between the colours instead, which led us to our current code that we eventually used for the final product shown and explained in Figure 17 and Figure 18 from 3.2.3. Finding the shortest distance from the unknown colour to a calibrated set of values for all 6 colours was much more accurate than the method in Figure 23 in 3.2.4.

3.3. Ultrasonic and Infra-red Sensor

3.3.1. Overview

The ultrasonic and IR sensors are used as the two proximity sensors for the mBot. The ultrasonic sensor is mounted on the left of the mBot and the IR sensor is mounted on the right. When the mBot deviates from its straight line path and tilts towards the left, the ultrasonic sensor will be responsible for detecting that the distance between the left wall and left side of the mBot is too little, and nudges the mBot to the right back to its original path. Similarly, the IR sensor is responsible for doing the same on the right side of the mBot.

3.3.2. Circuit

The ultrasonic sensor only requires digital pins on the mCore, thus we simply connect it to the mCore using the RJ25 cable. The design for the IR circuit is shown in Figure 24 below.

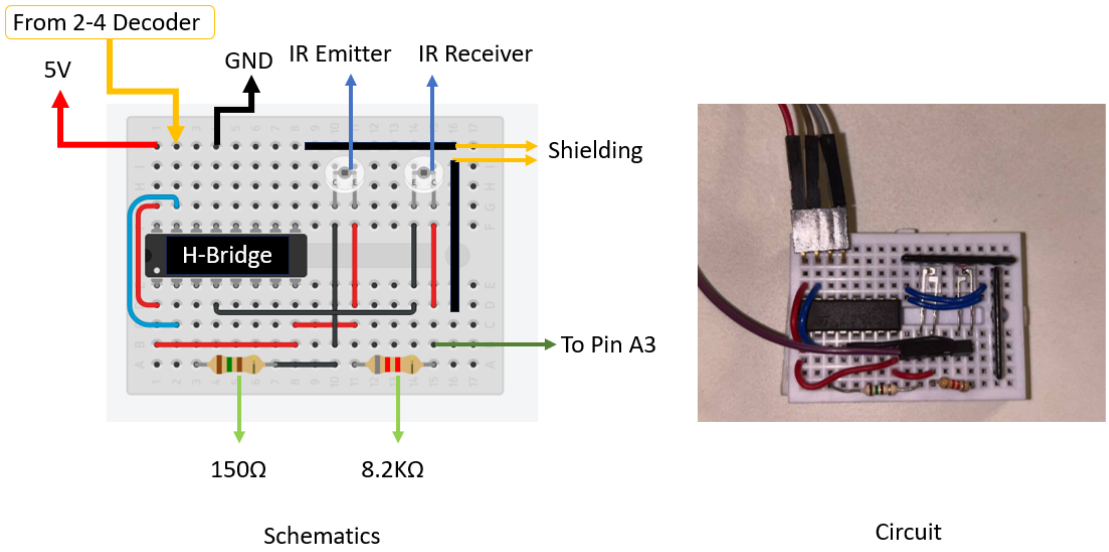


Figure 24: IR sensor schematic and circuit

Shieldings are placed around the emitter and receiver to block off excess ambient light which provides an accurate reading from the receiver and protects the IR circuit should the side of the mBot crash onto the wall and damage it.

Moreover, the emitter and receiver are placed a column apart (refer to the IR circuit in Figure 24). This ensures that the receiver can detect sufficient infrared rays which are

reflected off the wall from the emitter, and not excessive infrared rays directly from the emitter itself. A small piece of cardboard was placed between the emitter and receiver initially, to get a better reading from the receiver. However, we found that the cardboard blocked off a majority of the infrared rays for the receiver to detect and hence we concluded that its absence provided a much more accurate reading.

3.3.3. Code

The overview of the algorithm for the proximity sensors is shown in Figure 25 below.

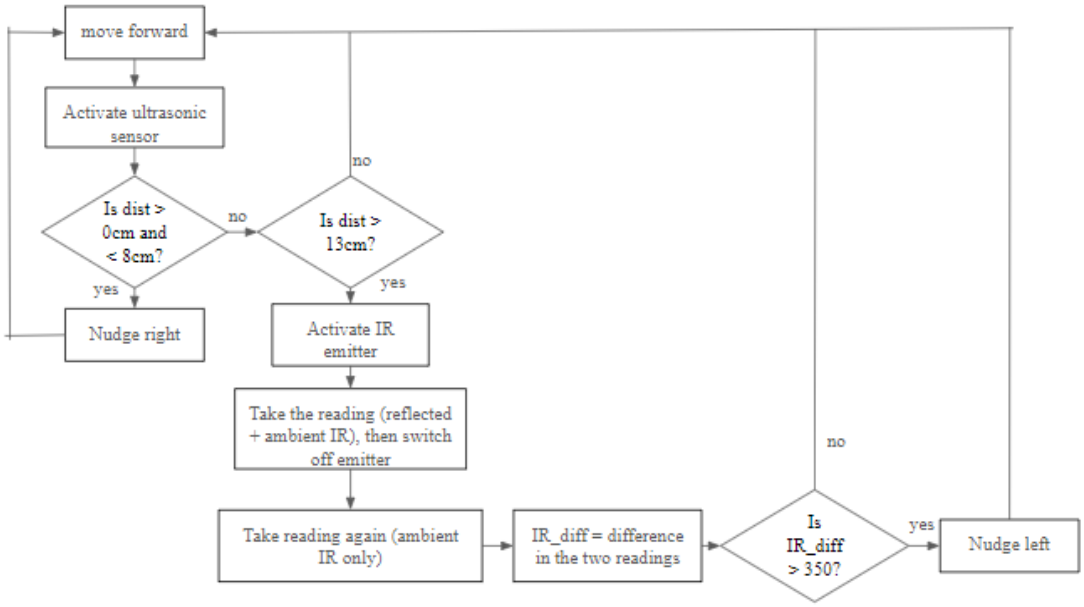


Figure 25: Flowchart for the algorithm for proximity sensing

Ultrasonic Sensor

While the mBot is moving forward in a straight line, the ultrasonic sensor will repeatedly emit and receive pulses to determine the distance the mBot is to the left wall. The reading from the ultrasonic sensor is then converted to centimetres and stored in a variable “dist”. If “dist” is more than 0 cm and less than 8 cm, the mBot will nudge back to the right to realign it to its straight line path. The reason why we checked if dist is more than 0 cm is because when there is no wall on the left of the mBot, “dist” would be 0 cm. Thus this would avoid the case where the mBot nudges to the right when there is no wall.

IR Sensor

If dist is more than 13 cm, this means that the mBot is deviating to the right as the distance of the mBot to the left wall is large. In this case, the IR sensors will be activated to detect for the right wall and nudge the mBot back to the left. However, due to ambient IR present, the voltage of the IR receiver will be affected as seen in Figure 26.

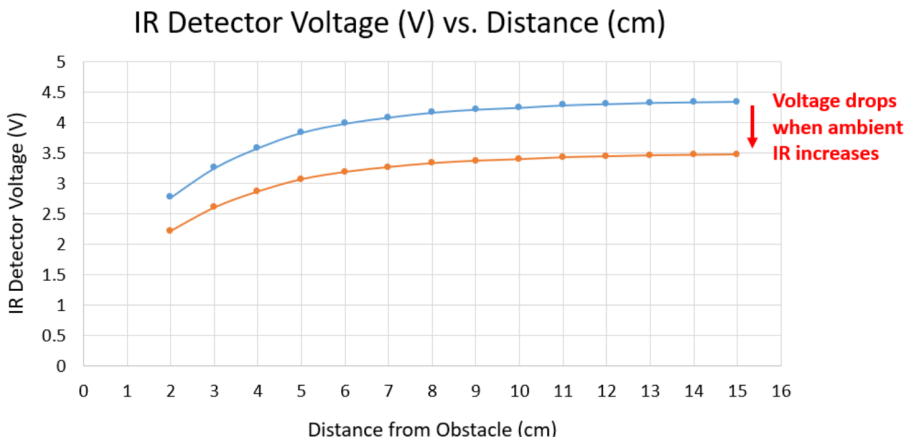


Figure 26: Effect of ambient IR on the voltage of the IR receiver

When the IR sensor is activated, we use the 2-to-4 decoder to turn on the IR emitter. The voltage of the IR receiver is then read and stored in a variable "val". We then turn off the IR emitter after the reading is taken. The voltage of the IR receiver is taken again which will represent the voltage that is solely due to ambient IR. We then take the difference between the two readings and store the result in another variable "IR_diff". This will take into account the ambient IR which will make our readings more accurate. After testing we found that for an appropriate distance between the mBot and the right wall, "IR_diff" is measured to be 350. Thus we can code it such that when "IR_diff" is less than 350, meaning the mBot is getting closer to the right wall, the mBot nudges back to the left. The combined code for both the ultrasonic sensor and the IR sensor is shown below in Figure 27.

```
#define IR_RECEIVER A3

pinMode(ULTRASONIC, OUTPUT);
digitalWrite(ULTRASONIC, LOW);
delayMicroseconds(2);
digitalWrite(ULTRASONIC, HIGH);
delayMicroseconds(10);
digitalWrite(ULTRASONIC, LOW);

pinMode(ULTRASONIC, INPUT);
long duration =  pulseIn(ULTRASONIC, HIGH, TIMEOUT); // Read ultrasonic sensing distance
float dist = (duration / 2.0 / 1000000) * SPEED_OF_SOUND * 100; //Converts the distance to centimetres
//Serial.println(dist);
if (dist > 0.0 && dist < 8.0) { //Checks whether mBot is too close to left wall
  nudgeRight();
}
else if (dist > 13.0) {
  decoder_state(3);   //Turn on IR emitter
  delay(5);
  int val = analogRead(IR_RECEIVER);  //Read reflected IR and ambient IR
  decoder_state(0);   //Turn off IR emitter
  delay(5);
  int IR_diff = analogRead(IR_RECEIVER) - val; //Read ambient IR and calculate the difference between the two readings
  //Serial.println(IR_diff);
  if (IR_diff > 350) {  //Checks whether mBot is too close to right wall
    nudgeLeft();
  }
}
```

Figure 27: Code for ultrasonic and IR sensors

The code for nudging to the left and right was simply to stop one of the motors, depending on which way it is nudging towards, while letting the other motor run. This will cause the mBot to steer towards the direction that it wants to nudge towards. This is depicted in Figure 28 below.

```
void nudgeLeft() {  // Code for nudging slightly to the left for some short interval
  leftMotor.run(0);
  rightMotor.run(230);
}
void nudgeRight() {   // Code for nudging slightly to the right for some short interval
  leftMotor.run(-230);
  rightMotor.run(0);
}
```

Figure 28: Code for nudging left and right

3.3.4. Improvements

Initially, we noticed that although the mBot is able to nudge back when it gets too close to the walls of the maze, it still moves in a zigzag manner. We improved the movements of the mBot for it to move straight by changing the threshold values where it nudges left and right. This was the reason we decided to use the threshold values of 0 cm and 8 cm for nudging right, and 13 cm for nudging left (Refer to 3.3.3) as compared to previous values that we tried during testing (threshold values that were smaller than 8 cm and larger than 13 cm). This would mean that the allowable range of distance where the mBot does not nudge would just be 5 cm (from 8 cm to 13 cm), making it as tight as possible. Thus when the distance of the mBot with the left wall of the maze is outside of this allowable distance, the mBot would immediately nudge back to correct its path. This way

the mBot is more efficient in detecting the walls of the maze, allowing it to travel as straight as possible. Figure 29 shows a visual representation of how these conditions make the mBot travel as straight as possible.

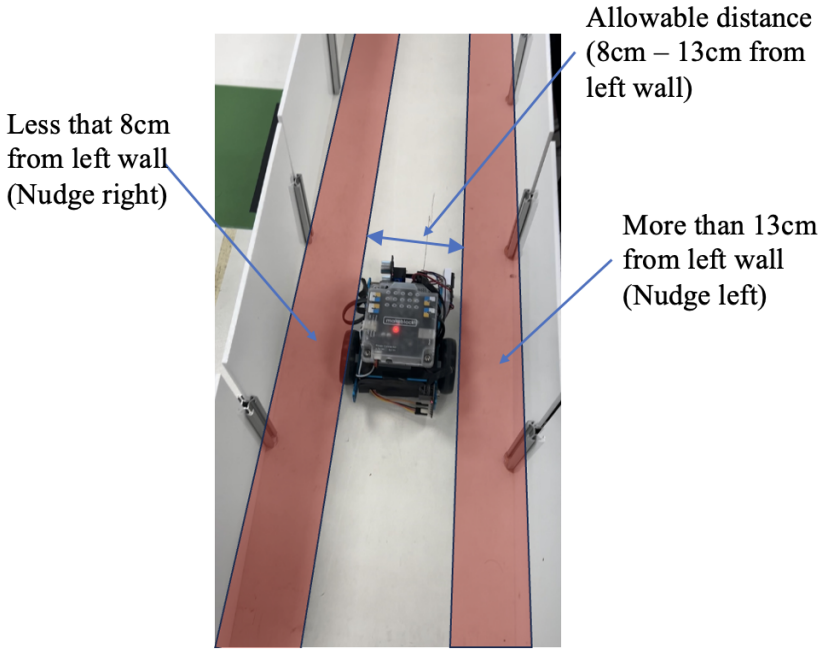


Figure 29: Visual of the tight conditions that make the mBot travel straight

4. Work Division Among Members

The table below shows the distribution of workload between the team members.

Member	**Contribution**
Beal Ng Zhi Jie	Report, Robot Assembly
Benjamin Thomas Komatt	LDR Sensor Design, Robot Assembly, Sub Programmer, Report
Brandon Kang	Main Programmer (Implementation of main algorithms), Troubleshooting of Hardware, Report
Bu Yin Shuang	Robot Assembly, IR Sensor Design, Skirting and Chimney Design, Report

Table 3: Distribution of workload

5. References

1. NUS CG1111A Notes and Weekly handouts (2023)
   [Accessed on 18 November 2023]