CG1111A Engineering Principles and Practice I

The A-Maze-ing Race Project Report 2021

Studio Group: 2
Section Number: 6
Team Number: 3

Team Members:

Name	Student ID
Bui Duc Thanh	A0243318J
Tan Yu Xiang, Gareth	A0233682A
Tan Yi Zhe	A0239076X
Wang Tingjia	A0237903A

Table of Contents

1 Introduction

We were tasked to create a robot that is capable of completing the A-maze-ing race. It must be able to move in a straight line, stop when required, detect colours and do turns based on the colours, and play a tune at the end of the maze. This report details different subsystems of the mBot, the difficulties we faced in this project and how we overcame them.

2 Overall Algorithm

Figure 1 below is a flowchart of our overall algorithm our mBot uses to solve the maze.

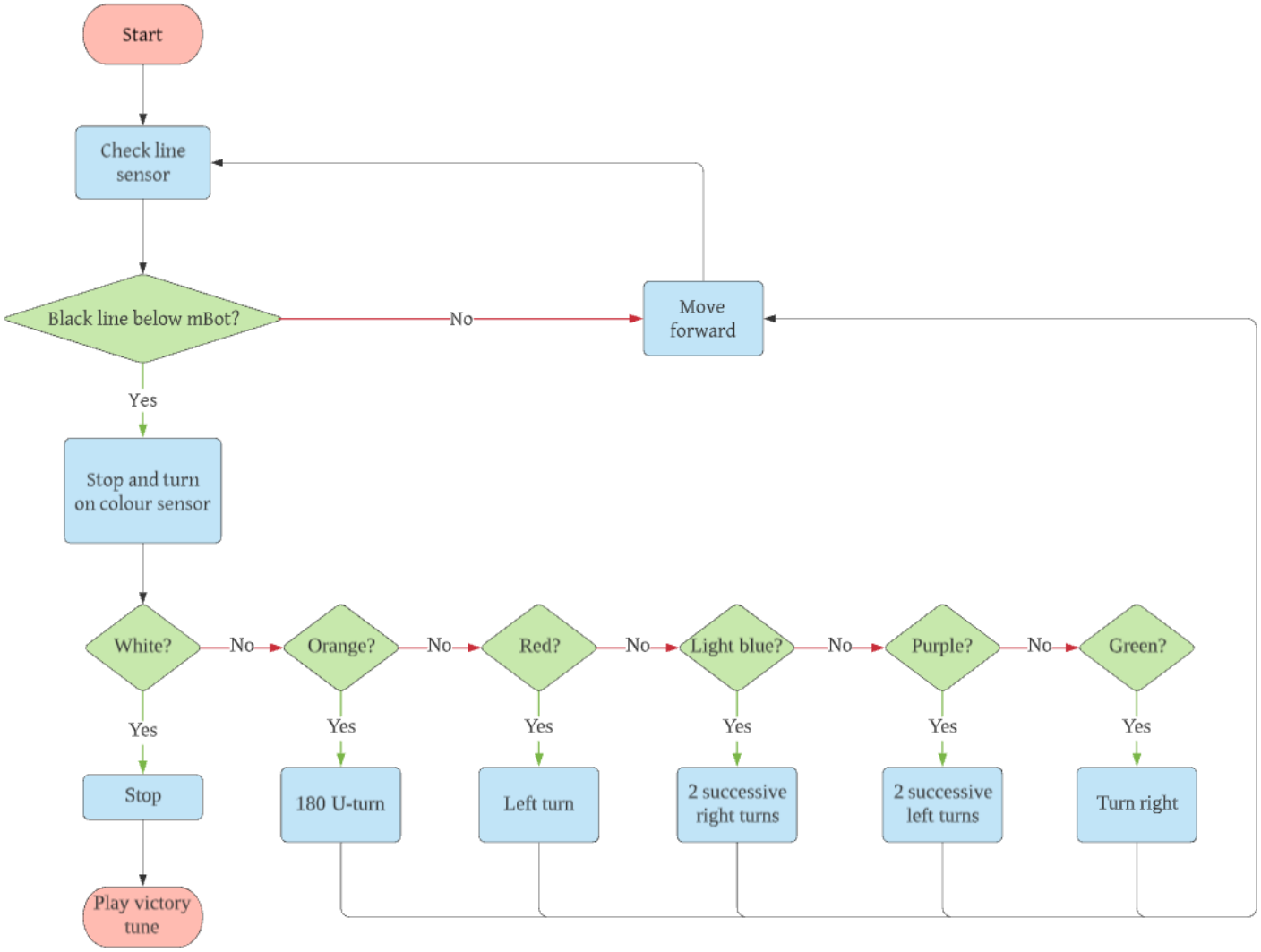


Figure 1 Overall algorithm

3 Subsystems Implemented

The main component allowing the implementation of the various subsystems is the HD74LS139P 2-to-4 Decoder IC Chip. This chip allowed the use of two signal pins on the mBot's mCore to control all the various subsystems in place. This also enabled us to control when these subsystems were to be turned on and off.

3.1 Movements

3.1.1 Overview

There were three main types of movements the mBot was required to perform: moving forward, stopping and turning. The components used to determine these movements are as follows: Makeblock Ultrasonic Sensor, Makeblock Line Sensor, InfraRed (IR) Emitter and Detector, and a Colour Sensor.

3.1.2 Moving Forward

In the beginning, both motors were initialised to operate at the same speed, but it was noted that the two motors of the mBot moved in opposite directions. Thus, one motor would be at a positive value while the other would be at a negative value.

An ultrasonic sensor and an IR sensor were used to ensure the mBot does bump into the walls of the maze - we will go more into detail about this later in the report.

3.1.3 Stopping

The Makeblock Line Sensor was used to determine when the mBot should come to a stop. The line sensor was used to detect a black strip, telling the mBot to stop moving. The implementation of this can be seen in Figure 2.

```
bool black_line() {
  int sensor_state = lineFinder.readSensors();
    switch(sensor_state) {
    case S1_IN_S2_IN: Serial.println("S1_IN_S2_IN"); break;
    case S1_IN_S2_OUT: Serial.println("S1_IN_S2_OUT"); break;
    case S1_OUT_S2_IN: Serial.println("S1_OUT_S2_IN"); break;
    case S1_OUT_S2_OUT: Serial.println("S1_OUT_S2_OUT"); break;
    default: break;
  }
  if (sensor_state == S1_IN_S2_OUT || sensor_state == S1_OUT_S2_IN || sensor_state == S1_IN_S2_IN){
    Serial.println("Black line");
    return true;
  }
  Serial.println("Not black line");
  return false;
}
```

Figure 2 Code snippet for detecting a black line

Once the mBot detects a black line, it will come to a stop. A colour sensor would then be activated, determining which direction the mBot has to turn. The colour sensing will be further explained in section 3.2.

3.1.4 Turning

There are 5 types of turns the mBot was required to do: right turn, left turn, 180 degree U-turn in the same grid, two successive left-turns in two grids, and two successive right-turns in two grids. The table below shows the different colours and the corresponding movements the mBot was required to do.

Colour	**Movements**
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 1: Colours and the corresponding movements of the mBot

In section 3.1.2, both motors on the mBot were initialised to the same value, and one motor is at a positive value while the other is at a negative value. This would affect how we implement our various turning functions. The figure below shows the implementation of all the turns.

```
void rightTurn() {
  rMot.run(-motorSpeed);
  lMot.run(-motorSpeed);
  delay(550);
}

void leftTurn() {
  lMot.run(motorSpeed);
  rMot.run(motorSpeed);
  delay(550);
}
void uTurn() {
  lMot.run(-motorSpeed);
  rMot.run(-motorSpeed);
  delay(1100);
}

void twogrid_rightTurn() {
  rightTurn();
  lMot.run(-180);
  rMot.run(158);
  delay(930);
  rightTurn();
}

void twogrid_leftTurn() {
  leftTurn();
  lMot.run(-180);
  rMot.run(158);
  delay(850);
  leftTurn();
}
```

Figure 3. Code snippet for the various turn operations

For the left and right turns, we set both the motors to be either negative or positive respectively, so they would move in opposite directions in a short duration to complete the turn. For the U-turn, both motors are set to a negative value but a slightly longer duration than a normal right turn so it can complete a 180 degree turn.

For the two successive turns in two grids, we call either the left or right turn function, followed by setting the motors to move forward for a short duration, and finally calling the left or right turn function once again.

3.1.5a Difficulties

Initially, a "motorSpeed" variable was used to control the speeds of both motors, where they were set to operate at the same speed, as mentioned earlier. However, one problem faced was that the mBot did not move in a straight line, as it was consistently moving slightly towards the left.

3.1.5b Solutions

It was concluded that the motor on the right-side of the mBot moves at a slightly higher RPM compared to the left-sided motor. Thus, the right motor was slowed down by assigning it a lower speed so that the mBot can move in a straight line.

After several rounds of testing, we decided to use the value -180 for the left motor, and 165 for the right motor. This also affected the values used for implementation of the 5 turns in section 3.1.4.

3.2 Colour Sensing

3.2.1 Overview

2 main components were used for the colour sensor: a Common-anode RGB LED lamp and a Light Dependent Resistor (LDR). Figures 4 and 5 below show our circuit design for the colour sensor.

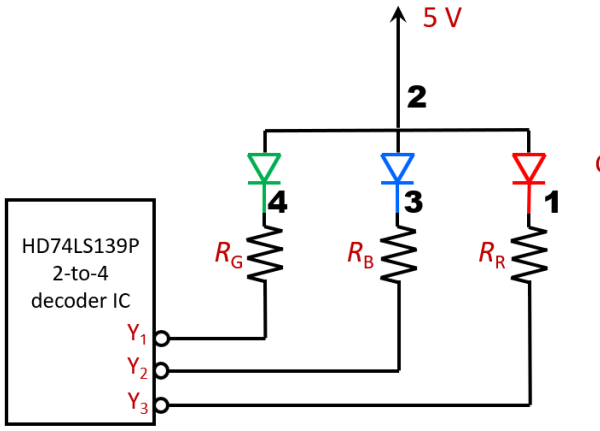


Figure 4. Controlling the RGB LED Lamp

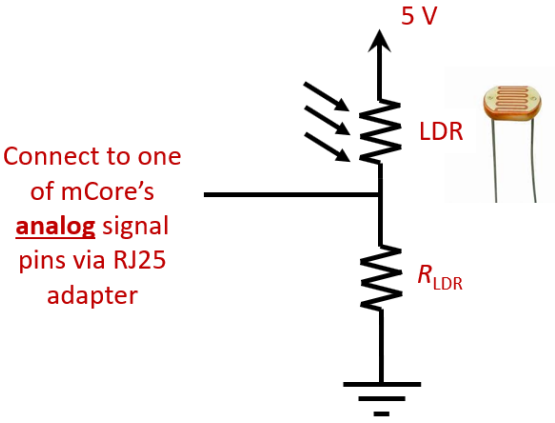


Figure 5. LDR circuit for detecting light intensity

3.2.2 Detection of RGB values

For the detection of RGB values, using the circuit as seen above, we first obtained the RGB values for the black and white colours, as seen in Figure 6, using the calibrate() function as seen in Figure 7. The difference was then stored in the "blackArray[]" and "whiteArray[]" respectively. This provided a basis for comparison for the different colors later on. The "greyArray[]", in Figure 6, is the difference in values between the white and black array.

At each colour challenge, the mBot will detect a black line and come to a stop, and will proceed to read the RGB values of the colour below it using the test_color() function in Figure 9. The readings are then stored inside the "colorArray[]".

```
//pre-calibrated before experiment
float colorArray[] = {0,0,0};
float whiteArray[] = {505,495,480};
float blackArray[] = {390,370,360};
float greyArray[] = {115,135,120};
```

Figure 6. Code snippet for our colour arrays to store the RGB value

```
void calibrate() {
    //turn on red first (Y3)
    Serial.print("R = ");
    analogWrite(PIN_A, 255);
    analogWrite(PIN_B, 255);
    delay(RGBWait);
    Serial.println(getAvgReading(15));
    //turn on green next (Y1)
    Serial.print("G = ");
    analogWrite(PIN_A, 255);
    analogWrite(PIN_B, 0);
    delay(RGBWait);
    Serial.println(getAvgReading(15));
    //turn on blue next (Y2)
    Serial.print("B = ");
    analogWrite(PIN_A, 0);
    analogWrite(PIN_B, 255);
    delay(RGBWait);
    Serial.println(getAvgReading(15));
}
```

Figure 7. Code snippet to calibrate our black and white backgrounds

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

Figure 8. Code snippet for reading of values from the LDR

```
void test_colors() {
    //turn on red first (Y3)
    Serial.print("R = ");
    analogWrite(PIN_A, 255);
    analogWrite(PIN_B, 255);
    delay(RGBWait);
    colorArray[0] = (getAvgReading(15) - blackArray[0])/(greyArray[0])*255;
    Serial.println(colorArray[0]);

    //turn on green next (Y1)
    Serial.print("G = ");
    analogWrite(PIN_A, 255);
    analogWrite(PIN_B, 0);
    delay(RGBWait);
    colorArray[1] = (getAvgReading(15) - blackArray[1])/(greyArray[1])*255;
    Serial.println(colorArray[1]);

    //turn on blue next (Y2)
    Serial.print("B = ");
    analogWrite(PIN_A, 0);
    analogWrite(PIN_B, 255);
    delay(RGBWait);
    colorArray[2] = (getAvgReading(15) - blackArray[2])/(greyArray[2])*255;
    Serial.println(colorArray[2]);

}
```

Figure 9. Code snippet for reading and storing the RGB value at each colour challenge

3.2.3 Identification of Colours

Following the reading of the colours, the actual colour is determined based on its RGB values using the code in Figure 10. The comparisons in the code below were determined after multiple rounds of testing and experimentation.

It was realised that comparing the relative red against relative blue against relative green values instead of hard-coding the values was better in accounting for the change in brightness of the environment after multiple rounds of experimentation.

colorArray[0] == Red value
colorArray[1] == Green value
colorArray[2] == Blue value

```
void move_off() {
  if (colorArray[0] > 200 && colorArray[1] > 200 && colorArray[2] > 200) {
    Serial.println("White Detected");
    rMot.stop();
    lMot.stop();
    victoryTune();
  }
  else if (colorArray[0] > colorArray[1] * 2 && colorArray[0] > colorArray[2] * 2) {
    if (colorArray[1] > colorArray[2]) {
      Serial.println("Orange detected");
      uTurn();
    }
    else {
      Serial.println("Red Detected");
      leftTurn();
    }
  }
  else if (colorArray[2] > colorArray[1]) {
      if (colorArray[2] > colorArray[0] + 100) {
        Serial.println("Blue Deteced");
        twogrid_rightTurn();
      }
      else {
        Serial.println("Purple Detected");
        twogrid_leftTurn();
      }
    }
  else if (colorArray[1] > colorArray[0] && colorArray[1] > colorArray[2]) {
    Serial.println("Green detected");
    rightTurn();
  }
}
```

Figure 10. Code snippet of how we determined the colours based on their RGB Values

3.2.4.a Difficulties

There was difficulty getting consistent values from the colour sensors due to ambient light and changing environments, due to testing of the colour sensor at home and in a lab. Under these different environments and weather conditions, the colour values were different from the values expected.

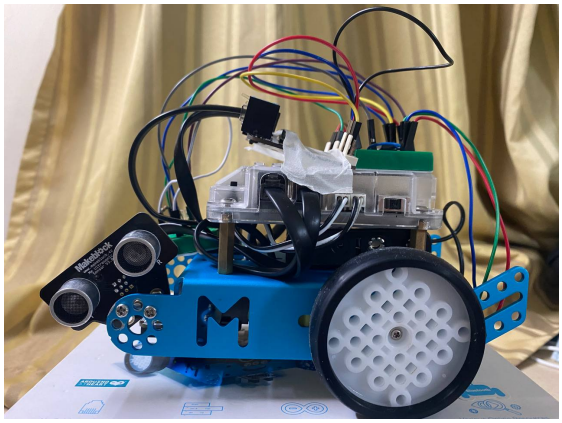
3.2.4.b Solutions

To address this, black paper was taped around the LDR to prevent ambient light from affecting the readings. Secondly, in each new environment, the readings were calibrated with reference to a black colour and a white colour. Lastly, instead of trying to compare with base values (i.e.if red > 300), we compared our RGB values relative to each other (i.e. if red > green). This proved to be more consistent in determining the colours as we did not need to update the different base values whenever the mBot was placed in a different environment. With these changes, the colour sensor proved to be more reliable and consistent.

3.3 Ultrasonic and Infrared sensors

3.3.1 Overview

To ensure the mBot travels in a straight line, we used an infrared sensor and ultrasonic sensor to adjust the speed of each wheel such that it turns away from the walls. The figures below show the sensors on the mBot.



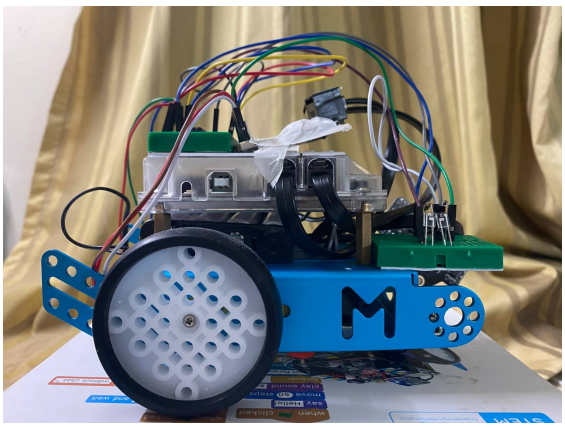


Figure 11. Left-side view of the mBot

Figure 12. Right-side view of the mBot

3.3.2 Ultrasonic Sensor

An ultrasonic sensor was installed on the mBot to measure the distance between the left walls of the maze and the mBot. The value obtained from the ultrasonic sensor was through Port 2 of the mCore.

3.3.3 Infrared (IR) Sensor

A self-built IR sensor was installed on the right side of the mBot to measure the distance between the right walls of the maze and the mBot. The value obtained from the IR sensor was through pin A1 on the mCore. The figure below shows the circuit design of the IR sensor.

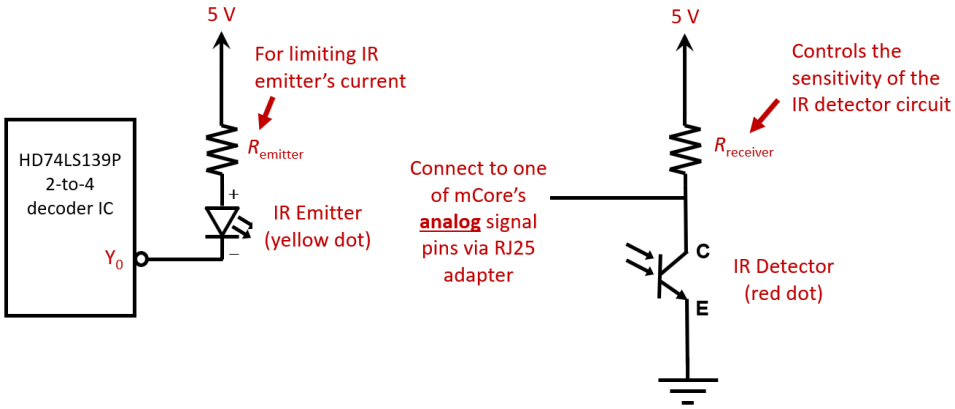


Figure 13. Circuit design of the IR sensor

Below is the actual IR circuit.

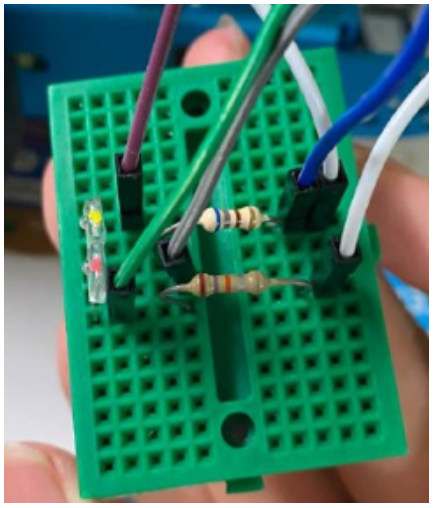


Figure 14: Actual IR circuit

The negative polarity of the IR emitter was connected to pin Y_0 of the decoder IC chip. This allowed the emitter to be turned on intermittently, so as to account for ambient IR in the maze. To turn the IR sensor on, we implemented the code “analogWrite(PIN_A, 0);” and “analogWrite(PIN_B, 0);”.

The resistance values of the IR circuits were chosen to be 1000Ω each as it resulted in the most consistent readings through multiple rounds of testing.

3.3.4 Keeping the mBot straight

To keep the mBot straight, the sensors had to be calibrated first. Multiple trial runs were conducted so each sensor’s output can be interpreted correctly to keep the mBot on track.

On the left side of the mBot with the ultrasonic sensor, the in-built function, ultrasonic.distanceCm(), was used to measure the distance between the left wall and the mBot. Slight adjustments were made based on measuring the distance between the bot and the left wall.

On the right side of the mBot with the IR sensor, the distance between the car and the wall was approximated based on the outputs given by the detector through pin A1 of the mCore. These outputs were converted into distance in centimetres to make approximating the distance between the right wall and the mBot easier..

If the output from the sensors indicated that the mBot was travelling too close to a left or right wall, the speed of the respective motor was adjusted. The mBot is considered to be at a central position if the values from the ultrasonic sensor and IR sensor are within the safe range that was determined in the previous section 3.3.4. The motor speeds are adjusted in accordance with the table below.

	Central Position	Too close to left wall	Too close to right wall
Left Motor Speed	-180	-175	-120
Right Motor Speed	165	120	175

Table 2. Speed of left and right motor in different scenarios

As shown in the table above, when the motor moves too close to the left wall, the right motor speed is decreased, allowing the left side of the mBot to catch up. This applies to the right side of the mBot. The changes in speeds of the motors were as minimal as possible so as to prevent the mBot from moving in a zig-zag manner. Figure 15 below shows the code implementation of the ultrasonic and infrared sensors.

```
void moveForward() {
  int ir_value = infrared();
  // the conditions when the mBot is centralized using both sensors
  // the condition ultrasonic.distanceCm() >= 16 is to avoid redundant checks when the wall is missing
  if ((ultrasonic.distanceCm() >= 10 && ultrasonic.distanceCm() <= 11.5 || ultrasonic.distanceCm() >= 16) && infrared() <= 945) {
    lMot.run(-180);
    rMot.run(165);
    Serial.println("Normal Distance");
    Serial.println(ultrasonic.distanceCm());
    Serial.println(infrared());
  }
  // the conditions when the mBot is too close to left side using only the ultrasonic sensor
  if (0 < ultrasonic.distanceCm() && ultrasonic.distanceCm() < 10){
    lMot.run(-175);
    rMot.run(120);
    Serial.println("Too Close to left side");
    Serial.println(ultrasonic.distanceCm());
  }
  // the condition when the mBot is too close to right side using only IR sensor
  if (ir_value > 945) {
    lMot.run(-120);
    rMot.run(175);
    Serial.println("Too Close to right side");
    Serial.println(ir_value);
  }
  analogWrite(PIN_A, 255);
  analogWrite(PIN_B, 255);
  Serial.println("IR disabled");
  delay(100);
}
```

Figure 15: Implementation of ultrasonic and IR sensors

3.3.5a Difficulty

It was noted that when the ultrasonic sensor was too close to the maze walls, it did not operate optimally, since the minimum operating distance of the ultrasonic sensor is 3cm.

Initially, the ultrasonic sensor was positioned on the far left of the mBot. This caused the distance between the ultrasonic sensor and the wall to be too close (around 5cm), preventing the ultrasonic sensor from properly detecting should the mBot move even the slightest distance towards it. Thus, the mBot was not able to adjust its course properly.

3.3.5b Solution

The ultrasonic sensor was thus shifted to a more central position within the mBot, as shown in Figure 16 below, instead of positioning it all the way to the left. To accomplish this, additional nuts were used to properly secure the ultrasonic sensor.

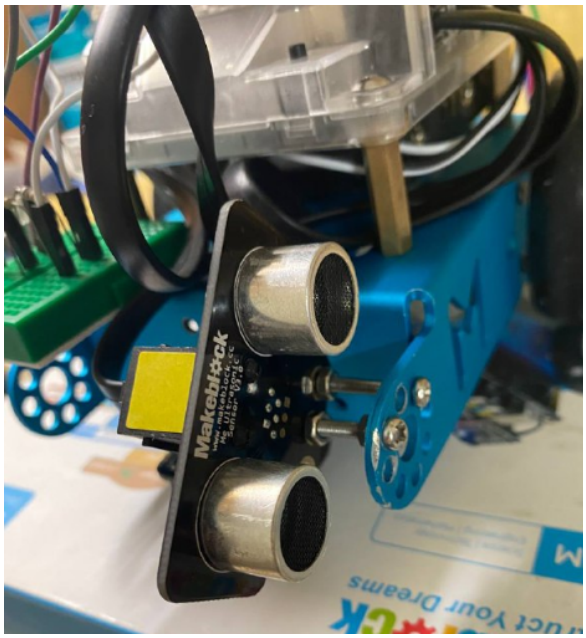


Figure 16: Ultrasonic Sensor, secured in a more central position

This allowed for more precise control of the mBot as the constraint of the minimum operating range of the ultrasonic sensor is negated. Previously, when the mBot was less than 3cm away from the wall, the wall would not be detected. Now, the distance of the sensor from the wall is 5cm, allowing the mBot to more accurately measure the distance between the wall and the mBot to adjust its course accordingly.

3.4 Victory Tune

Upon stopping at a black line and detecting the colour of the grid below the mBot to be white (i.e. the mBor has reached the end of the maze), we call the victoryTune() function (Figure below) to play the victory tune.

```
void victoryTune(){
  // the melody array contains the frequencies of notes for the song, which are initialized at the beginning of the code.
  int melody[] = {
    NOTE_E5, NOTE_DS5, NOTE_E5, NOTE_DS5,
    NOTE_E5, NOTE_B4, NOTE_D5, NOTE_C5,
    NOTE_A4, NOTE_C4, NOTE_E4, NOTE_A4,
    NOTE_B4, NOTE_E4, NOTE_GS4, NOTE_B4,
    NOTE_C5, 0, NOTE_E4, NOTE_E5, NOTE_DS5,

    NOTE_E5, NOTE_DS5, NOTE_E5, NOTE_B4, NOTE_D5, NOTE_C5,
    NOTE_A4, NOTE_C4, NOTE_E4, NOTE_A4,
    NOTE_B4, NOTE_E4, NOTE_C5, NOTE_B4,
    NOTE_A4, 0, NOTE_B4, NOTE_C5, NOTE_D5,

    NOTE_E5, NOTE_G4, NOTE_F5, NOTE_E5,
    NOTE_D5, NOTE_F4, NOTE_E5, NOTE_D5,

    NOTE_C5, NOTE_E4, NOTE_D5, NOTE_C5, NOTE_B4,
  };

  // 8 an eighteenth , 16 sixteenth.
  // negative numbers are used to represent dotted notes, the dot increases the duration of the basic note by half
  // the duration array has the same size with the melody array
  int duration[] = {
    16, 16, 16, 16,
    16, 16, 16, 16,
    -8, 16, 16, 16,
    -8, 16, 16, 16,
    8, 16, 16, 16, 16,
    16, 16, 16, 16, 16, 16,
    -8, 16, 16, 16,
    -8, 16, 16, 16,
    8, 16, 16, 16, 16,
    -8, 16, 16, 16,
    -8, 16, 16, 16,
    -8, 16, 16, 16, 8,
  };
// set the duration of a whole note to 1600 ms
  int wholenote = 1600;
  //find number of notes in the melody
  int notes = sizeof(melody) / sizeof(melody[0]);
  //define the variables for the duration of notes
  int noteDuration = 0;

  for (int noteNo = 0; noteNo < notes; noteNo += 1) {
    if (duration[noteNo] > 0) {
      // normal note
      noteDuration = wholenote / duration[noteNo];
    } else if (duration[noteNo] < 0) {
      // dotted notes
      noteDuration = wholenote / abs(duration[noteNo]);
      noteDuration *= 1.5; // increases the duration in half for dotted notes
    }

    // play 0.95 of the noteDuration, and 0.05 of the noteDuration is a pause
    buzzer.tone(8, melody[noteNo], noteDuration * 0.95);
    delay(noteDuration);
    // stop the tone playing
    buzzer.noTone(8);
  }
    delay(10000);
}
```

Figure 17. Code snippet of victory tune

4 Work Division

Due to COVID-19 restrictions, only 3 group members were allowed to work in the lab. Hence each member has contributed to multiple parts of the project.

Contribution	**Member**
Program code	Tan Yi Zhe, Wang Tingjia, Bui Duc Thanh, Tan Yu Xiang, Gareth
Movements (IR and ultrasonic sensor Implementation and calibration, Black line finder, turns)	Tan Yi Zhe, Bui Duc Thanh, Tan Yu Xiang, Gareth
Colour Sensor	Wang Tingjia
Victory Tune	Bui Duc Thanh

Table 3. Distribution of workload

5 References

[1] Available: https://github.com/Makeblock-official/Makeblock-Libraries
[Access: 18-Oct-2021]