National University of Singapore

CG1111A: Engineering Principles I

Academic Year 2023/2024 Semester 1

The A-maze-ing Race

Project Report

Studio B03 | Section 3 | Team 4

Name	Matriculation Number
Neeraj Kumbar	A0271737X
Kin Neo Jie Ming	A0273036J
Mark Neo Qi Hao	A0281484X

Table of Contents

1. Introduction

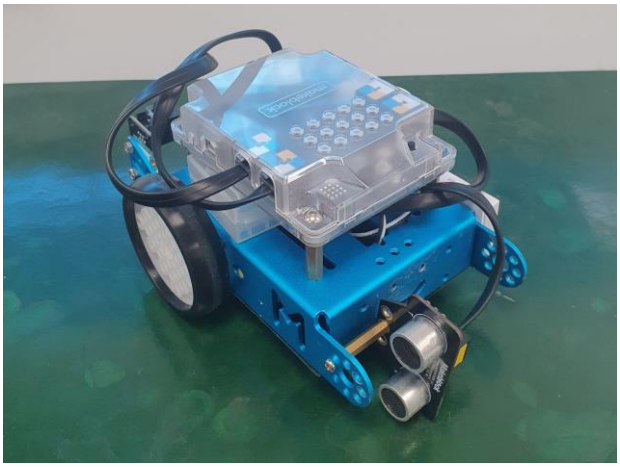


Figure 1: Our beloved mBot

In this project, our objective was to program and wire a robot with the capability to autonomously navigate through a randomly generated maze. The robot's functionality must be achieved through the integration of ultrasonic sensors and an infrared sensor, ensuring the maintenance of a straight course. A line sensor will be used to stop the robot upon reaching a black line, and an LED and LDR were utilized to differentiate the colours of the paper underneath, triggering distinct turns based on the colour detected. Finally, a victory tune needs to be programmed to play at the maze's conclusion. This report provides a comprehensive overview of the mBot's components, outlines the challenges encountered during the project, and discusses the strategies employed to overcome them.

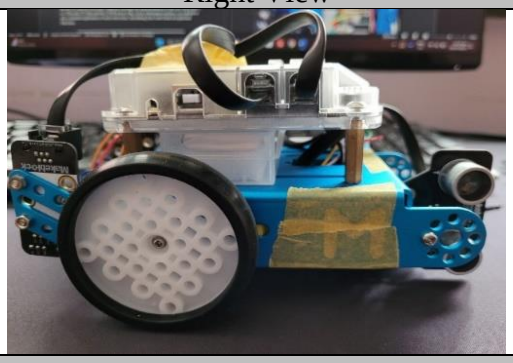
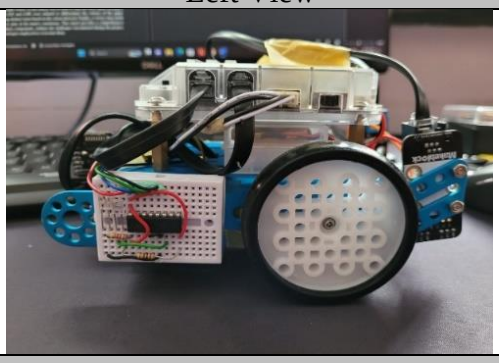
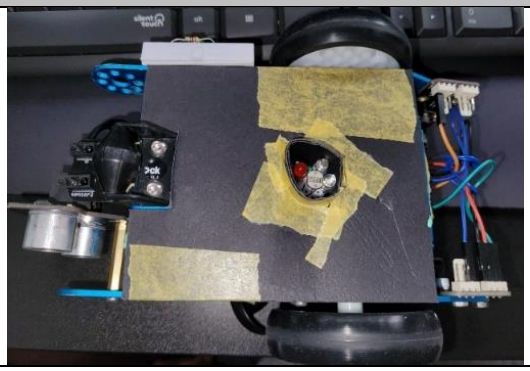
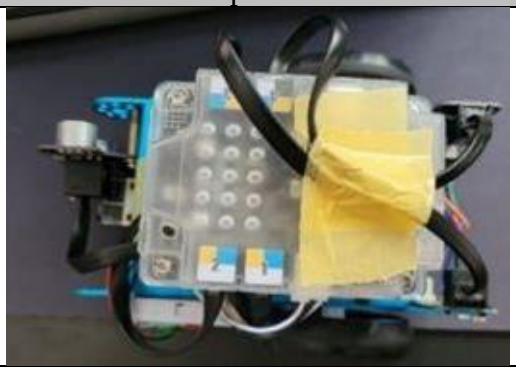
Right View	Left View
	
Bottom View	**Top View**
	

Figure 2: Pictures of mBot.

2. Maze-Solving Algorithm

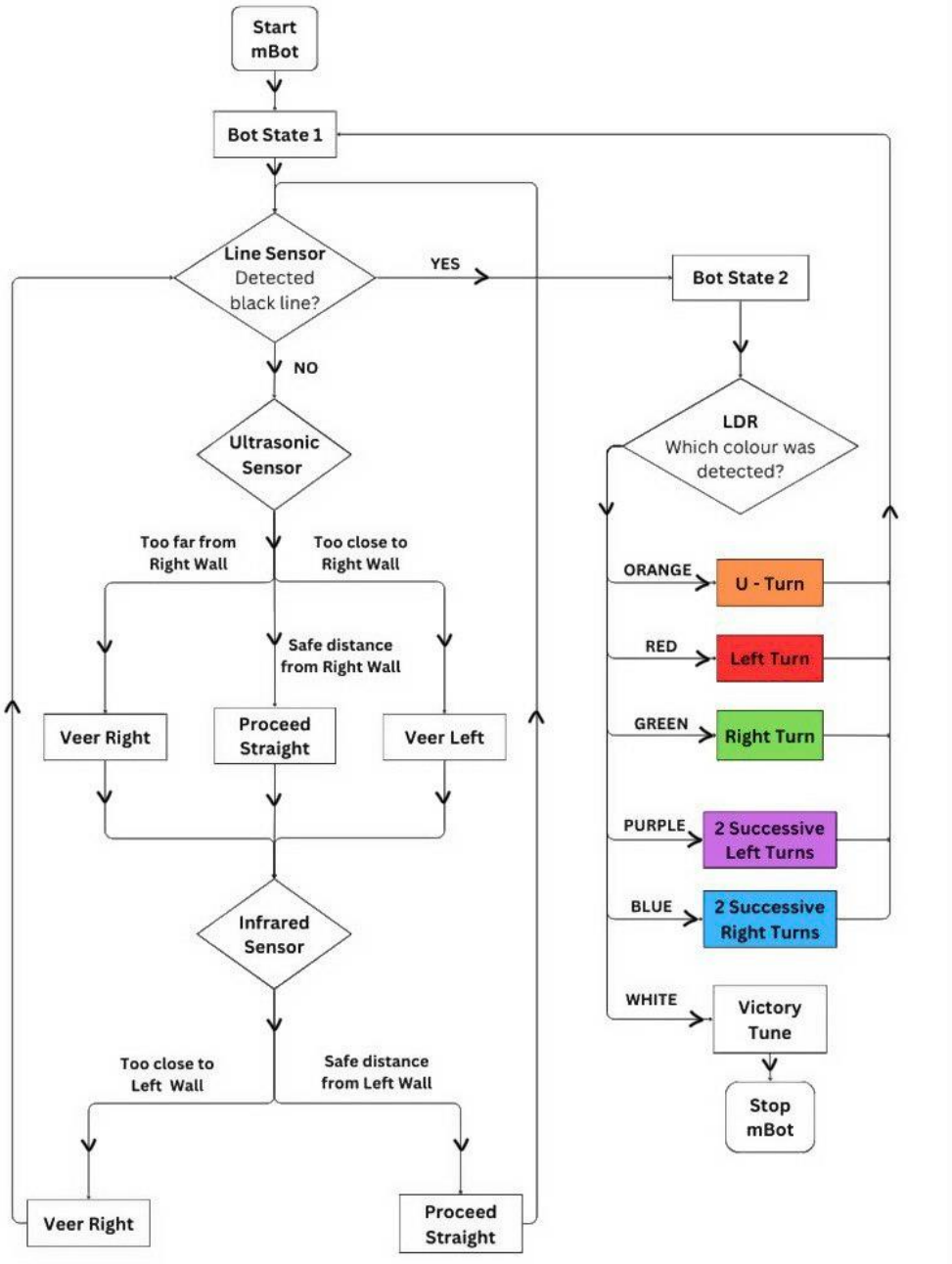


Figure 3: Algorithm used by the robot to solve the maze.

Our robot's operation involves two primary states: the movement state and the colour detection state. In the movement state, the robot consistently moves forward. The line sensor monitors for either a coloured grid or a black line, while both the ultrasonic and IR sensors collaborate to ensure the robot maintains a straight path during the forward movement.

Upon detecting a black line with the line sensor, the robot transitions into the colour detection state. The identification of the grid's colour triggers a corresponding action based on the colour's requirements. After executing the specific action, the robot will seamlessly transition back to the movement state and continues to solve the maze. An exception is made for the detection of a white grid, signifying the end of the maze.

3. mBot Navigation Subsystems

The mBot is comprised of three essential subsystems that play integral roles in facilitating its operations, each dedicated to specific functionalities: movement, colour detection, and path correction. These subsystems are intricately connected to the mBot's central processing unit, the mCore, and are efficiently controlled through the implementation of the *HD74LS139P 2-to-4 Decoder IC Chip* and the *L293D motor chip*. In the subsequent sections, we will delve into a detailed exploration of each subsystem, elucidating their individual navigational functions.

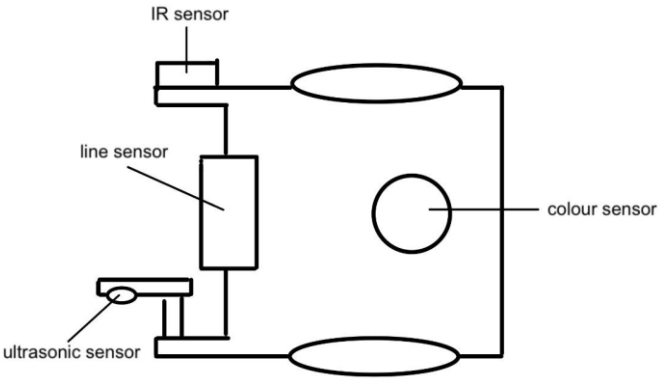


Figure 4: Location of subsystems.

3.1 Basic Movement Operations

The fundamental movement of the robot is facilitated by two Permanent Magnet DC (PMDC) motors strategically positioned beneath the chassis. These motors propel the robot forward by driving the two rear wheels, while a single unpowered front pivot wheel provides stability. Navigating through a maze involves executing various movement operations, including forward travel, left turns, right turns, and U-turns. The precision of these movements is further enhanced by the integration of sensors such as ultrasonic, infra-red, and line sensors. These sensors play a crucial role in ensuring the robot's path remains straight and facilitating the halting of the mBot for colour detection purposes.

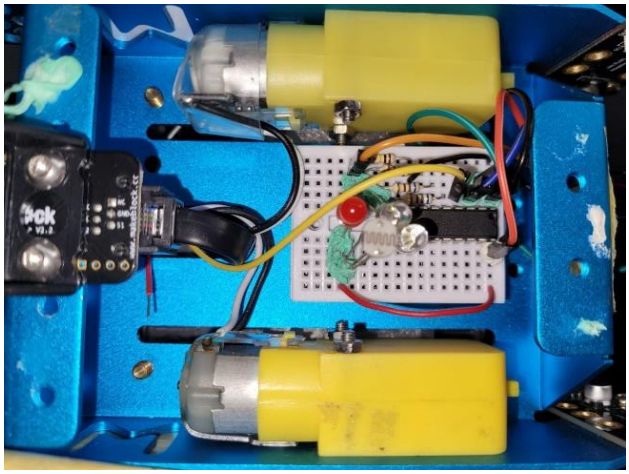


Figure 5: Motors

```
1. void moveForward() {
2.   leftMotor.run(-255); // Negative: Left wheel revolves forwards
3.   rightMotor.run(255); // Positive: Right wheel revolves forwards
4. }
5. void stopRobot() {
6.   leftMotor.stop();
7.   rightMotor.stop();
8. }
```

Figure 6: Code snippet for controlling PMDC motor.

The *moveForward()* function initiates forward movement of the robot by instructing the left and right motors to rotate the wheels in the required directions. Calling *stopRobot()* ceases the operation of both motors, resulting in the robot coming to a complete stop.

3.1.1 Static Turns

```
 1. void turnLeft() {
 2.   leftMotor.run(255); // Positive: Left wheel revolves backwards
 3.   rightMotor.run(255); // Positive: Right wheel revolves forwards
 4.   delay(300);
 5.   leftMotor.stop();
 6.   rightMotor.stop();
 7. }
 8. void turnRight() {
 9.   leftMotor.run(-255); // Negative: Left wheel revolves forwards
10.   rightMotor.run(-255); // Negative: Right wheel revolves backwards
11.   delay(325);
12.   leftMotor.stop();
13.   rightMotor.stop();
14. }
15. void uTurn(double distance_right) {
16.   if(distance_right <= 8) {
17.     // mBot turns to the Left by 180 degrees (on the spot)
18.     leftMotor.run(255); // Positive: Left wheel revolves backwards
19.     rightMotor.run(255); // Positive: Right wheel revolves forwards
20.     delay(590);
21.   } else {
22.     // mBot turns to the Right by 180 degrees (on the spot)
23.     leftMotor.run(-255); // Positive: Left wheel revolves forwards
24.     rightMotor.run(-255); // Positive: Right wheel revolves backwards
25.     delay(590);
26.   }
27. }
```

Figure 7: Code snippet for Basic turns and U-turn.

To achieve simple right-angled turns, we programmed the mBot to activate only one motor during the turn. When making a left turn, the right motor is engaged, while the left motor remains on standby. Conversely, for a right turn, the inverse is true. The process for a right-angled turn is akin to a 180-degree turn, with the key difference lying in the duration of the turn itself. Therefore, the primary challenge involved fine-tuning the duration for which the single motor is powered to ensure the robot completes either a 90 or 180-degree rotation. Our team addressed this challenge through a process of trial and error, gradually adjusting the motor's power duration to ensure the robot faced the correct orientation after each turn.

Additionally, to prevent the robot from colliding with the wall during its 180-degree U-turn, we added a condition to check if it was too near the right or left wall. If true, it will proceed to turn 180 degrees in the other direction instead, since there was more leeway of space for the robot to turn safely.

```
void successiveLeft() {
  // Turn mBot to the left by 90 degrees.
  turnLeft();

  // Then, move mBot forward by one tile.
  moveForward();
  delay(700);

  // After moving forward, stop motors and wait for a short duration for mBot to stabilise.
  leftMotor.stop();
  rightMotor.stop();
  delay(50);

  // Finally, turn mBot to the left by 90 degrees again.
  turnLeft();
}
```

Figure 8: Code snippet for Successive Left turns.

The robot is also tasked with performing consecutive left and right turns. For the double left turn sequence, it consists of a basic 90-degree left turn, a brief forward movement, followed by another 90-degree left turn. The left turns utilize the previously mentioned simple turn left function. The duration of the forward movement was also fine-tuned through a process of trial and error.

3.1.2 Difficulties faced for Motor Operation

In general, incorporating motor functions into both the physical robot circuit and the digital code was a relatively straightforward process. However, the calibration for the successive turn operations presented a greater challenge due to the narrow margin of error permitted. Since the functions were hard-coded, there was a significant risk of the robot colliding with a wall, as there was no path correction during the execution of the turn. This lack of correction made the robot susceptible to deviations caused by very slight differences in the starting position, leading to potential over-travel or over-rotation. Therefore, achieving precise successive turns not only required accuracy during the operation itself but also demanded the robot to be in a relatively precise position before initiating the turn. The ultimate challenge was not just executing the successive turns but also ensuring that the path correction was dependable enough to establish the necessary initial conditions for the turns.

Furthermore, an unforeseen challenge arose when we discovered that one of the motors provided to us was missing its axle. This absence rendered us incapable of securely attaching the wheel to the motor, resulting in an unstable wheel that significantly impeded the robot's performance. Unfortunately, this issue went unnoticed until later in the project timeline, causing a notable delay in our overall progress.

3.2 Dynamic Path Realignment

In addition to fundamental movement, the robot accomplished precise path adjustments within the maze through a synergy of the ultrasonic sensor and the IR sensor. The code consistently finetunes the robot's trajectory based on the measured values from these sensors.

3.2.1 Ultrasonic Sensor

The ultrasonic sensor, a component included in the mBot kit, is capable of measuring distances between the sensor and the object directly in front of it. Positioned on the front of the robot facing left, the ultrasonic sensor was utilized to gauge the distance between the robot and the left wall. The value obtained from the ultrasonic sensor was through port 2 of the mCore.

Figure 9: Ultrasonic sensor

```
double ultrasonic_dist() {
  pinMode(ULTRASONIC, OUTPUT);
  digitalWrite(ULTRASONIC, LOW);
  delayMicroseconds(2);
  digitalWrite(ULTRASONIC, HIGH);
  delayMicroseconds(10);
  digitalWrite(ULTRASONIC, LOW);
  pinMode(ULTRASONIC, INPUT);

  float duration = pulseIn(ULTRASONIC, HIGH, ULTRASONIC_TIMEOUT);

  float dist = -1;

  if (duration > 0) {
    dist = duration / 2.0 / 1000000 * SPEED_OF_SOUND * 100;
  } else {
    dist = -1;
  }

  return dist;
}
```

Figure 10: Code snippet for Ultrasonic Sensor

This function sends an ultrasonic pulse, measures the time it takes for the pulse to return, and then calculates the distance based on the speed of sound. The distance is returned, and if no valid measurement is obtained, -1 is returned.

3.2.2 Infra-Red Sensor

The infrared sensor was built by the team on a breadboard. We selected a $R_{receiver}$ value of 8000Ω and a $R_{emitter}$ value of 100Ω. Through numerous tests, this combination of resistances was determined to yield the most optimal performance for the IR sensor, ensuring both effectiveness and reliability.

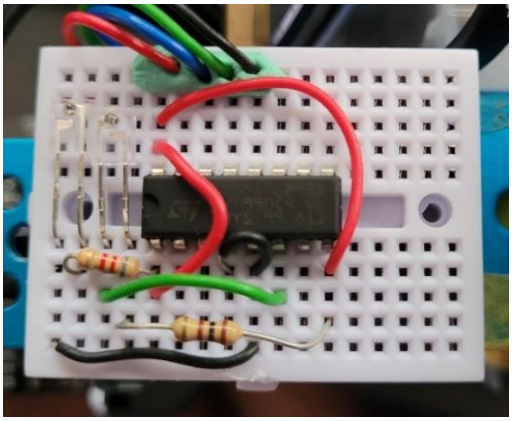


Figure 11: IR Sensor

For our IR sensor circuit, the negative terminal of the IR emitter was connected to the Y0 pin of the decoder IC chip. The same breadboard accommodates both our IR sensor and the L293D motor chip. To ensure the circuit seamlessly integrates with the robot's body, we carefully trimmed and organized the wires, the IR emitter and detector, and the resistors. Blue tack was then employed to secure all the wires in one place, minimizing the risk of any components hitting the maze walls. Being positioned on the right side of the robot, the IR sensor effectively detects distances between the robot and the right wall.

3.2.3 Integration of both Ultrasonic and IR sensor

Utilizing the two sensors, we could consistently measure the distances of both walls from our mBot, enabling the calculation of the mBot's position on the course. Leveraging the larger range and higher accuracy of the ultrasonic sensor, the mBot would primarily adjust its course based on its readings. Only when the IR sensor detected a value indicating the bot was too near to the left wall would there be a rightward adjustment to the robot's trajectory. This strategic combination allowed for effective and precise path adjustments in navigating the maze.

```
 1.  float shineIR() {
 2.     // Power on IR Emitter
 3.     analogWrite(A,LOW); //Setting A0 to High/Low
 4.     analogWrite(B,LOW); //Setting A1 to High/Low
 5.
 6.     delay(500);
 7.
 8.     float raw = analogRead(IRD);
 9.     float ans = (raw - 6.525)/19.826;
10.     return ans;
11.  }
```

Figure 12a: Code snippet for reading data from IR Sensor

In the above code, the IR emitter is activated, waits for a short duration, reads the analog voltage from the connected IR sensor, apply a calibration formula to convert the raw sensor reading into a distance value, and returns the distance value.

```
1.   //……
2.     }else if(distance_left < 9) { // Too close to the left, move right
3.        leftMotor.run(-255); // Negative: wheel turns anti-clockwise
4.        rightMotor.run(255 - dx*15); // Positive: wheel turns clockwise
5.        delay(50);
6.      }
7.
```

Figure 12b: Code snippet for Ultrasonic Sensor

In the above code, the IR sensor provides values corresponding to its distance from the left wall. A higher value signifies a greater distance, while a lower value indicates proximity to the wall. If the value is deemed too low, the robot will execute a sharp right turn to avert collision with the wall. This approach ensures that the robot responds dynamically to its surroundings, enhancing its ability to navigate the maze effectively.

```
1. if (distance_right < 9 && distance_right != -1) { // Left
2.       leftMotor.run(-255 - dx*15); // Negative: wheel turns anti-clockwise
3.       rightMotor.run(255); // Positive: wheel turns clockwise
4.       delay(50);
5. }
6. //……
```

Figure 13: Code snippet for Ultrasonic Sensor

The code snippet above outlines our programming approach for the ultrasonic sensor, calibrated at 9cm (considered rightward positive). This specific distance was determined as optimal, allowing the mBot to navigate the grids without encountering bumps along the course.

Regarding movement, when the mBot maintains precisely 9cm from the left wall, it proceeds forward in a straight line. If it exceeds this distance, the left motor slows down relative to the right motor, effecting a leftward correction. Conversely, if the distance falls below 9cm, the right motor slows down, causing a rightward correction. We aimed for subtle variations in motor speeds to ensure smooth path corrections.

3.2.4 Difficulties faced for Path Realignment

Developing a path correction algorithm for the mBot posed various challenges, notably due to Arduino's limited computational power. Designing an algorithm that ensures efficient real-time processing, accurate sensor inputs, and addressing calibration issues were crucial. Striking a balance between responsiveness and stability was challenging, as overly aggressive adjustments caused oscillations, while conservative responses led to imprecise positioning. Initially using fixed values for veering corrections proved insufficient, since extreme conditions required more time of the mBot to return to the straight path, prompting the introduction of a dynamic variable (dx) that increases or decreases the amount of veer required for efficient and effective path correction under varying conditions.

3.3 Grid & Colour Detection

Our robot employs a grid detection system comprising a line detector (Me Line Follower) and a colour sensor circuit that utilises an LDR and Red, Green and Blue LEDs.

3.3.1 Makeblock Line Detector

```
int sensorState = lineFollower.readSensors(); // read the line sensor's state
if (sensorState != S1_OUT_S2_OUT) { // run color sensor when mbot stops at black line
  stopRobot();

  enum Colours colour = detectColour();
  switch(colour) {
    case detectRed:
      turnLeft();
      break;
    case detectGreen:
      turnRight();
      break;
    case detectOrange:
      uTurn(distance_right);
      break;
    case detectPurple:
      successiveLeft();
      break;
    case detectBlue:
      successiveRight();
      break;
    case detectWhite:
      leftMotor.stop();
      rightMotor.stop();
      while(1) {
        celebrate(); // play "Never Gonna Give You Up"
      }
      break;
  };
}
```

Figure 14: Code snippet for Line Detector

The program continually monitors the state of the line follower. If the line follower's status shifts away from *S1_OUT_S2_OUT*, indicating detection of the colour black, the robot comes to a halt by stopping both its left and right motors. The line detector plays a pivotal role in identifying coloured grids by recognizing the black horizontal lines on each grid. Subsequently, the colour detection algorithm is executed to identify the colour beneath the robot. Depending on the detected colour, the robot takes specific actions assigned to the colour beneath the robot.

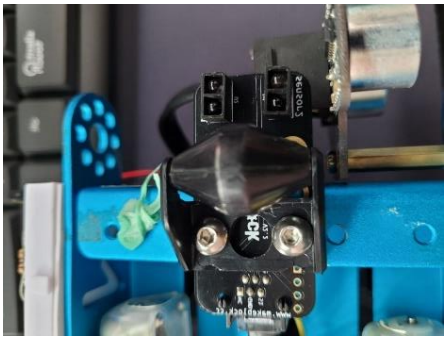


Figure 15: Line sensor

3.3.2 LDR Colour Sensor

Figure 16: LEDs and LDR inside the paper cylinder.

The colour sensor utilizes an RGB lamp and a light-dependent resistor (LDR). The RGB lamp sequentially illuminates in the sequence of red, green, and blue, with the reflected light from the ground being detected by the LDR. The lamp activation follows a specific order through a decoder IC, where Y_1, Y_2, and Y_3 are set to a low output in sequence. Subsequently, the sensor sends an analog value to the Arduino based on the voltage from the LDR circuit. We employed a 104kΩ resistor for the LDR circuit, and three 680Ω resistors for the LEDs. The resistor values were determined based on the respective LEDs' data sheets to ensure that the current flowing through the LEDs' during operation is at a suitable level for the colour sensor's functionality.

```
1. int detectColour() {
2.   // Shine Red, read LDR after some delay
3.   shineRed();
4.   delay(RGBWait);
5.   colourArray[0] = getAvgReading(5);
6.
7.   // Shine Green, read LDR after some delay
8.   shineGreen();
9.   delay(RGBWait);
10.  colourArray[1] = getAvgReading(5);
11.
12.  // Shine Blue, read LDR after some delay
13.  shineBlue();
14.  delay(RGBWait);
15.  colourArray[2] = getAvgReading(5);
16.
17.  // Turn the IR back on
18.  analogWrite(A, LOW);
19.  analogWrite(B, LOW);
20.
21.  // Red Array values
22.  colourArray[0] = (colourArray[0] - blackArray[0]) / (greyDiff[0]) * 255;
23.  Serial.println(int(colourArray[0]));
24.
25.  // Green Array values
26.  colourArray[1] = (colourArray[1] - blackArray[1]) / (greyDiff[1]) * 255;
27.  Serial.println(int(colourArray[1]));
28.
29.  // Blue Array values
30.  colourArray[2] = (colourArray[2] - blackArray[2]) / (greyDiff[2]) * 255;
31.  Serial.println(int(colourArray[2]));
32. }
```

Figure 17: Code snippet for turning RGB lamps on and obtaining values.

The function above sequentially activates the Red, Green, and Blue LEDs with accompanying delays for stabilization. LDR readings are then obtained for each colour and stored in an array. The final part of the code performs a conversion from voltage values to RGB values. The normalization process involves subtracting the corresponding black values *(blackArray)* from the colour readings and dividing the result by the corresponding grey differences *(greyDiff)*. This procedure scales the colour readings to remove environmental variations or baseline noise, ensuring a more accurate representation of the actual colour values. The final step of multiplying by 255 scales the values to the typical range for RGB colour representation.

3.3.3 Initial White and Black Calibration

```
1. void setBalance() {
2.   // Set white balance
3.   Serial.println("Put White Sample For Calibration ...");
4.   delay(5000);
5.
6.   // Scan the white sample.
7.   // Go through one colour at a time, set the maximum reading for each colour
8.   // Red, green, and blue to the white array
9.   for(int i = 0; i < 3; i++) {
10.    digitalWrite(A, LED_Array[i].A_val); // Setting A0 to High/Low
11.    digitalWrite(B, LED_Array[i].B_val); // Setting A0 to High/Low
12.    delay(RGBWait);
13.    whiteArray[i] = getAvgReading(5); // Get average of 5 readings and store in white
14.    Serial.print("White Array ");
15.    Serial.print(i);
16.    Serial.print(" Value: ");
17.    Serial.println(whiteArray[i]);
18.  }
19.
20.  Serial.println("Put Black Sample For Calibration ...");
21.  delay(5000); // Delay for five seconds for getting sample ready
22.
23.  // Next, scan black sample
24.  // Go through one colour at a time, set the maximum reading for each colour
25.  // Red, green, and blue to the black array
26.  for(int i = 0; i < 3; i++) {
27.    digitalWrite(A, LED_Array[i].A_val); // Setting A0 to High/Low
28.    digitalWrite(B, LED_Array[i].B_val); // Setting A0 to High/Low
29.    delay(RGBWait);
30.    blackArray[i] = getAvgReading(5); // Get average of 5 readings and store in white
31.    Serial.print("Black Array ");
32.    Serial.print(i);
33.    Serial.print(" Value: ");
34.    Serial.println(blackArray[i]);
35.    // The difference between the maximum and the minimum gives the range
36.    greyDiff[i] = whiteArray[i] - blackArray[i];
37.    Serial.print("Grey Array ");
38.    Serial.print(i);
39.    Serial.print(" Value: ");
40.    Serial.println(greyDiff[i]);
41.  }
42.  Serial.println("Colour Sensor Is Ready.");
43. }
```

Figure 18: Code snippet of Initial Calibration

This function serves as the initial calibration step for obtaining white and black reference values in a colour sensor system. The function prompts the user to place a white sample for calibration, and after a delay of 5 seconds, it scans the white sample. For each colour (red, green, and blue), it sets the corresponding LED values and records the maximum reading in the *whiteArray*. The process is then repeated for a black sample after another 5-second delay. The function calculates the range of values *(greyDiff)* for each colour by subtracting the corresponding black values from the white values. These reference values are crucial for subsequent colour detection, allowing the system to compensate for environmental variations and provide accurate colour readings. Upon completion, the function prints the calibrated values and signals that the colour sensor is ready for use. This calibration ensures reliable colour detection by establishing a baseline in the presence of varying lighting conditions.

	Red	Green	Blue
White	988	1009	992
Black	908	987	906
Grey Diff	80	22	86

Figure 19: Values obtained for white and black, stored in an array.

Therefore, when we fed these values into the serial monitor, it aids us in identifying and interpreting the readings obtained by the sensor during the colour grid detection process.

3.3.4 Differentiation and Recognition of Colours

To determine the RGB value range for the set of colour grids, we conducted a straightforward experiment. We measured the RGB values for each of the seven colour grids, collecting data from ten samples for each grid. Subsequently, we analysed the data to observe trends and establish a trendline. This trendline serves as a visual representation of the general behaviour of RGB values across the colour grids. Additionally, from the collected data, we extracted the minimum and maximum RGB values for each colour grid. This information provides a comprehensive understanding of the variability in colour representations and aids in setting reference ranges for subsequent colour detection.

Colour	R	G	B
Red	225 – 235	100 - 115	140 - 150
Orange	220 - 235	125 - 140	130 - 145
Blue	160 - 175	225 - 240	230 - 245
Purple	180 - 195	230 - 245	235 - 245
Green	110 - 125	235 - 245	210 - 225
White	240 - 265	245 - 275	240 - 270

Figure 20: Table of obtained RGB ranges

```
// If R, G, and B over 240 = WHITE
if (colourArray[0] >= 240 && colourArray[1] >= 240 && colourArray[2] >= 240) {
  led.setColor(255, 255, 255); // set both LEDs to WHITE
  led.show();
  return detectWhite;
}

// When GREEN is MAX
if (colourArray[1] > colourArray[0] && colourArray[1] > colourArray[2]) {
  if (colourArray[2] <= 230) {
    led.setColor(0, 255, 0); // set both LEDs to GREEN
    led.show();
    return detectGreen;
  } else if (colourArray[0] >= 180) {
    led.setColor(128, 0, 128); // set both LEDs to PURPLE
    led.show();
    return detectPurple;
  } else {
    led.setColor(0, 0, 255); // set both LEDs to BLUE
    led.show();
    return detectBlue;
  }
}

// When RED is MAX
else if (colourArray[0] > colourArray[1] && colourArray[0] > colourArray[2]) {
  if (colourArray[1] <= 120) {
    led.setColor(255, 0, 0); // set both LEDs to RED
    led.show();
    return detectRed;
  } else {
    led.setColor(255, 165, 0); // set both LEDs to ORANGE
    led.show();
    return detectOrange;
  }
} else if (colourArray[2] > colourArray[0] && colourArray[2] > colourArray[1]) {
  if (colourArray[0] >= 180) {
    led.setColor(128, 0, 128); // set both LEDs to PURPLE
    led.show();
    return detectPurple;
  } else {
    led.setColor(0, 0, 255); // set both LEDs to BLUE
    led.show();
    return detectBlue;
  }
}
```

Figure 21: Code snippet for differentiating colours.

This code defines a colour detection algorithm based on RGB values received from *colourArray*. The first condition checks if all three colour components (R, G, and B) are greater than or equal to 240, indicating a predominantly white colour. In such cases, it sets both LEDs to white using the RGB values (255, 255, 255), and returns a detection signal for white.

The subsequent conditions differentiate between various colours by comparing the intensity of each RGB component. When green is predominant (G > R and G > B), it further distinguishes between the other shades. If the blue component is below 230, the algorithm identifies it as green, setting the LEDs to green and returning a green detection signal. If the red component is relatively high (R >= 180), it identifies the colour as purple, setting the LEDs accordingly. Otherwise, it identifies the colour as blue.

Similarly, when red is the dominant colour (R > G and R > B), it distinguishes between pure red and orange based on the intensity of the green component. If green is below or equal to 120, it identifies the colour as red; otherwise, it identifies it as orange. If blue is the predominant colour (B > R and B > G), it again differentiates between purple, and blue based on the intensity of the red component. Overall, the code provides a versatile colour detection system with corresponding LED displays and detection signals for white, green, purple, blue, red, and orange.

3.3.5 Difficulties faced for Colour Detection

A challenge arose when distinguishing between blue and purple colours due to fluctuations in their RGB values, with the blue value occasionally being the maximum value and other times the green value taking precedence. This variability required the inclusion of an additional else if statement within the section handling green as the dominant colour. To address this issue effectively, the solution involved incorporating two distinct if statements - one for scenarios where blue is the maximum and another for cases where green is the maximum. Within the green-max function, specific criteria were established to differentiate between green, blue, and purple, ensuring accurate colour identification. Similarly, in the blue-max condition, the code was structured to discern between blue and purple. This approach ensures accurate categorization of colours based on the dominant component, whether it is the blue or green value.

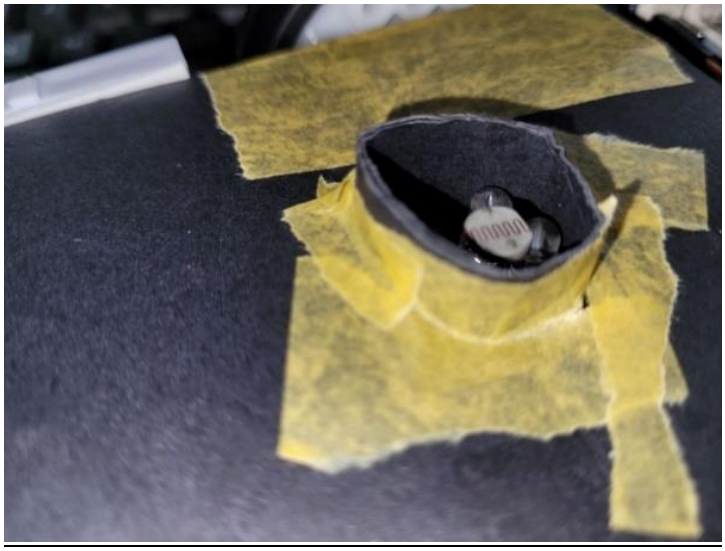


Figure 22: Chimney Design

In addition to calibrating the light sensor, we identified an issue where surrounding ambient light interfered with the sensor's readings, compromising its ability to accurately differentiate colours. To address this, we constructed a chimney using black masking tape and paper to shield the LDR from ambient light, as well as covering the entire open bottom of the mBot, ensuring more reliable and precise colour detection.

3.4 Miscellaneous

3.4.1 Victory Tune

```
void celebrate() {
  int notes = sizeof(melody) / sizeof(melody[0]) / 2;

  // Calculates the duration of a whole note in milliseconds
  int wholenote = (60000 * 4) / tempo;

  int divider = 0, noteDuration = 0;
  for (int thisNote = 0; thisNote < notes * 2; thisNote = thisNote + 2) {
    // Calculates the duration of each note
    divider = melody[thisNote + 1];
    if (divider > 0) {
      // Regular note, just proceed
      noteDuration = (wholenote) / divider;
    } else if (divider < 0) {
      // Dotted notes are represented with negative durations!!
      noteDuration = (wholenote) / abs(divider);
      noteDuration *= 1.5; // Increases the duration in half for dotted notes
    }

    // Only play the note for 90% of the duration, leaving remaining 10% as a pause
    buzzer.tone(melody[thisNote], noteDuration * 0.9);

    // Wait for the specified duration before playing the next note.
    delay(noteDuration);

    // Stop the waveform generation before the next note.
    buzzer.noTone();
  }
}
```

Figure 23: Code snippet of victory tune

Upon detecting a white colour in the colour detection mode, the robot ceases movement and initiates a victory tune. As the code doesn't revert to the moving state, the robot remains stationary on the white colour, simultaneously playing the victory tune. This indicates the successful completion of the maze and marks the conclusion of the robot's operation.

3.4.2 Neatness

Maintaining the impeccable organization of our robot's wirings was a meticulous process, achieved through a combination of strategic methods. Wire stripping was employed with precision to remove insulation, ensuring clean and secure connections. Cutting wires to optimal lengths not only reduced clutter but also enhanced the overall aesthetic of the robot. We employed a systematic taping approach to bundle and route wires neatly, preventing entanglement and facilitating easy troubleshooting. Tying wires in strategic locations further secured them in place, minimizing the risk of accidental disconnections or interference with moving parts. Additionally, the innovative use of blue tack played a key role in affixing small components, such as sensors and connectors, securely in position while maintaining a sleek appearance. Through these varied techniques, we achieved a harmonious blend of functionality and visual tidiness in our robot's wiring, ensuring not only efficient performance but also a professional and organized presentation.

3.4.2 Minor Components

In addition to the core components like motors and sensors, several auxiliary elements play a crucial role in ensuring the seamless operation of the robot. The battery serves as the powerhouse, providing the necessary electrical energy to drive the motors and power the electronic components. Well-organized wires and cables, meticulously routed and secured, facilitate efficient connectivity among various parts of the robot, promoting a neat and functional design. RJ25 connectors, with their standardized design, simplify the connection of modules and sensors, enhancing the modularity of the robot's construction. These components collectively form the backbone of the robot's functionality, enabling it to perform tasks with reliability and precision by ensuring a stable power supply, effective communication between modules, and an organized wiring infrastructure.

4. Work Division

In our team of just three members, seamless collaboration was achieved as each member assumed their designated roles right from the project's inception. Neeraj showcased exceptional proficiency in coding within the Arduino UNO software, efficiently employing shortcuts and demonstrating a thorough understanding of Arduino terminology. Consequently, he took the lead as the primary coder for the project. Mark, with his meticulous and precise approach, took charge of crafting both the IR and LDR circuits. His attention to detail ensured the creation of compact and tidy breadboards on the mBot, minimizing clutter and interference with other components. Meanwhile, Kin, with his creative vision and adept oversight, assumed the role of project director. He skilfully guided the team, orchestrating task priorities and overseeing the assembly of the mBot, inclusive of all components, breadboards, and wiring. This synergistic collaboration allowed our small team to achieve remarkable efficiency and success in our project endeavours.

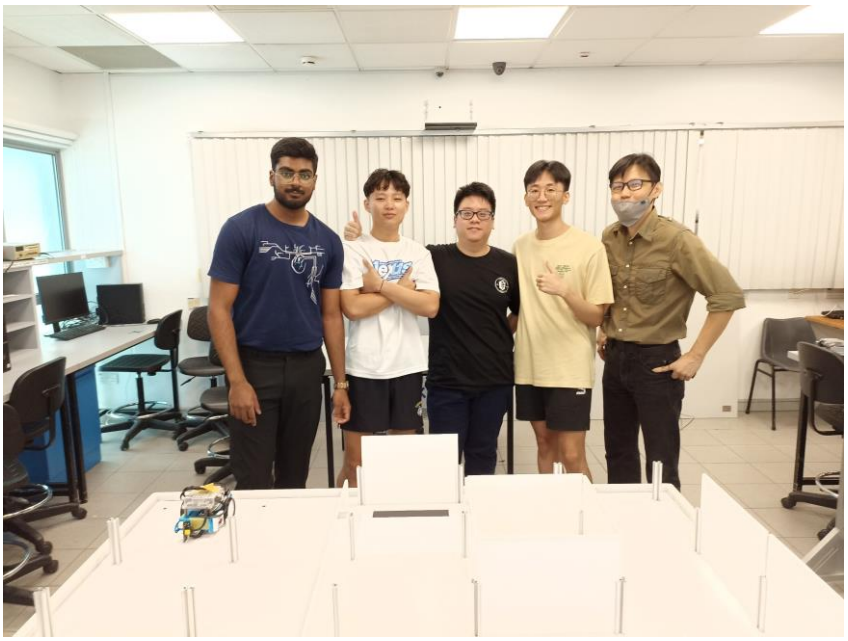


Figure 24: S3T4 with the 2 goats, Prof Henry Tan and TA Ryan Sim.

End

5. APPENDIX

Pictures for grading neatness.

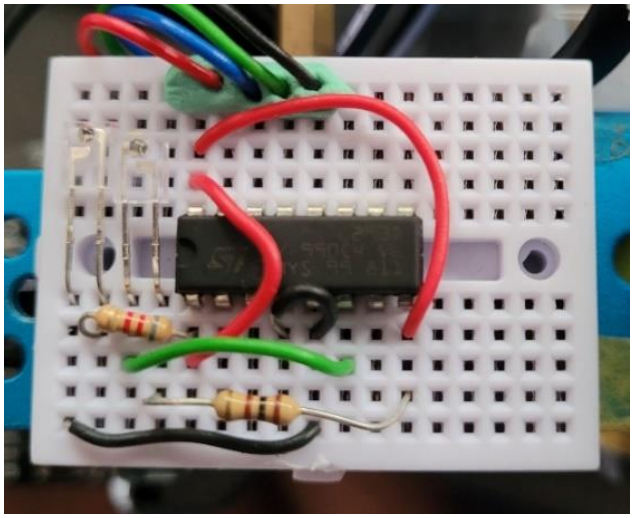


Figure 25: IR Sensor

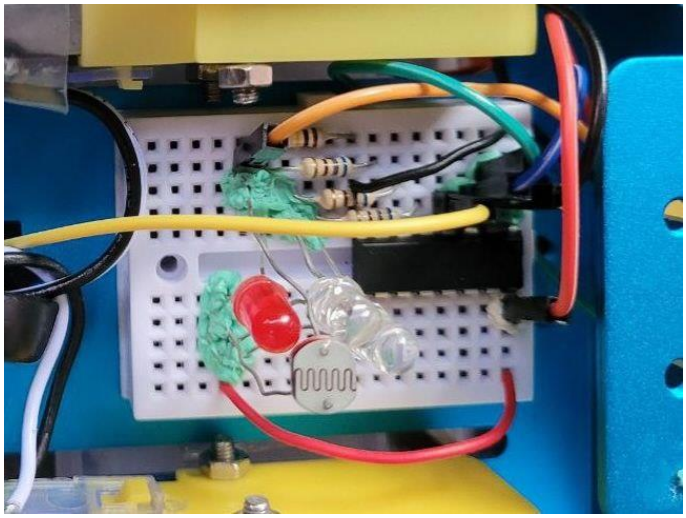


Figure 26: Colour Sensor

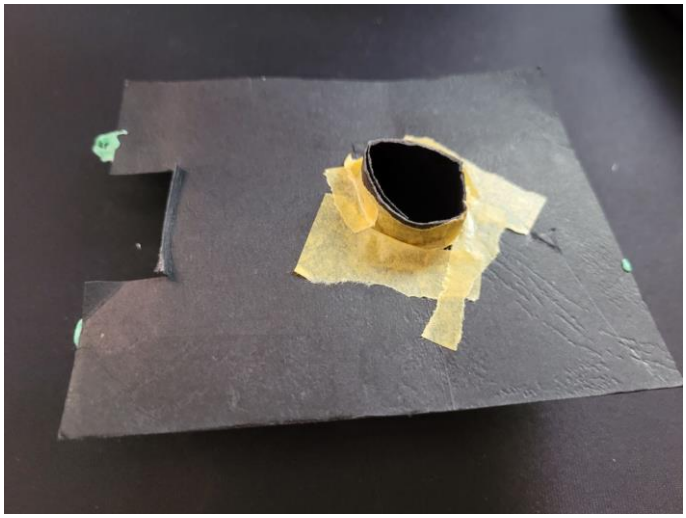


Figure 27: Skirting