CG1111A
ENGINEERING PRINCIPLES AND PRACTICE I

Studio Group : B06
Team : 4

YAO XIANG	A0299826E
YANG ZHENZHAO	A0309132N
YEE JIA JIE	A0301530B
YAP JIA WEI	A0307951B

---

1. mBot Pictorial Overview

1.1 Main Body

Figure 1.1: Left View

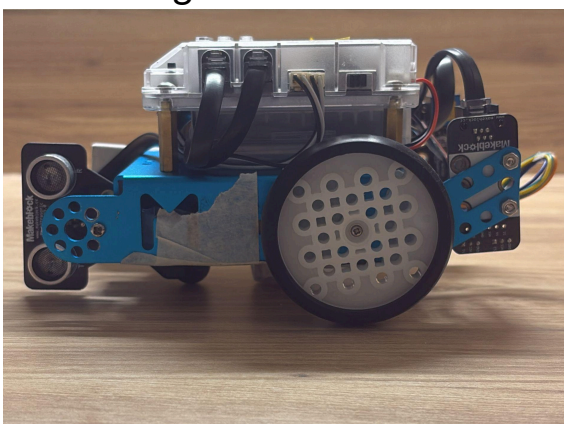


Figure 1.2: Right View

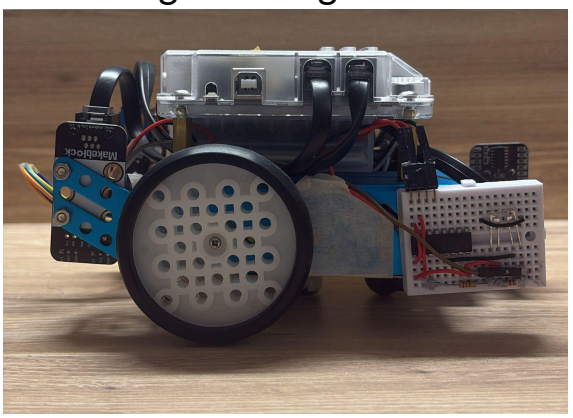


Figure 1.3: Top View

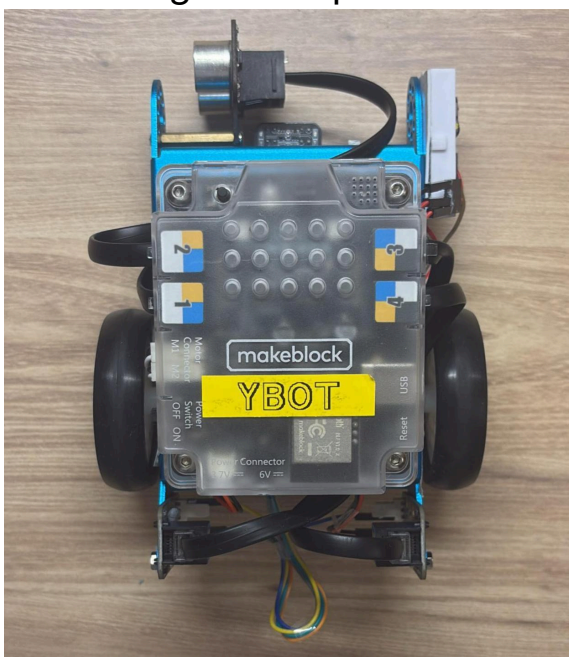


Figure 1.4: Bottom View

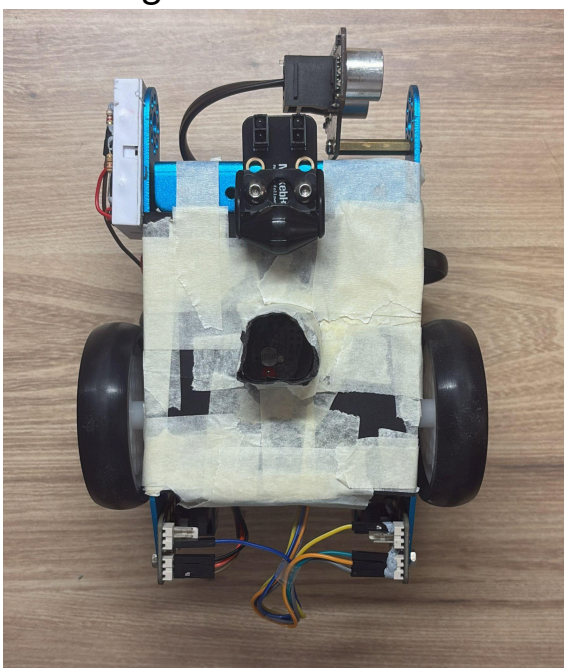


Figure 1.5: Front View

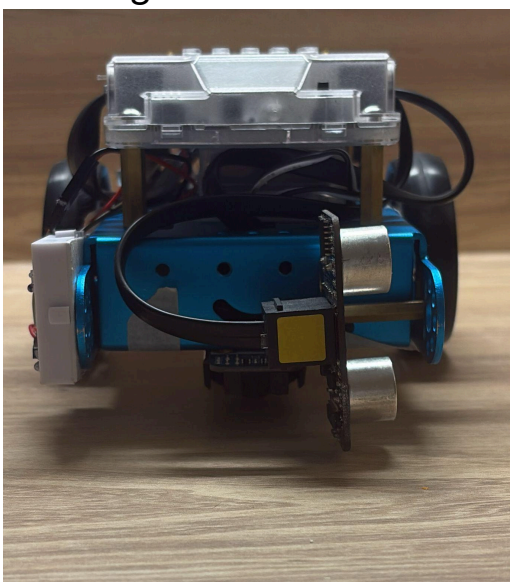
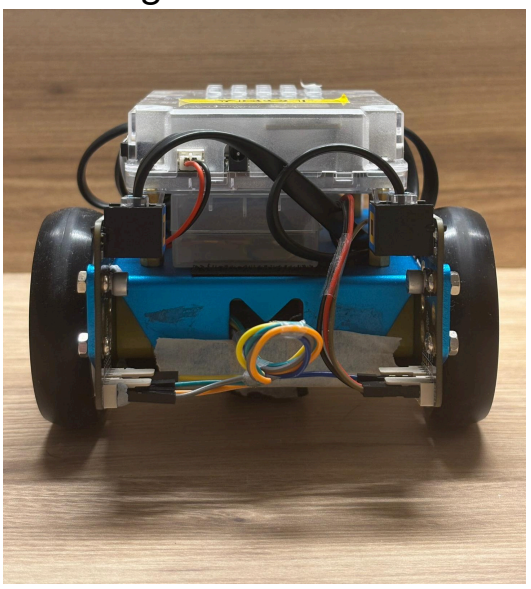


Figure 1.6: Back View



1.2 LDR Sensor Circuit

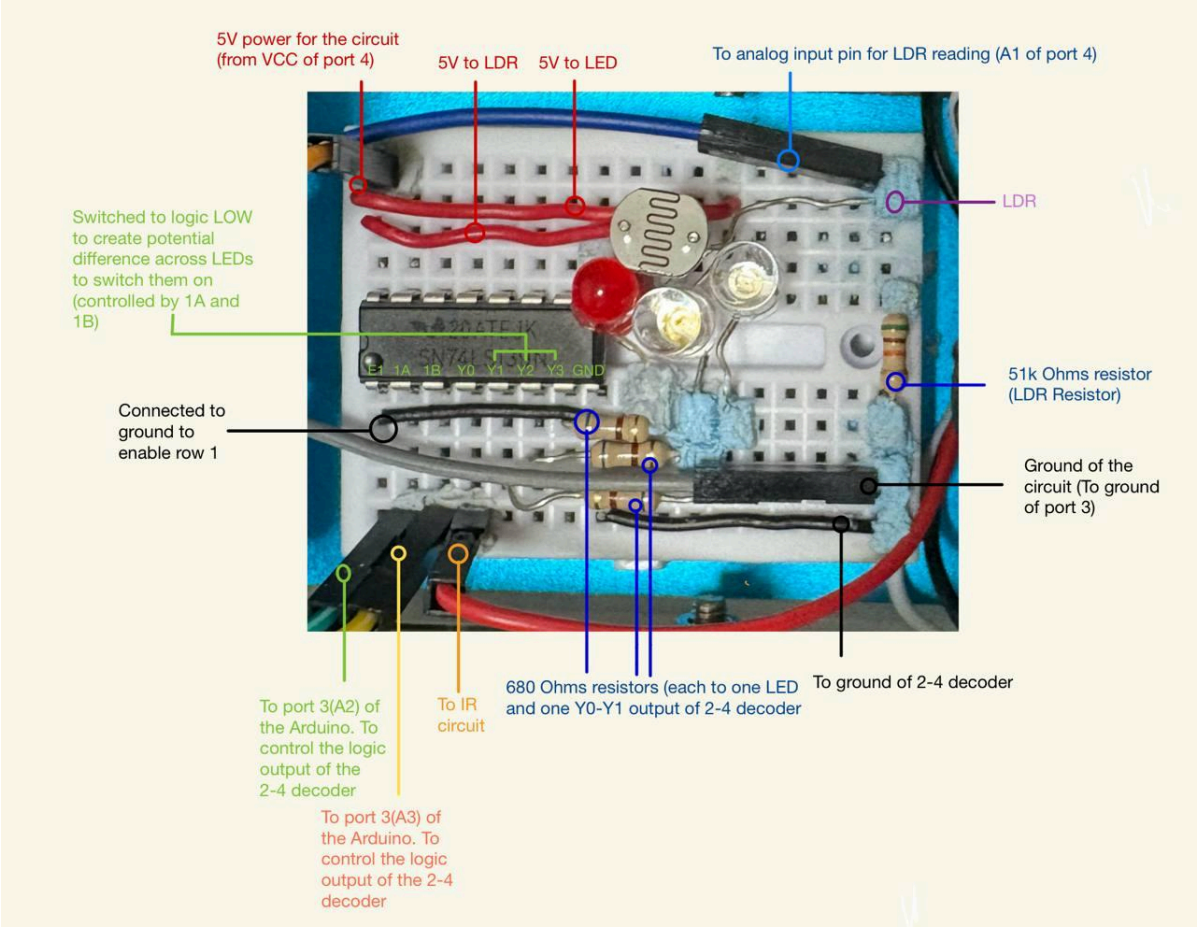


Figure 1.7: LDR Sensor Circuit

Circuit power supply

Ground and power of the LDR circuit are connected to the VCC and ground of port 3 via the RJ25 adapter.

- Power is connected to the VCC of the 2-to-4 decoder, the anode of the LEDs, and the LDR.
- Ground is connected to Enable 1G and the ground of the 2-4 decoder and the LDR resistor.

Logic control for LEDs

1A and 1B pins of the 2-4 decoder are used to control the logic for outputs Y0 - Y3 where a binary input gives 4 possible outputs.

1A and 1B pins are connected to port 3's A2 and A3 pins of the Arduino board respectively. A2 and A3 are set to output to control the 2-4 decoder's logic. LED's cathodes are connected to the 2-4 decoder's outputs via a resistor each.

When Y1, Y2, or Y3 is activated, a logic LOW will be the output, creating a potential difference across the LED, causing it to light up.

LDR Voltage reading

The analog pin A1 from port 4 of the Arduino is set as input and connected across the LDR via the RJ25 adapter to read the voltage value.

Key considerations in circuit design

LDR was placed on top of the 3 LEDs to ensure that LDR does not receive direct light from the LEDs and only light reflected off the coloured papers. This ensures the credibility of the LDR readings collected.

As the circuit is at the bottom of the Ybot, wires can become loose over time from gravity, mishandling, or sharp movements of the Ybot as it navigates the maze. This can result in inconsistent readings from LDR. To tackle this, blu-tacks are placed at parts with potential loose wires.

Resistor choices

LDR resistor

The 51kΩ resistor was chosen to match the 5-100kΩ variable resistance of the LDR so that the LDR circuit can give responsive readings according to the changes in colour.

LED resistors

680Ω resistors are used for our RGB LEDs as they give the desired brightness allowing the LDR circuit to be sensitive enough to the change of colour.

At the same time, this value keeps the current flowing through the 2-to-4 decoder below 8mA to prevent the 2-to-4 decoder from activating its self-protecting mechanism and outputting a positive voltage which can cause unpredictable readings.

To prove that our resistance is of a reliable value, we did some calculations:

Red LED:

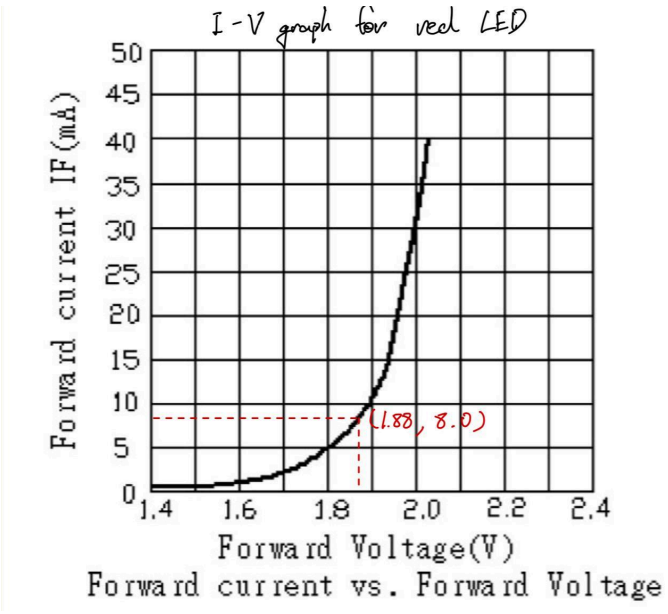


Figure 1.7a: Relation of Red LED with Forward Voltage(according to data sheet)

Maximum voltage difference across the LEDs and their respective resistors: 5.0V
Maximum current allowed to flow through: 0.008A
Voltage over LED lamp at maximum current = 1.88V
Min. resistance value: (5.0 - 1.88) / 0.008 = **390Ω**

Following the exponential upwards trend of I-V graph, we can safely assume that any resistance value above 390Ω for red LED is feasible.

Blue and green LEDs:

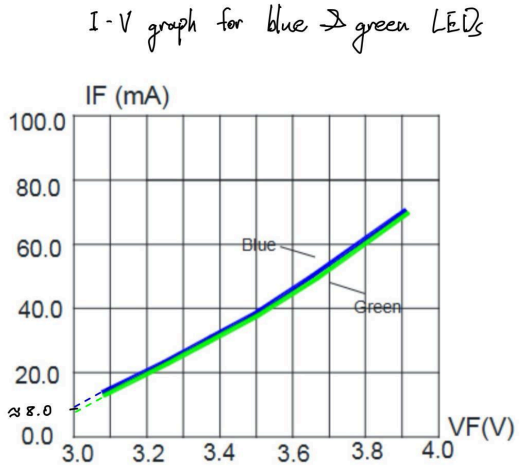


FIG.1 FORWARD CURRENT VS. FORWARD VOLTAGE.

Figure 1.7b: Relation of Green and Blue LED with Forward Voltage(according to data sheet)

Maximum voltage difference across the LEDs and their respective resistors: 5.0V

Maximum current allowed to flow through: 0.008A

Voltage over LED lamp at maximum current ≈3.0V (*via extrapolation of given data with trendline*)

Min. resistance value: (5.0 - 3.0) / 0.008 = **250Ω**

From above calculations, since our choice of **680Ω** on resistors for red, green and blue LEDs is higher than the minimum resistance requirement(*390Ω and 250Ω for red and green/blue respectively*), we can safely assume that our choice is feasible for the LDR colour sensor circuit.

Choice of method for securing Circuit connection

Figure 1.7c: A zoom in of *Figure 1.6: LDR Sensor Circuit*

Blu-tack is utilised due to its ability to stick firmly to anything while also capable of easy removal.

- Though appearing **visually unappealing**, it helped a lot in ensuring **consistent performance** by securing our connections.
- It is more reliable than tape when a situation requires securing connections in a narrow **space.**

1.3 Infrared Sensor Circuit

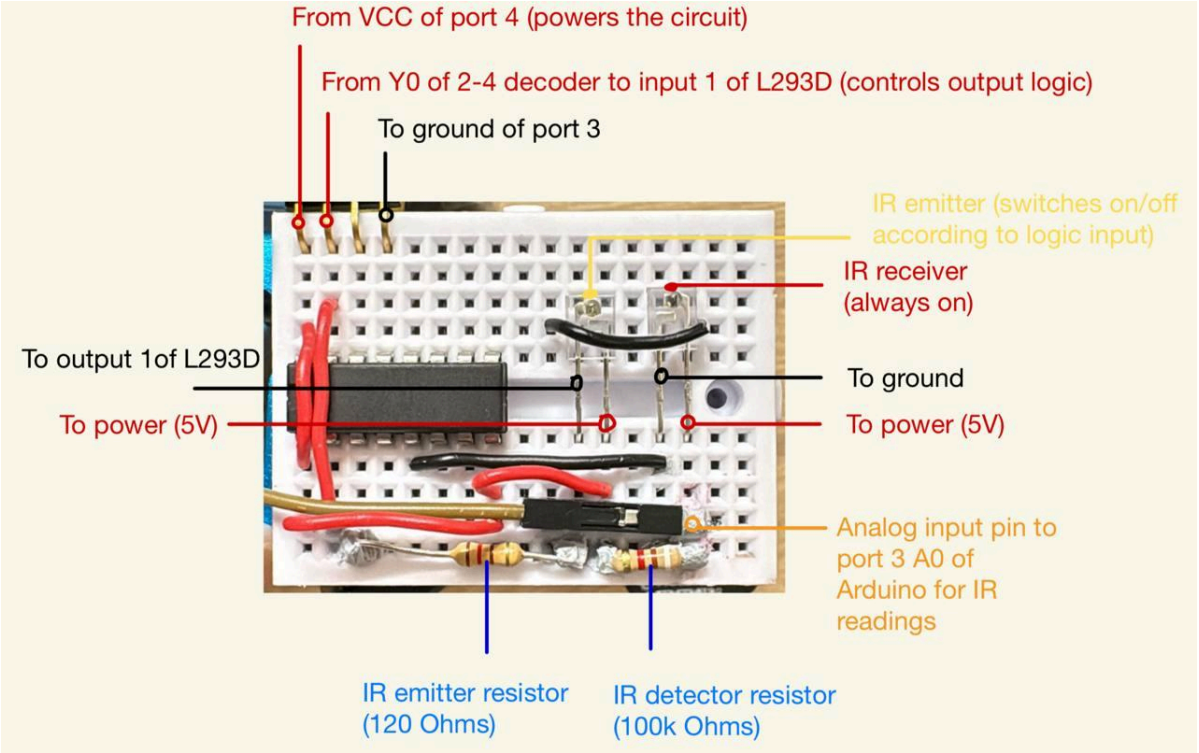


Figure 1.8: Infrared Sensor Circuit

Circuit power supply

The IR sensor circuit's power and ground are connected to the VCC and ground of port 4 via the RJ25 adapter respectively. The power is connected to the L293D H-Bridge's VCC1 and VCC2 and Enable1,2, and the anodes of the IR-emitter and the IR-detector.

Logic control for emitter and detector

The logic input for the L293D chip (Input 1) is connected to the Y0 of the 2-4 decoder in the LDR circuit to control the on and off of the IR emitter. The IR emitter will be switched on when the Y0 pin from the 2-4 encoder gives a logic low output, resulting in a LOW output at the output (output 1) of the LD293D H-Bridge which is connected to the IR emitter's cathode. This will thus result in a potential difference across the IR emitter, switching it on. On the other hand, the IR detector's cathode is connected to the ground, allowing it to be on all the time.

IR voltage reading

The analog pin from A0 of port 3 of the Arduino is set as input and connected across the IR detector to read the voltage from the IR detector.

Key considerations in circuit design

As the IR circuit is attached to the side of the robot, components such as the IR emitter and detector and their resistors are trimmed to ensure a tight fit preventing the components from touching the walls as the robot navigates the maze.

Components of the circuit are also secured to the breadboard using blu-tacks or bridging wires to prevent them from falling off if they scrape on the walls.

IR emitter and IR detectors are placed one breadboard column apart to prevent IR from the emitter from reaching the detector directly while ensuring sufficient reflected IR reaches the detector.

Resistor choice

Emitter

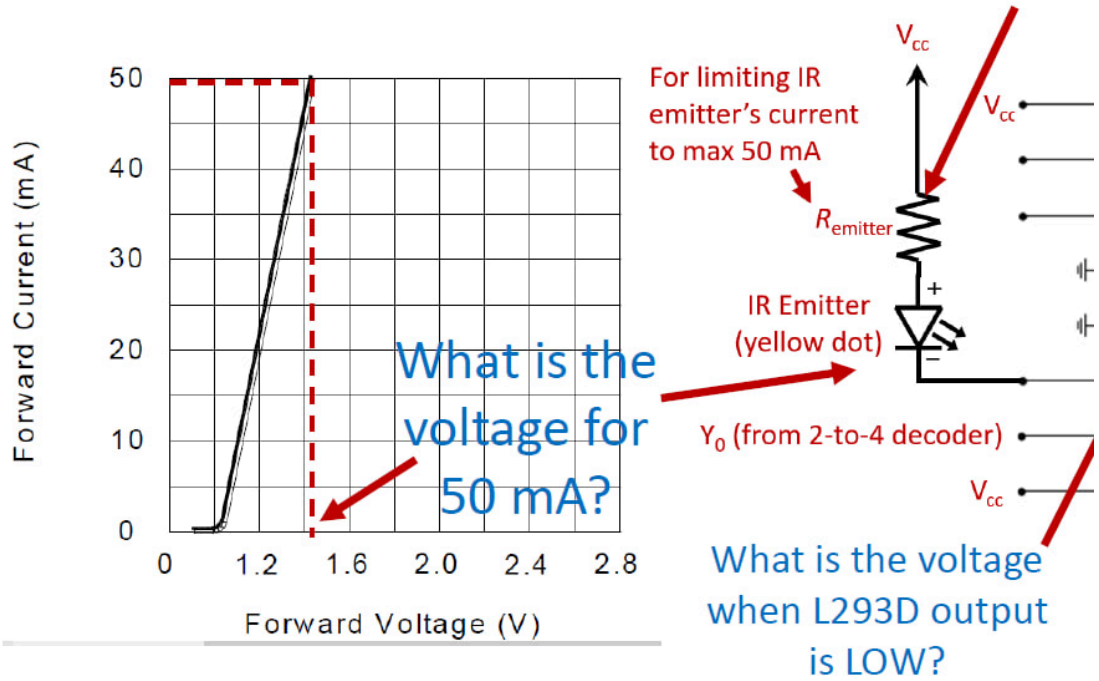


Figure 1.9: A guide from the amazing race PDF

The 120Ω resistor was used to ensure that a current of lower than **50mA** is passed through the IR emitter to avoid damaging it, we used a resistor of 120Ω.

Calculation:
Voltage from Arduino: 5V
Forward voltage of IR at max current*(50mA)*: 1.5V
Voltage across resistor: 5 - 1.5 = 3.5V
Min. resistor value: 3.5 / 0.05 = **70Ω**

Therefore, any resistance above **70Ω** would result in a current lower than 50mA hence, we have proven that 120Ω for our IR circuit's resistor is a reliable choice.

Detector
100k ohms resistor was tested by the team to be responsive enough yet not causing saturation which was therefore used for the IR detector unit.

---

2. Work Distribution

2.1 Table Of Distribution

The project team comprised **4** members and was completed as a team effort. Roles were assigned based on individual **expertise** to ensure efficient task execution. Tasks were **paired** accordingly to ensure thorough cross-checking and minimise errors.

Parts of mBot/Task	Team members mainly in-charge
Ultrasonic Sensor	Jia Wei and Zhen Zhao
Motor Movement (Turning and Adjusting)	Yao Xiang and Jia Jie
Infrared Sensor	Zhen Zhao and Jia Jie
Colour Sensor	Yao Xiang and Jia Wei
Main Code and Algorithm	Yao Xiang and Jia Wei
Circuits Design and Safe-Keeping of the Robot	Zhen Zhao and Jia Jie

Table 2.1: Work Distribution

3. Overall Algorithm

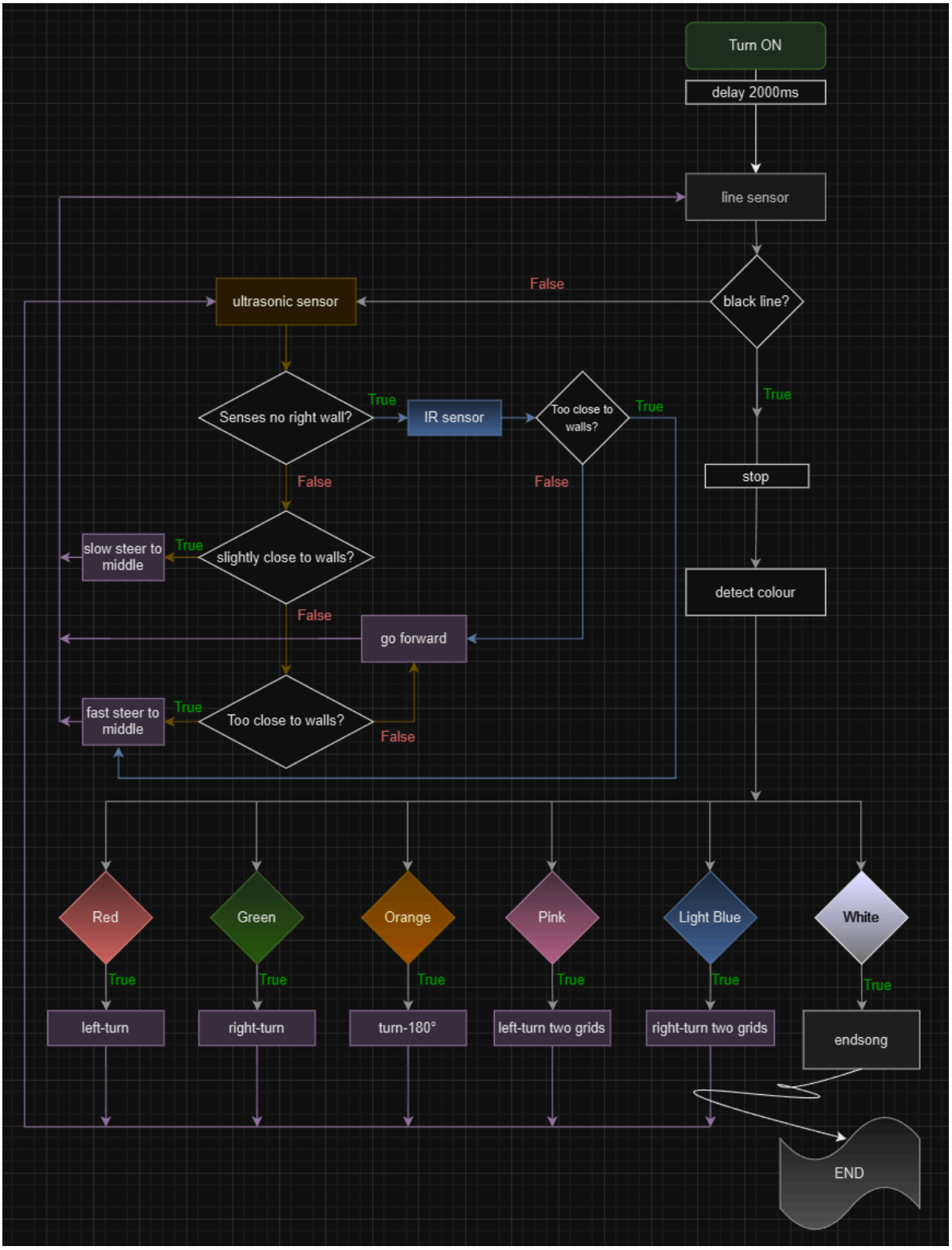


Figure 3.1: Overall Algorithm

Above is a **summary** of what the following sub-headings below will elaborate on.
Delay of 2s in set-up is set to allow ample time for us to prepare for the launch of Mbot.

3.1 Black Line Detection

This is the main loop that is being utilised for the entirety of the maze run:

```
void loop() {
  //checks for black line
  int yes_black = FindBlack.readSensors();
  //less than S1_OUT_S2_OUT means left sensor blacked or right sensor blacked or both blacked; if "none blacked", proceed to adjust_angle();
  if (yes_black < S1_OUT_S2_OUT) {
    stop_motor();
    check_colour();
  }
  else {
    adjust_angle();
  }
}
```

Figure 3.2: Main loop

Black line detection is the first part of the loop that holds the decision to whether colour detection(*check_colour*) should be called upon.

For optimisation and consistency,

```
if (yes_black < S1_OUT_S2_OUT) {
    stop_motor();
    check_colour();
  }
```

Figure 3.2a: A section of main loop on black line detection

We consider black line check as *true* as long as one of the 2 sensors of the black line detector senses black.

3.2 Colour Detection

Our colour detection proceeds after black line is checked:

- RGB LEDs are shone one by one in the order of red, green and lastly blue.

```
/*
 * Shine the Red LED
 */


void shine_red() {
```

```
  digitalWrite(DECODER_B, HIGH);
  digitalWrite(DECODER_A, HIGH);
}
/*
 * Shine the Green LED
 */

void shine_green() {
  digitalWrite(DECODER_B, HIGH);
  digitalWrite(DECODER_A, LOW);
}

/*
 * Shine the Blue LED
 */

void shine_blue() {
  digitalWrite(DECODER_B, LOW);
  digitalWrite(DECODER_A, HIGH);
}

/*
 * Identifies the colour given by the RGB values detected
 * in colour_array.
 * Colours are determined with the boolean analysis method on
 * either R or G or B values from colour_array.
 */
```

Figure 3.3: Code for shining RGB

- The 2-to-4 decoder IC chip is used to control the on-and-off logic of the RGB LEDs. The **red, green and blue** LEDs were connected to pins **Y3, Y2 and Y1** respectively. To activate one of the pins, it will output logic LOW, allowing current to flow through one of the LEDs, lighting it up.

Table 3.1: 2-to-4 Decoder Input/Output Logic

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

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

- The LDR then measures the reflected light from the coloured paper with each light source and outputs a voltage based on its change in resistance from the intensity of light reflected to its surface.

- The combination of these values determines the colour, and the mBot responds accordingly (*Refer to Table 3.2*).

- After completing the specified actions (except for detecting the colour white), the mBot adjusts its angle and the loop repeats.

Table 3.2: Turns

Colour	Action
Red	Left-Turn
Green	Right Turn
Blue	Two successive right-turns in 2 grids
Pink	Two successive left-turns in 2 grids
Orange	180° turn within the same Grid
White	Celebratory tune

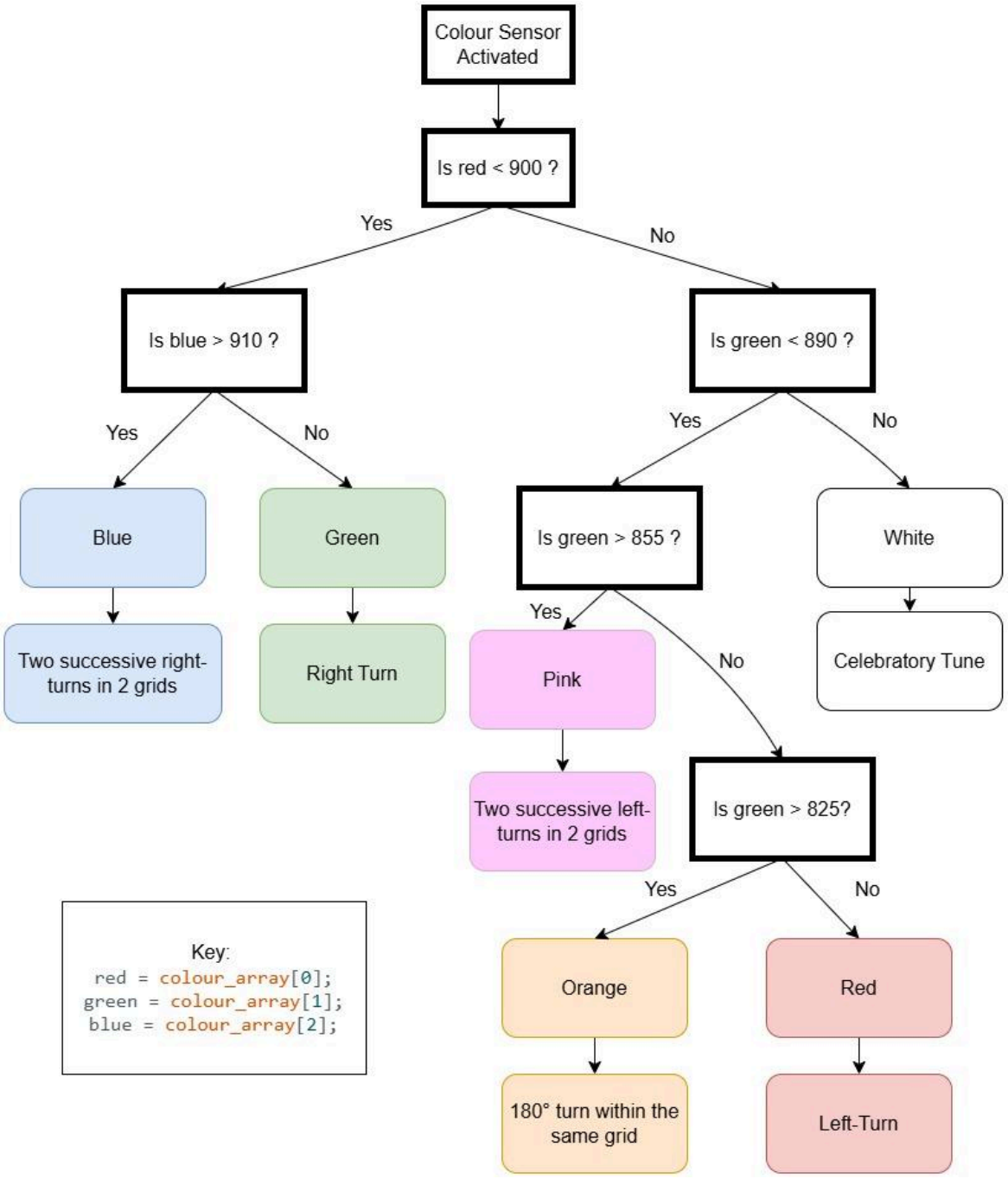


Figure 3.4: Flowchart showing the algorithm for colour detection

Summary of the boolean method utilised. Further description of the rigour and calibration procedure of such method is explored in (*refer to **4.1 colour sensor***).

```
/*
 * Identifies the colour given by the RGB values detected
 * in colour_array.
 * Colours are determined with the boolean analysis method on
```

```
 * either R or G or B values from colour_array.
 */
int identify_colour() {
  int red = colour_array[0];
  int green = colour_array[1];
  int blue = colour_array[2];


 if (red<900) {
    if (blue>910) {
      led.setColor(0, 0, 255);
      led.show();
    //  Serial.println("Blue");
       return BLUE;
    }


    else
    {
      led.setColor(0, 255, 0);
      led.show();
  //   Serial.println("Green");
      return GREEN;
    }
 } else
  {
      if (green < 890)
      {
        if (green > 855)
        {
          led.setColor(255, 192, 203);
          led.show();
          // Serial.println("Pink");
          return PINK;
        }
        else if (green > 825)
          {
            led.setColor(255, 165, 0);
            led.show();
            // Serial.println("Orange");
            return ORANGE;
          } else{
                led.setColor(255, 0, 0);
```

```
                led.show();
                //Serial.println("Red");
                return RED;
               }
      } else
      {
        led.setColor(255, 255, 255);
        led.show();
        //Serial.println("White");
        return WHITE;
       }
  }
}
```

Figure 3.5: Code for colour identification

This set of boolean is drawn by utilising the most stable values(*raw values from 2-to-4 decoder*) amongst (*R, G and B*) as a deciding factor for what colour it is.

- After trial and errors, we decided not to proceed with grayscale(*refer to below:* ***4.1 colour sensor***) and proceeded with using raw LDR values decoded by the 2-4 decoder for colour identification.

- Boolean is used as it has negligible running time, leading to more responsiveness

- We figured that the boolean method is the most stable and accurate as we can ignore at least one colour amongst (*R,G,B*) that gives unstable data while still getting accurate values. (*refer to below:* ***4.1 colour sensor***)

Music: *endsong()* is played after white colour is sensed during the colour detection process; marking the end of the maze run.

```
void endSong() {
#define NOTE_B0  31
#define NOTE_C1  33
#define NOTE_CS1 35
#define NOTE_D1  37
#define NOTE_DS1 39
#define NOTE_E1  41
#define NOTE_F1  44
#define NOTE_FS1 46
#define NOTE_G1  49
#define NOTE_GS1 52
```

```
#define NOTE_A1  55
#define NOTE_AS1 58
#define NOTE_B1  62
#define NOTE_C2  65
#define NOTE_CS2 69
#define NOTE_D2  73
#define NOTE_DS2 78
#define NOTE_E2  82
#define NOTE_F2  87
#define NOTE_FS2 93
#define NOTE_G2  98
#define NOTE_GS2 104
#define NOTE_A2  110
#define NOTE_AS2 117
#define NOTE_B2  123
#define NOTE_C3  131
#define NOTE_CS3 139
#define NOTE_D3  147
#define NOTE_DS3 156
#define NOTE_E3  165
#define NOTE_F3  175
#define NOTE_FS3 185
#define NOTE_G3  196
#define NOTE_GS3 208
#define NOTE_A3  220
#define NOTE_AS3 233
#define NOTE_B3  247
#define NOTE_C4  262
#define NOTE_CS4 277
#define NOTE_D4  294
#define NOTE_DS4 311
#define NOTE_E4  330
#define NOTE_F4  349
#define NOTE_FS4 370
#define NOTE_G4  392
#define NOTE_GS4 415
#define NOTE_A4  440
#define NOTE_AS4 466
#define NOTE_B4  494
#define NOTE_C5  523
#define NOTE_CS5 554
#define NOTE_D5  587
#define NOTE_DS5 622
```

```
#define NOTE_E5  659
#define NOTE_F5  698
#define NOTE_FS5 740
#define NOTE_G5  784
#define NOTE_GS5 831
#define NOTE_A5  880
#define NOTE_AS5 932
#define NOTE_B5  988
#define NOTE_C6  1047
#define NOTE_CS6 1109
#define NOTE_D6  1175
#define NOTE_DS6 1245
#define NOTE_E6  1319
#define NOTE_F6  1397
#define NOTE_FS6 1480
#define NOTE_G6  1568
#define NOTE_GS6 1661
#define NOTE_A6  1760
#define NOTE_AS6 1865
#define NOTE_B6  1976
#define NOTE_C7  2093
#define NOTE_CS7 2217
#define NOTE_D7  2349
#define NOTE_DS7 2489
#define NOTE_E7  2637
#define NOTE_F7  2794
#define NOTE_FS7 2960
#define NOTE_G7  3136
#define NOTE_GS7 3322
#define NOTE_A7  3520
#define NOTE_AS7 3729
#define NOTE_B7  3951
#define NOTE_C8  4186
#define NOTE_CS8 4435
#define NOTE_D8  4699
#define NOTE_DS8 4978
#define REST      0

  int tempo = 114;

  int melody[] = {

    NOTE_A4, 16, NOTE_B4, 16, NOTE_D5, 16, NOTE_B4, 16,
```

```
    NOTE_FS5, -8, NOTE_FS5, -8, NOTE_E5, -4, NOTE_A4, 16, NOTE_B4, 16, NOTE_D5, 16,
NOTE_B4, 16,
    NOTE_A5, 4, NOTE_CS5, 8, NOTE_D5, -8, NOTE_CS5, 16, NOTE_B4, 8, NOTE_A4, 16, NOTE_B4,
16, NOTE_D5, 16, NOTE_B4, 16,

    NOTE_D5, 4, NOTE_E5, 8, NOTE_CS5, -8, NOTE_B4, 16, NOTE_A4, 4, NOTE_A4, 8, //23
    NOTE_E5, 4, NOTE_D5, 2, REST, 4,

  };

  int notes = sizeof(melody) / sizeof(melody[0]) / 2;

  int wholenote = (60000 * 4) / tempo;

  int divider = 0, noteDuration = 0;
  for (int thisNote = 0; thisNote < notes * 2; thisNote = thisNote + 2) {
    divider = melody[thisNote + 1];
    if (divider > 0) {
      noteDuration = (wholenote) / divider;
    } else if (divider < 0) {
      noteDuration = (wholenote) / abs(divider);
      noteDuration *= 1.5;
    }
    buzzer.tone(melody[thisNote], noteDuration);
  }
}
```

Figure 3.6: End Song

Once the colour is identified, movement will commence.

```
void action(int colour) {
  if (colour == RED) {
    left_turn(TURNING_TIME);
    return;
  }

  if (colour == GREEN) {
    right_turn(TURNING_TIME);
    return;
  }

  if (colour == ORANGE) {
    rotate_back(TURNING_TIME);
    return;
  }

  if (colour == PINK) {
    left_turn_2_grid(TURNING_TIME+12, FORWARD_TIME);
    return;
  }

  if (colour == BLUE) {
    right_turn_2_grid(TURNING_TIME, FORWARD_TIME);
    return;
  }

  if (colour == WHITE) {
    endSong();
    return;
  }
}
```

Figure 3.7: Colour Detection for movements

3.3 Movements

```
void stop_motor(){
  leftMotor.stop();
```

```
  rightMotor.stop();
}
/*
 * Make the robot go forward.
 */
void go_forward() {
  leftMotor.run(-motorSpeed); //left always negative for forward
  rightMotor.run(motorSpeed);
}
/*
 * Moves the robot backwards by a given time(delay).
 * Stop the motors afterwards.
 */
void reverse(int time) {
  leftMotor.run(motorSpeed);
  rightMotor.run(-motorSpeed);
  delay(time);
  stop_motor();
}
/*
 * Turn the robot right by a given time(delay).
 * Stop the motors afterwards.
 */
void right_turn(int time) {
  leftMotor.run(-motorSpeed);
  rightMotor.run(-motorSpeed);
  delay(time);
  stop_motor();
}
/*
 * Turns the robot left by a given time(delay).
 * Stop the motors afterwards.
 */
void left_turn(int time) {
  leftMotor.run(motorSpeed);
  rightMotor.run(motorSpeed);
  delay(time);
  stop_motor();
}
/*
 * Turns the robot right by a given time(turn_time),
 * moves it forward by a given time(fwd_time),
```

```
 * do a stop to break motion
 * and then turn it right again by a given time(turn_time).
 */
void right_turn_2_grid(int turn_time, int fwd_time) {
  right_turn(turn_time);
  go_forward();
  delay(fwd_time);
  stop_motor();
  right_turn(turn_time);
}
/*
 * Turns the robot left by a given time(turn_time),
 * moves it forward by a given time(fwd_time),
 * do a stop to break motion
 * and then turn it left again by a given time(turn_time).
 */
void left_turn_2_grid(int turn_time, int fwd_time) {
  left_turn(turn_time);
  go_forward();
  delay(fwd_time);
  stop_motor();
  left_turn(turn_time);
}
/*
 * Turns the robot to the right by a given time(tuned with '/2-20' for eg so that the
Mbot can turn exactly 90 degrees),
 * move back for 100ms so that Mbot can adjust to tight angles
 * Turn the robot right again by 90 degrees, this time a little less as our Mbot appears
to require that to turn exactly 90 degrees
 * Stop motor to clear previous turn command; wait for next command
 */
void rotate_back(int time) {
  right_turn(time / 2 - 20);
  reverse(100);
  right_turn(time / 2 - 25);
  right_turn(time);
  stop_motor();
}
```

Figure 3.8: Movement code w.r.t colours sensed

Our movement code acts upon the colour detected, executing respective turns with *TURNING_TIME* kept constant at 300ms (*any further changes are done by using simple arithmetic on TURNING_TIME for efficiency of calibration*).

```
right_turn(time / 2 - 20);
```

- *rotate_back* function is designed to: *turn→ move back→ turn again* such that we can navigate tight corners and reduce the risk of bumping into walls

- The rest of the movement functions are kept standard with only delays and *TURNING_TIME* being modified to navigate the maze without issues. Values for *TURNING_TIME* are decided through experimentation and fine-tuning. No maths is implemented because it is trivial.

- Delays are kept minimal but sufficient so that we do not dampen its reaction time but also allow sufficient time to move.

3.4 Angle Adjustments

At the end of every main loop, distance from the middle point of the maze will be checked for deviation from the middle of the path in the maze; the angle of forward movement would thus be adjusted via steering actions. This is to ensure proper orientation of the robot.

```
void adjust_angle() {
  float distance = distance_left();
  float distance_r = distance_right();



  if (distance > 15.5 || distance < 0.0) {
    //fast steer to left
    if(distance_r > 800){
      leftMotor.run(0.3 * -motorSpeed);
      rightMotor.run(motorSpeed);
      return;
    //fast steer to right
    }else if(distance_r<=500){
      leftMotor.run(-motorSpeed);
      rightMotor.run(0.3 * motorSpeed);
      return;
    }
    go_forward();
    return;
```

```
  }
  //using 7cm as the mid point
  //fast steer to right
  if (distance < 3) {
    leftMotor.run(-motorSpeed);
    rightMotor.run(0.3*motorSpeed);
    return;
  }
  //slow steer to left
  if (distance > 9) {
    leftMotor.run(-0.7*motorSpeed);
    rightMotor.run(motorSpeed);
    return;
  }
  //fast steer to left
  if (distance > 11) {
    leftMotor.run(-0.3*motorSpeed);
    rightMotor.run(motorSpeed);
    return;
  }
  //slow steer to the right
  if (distance < 5) {
    leftMotor.run(-motorSpeed);
    rightMotor.run(0.7*motorSpeed);
    return;
  }
  //if position correct, proceed forward
  go_forward();
}
```

Figure 3.9: General Code for angle adjustment

```
float distance_right() {
  analogWrite(DECODER_A, LOW);
  analogWrite(DECODER_B, LOW);
  float reading = analogRead(IR_RECEIVER_PIN) ;
  return reading;
}
```

Figure 3.9a: Code for distance_right measured by IR sensor

```
float distance_left() {
```

```
  digitalWrite(ULTRASONIC_PIN, LOW);
  delayMicroseconds(2);
  digitalWrite(ULTRASONIC_PIN, HIGH);
  delayMicroseconds(100);
  digitalWrite(ULTRASONIC_PIN, LOW);

  pinMode(ULTRASONIC_PIN, INPUT);
  long duration = pulseIn(ULTRASONIC_PIN, HIGH, 3000); //3000s is the
max timeout duration
  float distance = (((float)SPEED_OF_SOUND * (float)duration / 10000.0)
/ 2.0) - 3.5;
    //Serial.println(duration);
    //Serial.print("distance: ");
    //Serial.println(distance);
    //Serial.println("cm");
   delay(10);
//short delay to smoothen the adjusting frequency while also
maintaining high levels of responsiveness
  return distance;
}
```

Figure 3.9b: Code for distance_left measured by Ultrasonic Sensor

Note: *All distance values are based on calibrated data from Ultrasonic sensor calibration (**4.2 ultrasonic sensor**). **They deviate from actual distance.***

Our angle adjustment code primarily relies on data from the ultrasonic sensor.

Steer

As shown in figure 3.9, we split angle adjustment into **2** steering motions: **slow steer** and **fast steer** depending on the following positioning of the Mbot:

- When **>2cm && <4cm** to the **left** or **right** from **measured middle position** within the maze
  → Accelerate to the **right** or **left** respectively using **slow steer**
- When **>4cm** to the **left** or **right** of the **measured middle position** within the maze
  → Accelerate to the **right** or **left** respectively using **fast steer**

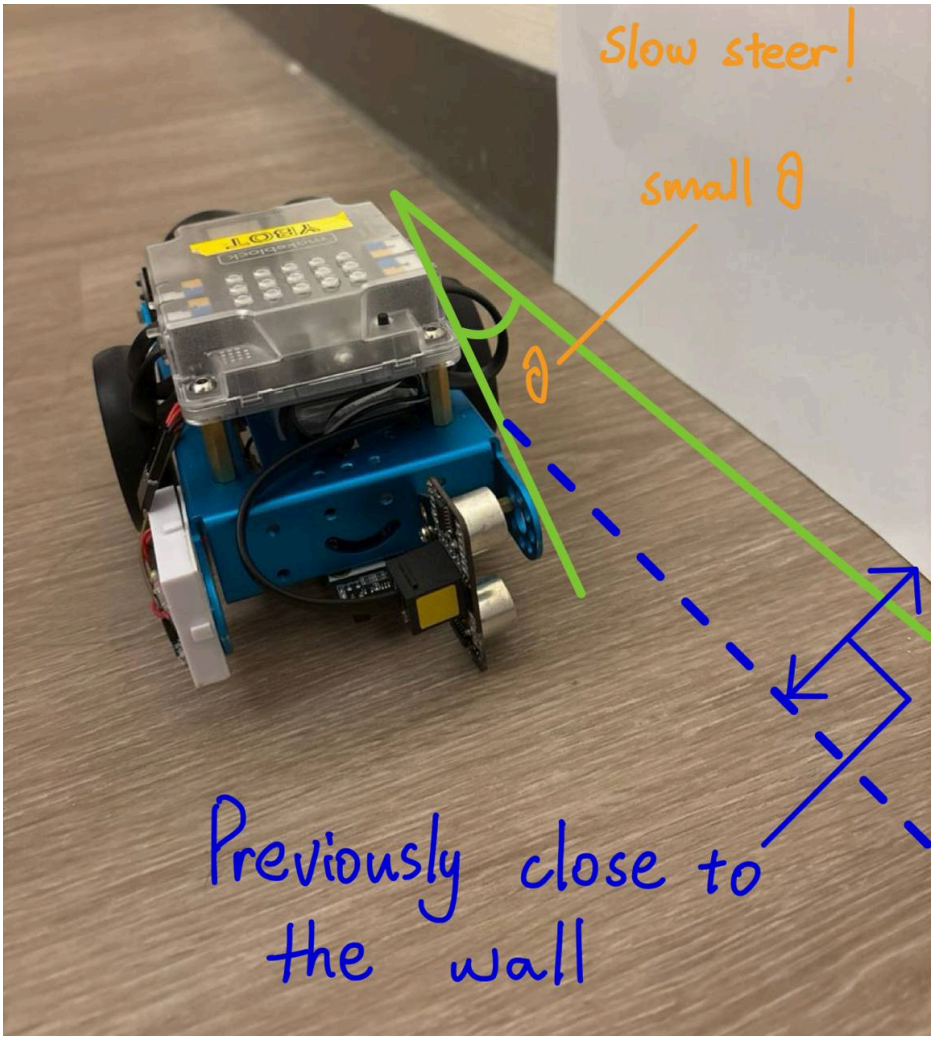
Note:

*- **Middle point** is pre-measured and walls are estimated to be **7cm** from mid-point due to the offset caused by the distance of ultrasonic sensor/ ir sensor from the the right wall/left wall respectively as*

well as taking into account the NULL detection range (too close to sensor). (such data are measured off trial-and-error)

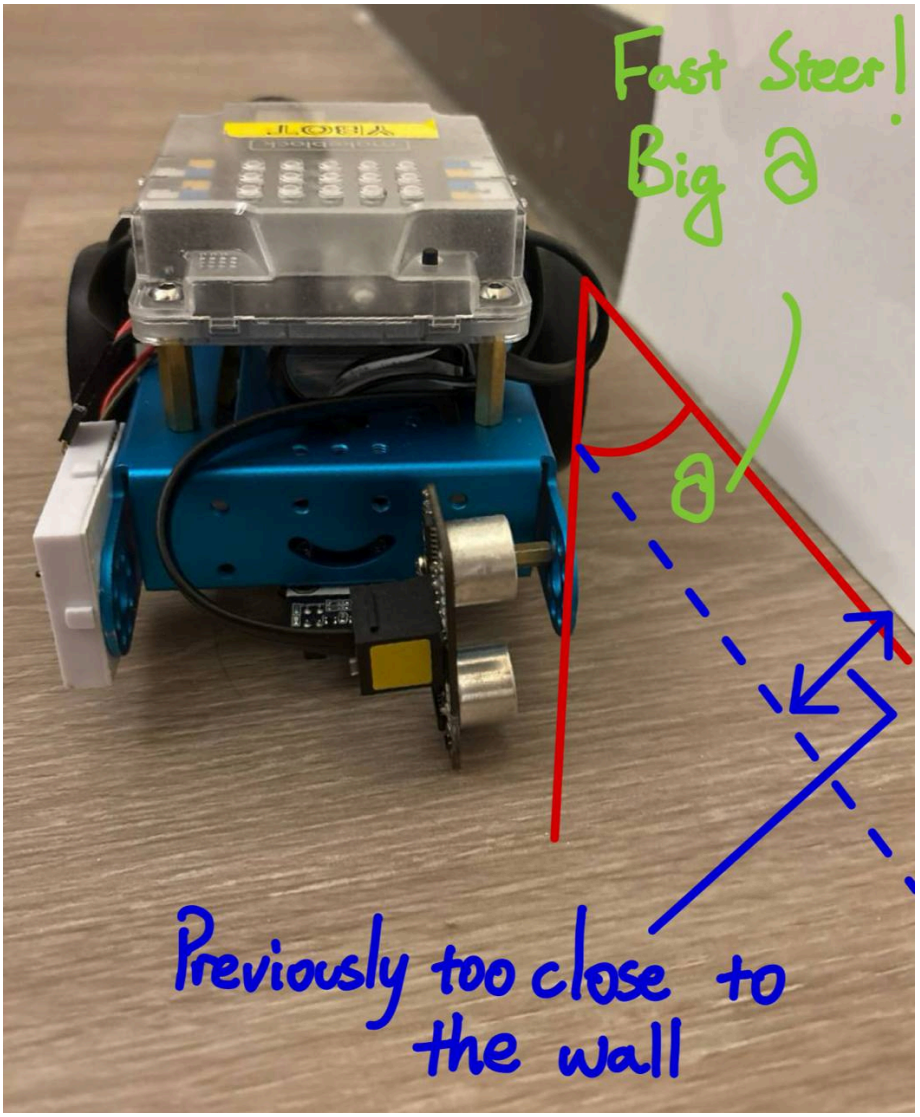
*- **Steer**: one wheel slow, other wheel fast*

Slow Steer



Slow steer is utilised such that Mbot can slowly adjust back to proper position without abrupt movements.

Fast Steer



Fast steer is utilised when Mbot is very close to the wall, thus requiring a quick adjustment to prevent Mbot from bumping against the wall

Forward movement proceeds if the position of Mbot is within the acceptable range. (+-2cm from middle)

```
go_forward();
```

Backup adjustments

We relied heavily on the ultrasonic sensor because of its accuracy and consistency of data as compared to the IR sensor. As such, we only use IR sensor in the case that ultrasonic sensor cannot sense any wall (no left wall):

```
if (distance > 15.5 || distance < 0.0) {
    if(distance_r > 800)
    {
```

```
      leftMotor.run(0.3 * -motorSpeed);
      rightMotor.run(motorSpeed);
      return;
    }else if(distance_r <= 500){
      leftMotor.run(-motorSpeed);
      rightMotor.run(0.3 * motorSpeed);
      return;
    }
    go_forward();
    return;
  }
```

Figure 3.9c: An excerpt from *Figure 3.9*

Note: distance *is referred to as distance from left wall*

In such situations, we proceed with only 1 steering motion as the IR sensor is not as accurate.

- When more than 4cm away from the middle position, which is <3cm or >11cm from the right wall

  (3cm and 11cm, as well as every distance used in code are calibrated distance that is acknowledged by the ultrasonic sensor, though it differs from the actual distance; it is consistent)

  → Accelerate to the left or right respectively using **fast steer**

(according to calibrated data, 800 and 500 corresponds to Mbot being 3cm and 11cm from the right wall respectively.)

Forward movement proceeds only if the position of Mbot is within the acceptable range. (*within 4cm from middle position as only fast steer is utilised*)

```
go_forward();
    return;
```

---

4. Calibration and Improvements

4.1 Colour Sensor

Data collection method 1

Our first approach was to perform a two-step calibration process to optimise colour detection.

1. We recorded RGB values for white paper (maximum reflectance) and for black paper (minimum reflectance) respectively.
   - The difference between the RGB values of black and white papers, known as the grey difference, are calculated and stored in an array as a reference range.

2. We then measured RGB values for all target colours.
    - The range of each RGB values were converted to 255 through the following formula:

$$\text{Normalized Value} = \frac{\text{RGB Value} - \text{Black RGB}}{\text{Grey Difference}} \times 255$$

3. Lastly, **5 sets of data** are recorded for each target colour then they are averaged to obtain more reliable data that accounts for spread and deviation

Table 4.1: Data Sample with Grey Difference

Color	Num	W R	W G	W B	B R	B G	B B	R	G	B
Green	0	**967**	**963**	**981**	**908**	**871**	**890**	**154**	**208**	**148**
Green	1	**959**	**961**	**981**	**905**	**871**	**889**	**174**	**215**	**149**
Green	2	961	962	981	905	870	890	162	212	148
Green	3	**960**	962	981	905	871	890	159	212	148
Green	4	961	962	980	909	873	891	161	212	148
AVG								162	211.8	148.2
Blue	0	**968**	**963**	**981**	**903**	**872**	**891**	**141**	**207**	**240**
Blue	1	**960**	**963**	**981**	**897**	**870**	**890**	**170**	**208**	**240**
Blue	2	961	962	981	908	872	891	168	209	240
Blue	3	960	962	981	904	873	892	159	209	240
Blue	4	956	962	981	896	870	891	186	210	240
AVG								164.8	208.6	240
Red	0	**969**	**963**	**981**	**901**	**872**	**892**	**255**	**94**	**85**
Red	1	**959**	**962**	**981**	**908**	**872**	**891**	**305**	**93**	**87**
Red	2	959	962	981	901	871	891	280	92	86
Red	3	957	961	981	901	871	891	306	96	86
Red	4	960	962	981	900	872	891	285	93	87
AVG								286.2	93.6	86.2
Pink	0	**964**	**962**	**981**	**901**	**872**	**892**	**275**	**226**	**229**
Pink	1	**959**	**963**	**981**	**902**	**874**	**893**	**293**	**222**	**228**
Pink	2	957	962	981	910	873	892	330	224	229
Pink	3	962	962	981	902	872	892	289	225	229
Pink	4	958	961	980	908	872	892	326	229	231
AVG								302.6	225.2	229.2
Orange	0	**968**	**962**	**980**	**911**	**875**	**893**	**259**	**152**	**96**
Orange	1	**958**	**962**	**981**	**909**	**873**	**891**	**322**	**152**	**101**
Orange	2	961	963	981	908	873	892	299	153	99
Orange	3	960	962	981	907	873	892	303	154	100
Orange	4	965	963	981	910	874	893	268	151	98
AVG								290.2	152.4	98.8
White	0	967	962	981	906	874	892	269	269	258
White	1	960	962	980	898	874	891	296	269	261
White	2	959	961	980	911	876	892	313	273	261
White	3	959	962	980	907	874	892	308	269	260
White	4	959	963	981	908	874	892	310	269	258
								299.2	269.8	259.6

Colour identification method

(Euclidean vs Boolean)

Euclidean

We initially utilised the concept of Euclidean distance to differentiate between each target colours.

This method calculates the straight-line distance between two colours in a 3D RGB colour space. Each colour is treated as a point with coordinates (R, G, B).

- The function will take in the RGB values of each target colour and get the distance between the **RGB colour** and the **target colour.**

- The returned value represents how far apart these 2 colours are in 3D space. (*closer means more similar*)

```
float calculateDistance(int inputColor[3], colour_array[3]) {
    float distance = 0;
    for (int i = 0; i < 3; i++) {
        distance += pow(inputColor[i] - colour_array[i], 2);
    }
    return sqrt(distance);
}
```

Figure 4.1: Prototype Euclidean Distance Function

However, the LDR reading for each **target** colour showed big signs of inconsistency in data when shone with red RGB(as shown in table 4.1), resulting in inaccuracies in identifying the correct target colour due to the method's dependence on all components (R, G and B).

Additionally, due to the involvement of squaring and square-rooting in the code, it resulted in slow speed in identifying each colour.

Thus, we decided to employ another method of colour identification.

Boolean

We settled on using the boolean method in which we compare the R, G and B components individually with the range obtained from the data sample.

- We separated the colours by first considering the **greatest** determining colour component that can distinctly separate the colours in question from one another. (*red ==220 in this case; Table 4.1*)

- Then, we obtained the **midpoint** between one of the 3 RGB components per colour, which would be used as a deciding factor that differentiates the colours from each other.

```
int identify_colour() {
  int red = colour_array[0];
  int green = colour_array[1];
  int blue = colour_array[2];



  if (red > 220) {
    if (green < 130) {
      Serial.println("Red");
      return RED;
    }
    else if (blue > 160 && blue < 250) {
      Serial.println("Pink");
      return PINK;
    }
    else if (blue < 160) {
      Serial.println("Orange");
      return ORANGE;
    }
    else {
      Serial.println("White");
      return WHITE;
    }
  }
  else {
    if (blue > 210){
      Serial.println("Blue");
      return BLUE;
    }
    else
    {
      Serial.println("Green");
      return GREEN;
    }
  }
}
```

Figure 4.2: Prototype Boolean Colour Identification Function

This method has proved us greater accuracy as we depended less on the R component which gave varying results. Since B and G components were more consistent throughout multiple trials of readings, we attained greater success in identifying the target colours accurately and consistently.

Drawbacks

During test runs in the maze, we encountered inaccuracies of the robot in identifying the correct colours consistently. This was caused by deviating LDR readings of the target colours when we troubleshooted using the serial monitor. We attributed this variance in LDR readings to the **amplification of inaccuracies** when using the grayscale normalisation.

- When the raw values of each RGB for the target colours are scaled to 255, any small error in the measured black or white RGB values is proportionally applied to all scaled values.

- For example, a slight miscalculation in the grey difference could make small inaccuracies in the raw data appearing larger after scaling.

- To counteract this, we decided to use raw RGB values for each target colour in our boolean function to reduce the risk of discrepancies amplified by the greyscale normalisation.

Final data collection method with implementation of boolean analysis

This brings us to our second and final data collection method.

- Instead of stamping the Mbot onto each of the coloured papers, we get the Mbot to:
    1. Run onto the paper
    2. Check colour
    3. Print to serial monitor
    4. Reverse
    5. Repeat above procedure

As such we can simulate the real-time colour detection procedure while recording down the LDR values for (R,G,B) when shone onto each of the coloured papers.

Below is the procedure of calibration:

1. Mbot will run forward on default
2. Upon black line detection, *check_colour()* function will commence:

```
void check_colour() {
```

```
  for (int i = 0; i < 3; i++)
  {
    if (i == 0)
    {
      shine_red();
    }


    else if (i == 1)
    {
      shine_green();
    }


    else
    {
      shine_blue();
    }

    int total = 0;
    int flag = 5;


    delay(200);


    for (int j = 0; j < 5; j++)
    {
      long reading = (analogRead(LDR_PIN)); /*- black_array[i]) * 255) / (white_array[i] -
black_array[i]);*/
      //previously used grey diff, now no more
      //Serial.println(reading);


      if (reading <= 0)
      {
        flag -= 1;
      }


      if (reading > 0)
      {
        total += reading;
      }
```

```
    }
   //avg LDR value per R,G,B = total/flag
    colour_array[i] = total / flag;

   Serial.println(colour_array[i]);



    digitalWrite(DECODER_B, LOW);
    digitalWrite(DECODER_A, LOW);



    delay(100);
  }
   // Proceeds to send colour array to identify the colour
    int colour = identify_colour();
   // action will be taken depending on what the colour is
    action(colour);
}
```

Figure 4.3: Code for checking the raw LDR values (proceeds to call *identify_colour()*)

Similar to before, arduino will take an average of **5** LDR readings per R,G and B. Then, it will **print** to serial monitor each of the RGB values for the coloured paper used.

After detecting the colours, Mbot will proceed to *identify_colour()* where the main point is to shine the detected colour for us to verify its accuracy.

```
if (blue>910) {
     led.setColor(0, 0, 255);
     led.show();
   //  Serial.println("Blue");
      return BLUE;
    }
```

Figure 4.4: an excerpt from *Figure 3.5*, (*shines blue when detected*)

3. Then, it will proceed with *action()* where we commented out every colour-response actions intended and only allow the Mbot to *reverse()*, stop for a while, *go_forward()* again to carry on with collection of the next set of data.

```
void action(int colour) {
```

```
  if(colour<10){
    reverse(200);
    stop_motor();
    delay(200);
    go_foward();
  }
  /*
  if (colour == RED) {
    left_turn(TURNING_TIME);
    return;
  }

  if (colour == GREEN) {
    right_turn(TURNING_TIME);
    return;
  }

  if (colour == ORANGE) {
    rotate_back(TURNING_TIME);
    return;
  }

  if (colour == PINK) {
    left_turn_2_grid(TURNING_TIME+12, FORWARD_TIME);
    return;
  }

  if (colour == BLUE) {
    right_turn_2_grid(TURNING_TIME, FORWARD_TIME);
    return;
  }

  if (colour == WHITE) {
    endSong();
    return;
  }
  */
}
```

Figure 4.5: Modified movement code(*for calibration)*

As described above, data is collected via the **serial monitor at 9600 baud**.

We allowed the Mbot to check colour repeatedly by itself using the loop of movements. However, since the repeated data are almost constant, we only took down 5 of the data where the Mbot faced the most deviation from the mean value per coloured paper to observe data spread.

- Such a method is used because the LDR faces shifts in data values after some time but is very consistent in the short period of calibration period. Hence, we chose to take the data of the most spread from mean such that our boolean method can account for data deviation.

- Since we do not use grey difference, we do not need to check white and black paper before checking the coloured paper. Hence, such movement looping is the most **efficient** method for calibration. It also prevents the **abnormal data deviation** due to startup deviation as the Mbot will continuously be moving (*and never turned off*) during the whole process of data collection.

- Not just that, it's intended to be conditioned exactly the same as the scenario of the maze run where we calibrated the Mbot on the maze itself by getting it to run back and forth onto each and every of the coloured papers.

Table 4.2: Final data collected after calibration before graded run(*before value tuning)*

Colour	R	G	B

Green	0	894	874	902
Green	1	894	874	902
Green	2	894	874	902
Green	3	895	876	904
Green	4	895	876	904
		894.4	**874.8**	**902.8**
Blue	0	893	871	927
Blue	1	894	871	927
Blue	2	893	871	927
Blue	3	890	866	922
Blue	4	892	871	927
		892.4	**870**	**926**
Red	0	921	812	890
Red	1	921	811	890
Red	2	921	815	892
Red	3	920	812	891
Red	4	921	812	890
		920.8	**812.4**	**890.6**
Pink	0	925	878	922
Pink	1	923	874	921
Pink	2	923	878	822
Pink	3	924	877	921
Pink	4	923	875	921
		923.6	**876.4**	**901.4**
Orange	0	925	842	895
Orange	1	925	841	894
Orange	2	926	843	895
Orange	3	924	841	895
Orange	4	925	841	895
		925	**841.6**	**894.8**
White	0	926	907	936
White	1	926	908	937
White	2	927	908	937
White	3	926	907	936
White	4	927	908	937
		926.4	**907.6**	**936.6**

As shown above, all data collected faced much lower deviation without grayscaling.

Furthermore, data is consistent despite the attempt to take data with **maximum deviations from mean**, this is an indicator that our collection method is a reliable approach to proceed with colour identification.(*boolean methodology*)

```
int identify_colour() {
  int red = colour_array[0];
  int green = colour_array[1];
  int blue = colour_array[2];

 if (red<907) {
    if (blue>913) {
      led.setColor(0, 0, 255);
      led.show();
    //  Serial.println("Blue");
       return BLUE;
    }

    else
    {
      led.setColor(0, 255, 0);
      led.show();
  //   Serial.println("Green");
      return GREEN;
    }
 }
   else
   {
    if (green < 892)
    {
      if (green > 859)
      {
        led.setColor(255, 192, 203);
      led.show();
    //  Serial.println("Pink");
        return PINK;
      }

      else if (green > 828)
      {
        led.setColor(255, 165, 0);
        led.show();
        // Serial.println("Orange");
      return ORANGE;
      }
```

```
      else
      {
        led.setColor(255, 0, 0);
    led.show();
    //    Serial.println("Red");
    return RED;
      }
    }

    else
    {
      led.setColor(255, 255, 255);
        led.show();
    //  Serial.println("White");
        return WHITE;
    }
  }
}
```

Figure 4.6: Boolean Code (*w.r.t the data shown above in Table 4.2*)

Further value tuning was applied after this code as data has shifted after some time has passed.

Outcome

Immediate results returned, needless to say, putting the calibrated Mbot into the maze with the well-tuned boolean successfully completes the maze. (Though after some time, data will deviate again and more calibration is needed such will be explored in **5.2 Colour Detection Inaccuracies**)

4.2 Ultrasonic Sensor (placed on the left)

Key considerations

Before calibration, we took **2** key characteristics of the ultrasonic sensor into account:

1. **<u>Sensor's offset from left and right wall</u>**

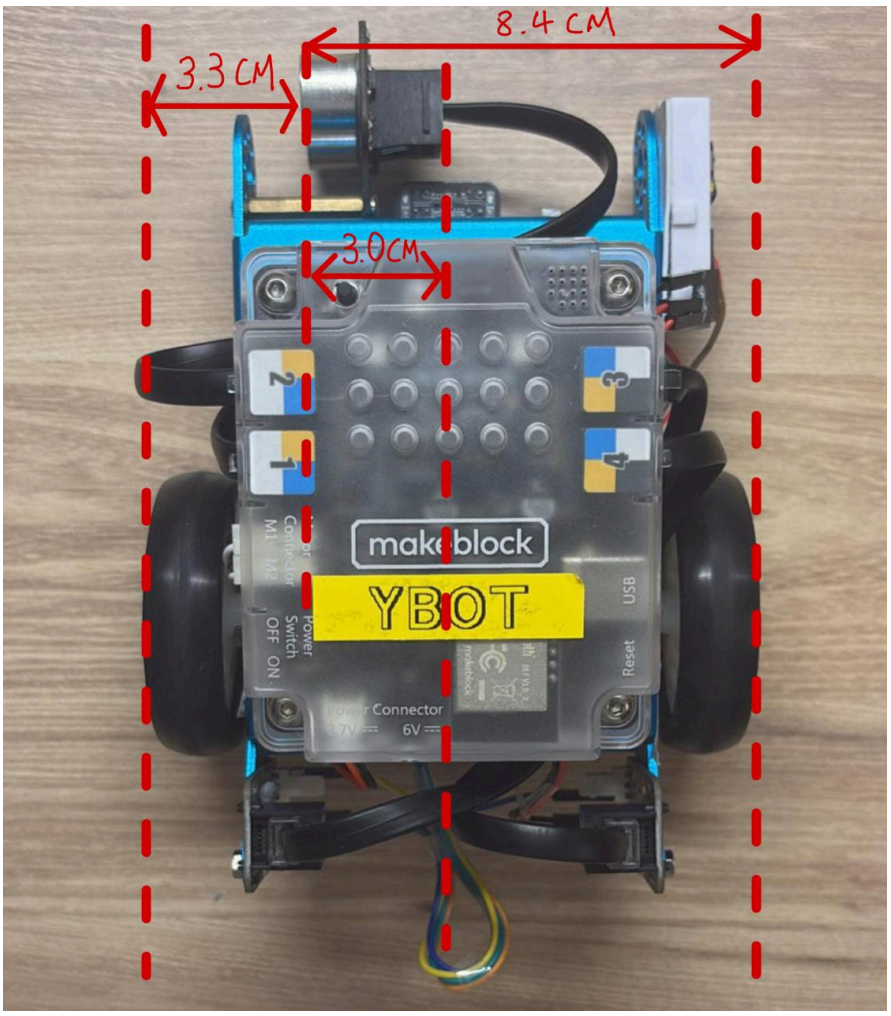


Figure 4.7: Mbot's dimensions

Measured width of maze (between any 2 walls)= 31cm

For ultrasonic sensor to fully prevent Mbot from touching the left or the right wall, we kept in mind that the Mbot has to start steering to the middle immediately before ultrasonic sensor feedbacks *close to but above* 3.3cm from the left wall or *close to but below* 22.6cm (*31-8.4=22.6*) from the right wall

2. **Sensor's null-detection range**

After experiments, the null detection range is found to be within 3cm proximity ultrasonic sensor.

We fully understand that the sensor has a null range but this will not affect us because we will be steering before 3.3cm anyways.

Calibrations

According to our pre-calculations, the Mbot should have a middle position of 15.5cm from left wall where the ultrasonic sensor should sense 12.5cm(*15.5-3=12.5*) and hence decided on a +- 2cm tolerance range from this position; slow steering when breaching the **2cm boundary**, fast steering when breaching the **4cm boundary** all w.r.t the middle position.

However, through experiments, we realised that the ultrasonic sensor's detection range, though consistent, **deviates from measured values**. Hence, we calibrated again and came up with the following set of values for our angle adjustment code with relation to sensor's data; all while maintaining the **boundary logic** above.

Mainly, the midpoint detected by the ultrasonic sensor is given as **7cm** and the maximum distance before touching the right wall is **15.5cm** as per sensor feedback.

Thus with the measured info, here is a summary of the **distance-range** we concluded upon.

distance<3	**distance<5 and distance>=3**	**distance >=5 or distance<=9**	**distance>9 and distance<=11**	**distance>11**	**distance>15.5 or no distance detected**
Fast steer right	Slow steer right	Go forward	Slow steer left	Fast steer left	No wall on left, rely on IR

Delay management

With regards to the delay resulting in slow responsiveness of the Mbot, the delays in our ultrasonic sensor heavily affected the responsiveness part of our runs till we realised it.

Method of delay calibration: **trial and error** (*elaborated in **5.1 slow responsiveness of line detector***)

4.3 IR Proximity Sensor (placed on the right)

Proximity accuracy

The IR sensor exhibits **inaccuracies** for **long ranges of detection**. A graph of distance in cm against the value of the IR sensor received from *AnalogRead()* was obtained to determine the threshold distance from the right wall in which the robot should adjust its trajectory back to the original straight path.

Table 4.3: Table with data for IR Sensor

Distance	IR Reading
0	1020
1	950
2	876
3	790
4	773
5	650
6	644
7	628
8	601
9	579
10	537
11	502
12	501
13	502

As shown in table 4.3, due to the nature of the IR sensor values being **non-linear** to proximity to the right wall, we decided that utilising any more than 2 values to set a range for fast steer is inconsistent and hazardous to Mbot's movement.

This led to us going with **800** and **500** as the two values to fast steer left and right respectively towards the middle during the short period where ultrasonic cannot sense the wall.

- **500** is selected as the data has negligible changes starting from 11 cm onwards hence any data beyond 11cm cannot be relied on.
  11cm is also just nice decently close to the left wall.

- **800** is selected as the difference between each value starting from 3cm to 0cm is very distinct with a difference of ~100, thus, through multiple tests, we decided that this value is consistent enough to act as our threshold for steering action.

Complementing ultrasonic sensor

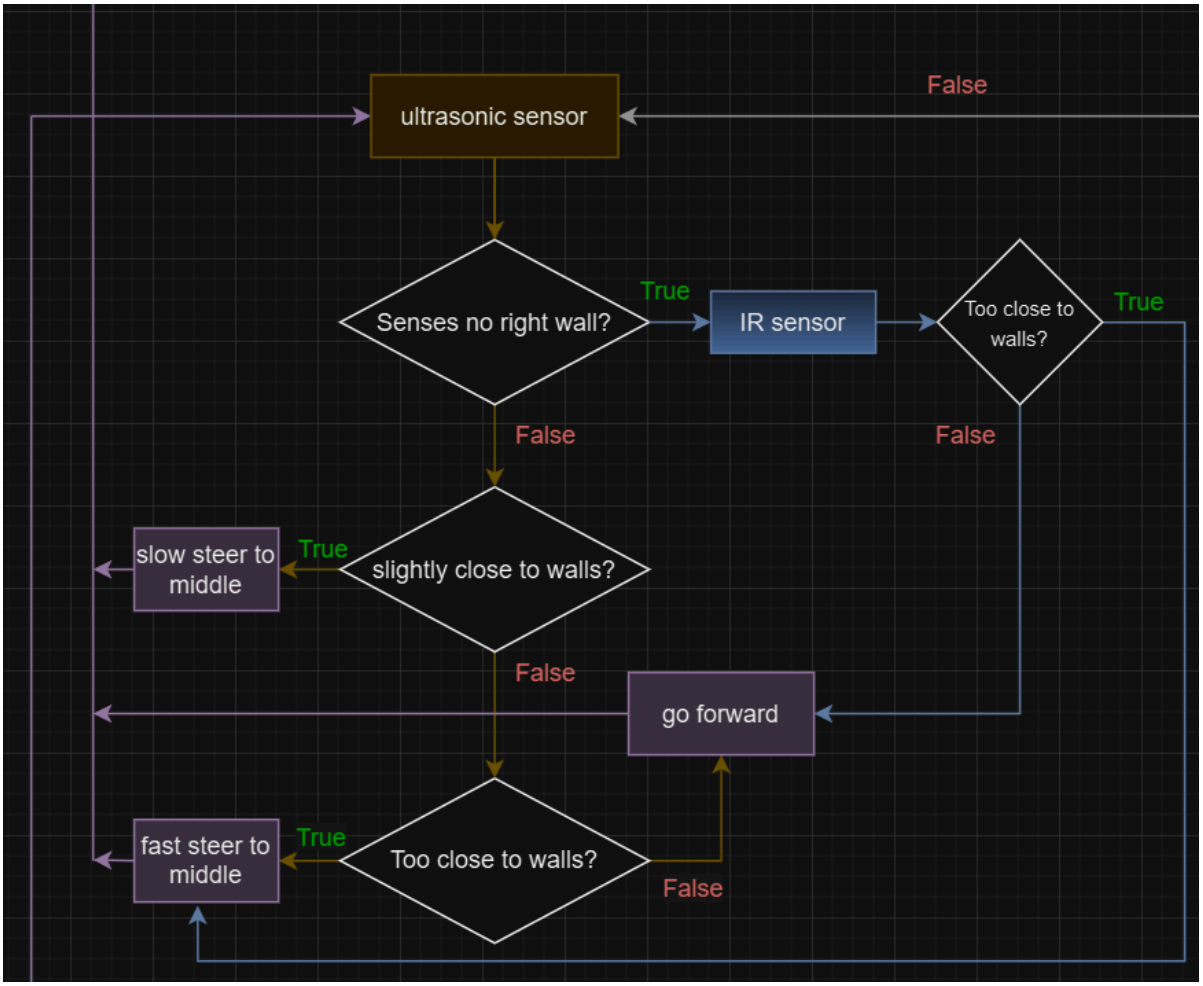


Figure 4.8: An excerpt of *Figure 3.1: Overall Algorithm*

As shown in the flowchart diagram in Figure 4.8:

- Due to the **low accuracy** of IR for large ranges of detection, we decided to only settle with the IR sensor **if and only if** the ultrasonic sensor does not detect any wall (>15.5cm according to ultrasonic sensor).

- Only **2 threshold distances and corresponding value of raw IR sensor** (500 and 800) are utilised as those are the only points of **consistency**, hence its logical for us to only rely on IR to steer when it is **too close** to right wall/left wall (<3cm or >11cm from right wall after accounting for deviations and offsets).

5. Overcoming Difficulties

5.1 Slow Responsiveness of Line Detector

Difficulty:

The Mbot frequently overshot the black lines and hit the wall ahead.

```
float distance_left() {
  digitalWrite(ULTRASONIC_PIN, LOW);
  delayMicroseconds(2);
  digitalWrite(ULTRASONIC_PIN, HIGH);
  delayMicroseconds(100);
  digitalWrite(ULTRASONIC_PIN, LOW);
  pinMode(ULTRASONIC_PIN, INPUT);
  long duration = pulseIn(ULTRASONIC_PIN, HIGH, 3000); //3000s is the max timeout duration
  float distance = (((float)SPEED_OF_SOUND * (float)duration / 10000.0) / 2.0) - 3.5;
    //Serial.println(duration);
    //Serial.print("distance: ");
    //Serial.println(distance);
    //Serial.println("cm");
   delay(300);//shorten this if needs to be more responsive
  return distance;
}
```

Figure 5.1: A prototype of *Figure 3.9b: Code for distance_left measured by Ultrasonic Sensor*

Root Cause:

Previously, we chose to put a **300ms** delay (*highlighted in code)* so that the Mbot is not overly reactive to signal fluctuations hence detecting inaccurate values at times; this will lead to jittery movements which could cause penalisation during the maze run.

This also allows the sensor to have ample time to send the next signal without overlapping(*if no delay*).

However, this has led to the following consequence:

- While being accurate in the feedback to the arduino regarding the distance from wall, the delay led to the next function which senses the black line being called too late.

Solution Implemented:

We optimised the delay value to satisfy **both** conditions:

- Delay cannot be too short leading to jittery movements
- Delay cannot be too long leading to black line check being done too late

As such, through the procedure of **halving the delay each time** and figuring whether we should opt for **higher or lower** delay value through testing(*binary search*), we realised all we needed was a delay of 10ms to keep it from jittery movements, while maximising black line detection consistency.

```
delay(10);//shorten this if needs to be more responsive
```

Outcome:

This increased the frequency of black line detection, improving the overall responsiveness of the line detector.

5.2 Colour Detection Inaccuracies

Difficulty:

The colour detection module frequently returned **inconsistent RGB readings** during data collection and calibration.

Root Cause:

- Environmental factors like changes in **ambient light intensity**.

- Wear and tear on the black paper skirting around the LEDs and LDR, **increasing exposure to ambient light**.

Initial Attempt:

Installed a black paper skirting around the LEDs and LDR to block out ambient light, but its effectiveness decreased over time due to wear and tear.

Solution Implemented:

We frequently changed and attached additional layers of skirting around the colour detection system reinforced with tape whenever there is wear and tear.

- This is done to make sure that the skirting is always nearly touching the ground allowing negligible ambient light to affect our colour detection.

Outcome:

This improved the reliability of the robot in correctly identifying the colours by minimising the effects of ambient light.

Figure 5.2: Reinforced skirting

5.3 Wiring Connections

Issue Identified:

Excessive and **disorganised** wiring in the custom-built circuit caused:

- Difficulty in troubleshooting and ensuring secure connections.
- Loose wires getting caught on wheels or maze walls, causing the bot to steer off course.
- Increased debugging time and risk of errors like short circuits or incorrect connections.

Root Cause:

- Long, protruding wires and lack of proper wire management.
- Loose connection of wires

Solution Implemented:

- Trimmed the wires to appropriate lengths.
- Taped down loose wires to prevent tangling and interference.
- Utilised Blu-tack to secure connections (shown in Figure 1.6a)

Outcome:

- Achieved a neater and sturdier circuit design.
- Minimised disruptions caused by wiring issues.
- Improved stability, reliability, and ease of debugging.

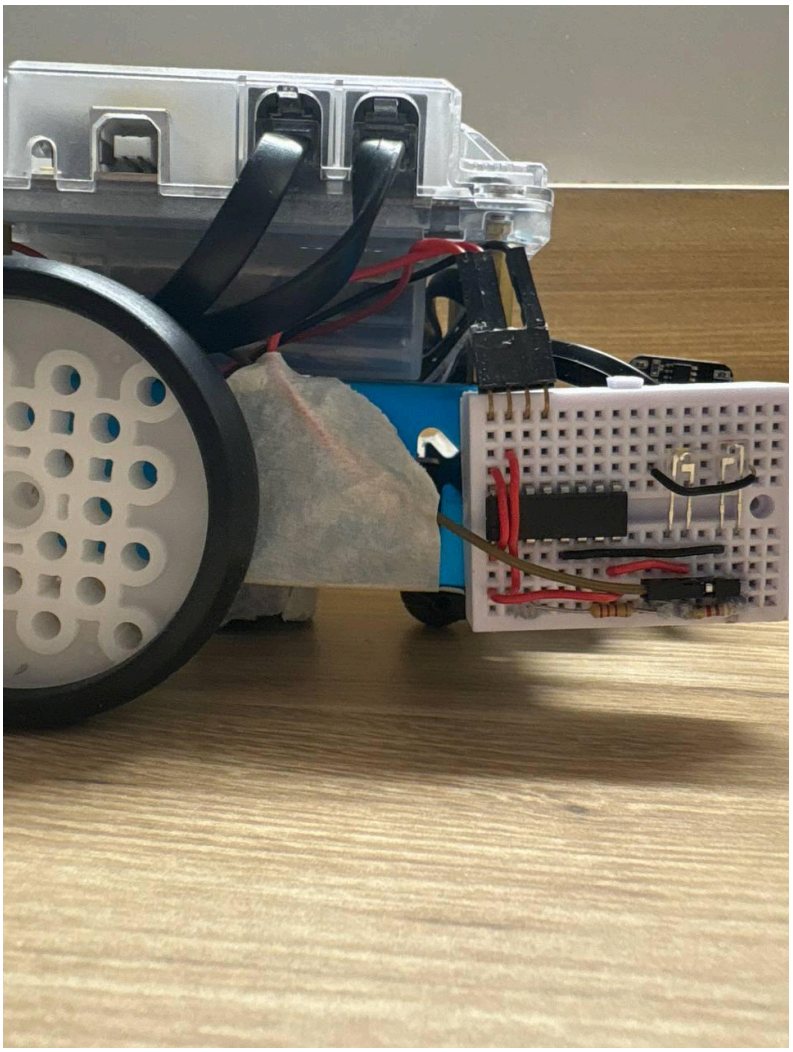


Figure 5.3: Proper wire management using tapes and jumper wires to secure connections

---

6. Conclusion

In conclusion, this project provided a comprehensive exploration of sensor integration and control mechanisms for the design of a colour-responsive mBot capable of autonomously navigating a maze. Through a systematic approach to sensor selection, algorithm

development, and iterative calibration, the mBot achieved the ability to detect and respond to different colours while maintaining a straight path and having directional control. Key sensors used in this project included a Light Dependent Resistor (LDR) for colour detection, an ultrasonic sensor for precise distance measurements, and an Infrared (IR) sensor for short-range wall detection. Each of these components contributed to a comprehensive system, where sensor feedback was used to adjust the robot's position, direction, and speed, ensuring that it could navigate turns and obstacles accurately.

However, the project faced several challenges, which underscored the complexity of building reliable autonomous systems. One major obstacle was optimising the response time of the line detector and colour sensors to maintain real-time performance without compromising accuracy. Initial delays in sensor response led to overshooting black lines and inaccurate colour identification, issues that were addressed by systematically reducing delays in sensor input processing and implementing calibration techniques to account for deviations in sensor readings. Similarly, colour detection was refined through extensive data collection and the choice of a Boolean-based decision-making algorithm over more complex methods, such as Euclidean distance calculations, which were computationally intensive and prone to inaccuracies given the mBot's hardware limitations. These adjustments resulted in improved responsiveness and stability, allowing the robot to perform reliably within the maze environment.

Also, wiring and physical design posed additional difficulties, as excessive and loosely managed wiring led to intermittent connection issues and disruptions in the bot’s movement. This was resolved through optimised wire management and physical reinforcement, which reduced the risk of disconnection and improved the bot’s overall durability. The use of reinforced skirting around the LDR sensors was also instrumental in minimising the impact of ambient light on colour detection accuracy, further enhancing the reliability of the mBot in the maze.

In addition, the calibration process for each sensor was iterative and thorough, involving the collection of extensive data to establish reference ranges for each target colour, and multiple rounds of testing to ensure the robot’s ability to navigate the maze without error. For instance, the ultrasonic sensor’s readings were calibrated to account for offset and data deviation ranges, enabling accurate steering adjustments that kept the robot centred within the maze path. Additionally, the IR sensor provided a reliable fallback for short-range detection, enhancing the mBot’s performance in scenarios where the ultrasonic sensor could not detect nearby walls or if the walls were removed from the maze.

Ultimately, this project demonstrated the successful integration of hardware and software components that reflects the main goal of the module CG1111A by giving us the opportunity to create a responsive, autonomous robot capable of fulfilling specific navigational tasks. The project’s outcomes highlight the importance of iterative testing, sensor calibration, and physical design considerations in robotics. Overall, this project serves as a valuable experience with practical insights in designing robotic systems that can be implemented in the real world.