CG1111A

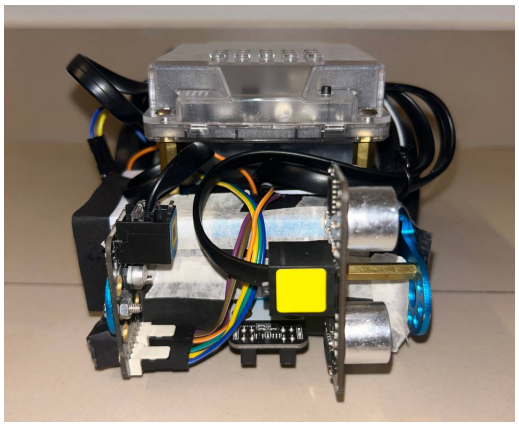
Final Project Report

Studio Group B01-S2-T2

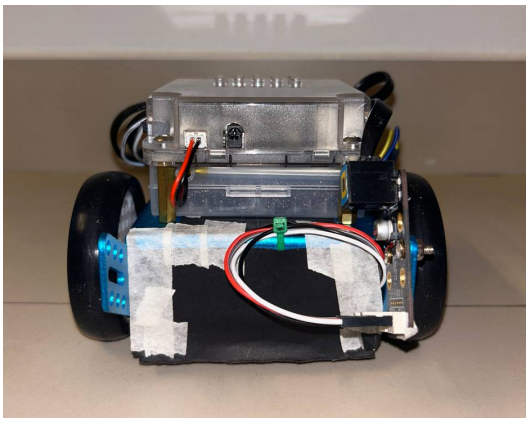
Name:	Student Number:
Chan Sheng Bin	A0273241M
Cao Junbo	A0288578B
Carvalho Andreus Roby	A0269927H
Bui The Trung	A0276103M

Table of Contents

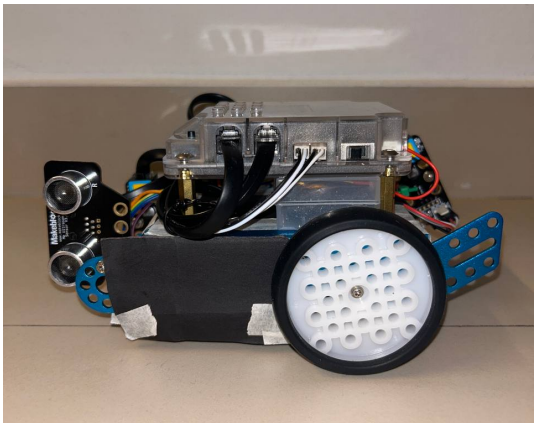
1. mBot Pictures



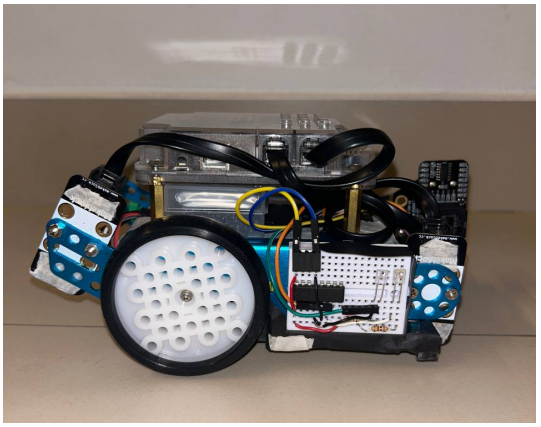
Front View



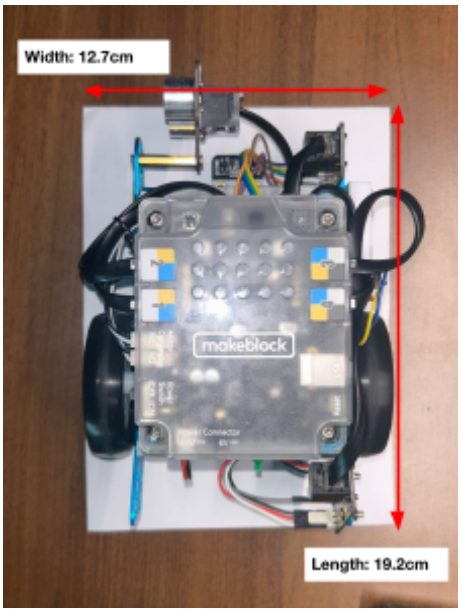
Back View



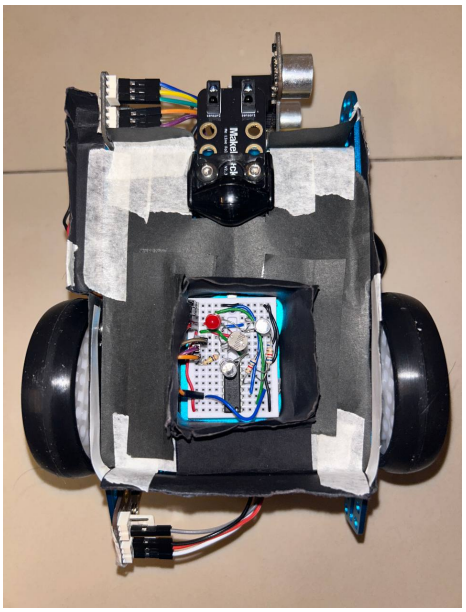
Side View (Left)



Side View (Right)



Top View



Bottom View

Figure 1: mBot Pictures

2. Robot Design

In this section, we will be covering the design process for our custom sensor circuits which includes the Infrared (IR) Proximity Sensor and Colour Detector. Moreover, this section will go in detail about some additional design decisions that we have taken to improve the performance of our mBot.

2.1 Sensor Breadboard Circuit Design Overview

Figure 2 shows the overall circuit connection of our mBot in TinkerCAD.

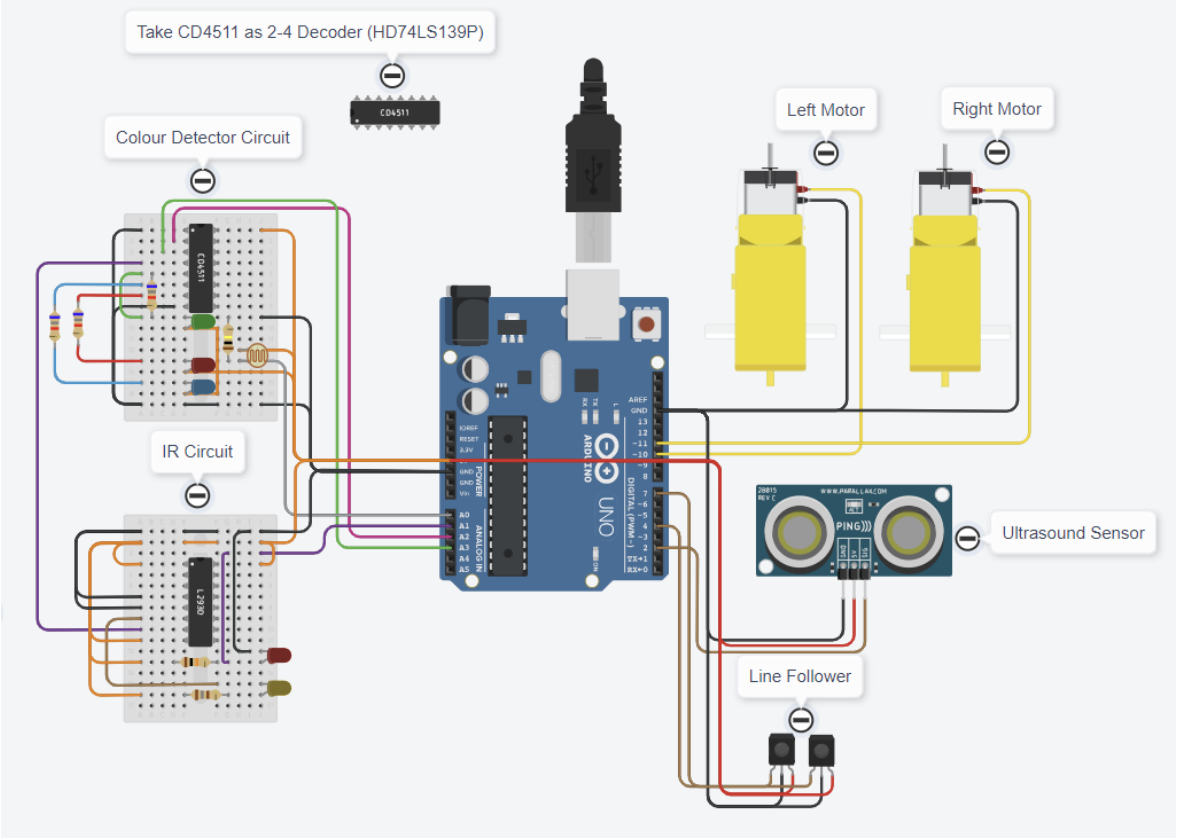


Figure 2: Circuit Connection for entire robot

2.2 HD74LS139P 2-to-4 Decoder IC Chip

Using the HD74LS139P (2-to-4 decoder) IC chip, we are able to control four outputs with two inputs. We have chosen ports A2 and A3 of the mCore as input pins for the chip.

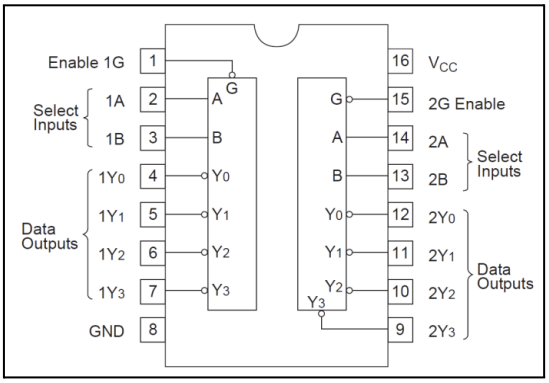


Figure 3: Pin Layout of HD74LS139P 2-to-4 decoder IC chip

We have chosen the following connection:

1A: Port A2
1B: Port A3
Y0: IR Emitter
Y1: Red LED
Y2: Green LED
Y3: Blue LED

The table below shows the function table summarising the behaviour of the 2-to-4 decoder and how it controls the respective components.

Inputs		Outputs				Component
1A (A2)	1B (A3)	Y0	Y1	Y2	Y3	
L	L	L	H	H	H	Emitter ON (through L293 Chip)
L	H	H	L	H	H	Green LED ON
H	L	H	H	L	H	Blue LED ON
H	H	H	H	H	L	Red LED ON

Figure 4 shows the connection between the custom sensors with the 2-to-4 Decoder IC Chip and the two RJ25 adapters.

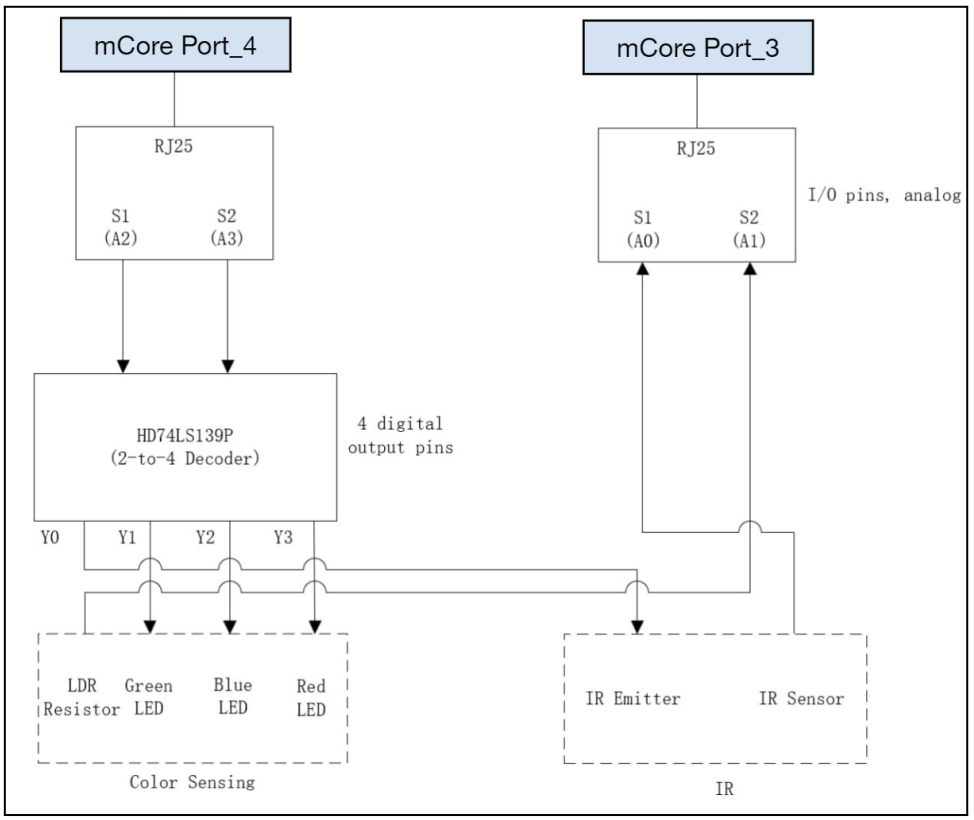


Figure 4: Connection between mCore and Custom Sensors

In the following sections, we are going to elaborate on our design processes and considerations for our custom sensor circuits.

2.3 Design Process and Considerations

In this section, we will be going through the design considerations that went into our custom sensors which are the colour detection sensor and IR sensor.

2.3.1 Colour Detection Sensor Circuit

2.3.1.1 Objective of Circuit

The colour detection sensor circuit will be used to give our mBot the ability to identify the colour of waypoints accurately to perform respective steering operations to navigate and complete the maze.

2.3.1.2 Current Limiting Resistors for LEDs

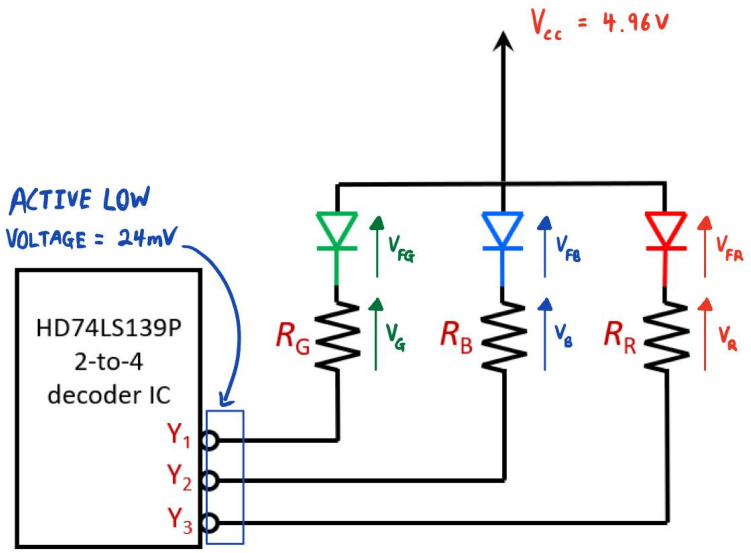


Figure 5: LED circuit connection with HD74LS139P Chip

Before we calculate the required resistance needed for our circuit, we first measure the exact values for Vcc and the active low voltage of the 2-to-4 decoder IC chip.

Since the maximum allowable current for the 2-to-4 decoder chip is 8 mA, we initially decided to design our circuit such that we achieve a current of 7 mA.

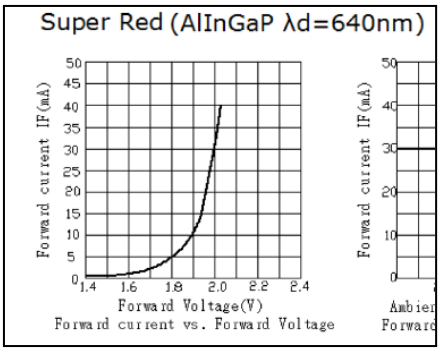


Figure 6: IV Characteristics for Red LED

Looking at the IV Characteristics for Red LED, we see that for a current of 7 mA, our red LED would have a forward voltage of 1.9 V. Using this information, we can calculate the desired resistor for our requirement:

$$V_{Forward(Red)} = 1.9\,V$$

$$V_R = (4.96V - 0.024V) - 1.9V = 3.036\,V$$

$$V_R = I * R_R = 3.036\,V$$

$$R_R = \frac{3.036\,V}{7\,mA} = 433.7\Omega$$

To get a current of maximum 7 mA,

$$R_R > 433.7\Omega$$

We used 460Ω resistor as it was the closest resistor value that satisfies the condition.

We repeat the previous steps for both the blue and green LED.

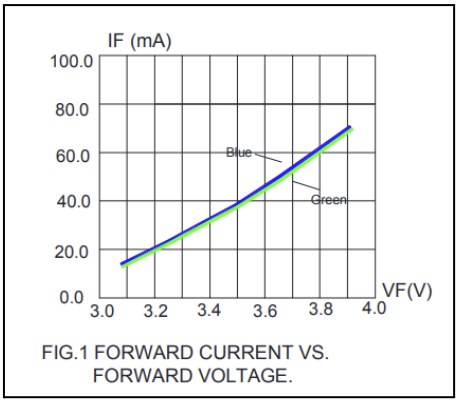


Figure 7: IV Characteristics for Blue and Green LED

Looking at the IV Characteristics for Blue and Green LED, we see that for a current of 7 mA, both LEDs would have a forward voltage of roughly 3.0 V. Using this information, we can calculate the desired resistor for our requirement:

Blue LED	Green LED
$V_{Forward(Blue)} = 3\,V$	$V_{Forward(Green)} = 3\,V$
$V_B = (4.96V - 0.024V) - 3\,V = 1.936\,V$	$V_G = (4.96V - 0.024V) - 3\,V = 1.936\,V$
$V_B = I * R_B = 1.936\,V$	$V_G = I * R_B = 1.936\,V$
$R_B = \frac{1.936\,V}{7\,mA} = 276.5\,\Omega$	$R_G = \frac{1.936\,V}{7\,mA} = 276.5\,\Omega$

To get a current of maximum 7 mA both resistors for blue and green LED should be,

$$R_B > 276.5\ \Omega \qquad \text{and} \qquad R_G > 276.5\ \Omega$$

In the lab, we used 330Ω resistor as it was the closest resistor value available in the lab.

However, the calculation of the resistance is only for the minimum resistance to prevent exceeding of maximum current in the 2-to-4 decoder chip.

As we advanced into the testing of our colour detection circuit, we realised that all three LEDs are shining too bright. After a series of experimentation, $6.8k\Omega$ resistors are used for all three LEDs. More on this decision will be elaborated in the section on challenges faced.

In short, we had to increase the resistance of each current limiting resistor to reduce light intensity and improve reliability of our colour sensor.

2.3.1.3 Current Limiting Resistors for LDR

For our LDR Resistor, we used a $100k\Omega$ ohms resistor not only because it was previously used in our studios but it gives us reliable performance with this model of LDR.

2.3.1.4 Final Circuit Design

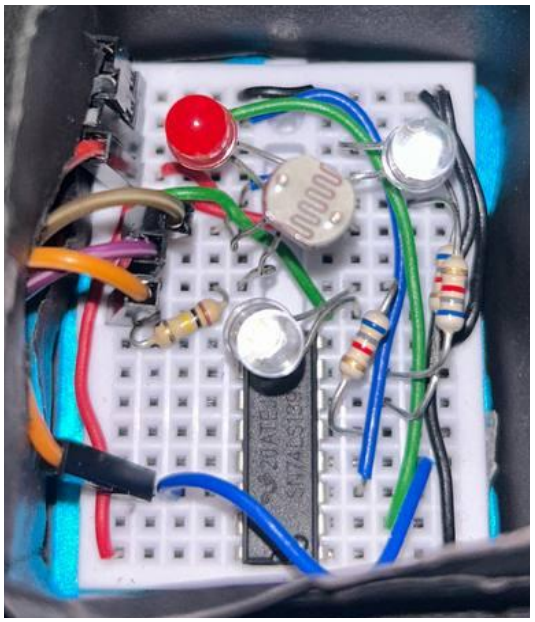


Figure 8: Colour Detection Circuit

2.3.2 IR Sensor Circuit

2.3.2.1 Objective of Circuit

The primary goal of the Infrared (IR) circuit is to make use of indirect incidence to determine proximity and ultimately prevent the mBot from colliding with the maze wall on one side. For our mBot, this circuit will be mounted to the right side to detect incoming walls on the right.

2.3.2.2 Current Limiting Resistor for IR Emitter

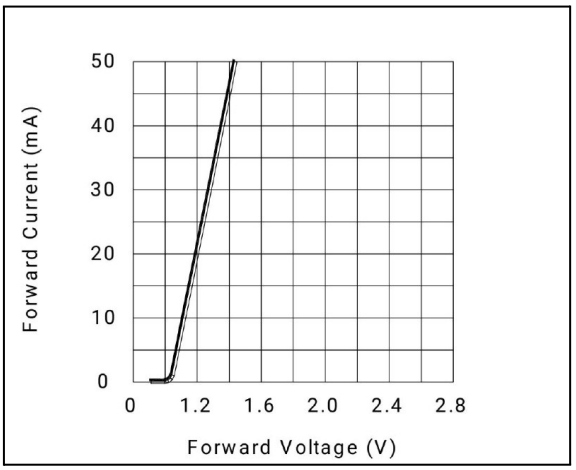


Figure 9: IR Emitter I-V Characteristic Graph

According to the datasheet for the IR emitter, the maximum rated continuous current is 50mA. From this we can calculate the lowest resistor value we can choose for our current limiting resistor.

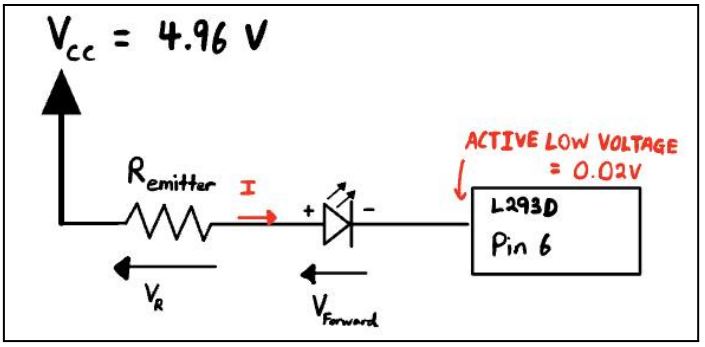


Figure 10: IR Emitter Circuit Connection

Suppose we have a current of 50mA flowing through the IR emitter,

According to the I-V Characteristic graph of the IR emitter, the forward voltage of the IR emitter is roughly 1.4V.

To find the voltage across the resistor, V_R,

$$V_R = (4.96V - 0.02V) - 1.4V = 3.54V$$

According to Ohm's Law,

$$V_R = I * R_{emitter}$$

$$R_{emitter} = \frac{V_R}{I} = \frac{3.54\,V}{50\,mA} = 70.8\Omega$$

To get a current less than 50 mA,

$$R_{emitter} > 70.8\Omega$$

We have chosen to work with a current of 20 mA as it is well below the maximum allowable current of 50 mA.

Using the same approach, we can find the resistance needed to get a current of 20mA.

According to the I-V Characteristic graph of the IR emitter, the forward voltage of the IR emitter is roughly 1.2 V.

$$V_R = (4.96V - 0.02V) - 1.2V = 3.52V$$

According to Ohm's Law,

$$V_R = I * R_{emitter}$$

$$R_{emitter} = \frac{V_R}{I} = \frac{3.52\,V}{20\,mA} = 176\Omega$$

Rounding off, we get:

$$R_{emitter} = 180\,\Omega$$

From this calculation, we decided to use a $180\,\Omega$ current limiting resistor for our IR emitter.

2.3.2.3 Current Limiting Resistor for IR Detector

For our IR Detector resistor, we initially tried a $8.2k\Omega$ resistor and found that a $10k\Omega$ resistor gives us a better sensor responsiveness.

2.3.2.4 Final Circuit Design

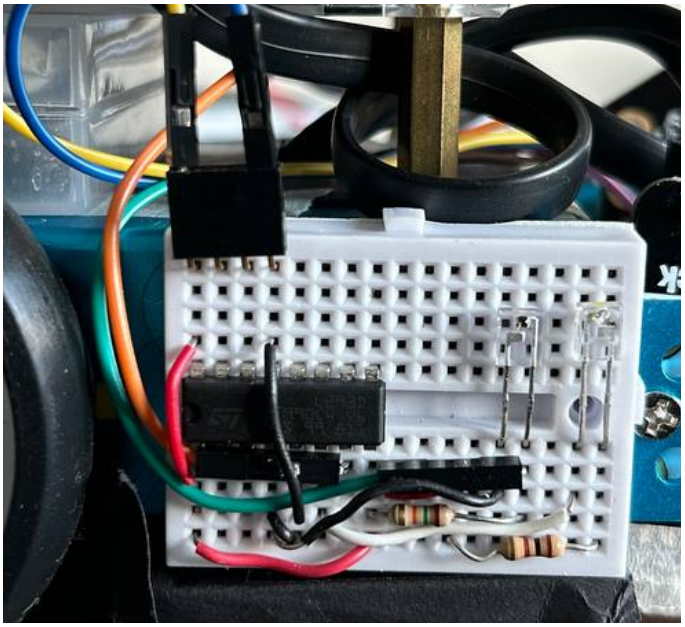


Figure 11: IR Proximity Sensor Circuit

2.3.3 Other Design Considerations

In this segment, we will be covering the different design considerations taken to enable our mBot to traverse the maze efficiently, without sacrificing motor speed or having any contact with maze walls.

2.3.3.1 Arrangement of Wires

Wire management is a crucial step to keep our robot design compact and neat. Compact design is especially important in preventing contact between wires and maze walls. Our team made use of cable ties to keep specific groups of cables together. Moreover, we made use of a common hole in the chassis for cables to pass through to simplify the shielding process while maintaining neatness.

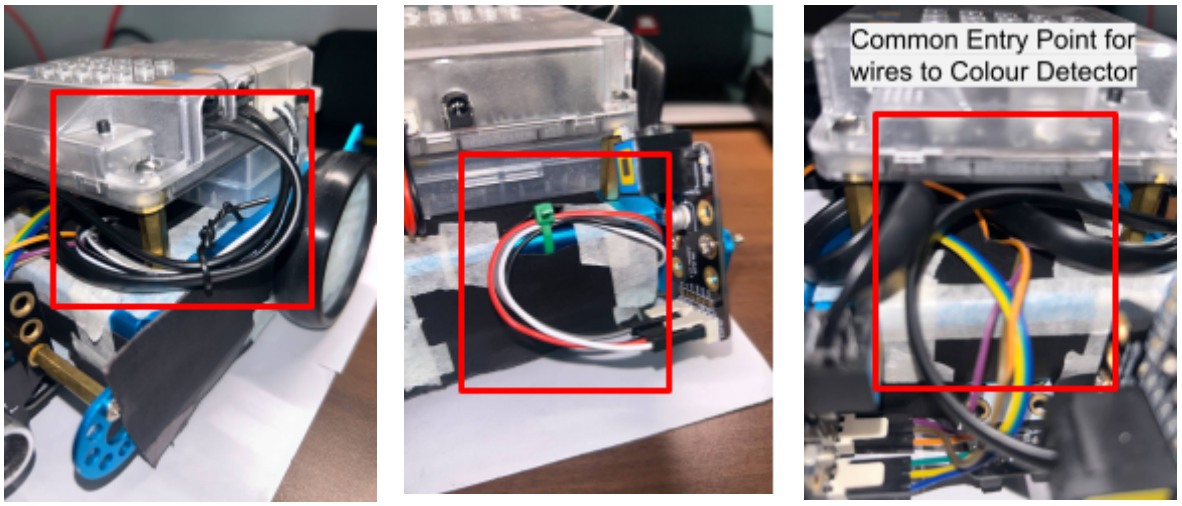
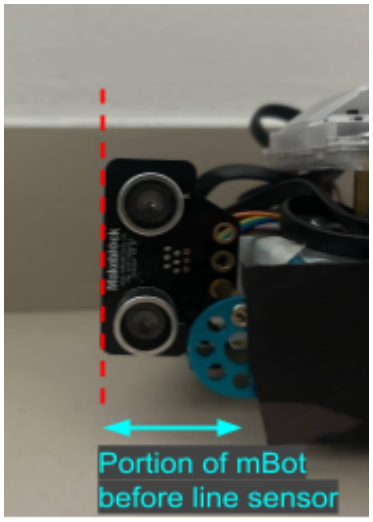


Figure 12: Wire Management

2.3.3.2 Placement of Ultrasound Sensor

Prior to the design phase of the project, we noted that the black strips of paper are placed 4 cm away from the wall. This means that we have to keep our robot design compact and reduce chances of a head-on collision with the front wall. Ultrasound sensor is mounted vertically so as to increase clearance in front of the mBot.

Moreover, through our past studios, we learn that our ultrasonic sensor has a minimum working range of at least 3 cm. We can account for this by adding an offset to the distance reading by shifting the placement of the ultrasound sensor. We mounted our ultrasonic sensor inwards using a brass stud as shown in the figure 13 below.



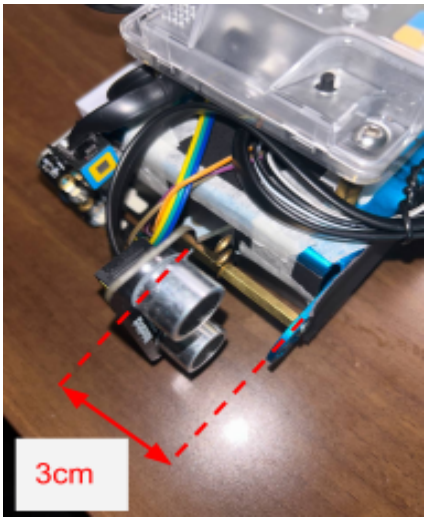


Figure 13: Placement of Ultrasound Sensor

2.3.3.3 Placement of Line Follower Sensor

The line follower sensor is used for the robot to stop the mBot at every waypoint. It does so by detecting the presence of a black strip of paper using the two pairs of IR transmitter and receiver at the front.

We decided that it would be better to move the line sensor forward so as to detect the black line early, giving our mBot slightly more time to come to a stop. This also meant that we can set our motor speed to the highest speed without having to worry about head-on collision with the wall as the mBot would be able to stop in time after the line follower sensor detects a black line.

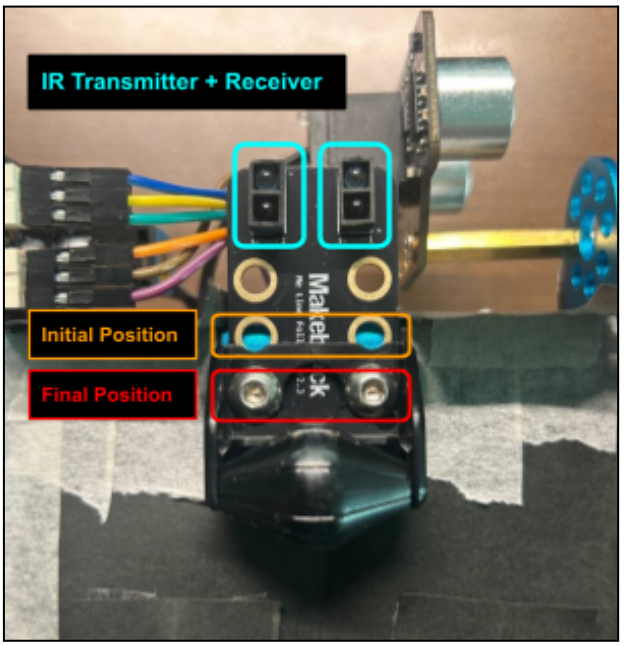
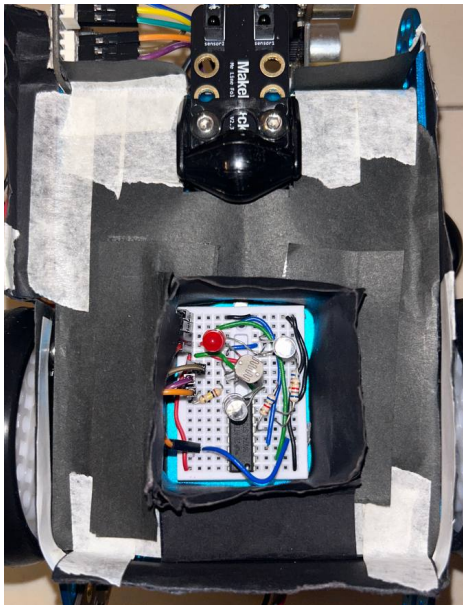


Figure 14: Placement of Line Follower Sensor

2.3.3.4 Shielding

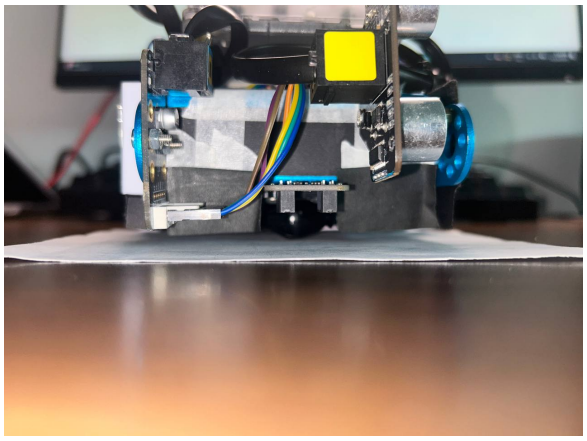


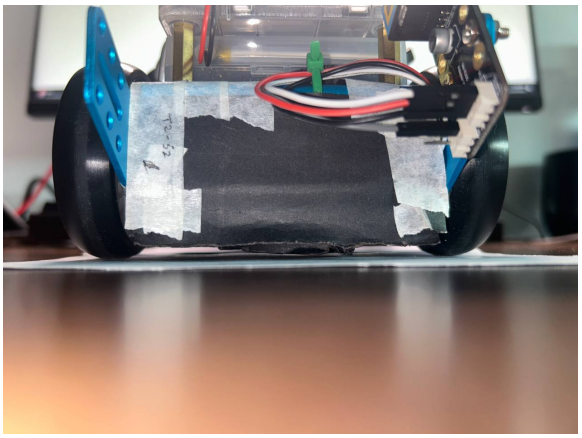
To improve the accuracy of our colour detection sensor, we have to block as much external light as possible. This is done by using black paper to create a shield or skirt.

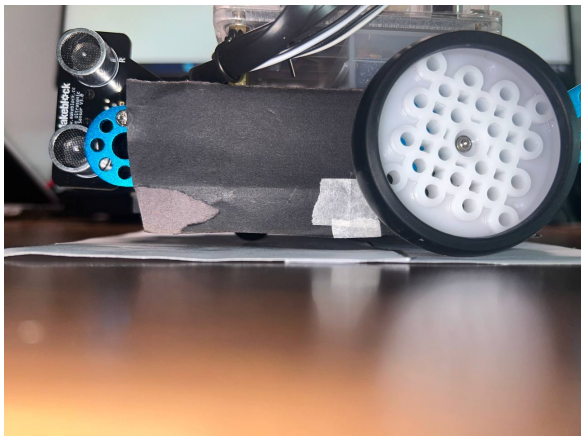
Besides creating walls at each side, we have decided to create a chimney enclosing our colour detection circuit.

It is ideal for the shielding to extend down to just above the maze surface, so as to block as much light as possible.

The following images show the shielding of the mBot from all sides.







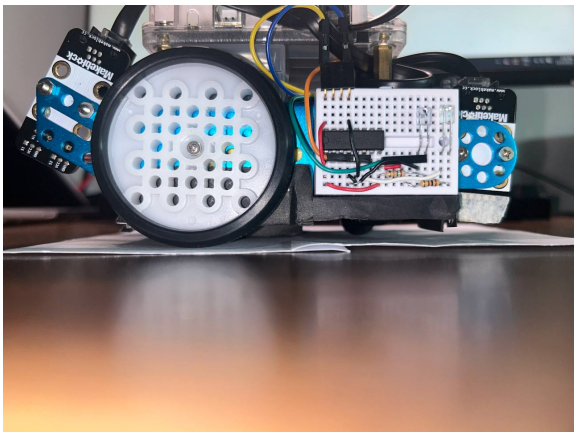


Figure 15: Shielding and Skirting for mBot

3. Maze Solving Algorithm

3.1 Algorithm Overview

In this section, we will describe the algorithm to navigate through the maze. The flowchart below shows the overall logic of the mBot Maze Solving Algorithm.

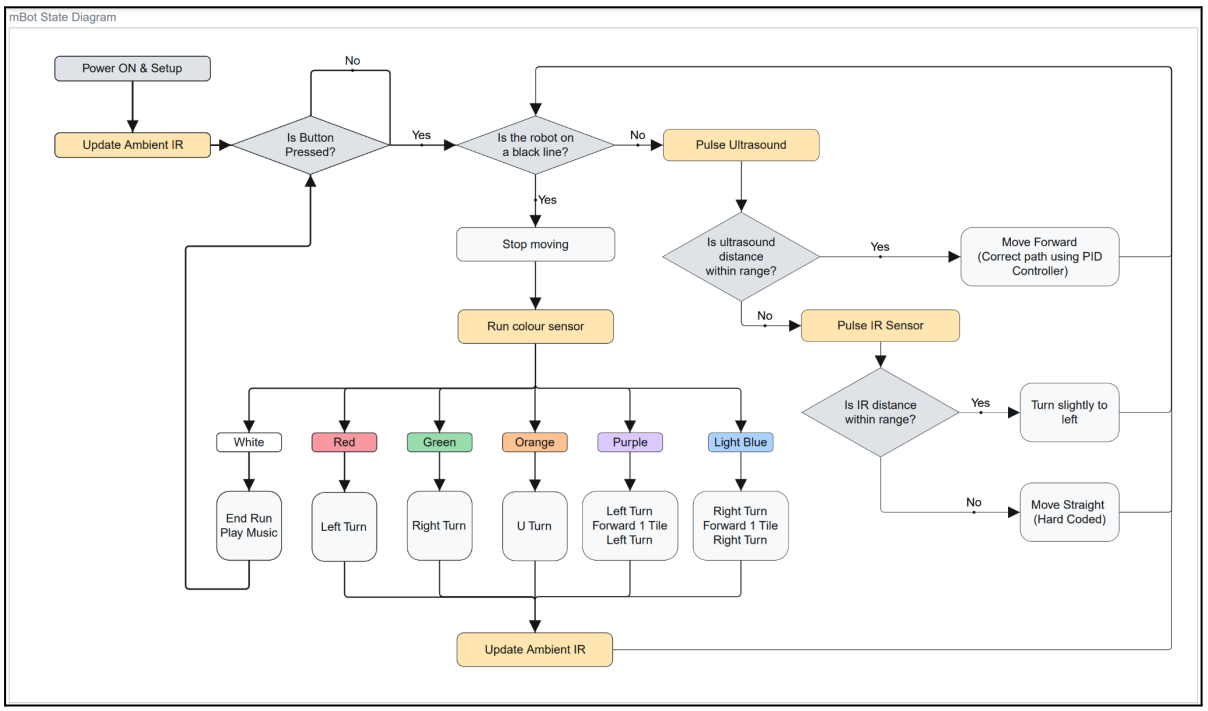


Figure 16: Flowchart for maze algorithm

Our robot algorithm can be broken down into 3 phases.

1) Idle (Standby)
2) Navigation (Searching of Waypoint)
3) Waypoint Execution

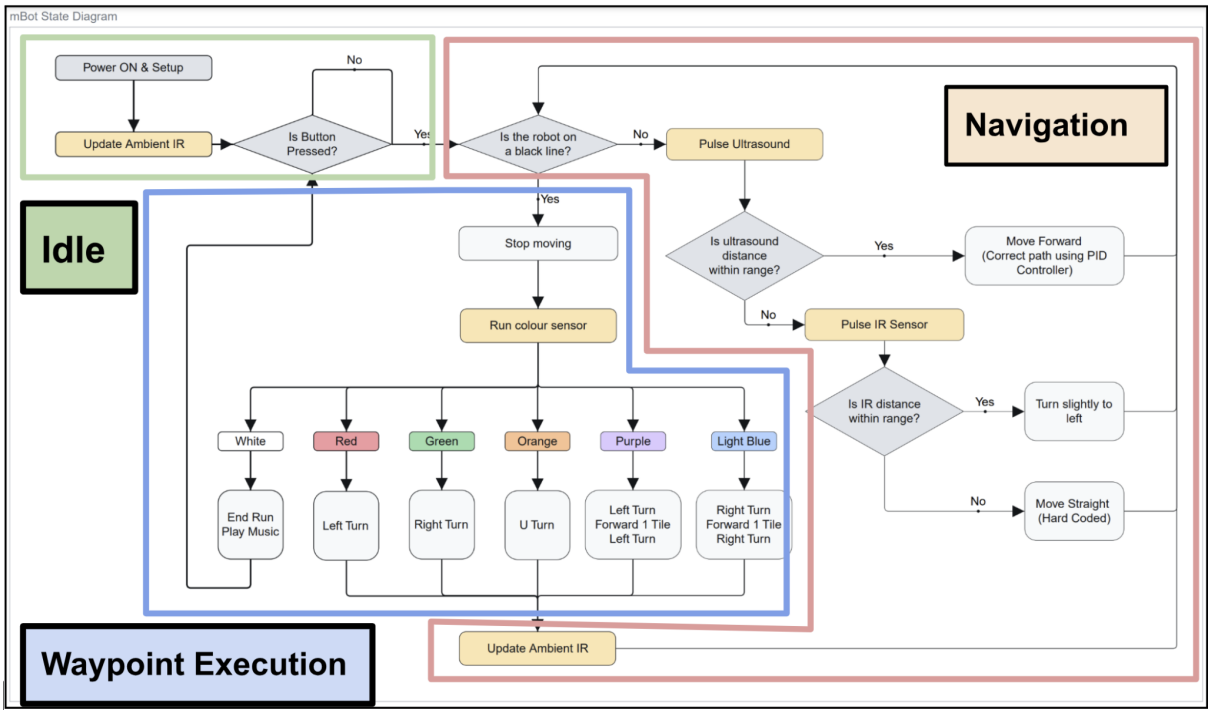
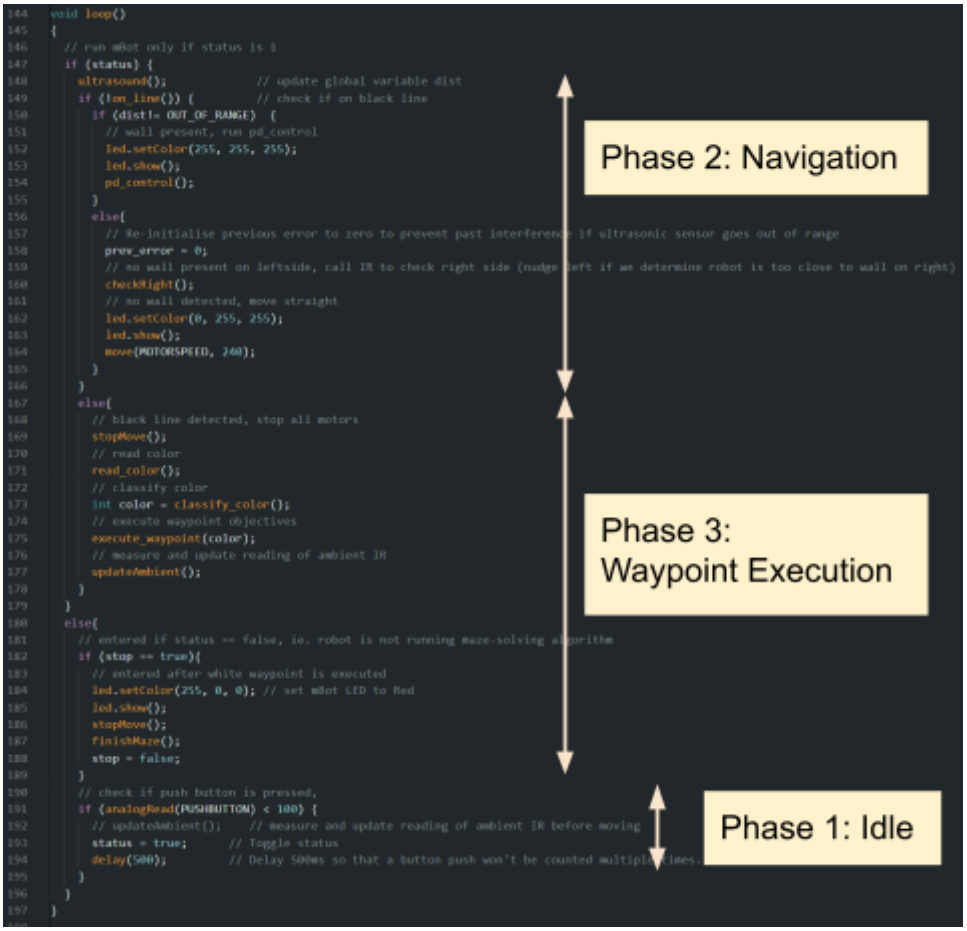


Figure 17: Flowchart for algorithm segments

The following sections will go through the code to better illustrate the flow of our maze solving algorithm. The main loop of the mBot is shown in the image below.



3.2 Phase 1: Idle

In the idle phase, the mBot waits for a button press before switching into navigation mode.

```
    // check if push button is pressed,
    if (analogRead(PUSHBUTTON) < 100) {
      // updateAmbient();    // measure and update reading of ambient IR before moving
      status = true;      // Toggle status
      delay(500);         // Delay 500ms so that a button push won't be counted multiple times.
    }
  }
}
```

3.3 Phase 2: Navigation

In the navigation phase, the mBot will first check if there is a black strip of paper detected by the line sensor. This checking is done by calling the on_line() function which uses the memCore.h library to call the read lineFinder.readSensors() function and returns true if there is a black line.

```
/**
 * This function detects for black line (waypoint)
 * @return Returns true if black line is detected, false if not detected.
 */
bool on_line() {
  sensorState = lineFinder.readSensors();
  if (sensorState != S1_OUT_S2_OUT) {
    return true;
  }
  return false;
}
```

If there is no black paper detected, the mBot will make use of the ultrasonic sensor to read the distance between the mBot and the left wall.

If there is a left wall, we call our PID Controller function to achieve wall-following to maintain smooth forward movement.

If there is no wall detected on the left, the mBot will call checkRight() to run our IR proximity detector. Within that function, the mBot will nudge left and away from the right side if our IR sensor determines that the right side of our mBot is approaching the right wall.

If our IR sensor determines that our mBot is not approaching the right wall, both motors of the mBot shall take in a fixed value to move forward normally. This whole process repeats until a black line is detected by the line sensor.

```
// run mBot only if status is 1
if (status) {
  ultrasound();             // update global variable dist
  if (!on_line()) {         // check if on black line
    if (dist!= OUT_OF_RANGE)  {
      // wall present, run pd_control
      led.setColor(255, 255, 255);
      led.show();
      pd_control();
    }
    else{
      // Re-initialise previous error to zero to prevent past interference if ultrasonic sensor goes out of range
      prev_error = 0;
      // no wall present on leftside, call IR to check right side (nudge left if we determine robot is too close to wall on right)
      checkRight();
      // no wall detected, move straight
      led.setColor(0, 255, 255);
      led.show();
      move(MOTORSPEED, 240);
    }
  }
```

3.4 Phase 3: Waypoint Execution

After the mBot detects a black line, it enters the waypoint execution phase in which it will stop moving and call read_color() to estimate the red, green and blue values of the current waypoint. Next, the mBot will run classify_color() to classify our new readings as a colour. It does so by returning a colour id (0, 1, 2, 3, 4, 5) which is passed to the function execute_waypoint() to execute the movement specific to the identified waypoint.

Lastly, the mBot updates the baseline IR which is the reading of the ambient IR before entering the navigation phase to traverse the rest of the maze.

The mBot goes back to idle phase after detecting a white waypoint. This is done by changing stop to true, which happens in the execute_waypoint() function when white waypoint is detected. Since a white waypoint signifies the end of a maze, the function finishMaze() is called to play a celebratory tune of our choice through the buzzer.

```
167       else{
168         // black line detected, stop all motors
169         stopMove();
170         // read color
171         read_color();
172         // classify color
173         int color = classify_color();
174         // execute waypoint objectives
175         execute_waypoint(color);
176         // measure and update reading of ambient IR
177         updateAmbient();
178       }
179     }
180     else{
181       // entered if status == false, ie. robot is not running maze-solving algorithm
182       if (stop == true){
183         // entered after white waypoint is executed
184         led.setColor(255, 0, 0); // set mBot LED to Red
185         led.show();
186         stopMove();
187         finishMaze();
188         stop = false;
189       }
```

The table and image below shows the different waypoint tasks and the implementation of execute_waypoint().

Waypoint Tasks:

Colour	Action/Task
Red	Left Turn
Green	Right Turn
Orange	U Turn
Purple	2 Successive Left Turns
Light Blue	2 Successive Right Turns
White	End Run

Code Implementation for execute_waypoint():

```
/********** Functions (Waypoints) **********/

/**
 * Function takes in the color of waypoint classified by mB
 * @param[in] color Classified color id
 */
void execute_waypoint(const int color)
{
  switch(color) {
  case 0:
    // code block for white
    led.setColor(255, 255, 255); // set both LED to WHITE
    led.show();
    stop = true;
    status = false;
    break;
  case 1:
    // code block for red
    led.setColor(255, 0, 0); // set both LED to RED
    led.show();
    turnLeft(TIME_FOR_LEFT_TURN);
    break;
  case 2:
    // code block for blue
    led.setColor(0, 0, 255); // set both LED to BLUE
    led.show();
    doubleRight(TIME_FOR_1_GRID_BLUE);
    break;
  case 3:
    // code block for green
    led.setColor(0, 255, 0); // set both LED to GREEN
    led.show();
    turnRight(TIME_FOR_RIGHT_TURN);
    break;
  case 4:
    // code block for orange
    led.setColor(153, 76, 0); // set both LED to ORANGE
    led.show();
    uTurn();
    break;
  case 5:
    // code block for purple
    led.setColor(153, 51, 255); // set both LED to PURPLE
    led.show();
    doubleLeft(TIME_FOR_1_GRID_PURPLE);
    break;
  default:
    // code block for no color classified
    led.setColor(0, 0, 0); // set both LED to OFF
    led.show();
    break;
  }
}
```

4. Subsystem: General Movement

In this section, we will be going through the use of the “MemCore.h” library to perform motor control to achieve our project requirements.

To perform motor control, we have to define the mCore ports that are used for the left and right motors.

```
MeDCMotor                leftWheel(M1);        // assigning LeftMotor to port M1
MeDCMotor                rightWheel(M2);       // assigning RightMotor to port M2
```

The figure below shows the constants defined in relation to mBot movement. For our mBot, we are using the maximum motor speed of 255. The constants for the various waypoint objectives are found through experimentation.

```
// Movement
#define MOTORSPEED                255
#define IR_THRESHOLD              480   // Amount of dip in value to determine proximity of wall on right side. (Difference between Amb
#define SIDE_MAX                  18    // Side distance threshold (cm)
#define TIME_FOR_LEFT_TURN        300   // The time duration (ms) for turning 90 degrees counter-clockwise      (for red waypoint)
#define TIME_FOR_RIGHT_TURN       300   // The time duration (ms) for turning 90 degrees clockwise              (for green waypoint)
#define TIME_FOR_1_GRID_PURPLE    640   // The time duration (ms) for moving forward by 1 grid                  (for purple waypoint)
#define TIME_FOR_1_GRID_BLUE      720   // The time duration (ms) for moving forward by 1 grid                  (for blue waypoint)
#define TIME_FOR_SECOND_LEFT_TURN 320   // The time duration (ms) for second 90 degrees counter-clockwise turn  (for purple waypoint)
#define TIME_FOR_SECOND_RIGHT_TURN 300  // The time duration (ms) for second 90 degrees clockwise turn          (for blue waypoint)
#define TIME_FOR_UTURN            530   // The time duration (ms) for turning 180 degrees clockwise             (for orange waypoint)
```

Our mBot uses a move() function which takes in the absolute speed (a value from 0 to 255) for both left and right motors. The function will invert the value passed for the left motor as the left motor requires an anti-clockwise rotation to move the mBot forward. The right motor will take in the positive value which will make it rotate in a clockwise manner to move the mBot forward.

```
/**
 * This function is a general movement function used to move robot forward.
 * Takes in values from -255 to 255 as input and writes it to the motors as PWM values.
 * @param[in] L_spd Left Motorspeed to be passed to left motor, positive value indicate forward, negative value indicate reverse
 * @param[in] R_spd Reft Motorspeed to be passed to right motor, positive value indicate forward, negative value indicate reverse
 */
void move(int L_spd, int R_spd) {
  if (stop == false) {
    leftWheel.run(-L_spd);
    rightWheel.run(R_spd);
  }
  else {
    stopMove();
  }
}
```

To stop both motors from spinning, we can simply invoke the stop() function provided in the MemCore.h library. We have created a simple function called stopMove() which will call stop() for each motor, allowing us to stop the mBot from moving.

```
/**
 * This function is called to stop both motors.
 */
void stopMove() {
  rightWheel.stop();
  leftWheel.stop();
}
```

Besides moving forward and stopping, our mBot needs to be able to perform 90 degree turns to traverse the maze. For a 90-degree left turn, both left and right motors shall turn clockwise. This means passing a positive value to both motors. Conversely, for a 90-degree right turn, both left and right motors shall take in negative values to turn counter clockwise.

To achieve exact 90 degree turns, we pass in constant time delays that are defined previously. The constants for left and right turns are TIME_FOR_LEFT_TURN (300ms) and TIME_FOR_RIGHT_TURN (300ms) respectively.

```
/**
 * Function allows mBot to make a 90 degrees counter-clockwise turn.
 * @param[in] time delay time in ms.
 */
void turnLeft(int time) {
  leftWheel.run(MOTORSPEED);
  rightWheel.run(MOTORSPEED);
  delay(time);
  stopMove();
}
```

```
/**
 * Function allows mBot to make a 90 degrees clockwise turn.
 * @param[in] time delay time in ms.
 */
void turnRight(int time) {
  leftWheel.run(-MOTORSPEED);
  rightWheel.run(-MOTORSPEED);
  delay(time);
  stopMove();
}
```

To turn 180-degrees, which will be used in the orange waypoint, we make use of our turnRight() function and pass in the time constant defined as TIME_FOR_UTURN which equates to 530ms.

```
/**
 * Function allows mBot to make a 180 degrees clockwise turn.
 */
void uTurn() {
  turnRight(TIME_FOR_UTURN);
}
```

5. Subsystem: Buzzer

This section covers the implementation for the buzzer which will be used to play our celebratory tune when our mBot has traversed the whole maze.

Firstly, we define the music pin which is found to be pin number 8. Next, we make use of the memCore.h library to define a buzzer object.

```
8   #define MUSIC_PIN          8
9   MeBuzzer                   buzzer;
```

Our mBot uses a custom built header file called “Notes.”' which contains the music notes and duration for each note. Inside this header file, we have two arrays called music_key[] and music_duration. Both arrays contain the keys and duration for each note respectively. The tune that was chosen is a chinese song called Juhuatai by Jay Chou.

```
97  // Sequence of Music Notes for Tune
98  static int music_key[] = {
99
100   NOTE_C3, NOTE_D3, NOTE_E3, NOTE_E3,
101   NOTE_G3, NOTE_A3, NOTE_A3,
102
103   NOTE_E4, NOTE_D4, NOTE_C4, NOTE_C4,
104
105   NOTE_A3, NOTE_G3, NOTE_A3, NOTE_G3,
106   NOTE_E3, NOTE_D3, NOTE_C3,
107
108   NOTE_G2, NOTE_C2,
109 };
110
111 // Duration for each Music Note for Tune
112 static int music_duration[] = {
113   8,  8,  8,
114   8, 8, 8, 8,
115
116   8, 8, 8, 8,
117
118   8, 8, 8, 8,
119   8, 8, 8,
120
121   8, 8
122 };
123 #endif
```

After creating our music tune inside Notes.h, we simply include the header file in our main program and create a function called finishMaze() that will iterate through each note in music_key[] and buzz that specific note for a duration set by music_duration[].

```
5   #include "Notes.h"    // Self-made header file to contain Celebratory Tune for Buzzer
```

```
526 /**
527  * This function plays a music tune of our choice.
528  */
529 void finishMaze() {
530   // keys and durations found in NOTES.h
531   for (int i = 0; i < sizeof(music_key) / sizeof(int); ++i) {
532     // quarter note = 1000 / 4, eighth note = 1000/8, etc. (Assuming 1 beat per sec)
533     const int duration = 1000 / music_duration[i];
534     buzzer.tone(MUSIC_PIN, music_key[i], duration);
535     delay(duration * 1.30);
536     buzzer.noTone(MUSIC_PIN);
537   }
538 }
```

6. Subsystem: Ultrasound Proximity Detection (Wall Tracking Algorithm)

6.1 Fundamental Concept

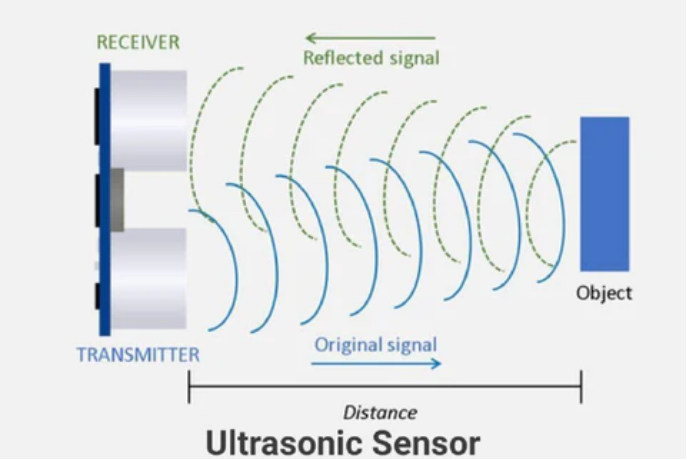


Figure 18: How an ultrasonic sensor works

The ultrasonic sensor works by emitting high-frequency sound waves and then measuring the time for the waves to bounce back after hitting an object. The distance from the sensor to the object can be calculated using the following formula:

$$Distance \ = \ speed\ of\ sound(\sim 340 m/s) \ * \frac{time\ taken\ for\ signal\ to\ come\ back}{2}$$

6.2 Initial Approach

Initial algorithm involved the mBot making a slight turn in the opposite direction (pausing forward movement) when the ultrasonic sensor was close enough to the wall.

However, such an approach to prevent the robot from colliding into the walls was naive because, although the mBot was successful in correcting its movement when too close to the wall, its movement ended up being very haphazard, slow and far from elegant.

6.3 Improved Approach (PID Controller)

6.3.1 Concept behind PID Controller

The Proportional-Integral-Derivative (PID) controller is a control loop mechanism that continuously employs feedback from sensors to determine the amount of correction required. In our context, we will be utilising the ultrasound as our primary sensor for the controller.

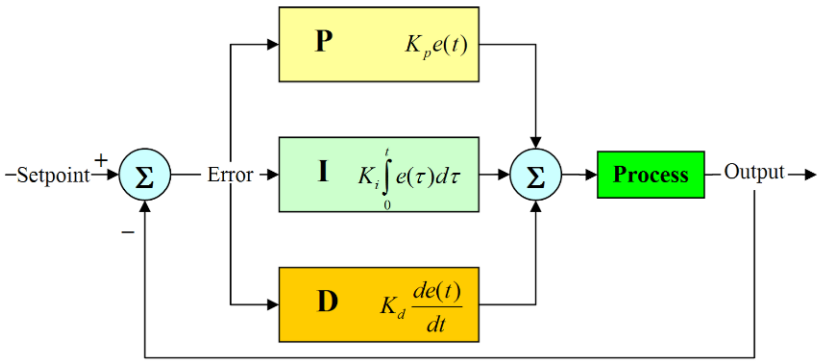


Figure 19: PID Flowchart

The general PID equation is as follows:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt}$$

The proportional component (Kp) responds to the current error, the integral component (Ki) considers the accumulation of past errors, and the derivative (Kd) component anticipates future errors based on the rate of change. The error is a function of time which represents the difference between the current position and its desired position.

In the context of wall following, the desired position is the midpoint between two maze walls. The error would simply be the difference between our desired position and the mBot's current position. Using this error, we can calculate the amount of correction needed to adjust the mBot's path such that we achieve a smooth and straight movement.

The table below shows the effects of the different components on the response behaviour.

Parameter	Rise time	Overshoot	Settling time	Steady-state error	Stability[11]
K_p	Decrease	Increase	Small change	Decrease	Degrade
K_i	Decrease	Increase	Increase	Eliminate	Degrade
K_d	Minor change	Decrease	Decrease	No effect in theory	Improve if K_d small

Figure 20: PID Constants and their effect on system response

We are excluding the Integral component in this project as the P and D component of PID is sufficient for us to perform consistent and reliable wall-following.

To optimise a PD Controller, we experiment and tune our constants for both the P and D components.

Standard manual tuning procedure:

1) Tune Proportionality gain to get a steady oscillation

2) Tune Derivative constant to get a close to critically damped response (no oscillation)

6.3.2 Implementation (PD Controller)

Our PD algorithm makes use of the ultrasonic sensor's distance measurement capability in void ultrasound( ) and if the distance is more than the desired distance (10.5cm), the distance variable gets assigned a large value of 100 cm to represent that it is Out of Range.

```
/********** Functions (Ultrasound & Line Sensor) **********/

/**
 * This function updates the global variable dist to current distance between the ultrasonic sensor and the closest object (wall) to it.
 * Sets dist to OUT_OF_RANGE if measured distance is out of threshold SIDE_MAX.
 */
void ultrasound() {
  dist = ultraSensor.distanceCm();
  if (dist > SIDE_MAX)
  {
    dist = OUT_OF_RANGE;
  }
  // Serial.println(dist);
}
```

The figure below shows the variables and constants used in our PD Controller. The tuning process to get the constant values for the P and D component will be elaborated at the end of this section.

```
/********** Constants & Variables for PID Controller (only PD is used) **********/

const double kp = 19;  // Proportional Gain/Constant  (P component of PID)
const double kd = 19;  // Derivative Constant         (D component of PID)

// Variables to hold final motorspeed calculated for each motor
int L_motorSpeed;
int R_motorSpeed;

const double desired_dist = 10.50;  // Desired distance between ultrasound sensor and wall to keep mBot centered in tile
double error;                       // Difference between current position and our desired distance
double prev_error = 0;              // Variable to store previous error, to be used to calculate change in error (For D component of PID)
double error_delta;                 // Difference between current error and previous error (For D component of PID)
double correction_dble;             // For calculation of correction for motors
int correction;                     // To be used to adjust input to motor
```

```
/**
 * Function used P and D components of PID to calculate the new PWM values for left motor and right motor.
 * Error is the difference between the robot's current position and the desired position of the robot.
 * After finding the error, we multiply the error by the proportional gain which helps us minimise this error.
 * Change of error (error_delta) helps us prevent oscillation by adding "damping" effect to our correction.
 */
void pd_control() {
  error = desired_dist - dist;                    // P Component of PID
  error_delta = error - prev_error;               // D Component of PID
  correction_dble = (kp * error) + (kd * error_delta);
  correction = (int)correction_dble;

  // Determine direction of correction and execute movement
  if (correction < 0) {
    L_motorSpeed = 255 + correction;
    R_motorSpeed = 255;
  } else {
    L_motorSpeed = 255;
    R_motorSpeed = 255 - correction;
  }
  move(L_motorSpeed, R_motorSpeed);

  //Initialise current error as new previous error (D Component of PID)
  prev_error = error;
}
```

In pd_control ( ), the error is calculated by getting the difference between desired_dist and the distance variable and error_delta is the change in error, which is found by taking the difference of current error and previous error.

Correction_dble combines the P and D component together, involving constants Kp = 19 and Kd = 19 which are values we arrived at through several rounds of fine tuning (doing multiple runs and changing the constants every round to achieve a better run).

Subsequently, correction_dble is converted to an integer “correction” as the motorSpeed variable can only be an integer value.

If “correction’ is negative, (left side of mBot is too far from wall), left motor speed is decreased so it moves closer to left wall and if ‘correction’ is non-negative (left side of mBot is too close to wall), right motor speed is decreased so it moves closer to right wall.

Lastly, prev_error is reassigned a new value which is the latest error that was calculated above. This loop goes for the entire duration for which the mBot is turned on, constantly ensuring that the mBot does not collide into the walls.

6.3.3 Calibration and Finetuning

Now that we have the software implementation of a PD controller, now we will be going through the process to calibrate and find the ideal values for our proportional gain and derivative constant. This tuning is done at the lab, where we observe the movement of the mBot and take note of the time taken for the mBot to adjust itself back to its setpoint.

The steps taken to finetune the PD Controller are as follows:

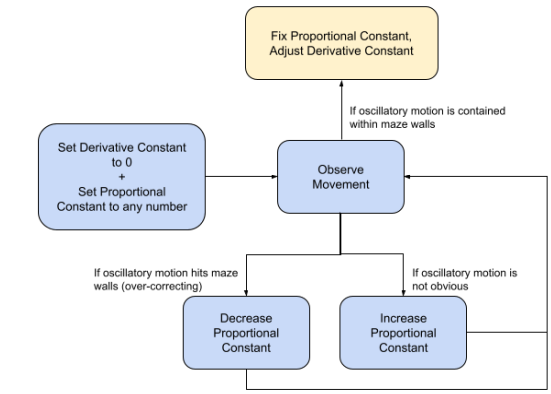


Figure 21: Process to tune Proportional Gain

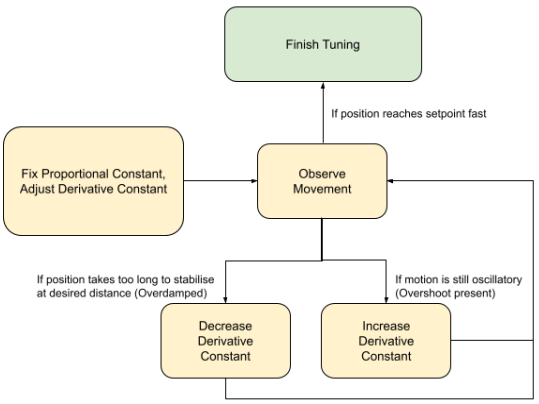


Figure 22: Process to tune Derivative Constant

By following the tuning procedure mentioned earlier, we arrived at a proportional constant of 19 and a derivative constant of 19.

7. Subsystem: IR Proximity Detection

7.1 Fundamental Concept

The IR detector can be viewed as a variable resistor that varies with the amount of IR received by the collector. Paired with an IR Emitter, we make use of indirect incidence to transmit IR from the emitter to the detector.

According to the circuit on the right, we are using the analogue input pin A1 of the mCore to measure the analogue voltage across the detector.

As the IR proximity sensor approaches a reflective surface, the detector will be able to receive more IR which increases its resistance. Since it is connected to a 10k resistor in series, we have a potential divider circuit.

In the absence of IR, we would have a higher voltage value close to Vcc as the detector resistance is low.

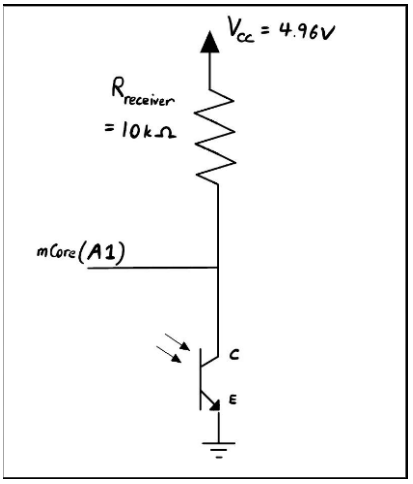


Figure 23: IR circuit

7.2 Initial Approach

Our initial approach directly uses the analog readings of the IR detector to determine proximity of a wall. We do so by comparing the measured voltage to a threshold value to determine that the right side of the mBot is within a certain distance to the wall.

```
274  void checkRight() {
275    // Turn ON IR Emitter
276    for(int i = 0; i < 2; i++){
277      digitalWrite(ledArray[i], 0);
278    }
279    delay(IRWait);
280    // Measure voltage across IR Detector
281    int irVolt = analogRead(IR);
282    // TURN OFF IR Emitter
283    for(int i = 0; i < 2; i++){
284      digitalWrite(ledArray[i], 1);
285    }
286    if (irVolt > IR_THRESHOLD)
287    {
288      // nudge left
289      move(215,255);
290      delay(5);
291    }
292  }
```

However, this approach does not consider the changes in the environment. The ambient lighting and nearby electronic appliances may affect the presence of IR in various parts of the maze. Due to inconsistent presence of IR, the voltage across the detector will be inconsistent as well, limiting our capability to determine proximity.

7.3 Improved Approach

7.3.1 Tackling Changes in Ambient IR

One solution to tackle changes in ambient IR is to regularly measure the amount of IR present in the environment. This is done by turning the emitter off and reading the analogue voltage across the detector. This voltage value is known as the baseline voltage, which shall be stored in a variable named ambientIR.

By keeping track of the baseline voltage, we will be able to calculate the amount of IR reflected to the IR detector from the IR emitter.

The steps to determine proximity using IR sensor will be:

1. Update baseline voltage
2. Turn IR Emitter ON
3. Measure analogue voltage of IR Detector
4. Turn IR Emitter OFF
5. Compare measured voltage with baseline voltage
6. If the difference exceeds the defined threshold, nudge mBot to the left.

7.3.2 Implementation

The figure below shows the time delay constants related to our IR proximity sensor.

The IR_THRESHOLD is the amount of “dip” in voltage that is required for our function to determine the presence of a wall on the right.

The value 480 corresponds to roughly a voltage of 2.34V. The derivation for this threshold will be covered later.

```
#define IRWait    20    // Time delay (in ms) before taking IR reading
#define IR_THRESHOLD              480   // Amount of dip in value to det
```

As mentioned in the previous section, we now include the function to update the ambient IR reading.

```
253  /**
254   * This function updates measured voltage value for ambient IR.
255   * Ambient IR is measured by first turning OFF the IR Emitter and measuring the voltage of IR Detector.
256   */
257  void updateAmbient(){
258    // Turn OFF IR Emitter
259    for(int i = 0; i < 2; i++){
260      digitalWrite(ledArray[i], 1);
261    }
262    // wait for voltage to stabilise before reading
263    delay(IRWait);
264    ambientIR = analogRead(IR);
265  }
```

Lastly, we add the additional step to find the difference between the baseline voltage and the measured voltage across the detector. Using this difference, we compare it with the defined threshold and if it exceeds that value, we will call move(215, 255) to nudge the mBot to the left.

```
272  void checkRight() {
273    // Turn ON IR Emitter
274    for(int i = 0; i < 2; i++){
275      digitalWrite(ledArray[i], 0);
276    }
277    delay(IRWait);
278    // Measure voltage across IR Detector
279    int irVolt = analogRead(IR);
280    // TURN OFF IR Emitter
281    for(int i = 0; i < 2; i++){
282      digitalWrite(ledArray[i], 1);
283    }
284  ··int·difference·=·ambientIR·-·irVolt;
285    if (difference > IR_THRESHOLD)
286    {
287      // nudge left
288      move(215,255);
289      delay(5);
290    }
291  }
```

7.3.3 Calibration and Finetuning

Now that we have looked at how the IR proximity sensor is implemented in code, we will be covering the calibration process and how we derived our threshold value.

Our team decided to choose a distance of 4 cm as our threshold distance between our IR sensor and the right wall.

We wrote a program that will change the mBot onboard LED to reflect that there is a wall detected on the right (i.e, difference between ambientIR and voltage across detector is beyond threshold value).

Using this, we could have a visual indicator of the rough distance required for our IR proximity sensor to detect the presence of a wall.

By placing a ruler between the wall and the IR sensor circuit, we can measure the distance needed for the IR sensor to detect a wall. Using this iterative approach, we arrived at our threshold value of 480 to detect a wall when it is 4 cm away from our IR sensor.

8. Subsystem: Colour Detection

8.1 Fundamental Concept

The LDR is essentially a light-controlled variable resistor whose resistance varies inversely proportional to the intensity of light. This means that if there is less light received by the LDR, its resistance will be higher.

In addition, different coloured surfaces reflect different amounts of light in each of the three primary colours that is red, green and blue. For a red surface, it would reflect more red light as compared to green and blue.

Using these two concepts, we can say that the colour of an object we perceive is the colour of light being reflected by that object while all other colours get absorbed.

Hence, by emitting a series of coloured pulses and reading the analogue voltage across the LDR for each colour pulse, we would be able to get a reading to quantify how much light is reflected for each colour. Moreover, we measure the LDR resistance for white and black samples beforehand so we can get a range of values which we can use to gauge the intensity for each colour. The intensity for each colour is described as a number between 0 and 255.

8.2 Initial Approach

After reading the analogue voltage across the LDR for each colour pulse. We proceeded to classify each colour by defining a range for each red, green and blue values.

For example, we defined the following range of values for the colour red:

```
401  bool withinRed()
402  {
403    if ( (55 <= blue && blue <= 140) && (100 <= green && green <= 165) &&(170 <= red && red <= 255)  )
404    {
405      Serial.println("Red Detected.");
406      return true;
407    }
408    return false;
```

To give another example, we defined the following range of values for the colour orange:

```
438  bool withinOrange()
439  {
440      if ( (50 <= blue && blue <= 130) && (150 <= green && green <= 210) &&(160 <= red && red <= 255)  )
441      {
442        Serial.println("Orange Detected.");
443        return true;
444      }
445      return false;
446  }
```

This method of classification proved to work for the maze on our table but we soon realise that our mBot is unable to classify the desired colour on other tables due to the difference in ambient lighting. With changes in ambient lighting our measured values for each colour is different. This is also partly due to insufficient shielding at the bottom of the mBot.

Moreover, this method of classification is tedious to implement and finetune. We have three values to tune for six different colours. Tuning these ranges to satisfy all mazes would be a great hassle. Hence, we decided to improve on this and apply a bit of a statistical approach to solve this problem.

8.3 Improved Approach

8.3.1 K-Nearest Neighbour (KNN) Algorithm

Suppose we have a 3D plot to describe any colour, using red, green and blue intensity for each of the axes. We would be able to pinpoint a coordinate for each colour.

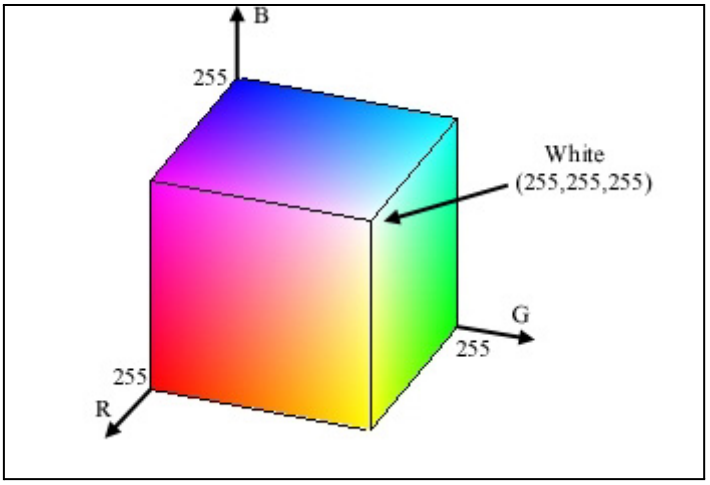


Figure 24: 3D Representation of RGB Colour Space

With a 3D coordinate for two colour samples, we would then be able to calculate the euclidean distance between the two colour samples and this distance metric helps us determine similarity between two colours. This leads us to the k-nearest neighbours (KNN) algorithm.

KNN is a simple yet powerful algorithm used for classification and regression tasks in many applications such as machine learning. It operates on the principle that similar data points tend to belong to the same class or have similar numerical values.

For us to make use of KNN, we first collect our dataset comprising sampled red, green and blue values for each coloured waypoint in each maze of each table in the lab. The samples collected can be found under Appendix I.

For our mBot, we are using the Euclidean distance metric to classify the colour of newly sampled colour. To simplify our process, we made use of excel to find the average intensity

of Red, Green and Blue for each Waypoint Colour. The result is 6 defined coordinates for each colour to be detected.

The KNN algorithm then calculates the Euclidean distance between the newly sampled colour and each of the defined colours, selecting the defined colour that gives us the smallest distances.

The calculation for Euclidean distance is as follows:

$$Distance\ Between\ Two\ Points\ in\ 3D\ Space\ = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

For example, in the figure below, we have defined coordinates for the colours: white, blue, green, red, orange, yellow. When we measure a new colour that gives us the coordinate of “new” as shown in the figure, we compare the Euclidean distance between “new” and all other defined coordinates. Since white gives us the smallest Euclidean distance, we can classify the newly sampled colour as white.

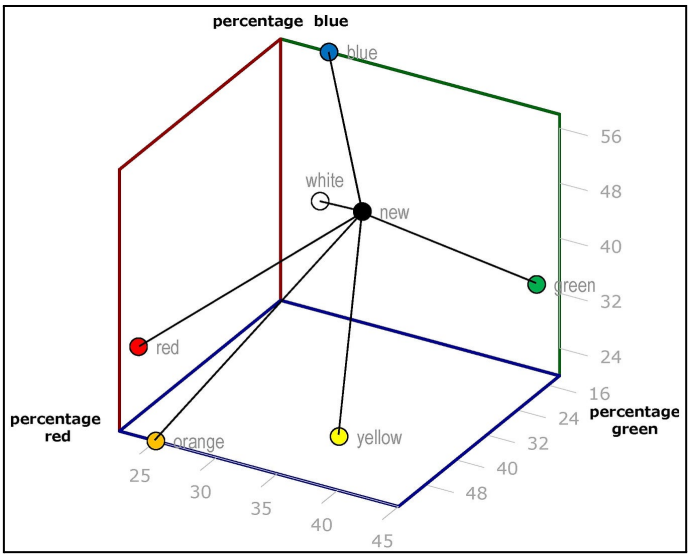


Figure 25: K-Nearest Neighbour using Euclidean Distance Metric

8.3.2 Implementation

First step of implementing colour detection is to calibrate our colour detector sensor by obtaining the analogue voltage across the LDR when we are sampling a white surface and black surface. This is done using the calibration program that was used in the previous studio on photoresistors.

The code implementation to control the individual LED colours and read the analogue voltage across the LDR is shown in the next two images.

The following code declares the necessary arrays for controlling the 2-to-4 decoder IC chip to turn respective LEDs ON and OFF.

```
/********** Color Detection **********/
// Pins controlling 2-4 Decoder
int ledArray[2] = { A2, A3 };

// Truth Table to control 2-4 Decoder (for LED Control):
int truth[3][2] =  { { 0, 1 },      // Blue LED ON
                     { 1, 0 },      // Green LED ON
                     { 1, 1 }       // Red LED ON, also used for IR Emitter OFF
                   };

char colourStr[3][5] = {"B = ", "G = ", "R = "};  // array of strings to aid debugging of measured RGB values
```

The following code implementation allows us to shine each coloured LED and measure LDR voltage for each respective LED. The getAvgReading() function is called to measure the LDR voltage.

```
/********** Functions (Color Detection) **********/

/**
 * Function turns on one colour at a time and measure LDR voltage for each col
 * Estimated RGB values are stored in global array colourArray.
 */
void read_color() {
  for(int c = 0; c <= 2; c++){
    // Serial.print(colourStr[c]);
    // TURN ON LIGHT
    for (int i = 0; i < 2; i++) {
      digitalWrite(ledArray[i], truth[c][i]);
    }
    delay(RGBWait);
    // 1 Average Reading is taken for color measurement as our color detection
    colourArray[c] = getAvgReading(1);
    //the difference between average reading and calibrated values for black s
    int result = (colourArray[c] - blackArray[c])/(greyDiff[c])*255;
    if (result > 255) {
      result = 255;
    }
    else if (result < 0) {
      result = 0;
    }
    colourArray[c] = result;
    // TURN OFF LIGHT
    for (int i = 0; i < 2; i++) {
      digitalWrite(ledArray[i], 0);
    }
    delay(RGBWait);
    // Serial.println(int(colourArray[c])); //show the value for the current c
  }
```

Function for getting average LDR reading:

```
/**
 * This function finds the average reading of LDR for greater accuracy of LDR readings.
 * @param[in] times Number of times to iterate and repeat before finding average
 * @return Returns the averaged LDR values across times iterations
 */
int getAvgReading(int times) {
  //find the average reading for the requested number of times of scanning LDR
  int reading;
  int total = 0;
  //take the reading as many times as requested and add them up
  for (int i = 0; i < times; i++) {
    reading = analogRead(LDR);
    total = reading + total;
    delay(LDRWait);
  }
  //calculate the average and return it
  return total / times;
}
```

Now we get into the classification portion of our subsystem. We have created a compound datatype using struct so we can group several variables into a single variable.

Next, we collect colour samples from every waypoint in the lab using the calibrated colour sensor. In total, we have 8 datasets for each colour and the average is taken as the defined colour for each label. The data collected can be found under the section on Appendix.

The declaration of said compound data type is as follows:

```
// Struct for each label to be used for color classification
struct Color {
  String name;
  int id; // 0 - white, 1 - red, 2 - blue, 3 - green, 4 - orange, 5 - purple
  uint8_t red;
  uint8_t green;
  uint8_t blue;
};

// Data points for each color, obtained by taking average of collected color samples.
// To be used in KNN classification using euclidean distance.
Color colors[] = {
  //  Label/name -   id - R  -  G  -  B
  {   "Red",     1,  220,  110,  135 },
  {   "Blue",    2,  180,  212,  225 },
  {   "Green",   3,  160,  169,  128 },
  {   "Orange",  4,  195,  170,  120 },
  {   "Purple",  5,  160,  163,  170 },
  {   "White",   0,  255,  255,  255 }
};
```

After defining red, green and blue values for each colour, we now have to write out the code that does the comparison between the defined colours with our newly sampled red green and blue values.

For each comparison, we calculate the euclidean distance and keep track of the minimum distance and the corresponding id/label of the likely colour. After comparing with all six defined colours, we would simply return the label of the likely colour which has the smallest euclidean distance to our newly sampled colour.

```
/**
 * Function looks at RGB values stored in colourArray[] and compare it with defined points for each color stored in colors[].
 * Color is classified by finding calculating the Euclidean distance for each known color and select the one with the minimum distance.
 * @return Returns the color id of classified color. [0 - white, 1 - red, 2 - blue, 3 - green, 4 - orange, 5 - purple, 6 - unknown]
 */
int classify_color() {
  int classified = 6;
  measured_blue = colourArray[0];
  measured_green = colourArray[1];
  measured_red = colourArray[2];

  // Calculate Euclidean distances for each known color
  double minDistance = 9999;  // Initialize with a large value
  //String classifiedColor;

  for (int i = 0; i < 6; i++) { // 6 is the number of known colors
    double distance = sqrt(pow(colors[i].red - measured_red, 2) + pow(colors[i].green - measured_green, 2) + pow(colors[i].blue - measured_blue, 2));
    if (distance < minDistance) {
      minDistance = distance;
      //classifiedColor = colors[i].name;
      classified = colors[i].id;
    }
  }
  // Serial.println("Classified as: " + classifiedColor);
  return classified;
}
```

Euclidean Distance based K-Nearest Neighbour

8.3.3 Calibration and Finetuning

Now that we have understood the implementation of the K-Nearest Neighbour algorithm in code, we will be going through the steps taken to calibrate and improve the robustness of the colour detection sensor.

The accuracy of the colour detection is dependent on the dataset that was collected. In our case, we wanted our mBot to be able to adapt to all different lighting in the lab. Hence, we went to collect colour samples from every waypoint in the lab using the calibrated colour sensor. We store the collected colour samples in an excel file in which we will calculate the average colour values.

Another part of fine tuning the colour sensor is to improve the precision of our colour sensor by using a resistor of larger resistance as current limiting resistor for the LEDs. A larger resistance resistor means less light reflected off a black paper during the calibration process and therefore leads to a greyArray of larger value. With higher values in greyArray, there will be a larger range of values to estimate the light intensity of current colour detected by the LDR. This is to be elaborated in the section on Challenges later on.

When our mBot fails to detect a colour correctly, we would use the Serial Monitor to display the measured intensity of each colour and compare with our datasets, if the value is off by at least 20, we would include it in our datasets and effectively adjust the defined values for that particular colour, increasing the chance of a successful identification using KNN algorithm for that new colour.

In total, we have 8 datasets for each colour and the average is taken as the defined colour for each label. The data collected can be found under the section on Appendix.

Moreover, to aid our troubleshooting and testing of our colour sensing functionality, we included the code to change the mBot's onboard LED to reflect the classified colour. This has proved to be a very useful visual indicator of our colour sensor performance.

9. Challenges Faced

9.1 Bright LEDs

Initially, our current limiting resistors for red, green and blue LEDs have a rather low resistance value of 460 ohms. However, we noticed that the LDR was reading high values when the LEDs were shining on a black paper, which meant that there is a large portion of light being received by the LDR and the black paper is reflecting too much light. This meant that our LEDs were too bright and we had to increase the resistance of our current limiting resistors to lower the light intensity of our LEDs.

The image below shows the calibration results when we used 460 ohms resistors. We see that greyDiff array values are rather small and it is important to have a big range so that we can have higher precision when converting measured LDR voltage to respective Red, Green and Blue intensity. A good value to have for each colour in greyDiff is at least 300.

Initial values from calibration program (with 460 ohms current limiting resistors).

```
//floats to hold colour arrays
float colourArray[] = {0,0,0};
float whiteArray[] = {1000.00,994.00,984.00};
float blackArray[] = {939.00,811.00,943.00};
float greyDiff[] = {61.00,183.00,41.00};
```

After experimenting with different resistor values, we found that 6.8k ohms resistor increases the values in greyDiff to values above 350, which improves our colour sensor precision.

Final values from calibration program (with 6.8k ohms current limiting resistors)

```
// Float arrays to store calibrated values for color arrays
float colourArray[] = {0,0,0};
float whiteArray[] = {928.00,908.00,766.00};
float blackArray[] = {566.00,473.00,308.00};
float greyDiff[] = {362.00,435.00,458.00};
```

9.2 Skidding Wheels

In the midst of our testing, we noticed that the mBot skidded when performing a 90 degrees turn on a smoother table. This happened when the mBot executed a purple or blue waypoint which required the mBot to perform a 90 degree turn on the maze surface, rather than on the colour paper, which is of similar smoothness in every maze.

We noticed that the dust particles accumulated on the tyres reduce friction between the tyre and the maze surface. Our solution was to wipe away the dust on the tyres of our mBot every few rounds of testing or use. It was a simple but effective solution, without having to reduce our motor speeds.

9.3 Impact of Battery Voltage on Robot Performance

Throughout our testing, we noticed that after repeated use of the mBot, the movement of the mBot becomes less responsive, resulting in less precise movement. This is due to the depletion of battery voltage which has lowered the output voltages of the mCore. A lowered output voltage not only impacts basic movement of the mBot, but also the sensor performance.

Our solution to tackle this challenge is to ensure that the robot was fully charged before every lab session. This is important as we should reduce inconsistencies in movement as well as sensor performance so that our test runs are more reliable.

9.4 Low effective range for IR Proximity Detector

As we were testing the IR sensor circuit, we noticed that the IR Proximity Detector works for a short effective distance of roughly 4 cm. This is because the voltage across the detector does not scale linearly with distance beyond 4 cm. Hence we decided to prioritise using the ultrasonic sensor for the majority of the maze navigation as it would be difficult to translate the voltage across the IR detector into exact distance between wall and robot.

The diagram below shows the scatter plot of voltage across the detector against the distance between the sensor and maze wall.

Distance (cm)	Detector Voltage (V)
1	0.85
2	1.7
3	2.73
4	3.16
5	3.38
6	3.65
7	3.71
8	3.78
9	3.83
10	3.85
11	3.89
12	3.89
13	3.91
14	3.92
15	3.92

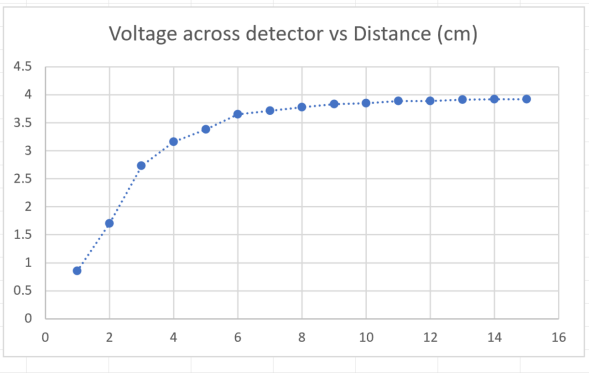


Figure 26: Voltage-distance relationship for IR detector

9.5 Impact of Ambient Lighting on Colour Detection Sensor

Despite the rigorous tuning and testing for our mBot, our mBot was unable to perform accurate colour detection in our second maze of the actual evaluation run. This is a good reminder that no sensor can be perfect and the performance of the KNN algorithm is based on how representative the datasets are to the actual environment. There will always be slight deviations in the environment with time and it so happens that these slight deviations prevented our KNN algorithm from classifying the colours accurately. This is also a reminder that more emphasis should be placed on designing the shielding so as to minimise the impact of ambient lighting on the colour detection sensor.

10. Conclusion

In conclusion, this project gave us the opportunity to apply our knowledge and skills learnt from past labs. This is evident in the design process of the circuits where we needed to apply electronics principles to reason and justify design considerations.

Moreover, this project required us to work with online documentation to better understand the components given to us. The ability to read and understand documentation is a crucial skill for an engineer as it helps with our design for both hardware assembly and software implementation.

Furthermore, this project gave us numerous opportunities to explore different ways of tackling problems. Some evident examples would be the use of K-Nearest Neighbour algorithm to solve a colour classification problem and the use of PID Controller to achieve smooth wall following.

Although we were unable to clear the second evaluation maze in the first attempt, it served as an important reminder that a sensor should undergo frequent calibration due to unforeseen changes in the external environment.

What we could have done better in:

- Better shielding and skirting to prevent the ambient light from reaching the LDR
- Greater emphasis on improving our datasets to improve reliability of colour detection sensor.

11. Division of Labour

Everyone contributed well in this project, notable contributions from each team member are listed below:

Name	Role
Chan Sheng Bin	• General Software Development and Testing • Implementation of Colour Detection Algorithm • Implementation of Wall-Tracking • General Construction of mBot • Written Report
Carvalho Andreus Roby	• Design of Maze-Solving Algorithm • Improvements to Shielding • Written Report
Cao Junbo	• Colour Detection Circuit Design • Implemented Victory Tune for Buzzer • Written Report
Bui The Trung	• Wire Management • IR Circuit Design

Appendix

Data Set for KNN Algorithm

Calibrated Values			
	R Value	G Value	B Value
WhiteArray	766	908	928
BlackArray	308	473	566
GreyDiff	458	435	362

Colour	R Value	G Value	B Value
Red	221	139	134
	217	127	130
	220	110	133
	233	121	136
	224	127	138
	218	124	127
	224	116	137
	217	125	137
Average	221.75	123.625	134
Green	141	165	124
	151	165	127
	147	175	134
	163	177	110
	166	160	128
	168	158	132
	165	172	129
	164	174	135
Average	158.125	168.25	127.375

Blue	186	215	226
	151	215	227
	178	205	220
	174	210	226
	166	216	228
	168	216	228
	174	214	223
	182	218	229
	178	211	217
Average	173	213.3333333	224.8888889
Orange	193	177	128
	227	175	124
	210	168	118
	216	177	113
	216	174	112
	207	164	133
	229	177	126
	220	174	147
Average	214.75	173.25	125.125
Purple	159	168	170
	162	168	181
	164	157	171
	156	149	169
	167	172	180
	178	174	183
	166	153	174
	172	158	170
Average	165.5	162.375	174.75
White	255	255	251
	251	255	255
	235	250	245
	247	255	248
	250	255	255
	255	253	255
	240	254	250
	255	255	255
Average	248.5	254	251.75