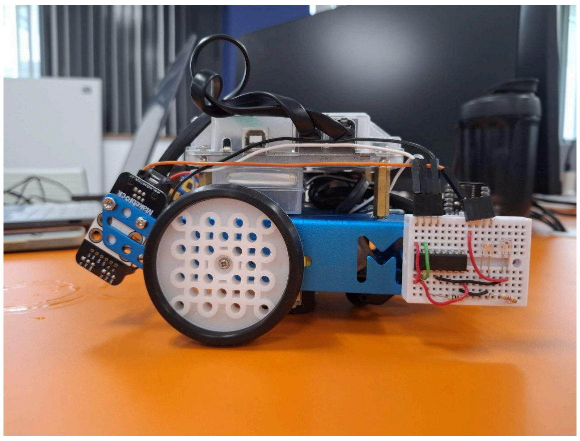
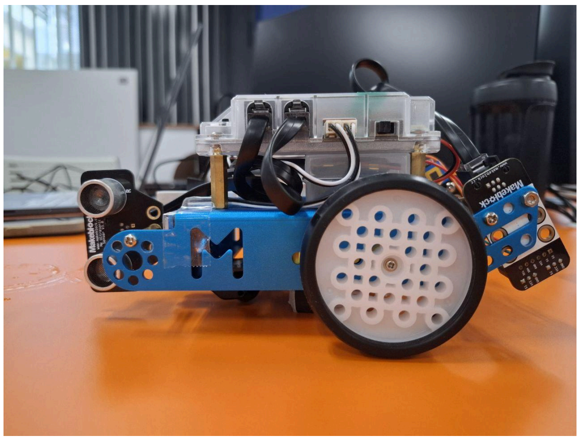
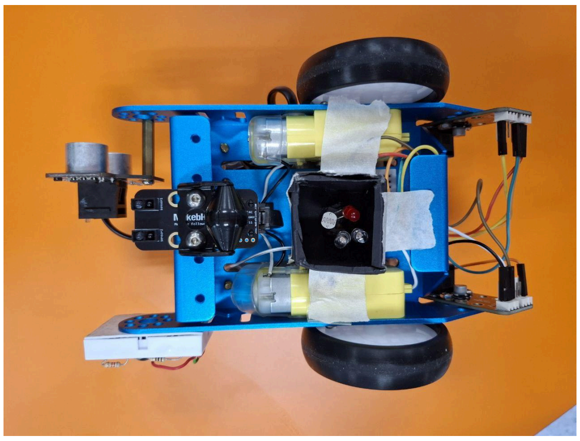
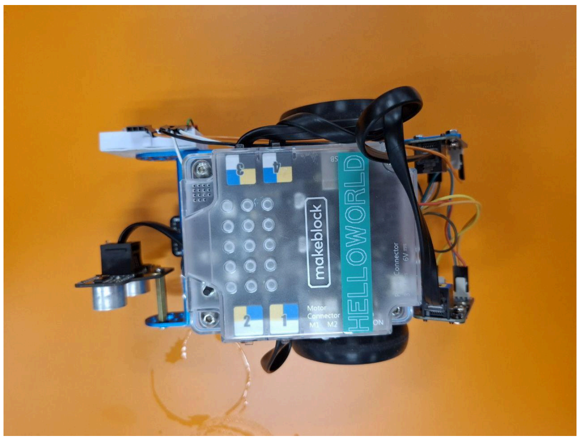
CG1111A A-maze-ing Race Project 2024

Studio Group No	B06
Section No	S2
Team No	T4 (Hello World)
Team Members	Zheng Kaiwen Zhu Wenbo Zhu Yicheng

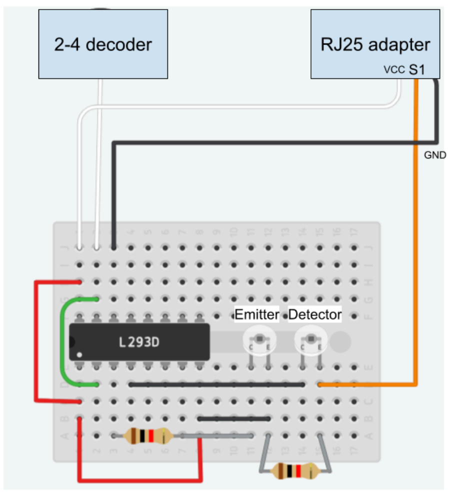
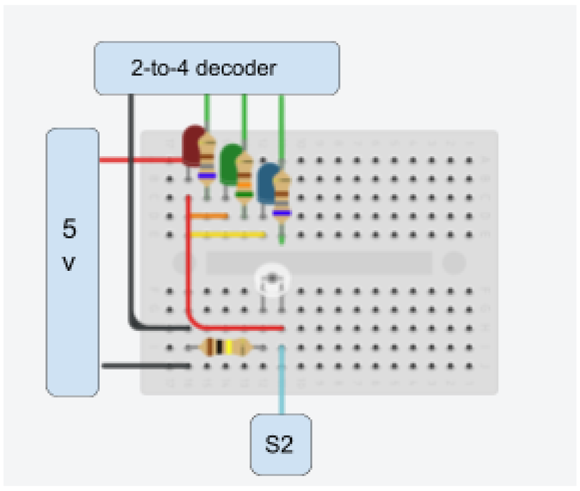
TABLE OF CONTENTS

1. Pictures

1.1. Pictures of the mBot

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

1.2. TinkerCAD model of the circuits used

IR sensor circuit	Colour sensing circuit
 2-4 decoder RJ25 adapter vcc S1 GND L293D Emitter Detector	

2. Overall algorithm

Full code: https://github.com/mendax1234/CG1111A-Final-Project

2.1. Flowchart of algorithm

The goal of the project is to navigate the mBot through a maze while ensuring it does not collide with the walls using the ultrasonic and IR proximity sensors, completing missions pre-determined by the coloured paper on the ground.

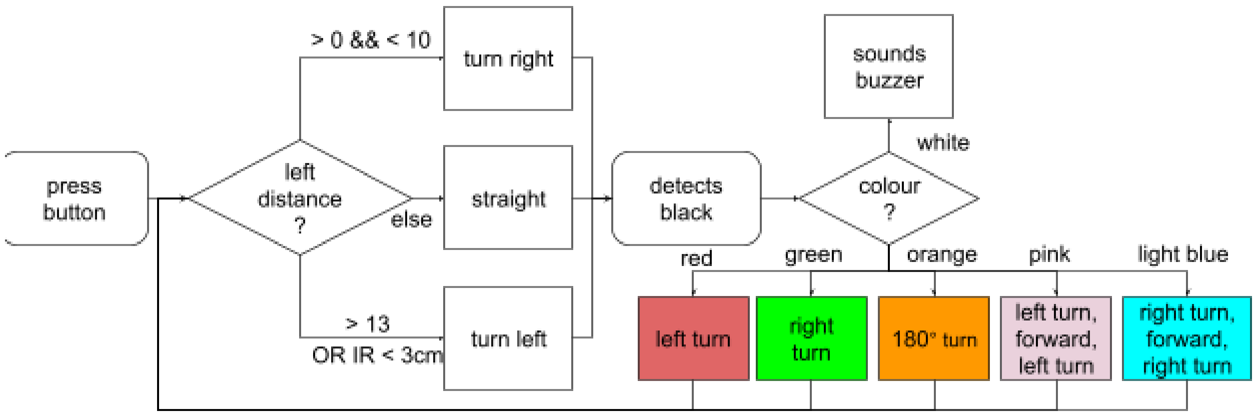


Figure 2.1: Overall Flow of the Algorithm

Our algorithm utilises the state machine to control the robot to do the challenges. The default state for our mBot is set to FORWARD. At the start of our program, we define a status for our onboard button and only when the onboard button is pressed, our robot will proceed to navigate through the maze. While doing so, it will keep track of the distance from the left. If the robot is too close to the left wall, it will speed up its left motor and slow down the right motor to correct to the right. Should the robot be too close to the right, it will do the opposite to correct itself. If there are no walls or the robot is just right between the walls, it will keep going straight.

Once the line detector detects a black line, the state will be changed to CHALLENGE, and the robot will stop. The red, green, and blue LEDs will then be turned on sequentially to read the colour of the paper. It will then do the challenge according to the colour that is detected, as seen in Figure 2.1.

Should the colour sensor detect white, it will sound the buzzer to signal the completion of the maze and set the state to FINISH. Otherwise, after the challenge is completed, the state will be set to FORWARD again and the robot will continue to navigate through the maze until the next challenge.

We will go into more detail on the individual algorithms in 3. Subsystem algorithm.

2.2. File Structure

To keep our code clean and easy to manage, we divided the project into multiple files. This leverages the Arduino IDE's property of compiling all files (except for the files under the subdirectory) in the same folder into a single project in alphabetical order. Each file groups related functions for specific purposes. The implementations are as follows:

File Name	Description
project.ino	The main program which contains the core control logic, including state machine control.
a_utility.ino	Colour matching algorithm
b_navigation.ino	Motion for the motors, including auto line correction & complex turning algorithms to overcome compound turn and fringe left turn cases
c_periperhals.ino	Buzzer sound & onboard Arduino display colour
d_sensors.ino	Take in values from sensors & output readings to be used for navigation & colour sensing
e_test_func.ino	Write functions to test constants
color_calibration.ino	This file contains code to calibrate our colour-sensing circuit. It is separated from the main project.ino and will be run separately from project.ino.

The folder structure:

```
.
├── README.md
├── a_utility.ino
├── b_navigation.ino
├── c_periperhals.ino
├── color_calibration
│   └── color_calibration.ino
├── d_sensor.ino
├── e_test_func.ino
├── images
│   ├── left-side.jpg
│   ├── maze.jpg
│   ├── no-walls.mp4
│   ├── right-side.jpg
│   ├── turn-colors.jpg
│   └── turns.jpg
└── project.ino
```

3. Subsystem algorithm

3.1. Navigation

3.1.1. Walk Straight

During our experiment, we noticed that our robot could not move in exactly a straight line even if we gave the two motors the same output. We think this is probably due to 1) the difference between the friction of the two wheels, and 2) the difference between the rotational speed of the two motors (even if we set the same speed for them). To simplify our analysis, we assume that the friction of the two wheels is nearly the same. So, to solve this problem, we just need to find the small difference between the speed of the two motors. This is easy to get since we just need to add two deviation constants and do some tests to get these constants. This is implemented in move_forward() (Code 3.1.1) below:

```
#define RIGHT_DEVIATION 20
#define LEFT_DEVIATION 0

void move_forward() {
  leftMotor.run(-motorSpeed - LEFT_DEVIATION);
  rightMotor.run(motorSpeed + RIGHT_DEVIATION);
}
```

Code 3.1.1: Walk Straight

Besides this function, we still have another method that helps our robot move in a straight line, which utilises the state machine control and our wall avoidance algorithm. We will dig into it later after we talk about the wall avoidance algorithm in section 3.1.2. and the structure of our state machine control in section 3.1.4.

3.1.2. Wall Avoidance

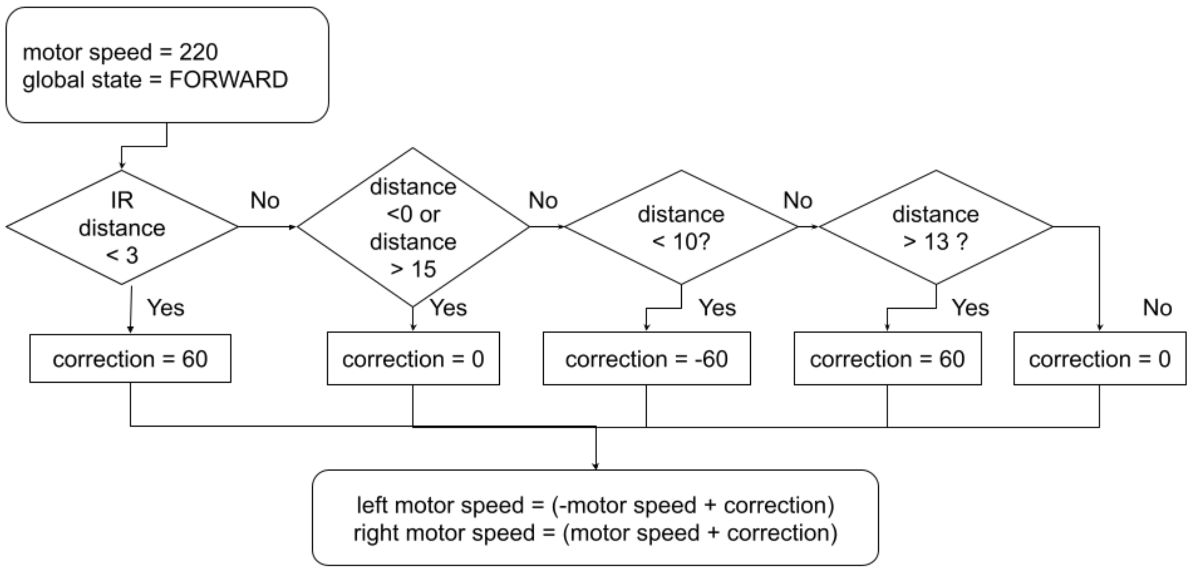


Figure 3.1.2 Overall Flow of the Wall Avoidance Algorithm

Referring to the idea of using the PID algorithm, we think the core of the PID algorithm is to get the so-called correction, and this correction will be based on the “error”, which is our distance from our desired range (in our project, the setpoint is 10-13cm). With this correction value, we can add it to our motor speed to correct the deviation from our desired range so that we can always maintain our robot in the desired range, which is called “wall avoidance” in our project.
Firstly, based on Figure 3.1.2, we implement our desired range in within_range() (Code 3.1.2a) below:

```
/* Auto correction function */
int within_range() {
  double distance = left_distance();
  int ir_diff = ir_read();
  if (ir_diff > IR_TOO_CLOSE) {
    // IR Correction
    return CORRECTION_SPEED;
  }
  if (distance < 0.0 || distance > 15.0) {
    // No wall
    return 0;
  }
  if (distance < 10) {
    // Too close
    return -CORRECTION_SPEED;
  }
  if (distance > 13) {
    // Too far
    return CORRECTION_SPEED;
  }
  // Within Range
  return 0;
}
```

Code 3.1.2a: Within Range

Next, to simplify the analysis (In the PID method, the only thing that is different is the PID algorithm will calculate a varying CORRECTION_SPEED), we hard-coded the CORRECTION_SPEED to 60 based on our experiment. Using this speed, we can modify our move_forward() a bit to take the correction into account. The idea is that, if the robot is close to the left wall, it needs to turn right, this is achieved by increasing the left motor speed and decreasing the right motor speed. In our code, it is shown that we always add the correction to both our motors and if the robot is too close to the left wall, the CORRECTION_SPEED will be negative and vice versa. It is implemented in our move_forward_correction() (Code 3.1.2b) below:

```
void move_forward_correction(int correction) {
  leftMotor.run(-motorSpeed + correction);
  rightMotor.run(motorSpeed + correction);
}
```

Code 3.1.2b: Move Forward with Correction

The advantage of this approach is that the correction algorithm is speed-based rather than angle-based. Instead of making the mBot turn by a specific angle to avoid walls (this will introduce some delay to the whole system), this method dynamically adjusts the motor speeds while the mBot is in motion. This results in smoother and more fluid movement for the mBot.

3.1.3. Turning Degree

Since the turning degree function is achieved by specifying the time for the two motors to rotate at the same speed, to simplify our future work as well as make our code clean and neat, we decided to write a function to instruct the robot to turn a certain degree using the time constant, which is called TURNING_TIME_MS. This constant means the time (in ms) for the robot to turn 90 deg. Our code to implement this function is as follows (Code 3.1.3):

```
void turn_deg(int side, int angle) {
  // 0: Turn Left, 1: Turn Right
  if (side == 0) {
    leftMotor.run(motorSpeed);
    rightMotor.run(motorSpeed);
    delay((double)angle / 90.0 * TURNING_TIME_MS);
    stop();
  } else {
    leftMotor.run(-motorSpeed);
    rightMotor.run(-motorSpeed);
    delay((double)angle / 90.0 * TURNING_TIME_MS);
    stop();
  }
}
```

Code 3.1.3: Turn a certain degree

3.1.4. State Machine Control

As mentioned above, we have used state machine control in our robot code since it makes it clean, neat, and easy to debug. The main state in our robot is the Motion state, we have also defined the colours as enums, and they are implemented as follows (Code 3.1.4a):

```
enum Color {
  C_BLUE, C_GREEN, C_RED, C_ORANGE, C_PINK, C_WHITE
};
enum Motion {
  TWO_RIGHT, TURN_RIGHT, TURN_LEFT, U_TURN, TWO_LEFT,
CHALLENGE, FORWARD, FINISH
};
```

Code 3.1.4a: Motion state and colour state

These Motion states represent the different states of our mBot. The default state is FORWARD. The CHALLENGE state indicates that we are about to start the colour challenge and the FINISH state indicates that we have finished the maze running. The remaining five colour challenge states are mapped with the colour enums using the code shown as follows (Code 3.1.4b):

```
global_state = static_cast<Motion>(predicted_color);
```

Code 3.1.4b: Casting between Motion state and Predicted Colour

At the end of each colour challenge state, we will change our global state back to FORWARD. The whole state machine control logic is implemented as follows (Code 3.1.4c):

```
if (global_state == FORWARD) {
  int correction = within_range();
  if (correction == 0) move_forward();
  else move_forward_correction(correction);
  display_color(C_WHITE);
  if (has_reached_waypoint()) {
    stop();
    global_state = CHALLENGE;
  }
} else if (global_state == CHALLENGE) {
  readColor();
  int predicted_color = colour();
  display_color(predicted_color);

  if (predicted_color == C_WHITE) {
    stop();
    celebrate();
    global_state = FINISH; // Terminates program
  } else {
    global_state = static_cast<Motion>(predicted_color);
  }
}
else if (global_state == TURN_LEFT) turn_left_time();
else if (global_state == TURN_RIGHT) turn_right_time();
else if (global_state == U_TURN) uturn_time();
else if (global_state == TWO_LEFT) compound_turn_left();
else if (global_state == TWO_RIGHT) compound_turn_right();
delay(10);
```

Code 3.1.4c: State Machine Control Logic

Now, we will reveal the trick about how this state machine control can also help our mBot walk in a straight line. Notice that in our default state FORWARD, the moving logic is shown as follows:

```
if (global_state == FORWARD) {
  int correction = within_range();
  if (correction == 0) move_forward();
  else move_forward_correction(correction);
  display_color(C_WHITE);
  if (has_reached_waypoint()) {
    stop();
    global_state = CHALLENGE;
  }
}
```

Code 3.1.4d: Moving Logic under FORWARD state

First, we will calculate our correction using our wall avoidance algorithm within_range(), and if the return value (correction) is 0, we will use the move_forward() function we have introduced in section 3.1.1. to walk straight without applying any correction. Otherwise, we will apply the move_forward_correction() we have introduced in section 3.1.2. to adjust our robot so that it will not bump into the wall. We separated move_forward() and move_forward_correction() because adding deviations and corrections together to the movement logic caused unexpected behaviour, so separating these two functions makes our mBot move more smoothly in a straight line.

3.1.5. Implementing Navigation Logic

We defined some constants initially related to the motor:

	Constant	**Description**	
1	TURNING_TIME_MS	● The time for our mBot to turn 90 degrees	
2	LEFT_DEVIATION	● Used to correct the direction of the mBot ● Set to 0	Based on our experiment, our mBot has the tendency to turn right while moving in a straight line. This effect can be minimised by adding a deviation speed to the right motor, making it move slightly faster than the left motor. This will help the robot move straight forward without using our correction algorithm.
3	RIGHT_DEVIATION	● Used to correct the direction of the mBot ● Set to 20 to deviate to the right	

4	CORRECTION_SPEED	• Used to make our robot avoid bumping into the wall
5	TWO_LEFT_TURN_TIME_MS	• Used for Pink Challenge • Time for the mBot to stop before doing the next action in the Pink Challenge. This can make the action more accurate.
6	TWO_RIGHT_TURN_TIME_MS	• Used for Light-Blue Challenge • Time for the mBot to stop before doing the next action in the Light-Blue Challenge. This can make the action more accurate.
7	IR_TOO_CLOSE	• The IR Diff value when mBot is too close to the right wall • Set to 600 based on our experiment

3.2. Colour sensing

All colours are created by combining different amounts of the three primary colours: red, green, and blue. The colour of an object is determined by the intensity of red, green, and blue light reflected from its surface. For instance, a red object absorbs green and blue light while reflecting primarily red light, allowing our eyes to perceive it as red. Similarly, a colour sensor measures the reflected intensities of red, green, and blue light and identifies the object's colour by comparing the readings to the characteristic RGB components of light reflected by specific coloured surfaces, such as red, orange, pink, green, blue, and white tiles.

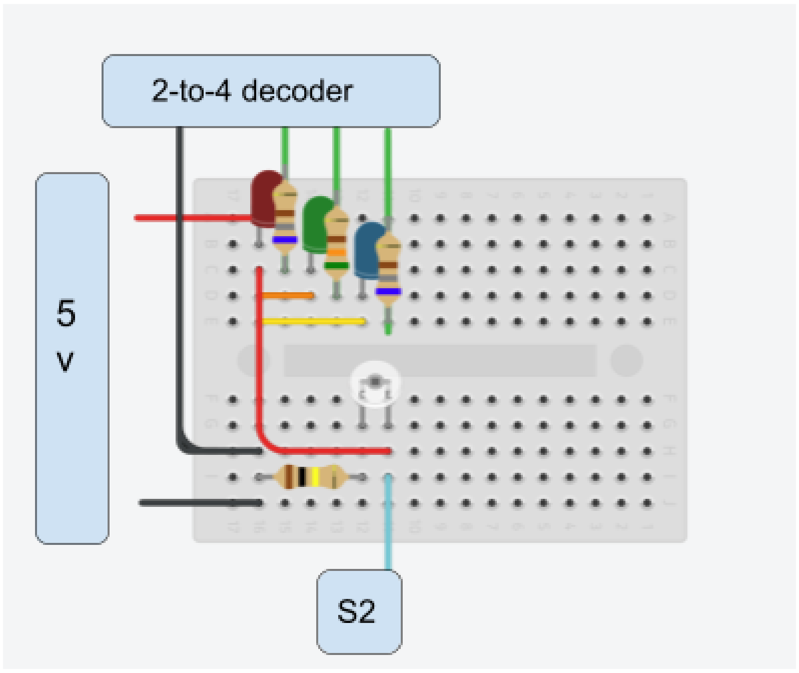


Figure 3.2a: Circuit for Color Sensor

In this project, we built a colour sensor using three separate LEDs, an LDR and a 2-to-4 decoder:

- Due to the limited number of available pins, we utilised a 2-to-4 decoder, allowing the control of three LEDs using only two pins (ldr_adaptor) from the mBot. The behaviour is as follows:
    - When S1 is high and S2 is high, the red light is activated.
    - When S1 is low and S2 is high, the green light is activated.
    - When S1 is high and S2 is low, the blue light is activated.
- At fixed intervals, we toggled the state (HIGH or LOW) of the two pins to switch on each LED. We then measured the voltage divider reading of the LDR through the ir_adaptor S2 pin.
- The program calculates these values to match the RGB colour code, with each value ranging from 0 to 255. To calibrate the colour sensor, we measured the white and black colours of the field. The white measurement represents the readings when all light is reflected, while the black measurement represents the readings when most light is absorbed. The greyscale is then determined by subtracting the black values from the white values.
- The resistors used in the circuit are as follows: 680Ω for the red light, 530Ω for the green light, 680Ω for the blue light, and 100kΩ for the LDR. These resistance values were chosen based on calculations to achieve a higher grayDiff value, indicating greater sensitivity.

As the values of the analog pin of the Arduino range from 0 - 1023 (0V - 5V), we have to convert the values to resemble the RGB colour code. The equation looks like this:

$$\frac{colourArray - blackArray}{greyDiff} * 255$$

In the getAvgReadingLDR(int times) function, the LDR is sampled three times, and the average is returned. Initially, we sampled it five times but found this to be time-consuming. Ultimately, we settled on three samples as a balance between accuracy and efficiency.

In the readColor(), the flowchart looks like below:

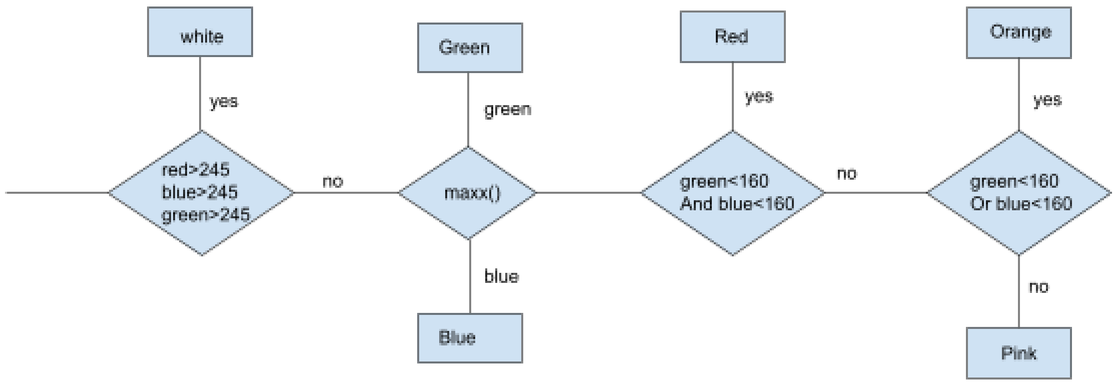


Figure 3.2b: Flowchart for readColor()

To give out this way to get colour. We measured the RGB value of each colour.

	Value of Red	Value of Green	Value of Blue
White	258	259	257
Blue	116	220	232
Green	124	228	150
Red	236	106	66
Orange	238	206	80.5
Pink	258	231	232

Table 3.2: RGB values for each colour challenge

After measuring the RGB value of each colour, we find that, for green, the value of green is the highest. And for blue, the value of blue is the highest. Then, for red, orange, and pink, the value of red is the highest. This is why we write a program called maxx() to detect the maximum value of a colour, the implementation (Code 3.2a) is shown as follows:

```
int maxx(){
  // Find the three colours with the highest proportion
  int max = 0;
  for (int c = 1; c <= 2; c++){
    if (colourArray[c] > colourArray[max]){
      max = c;
    }
  }
  return max;
}
```

Code 3.2a: Detect the maximum value of a colour

When the function returns 1, it indicates that the highest value corresponds to green, identifying the colour as green. Similarly, when it returns 2, the highest value corresponds to blue, identifying the colour as blue.

However, for red, orange, and pink, the highest value is always red. To differentiate these colours, we use the values of green and blue:

- Red: Both green and blue values are below 160.
- Orange: Either green or blue value is below 160.
- Pink: Both green and blue values are above 160.

Based on the above analysis, we implement our colour-matching algorithm (Code 3.2b) as follows:

```
int colour(){
  // Map the color measured within range
  for (int c = 0; c <= 2; c++) {
     colourArray[c] = (colourArray[c] - blackArray[c]) / greyDiff[c] *
255;
  }
  // Detect color
  int maxColor = maxx();
   if (colourArray[0]>245 && colourArray[1]>245 && colourArray[2]>245)
return C_WHITE;
  if (maxColor == 2) return C_BLUE;
  if(maxColor == 1) return C_GREEN;
  if (colourArray[1] < 160 && colourArray[2] < 160) return C_RED;
  if (colourArray[1] < 160 || colourArray[2] < 160) return C_ORANGE;
  return C_PINK;
}
```

Code 3.2b: Colour-matching algorithm

3.2.1. Colour Sensing Improving Robustness

Version	Description
Version 1: Does not have anything	Initially, we did not have a box around the LEDs and LDR. In this case, we found that the environment will affect the outcome a lot. Even if a person is standing beside the robot, the accuracy of it would be very low.

Version 2: Black box around the breadboard	In this case, we made a big box which surrounds the entire bottom of the machine. However, we find that the big box surrounding the entire bottom of the machine would cause drag force to the machine and affect its movement. If we decrease its length, there would be some light leakage into the box affecting the accuracy of LDR.
Version 3: Transparent adhesive	We then used transparent tape at the bottom of the box since the tape has a smooth surface which would decrease the drag. But the tape will reflect the light which also affects the accuracy of detecting colour.
Version 4: Small box surrounds LEDS and LDR	Finally, we chose to use a small box that nicely surrounds the LEDs and LDR. Since the area of the box touching the ground becomes smaller, the drag force becomes lower. It has a lower effect on the movement of the robot and little light leakage from the bottom.

3.3. IR Proximity Sensor

While the mBot is able to do most of the navigation using only the ultrasonic sensor, there are fringe cases where we cannot count on the ultrasonic sensor, i.e., when there is no wall on the left and the robot is about to hit the wall on the right. As such, we made full use of the assortment of sensors available to us by using the IR proximity sensor to aid the ultrasonic sensor in navigation as well.

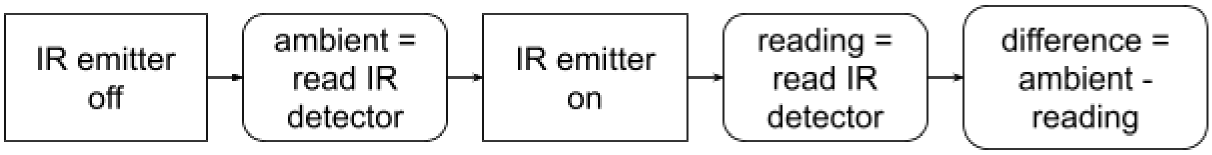

IR emitter off
ambient = read IR detector
IR emitter on
reading = read IR detector
difference = ambient - reading

Figure 3.3: IR sensor algorithm

As the IR detector will give extremely inconsistent values if read directly from, we instead take the difference between the reading with the IR emitter on and the ambient reading with the IR emitter off. This allows us to obtain more consistent values regardless of the change in ambient light. The function we implemented below (Code 3.3) will return the difference and the robot will analyse it to check if it is too close to the wall on the right.

```
#define IRWait 5
int ir_read() {
  ldr_adapter.dWrite1(HIGH);
  ldr_adapter.dWrite2(LOW);
  delay(IRWait);
  int ambient = ir_adapter.aRead1();
  ldr_adapter.dWrite1(LOW);
  ldr_adapter.dWrite2(LOW);
  delay(IRWait);
  int reading = ir_adapter.aRead1();
  int difference = ambient - reading;
  return difference;
}
```

Code 3.3: ir_read()

This allows our IR sensor to be used in conjunction with the ultrasonic sensor should the ultrasonic sensor fail when there is no wall on the left side. The robot will correct to the left if the IR sensor detects that it is too close to the wall.

More explanation on how we calibrated our IR sensor to help us navigate through will be explained below in section 4.2.

3.4. mBot LED

At the end of our maze running, we have designed some fancy functions to utilise the Buzzer to play the classic Nokia Melody and the Mario Game Melody while changing the LED colour according to the melody. To achieve this, we implement the function below in Code 3.4

```
int colorIndex = 0;  // Color index for cycling colors

void playNoteWithColor(uint16_t note, uint32_t duration) {
  int nextColorIndex = (colorIndex + 1) % 10;
  int *currentColor = ledColorsBuzzer[colorIndex];
  int *nextColor = ledColorsBuzzer[nextColorIndex];

  smoothColorTransition(currentColor, nextColor, duration);
// Smoothly transition LED colors
  buzzer.tone(note, duration);  // Play the note for its
duration

  colorIndex = nextColorIndex;  // Update the color index
}
```

Code 3.4a: Play the Melody with LED Colour

The smoothColorTransition() is implemented as follows:

```
// Interpolate and gradually change colors
```

```
void smoothColorTransition(int startColor[], int endColor[],
uint32_t duration) {
  const int steps = 50;  // Number of steps for a smooth
transition
  for (int i = 0; i <= steps; i++) {
    int r = startColor[0] + (endColor[0] - startColor[0]) * i
/ steps;
    int g = startColor[1] + (endColor[1] - startColor[1]) * i
/ steps;
    int b = startColor[2] + (endColor[2] - startColor[2]) * i
/ steps;
    RGBled.setColor(r, g, b);
    RGBled.show();
    // delay(duration / steps);
  }
}
```

Code 3.4b: Smooth the Colour Transition

With these two helper functions, we can replace the default buzzer.tone() with our custom playNoteWithColour() to play the melody with LED colour changing.

3.5. mBot Line Finder

The mBot has a default line finder which is mounted on PORT_2 by us to detect whether we have reached the black line or not.

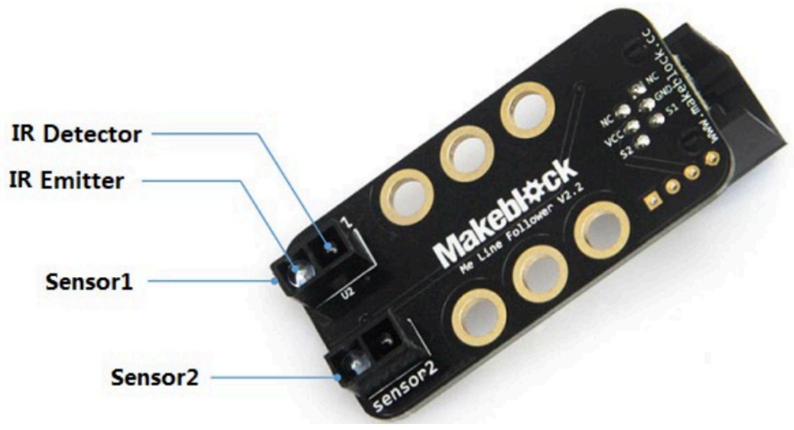


Figure 3.5: mBot Line Finder

This line finder will be useful since it can let mBot stop at the black stripe at the front of each colour tile. Our basic implementation is shown as follows (Code 3.5):

```
bool has_reached_waypoint() {
  /* Check if mBot line sensor has detected black line on the
floor */
  int sensor_state = lineFinder.readSensors();
  if (sensor_state == S1_OUT_S2_OUT)
  {
    return true;
  }
  return false;
}
```

Code 3.5: Implementation of mBot line finder

However, this is not our final version. The final version can be seen in section 6.3.

4. Calibrations

4.1. Colour Sensor

To calibrate the colour sensor, we only need to change the blackArray and the grayArray which is used to calculate the RGB value of the colour. In this case, we write a setBalance function (Code 4.1a) to calibrate the colour sensor. The code is shown below.

```
void setBalance(){
  // Calibration: To get whiteArray, blackArray and greyDiff
  Serial.println("Put White Sample For Calibration ...");
  delay(5000);
  Balance(0);
  Serial.println("Put Black Sample For Calibration ...");
  delay(5000);
  Balance(1);
  for(int i = 0; i <= 2; i++) {
    greyDiff[i] = whiteArray[i] - blackArray[i];
  }
  Serial.println("Colour Sensor Is Ready.");
  delay(1000);
}
```

Code 4.1a: Calibrate the colour sensor

In our Balance() function, we will use getAvgReadingLDR() to read the RGB values for our whiteArray and blackArray respectively. The implementation is shown as follows:

```
void Balance(int type){
  // 0: For White, 1: Black
  // Turn ON RED LED
  ldr_adapter.dWrite1(HIGH);
  ldr_adapter.dWrite2(HIGH);
  delay(RGBWait);
  if (type == 0) {
    whiteArray[0] = getAvgReadingLDR(3);
  } else if (type == 1) {
    blackArray[0] = getAvgReadingLDR(3);
  }
  // Turn ON GREEN LED
  ldr_adapter.dWrite1(LOW);
  ldr_adapter.dWrite2(HIGH);
  delay(RGBWait);
  if (type == 0) {
    whiteArray[1] = getAvgReadingLDR(3);
  } else if (type == 1) {
    blackArray[1] = getAvgReadingLDR(3);
  }

  // Turn ON BLUE LED
  ldr_adapter.dWrite1(HIGH);
  ldr_adapter.dWrite2(LOW);
  delay(RGBWait);
  if (type == 0) {
    whiteArray[2] = getAvgReadingLDR(3);
  } else if(type == 1) {
    blackArray[2] = getAvgReadingLDR(3);
  }
  delay(RGBWait);
}
```

Code 4.1b: Read the RGB values for whiteArray and blackArray

4.2. IR Proximity Sensor

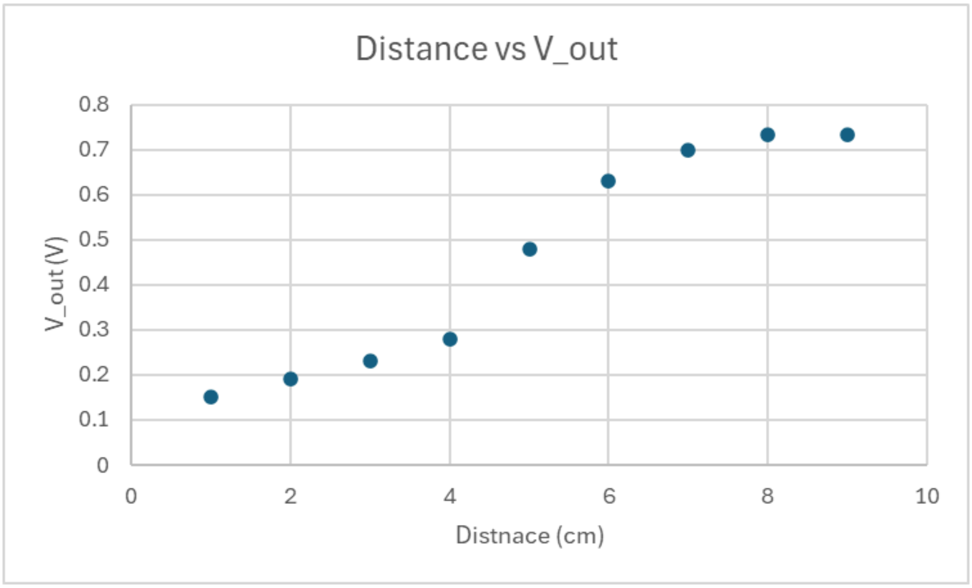


Figure 4.2: Voltage from IR detector vs distance(cm)

From the testing done on our circuit, the IR detector is most accurate up to around 8 cm (Figure 4.2). Due to the inconsistent nature of the IR detector (likely due to constant change in surrounding ambient light), we opt to only use the IR detector when the IR detector is extremely close (<4cm) to the wall as a backup plan for the ultrasonic sensor.

4.2.1 Interpolation Table

Based on our experiment, we construct the interpolation table for our IR difference vs distance as follows:

IR diff	814	791	735	696	643	564	510	473	479
Distance	1cm	2cm	3cm	4cm	5cm	6cm	7cm	8cm	9cm

Table 4.2.1: The interpolation table for IR difference vs. distance

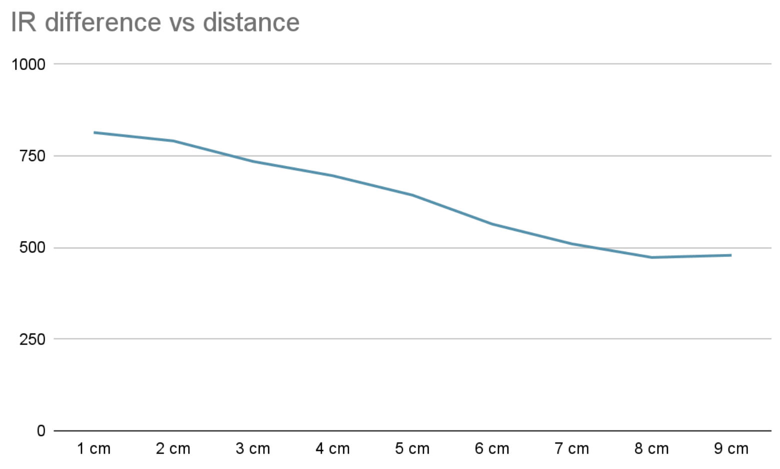


Figure 4.2.1: graph of IR difference vs distance from IR detector to wall

As seen above in the graph of IR difference (ambient minus reading when the emitter is on) vs distance, the relationship is roughly linear up to around 8 cm. Thus, we are able to use an interpolation table to estimate the distance between the robot to the wall on the right.

```
const int numValues = 9;
double xValues[10] = { 814, 791, 735, 696, 643, 564, 510,
473, 479 }; // Analog output for IR
double yValues[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }; //
Distance in cm
```

```
double getDistanceIR(float x)
{
  // The last boolean flag "clamp", true will limit the y
output in the range of the known values.
  double distance = Interpolation::Linear(xValues, yValues,
numValues, x, false);
  return distance;
}
```

Code 4.2.1: interpolation function to obtain the distance

Due to the linear relationship, we were able to use the interpolation library in Arduino to interpolate the analogRead values from the IR detector to the actual distance to estimate how close our robot is to the right and adjust accordingly when it is too close.

5. Work Division

Members	Roles
Yicheng	1. Designed and connected the IR sensor circuit. 2. Coded the IR detection algorithm 3. Contributed to the group report
Wenbo	1. Coded the navigation algorithm 2. Designed the state machine control to coordinate the operation of each mBot component. 3. Came up with the interpolation table method used in IR sensor 4. Contributed to the group report
Kaiwen	1. Designed and connected the colour-sensing circuit. 2. Coded the colour detection algorithm and colour calibration 3. Contributed to the group report

6. Challenges Faced

6.1. Inconsistent Colour Reading

While carrying out the calibration of colour readings, some problems arose that caused the robot to be unable to detect the correct colour correctly. We found out that there are three common problems:

Problems	Solutions
● The positions of the LEDS and LDR kept changing as the robot moved around. ● This causes the change in whiteArray and greyDiff, which would greatly affect the result.	● We recalibrated the colour values after every few runs. ● Although tedious, this is the most effective way to minimise the problem.
● The connection of the wire is loose. This would cause some LEDs' inability to light up. ● When the LEDs are unable to light up, the robot cannot obtain the correct colour values.	● We taped down the wires to ensure secure connections and the wires would not come loose.
● The pins of some LEDs come in contact which causes a short circuit. ● Since all LEDs are very close, their pins often contact each other.	● The tape was wrapped around the legs of the pins that often touched each other.

6.2. Poor accuracy of IR sensor

The IR sensor is often known to be inaccurate due to the amount of value affected by ambient light. For instance, when we were testing the V_{out} of the detector vs the distance to the object, someone standing nearby could easily affect the values. This is our first exposure to the unreliability of the IR sensor.

Originally, we chose to keep the IR emitter constantly on and analogRead the IR detector values directly. However, we quickly realised how that may be problematic. When we were testing the analogRead values from the IR detector with a piece of wall, we noticed that there was often a sudden change in values whenever we moved the IR detector out of the wall's shadow. This makes the value we obtain extremely inconsistent and can be affected by something as trivial as the brightness of the direction the robot is facing.

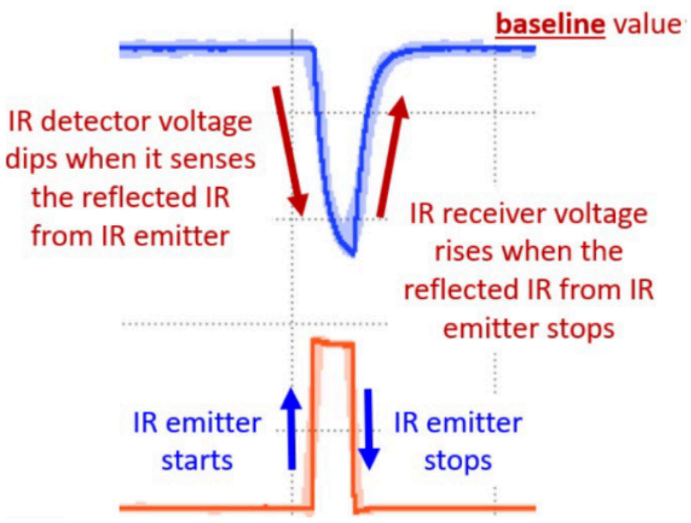

baseline value
IR detector voltage dips when it senses the reflected IR from IR emitter
IR receiver voltage rises when the reflected IR from IR emitter stops
IR emitter starts
IR emitter stops

Figure 6.2: Effect of turning on and off the IR emitter

As such, we decided to account for the ambient light in our code. We turn our IR emitter off and on intermittently to instead obtain the difference between the reading and ambient light as our new measurement of distance. This method, albeit not perfect, mostly accounts for the change in ambient lights in different environments (Figure 6.2). The ambient light still does affect the reading to an extent, but nowhere near the magnitude of reading the values directly.

6.2.1 Effect of IR detection on the overall code

One unexpected problem we faced when implementing our new IR algorithm was the delay it caused to the overall flow of our code. As we kept turning our IR emitter off and on, originally with a delay of 100ms, the delay caused our robot to display unnatural behaviours such as bumping into the wall when it was supposed to stop for the black line. We theorised that this is due to the delay causing an overall lag in our robot's algorithm. However, reducing the delay will mean that the IR reading will be less accurate, due to the lack of time for the IR reading to stabilise.

One potential solution to the delay issue is to use multiple threads in our robot code, with each thread handling a specific task, such as reading IR data, controlling the LED, and so on. However, based on our research, Arduino does not support native multi-threading, and its alternative approach appears to be somewhat time-consuming for us to test.

In the end, we decided it was not worth sacrificing the entire code for IR accuracy. We reduced the delay of IRwait to only 5ms. While the average IR readings increased by a magnitude of around 100, it is still workable since we only need the IR detector to sense when the robot is extremely close to the right and does not need it to be super accurate. With the reduced delay, however, the robot now has no problem with its overall algorithm and no longer suffers the bumps it previously displayed.

6.3. Poor accuracy of hard-coded motions

In this project, our mBot is required to perform specific hard-coded motions after detecting a colour. Since these motions are pre-programmed, they can easily become inconsistent and the consequence is that our mBot will easily bump into the wall. However, if the constants we have tuned are correct, we found that the mBot can perform the motions more accurately as long as it stops precisely at the "black line." As discussed in Section 3.5., our initial version of has_reached_way_point() does not achieve this functionality. Therefore, we further analysed the "state" change when the mBot reaches the black line. According to the mBot documentation, there are four distinct states, as outlined below:

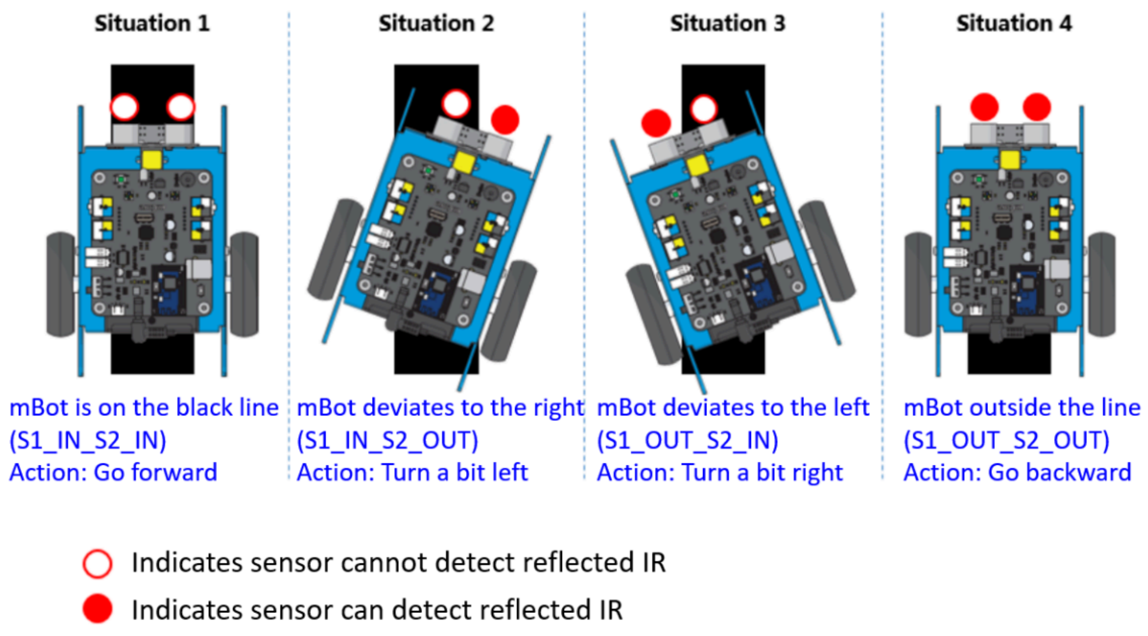


Figure 6.3a: The four different states for the mBot line finder

The ideal state transition should be from situation 4 directly to situation 1. However, in practice, external factors may cause the mBot to deviate from a straight path, leading to state transitions from situation 2 to situation 1 or from situation 3 to situation 1. To account for this error, we made a slight modification to our `has_reached_way_point()` function as follows:

```
bool has_reached_waypoint() {
  int sensor_state = lineFinder.readSensors();
  if (sensor_state != S1_OUT_S2_OUT)
  {
    if (sensor_state == S1_IN_S2_OUT)
    {
      // Turn left a bit
      turn_deg(0, 20);
    }
    else if (sensor_state == S1_OUT_S2_IN)
    {
      // Turn right a bit
      turn_deg(1, 20);
    }
    return true;
  }
  return false;
}
```

Code 6.3: Calibrate mBot at the black line

Now, our mBot can stop almost precisely at the black stripe in front of the colour tile, allowing it to follow the hard-coded motions more accurately.