Table of Contents

Table of Contents..... 1

1. Mbot Design & Construction.....3

2. Hardware Systems..... 6

2.1 Ultrasonic Sensor Setup and Calibration..... 7

2.2 IR Sensor Setup and Calibration.....8

2.3 LDR Sensor Setup and Calibration.....10

This is our final 2-to-4 decoder mapping, to ensure the IR emitter is always on..... 12

2.4 Line Sensor.....13

2.5 Wire Management..... 14

3. Algorithm..... 15

3.1 Config.h.....15

3.2 Main.h.....18

3.3 Proximity.h.....26

3.4 Motion..... 28

3.5 Waypoint.h.....31

3.6 Buzzer.h.....36

4. Calibration.....37

4.1 Colour Sensor Calibration.....37

4.2 Infrared (IR) Distance Sensor.....37

4.3 Ultrasonic Distance Sensor.....37

4.4 Movement Calibration.....38

5. Reflections..... 39

5.1 Work Division..... 39

5.2 Significant Challenges..... 40

5.3 Acknowledgements..... 42

1. Mbot Design & Construction

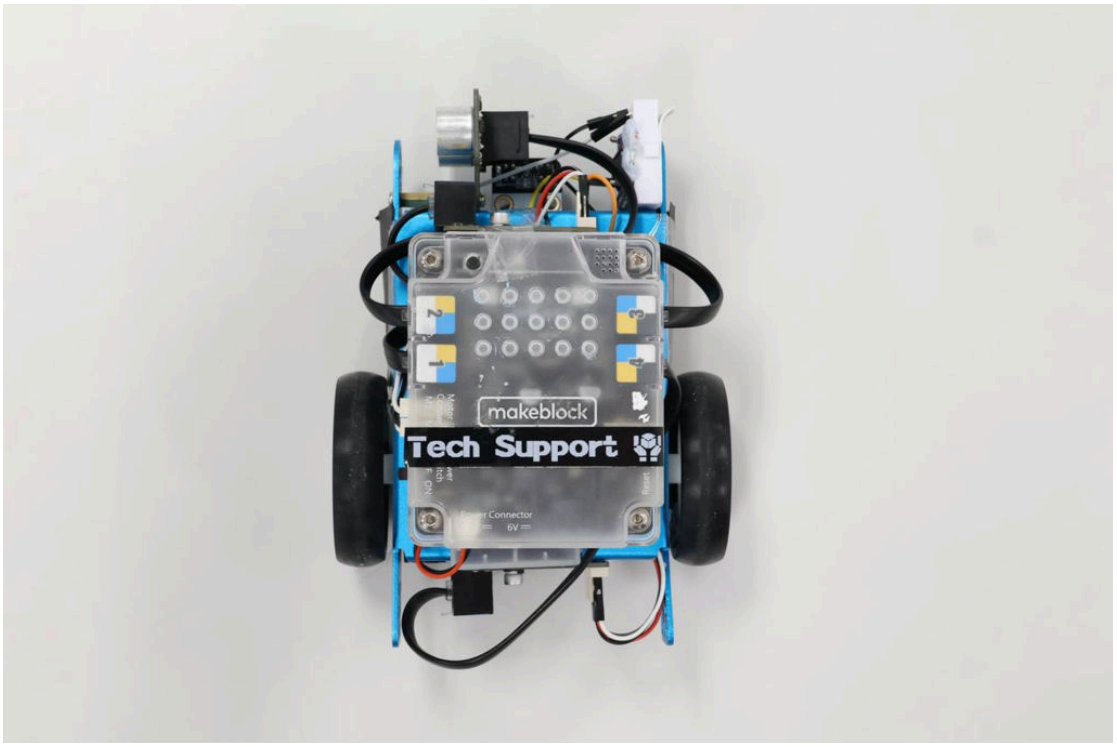


Figure 1 – Bird's Eye View of MBot

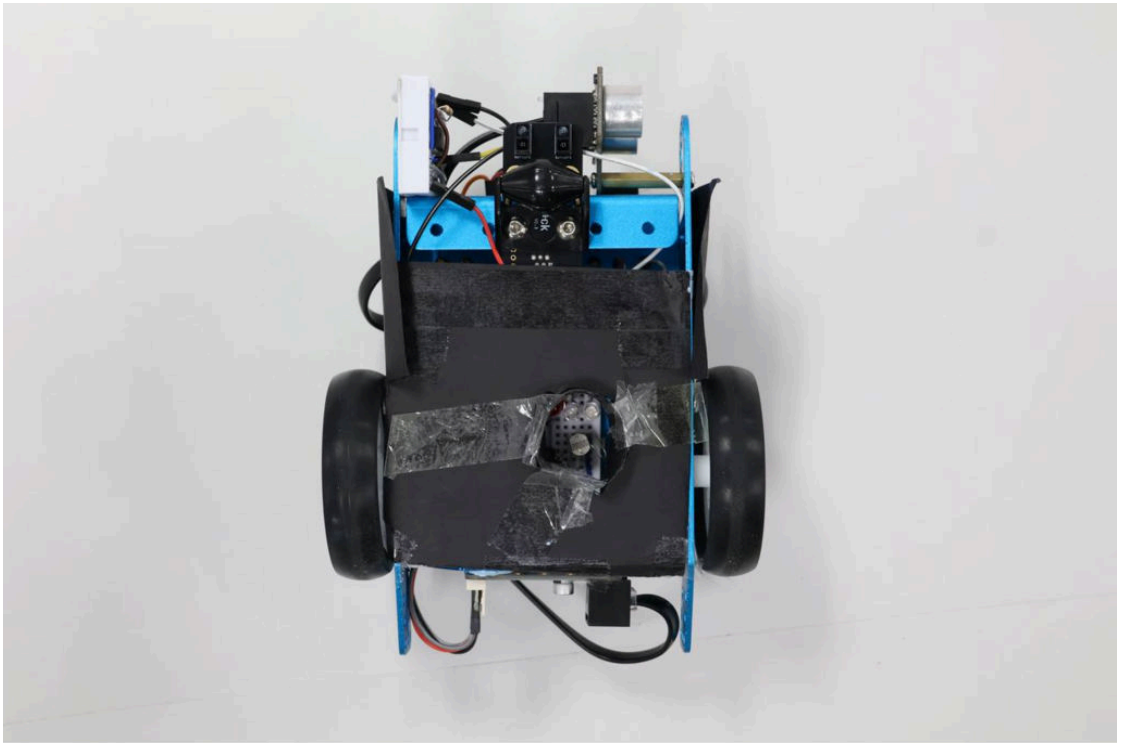


Figure 2 - Underside View of MBot with Skirting

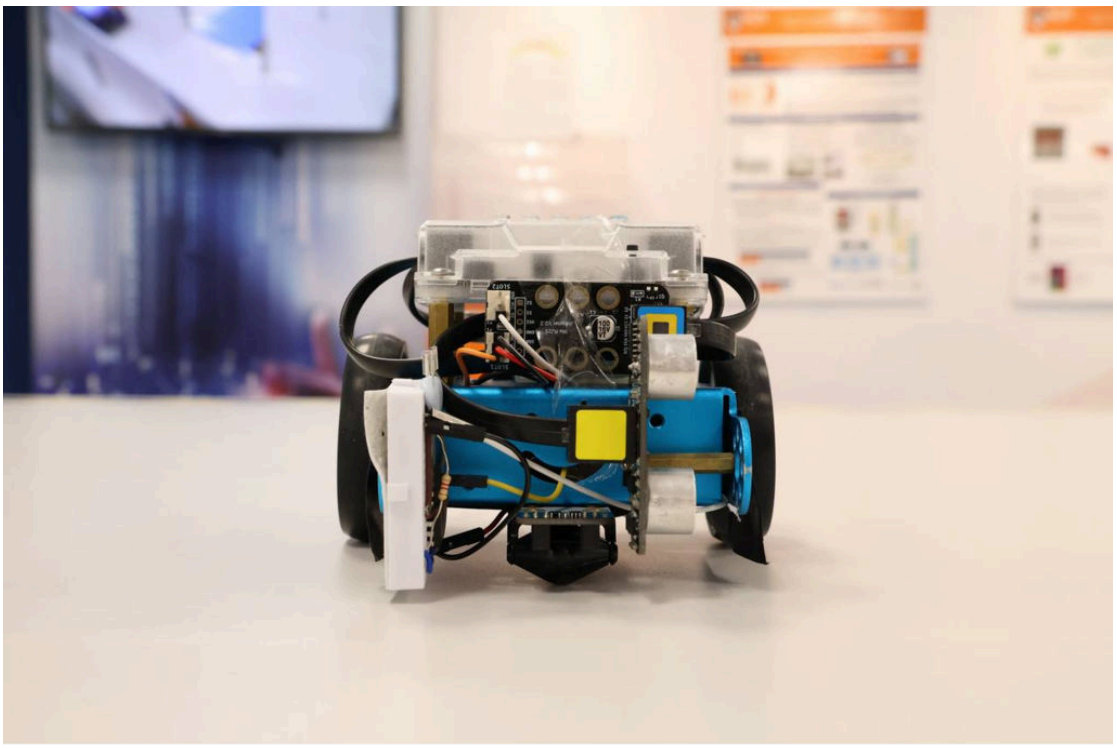


Figure 3 - Front View of MBot

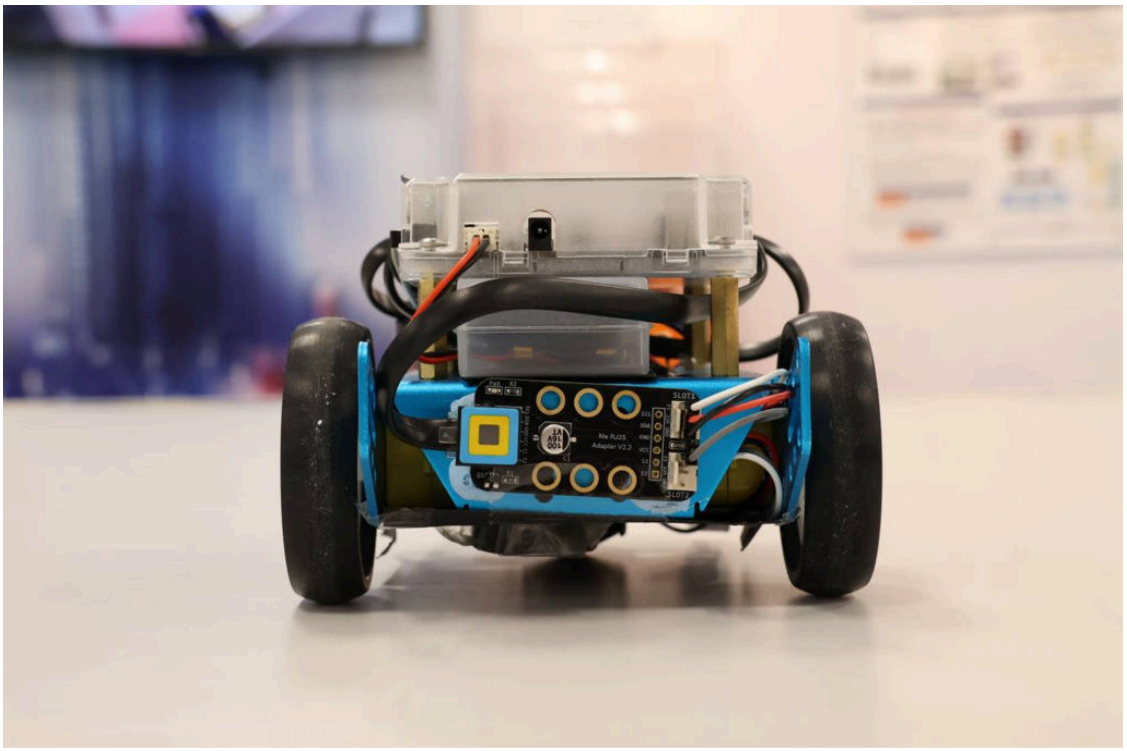


Figure 4 - Rear View of MBot

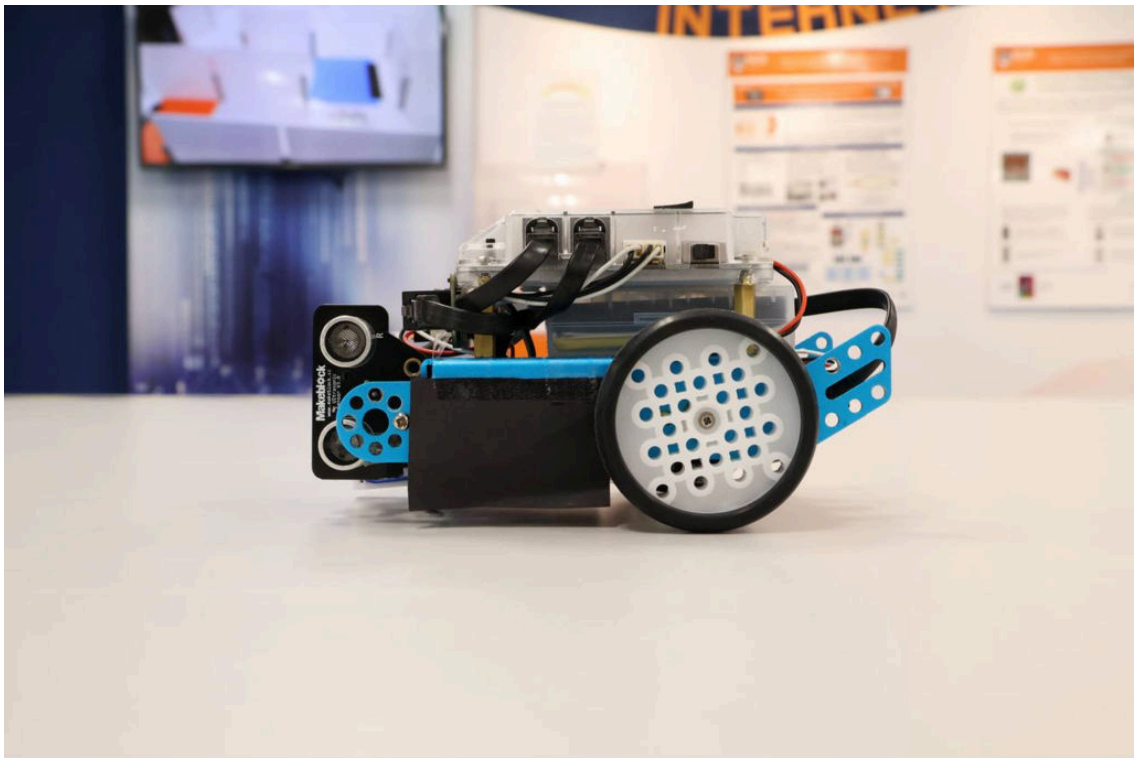


Figure 5 - Left View of MBot

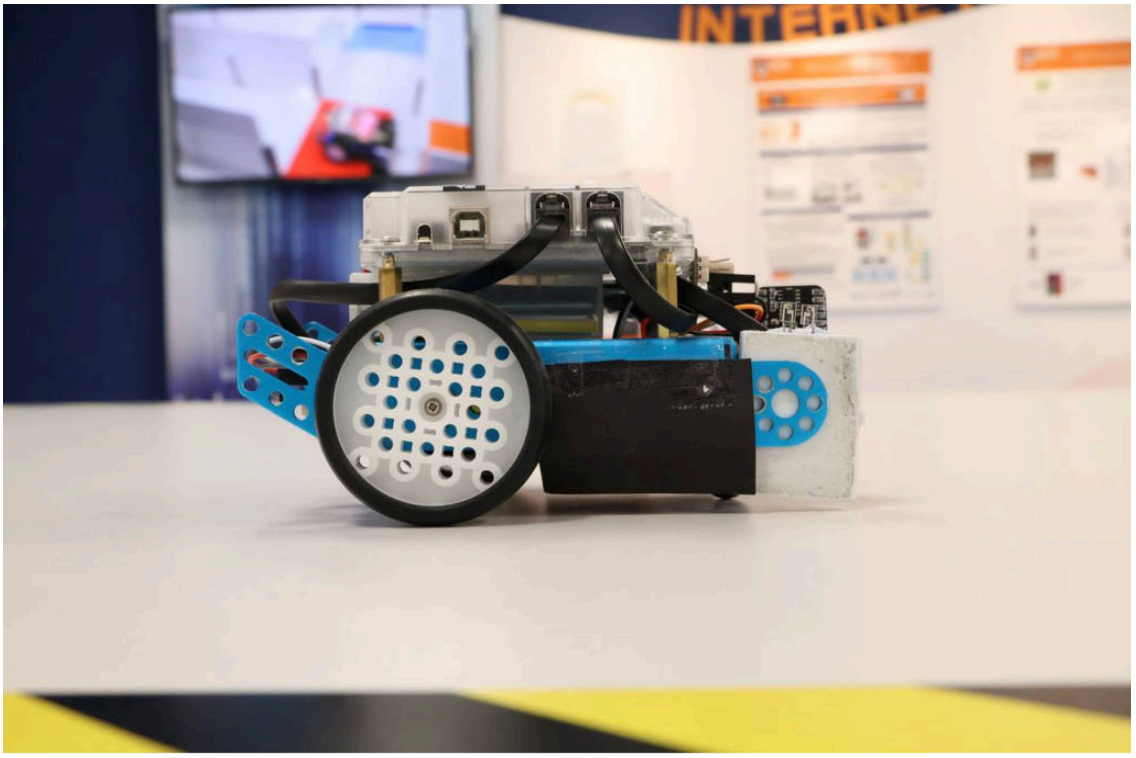
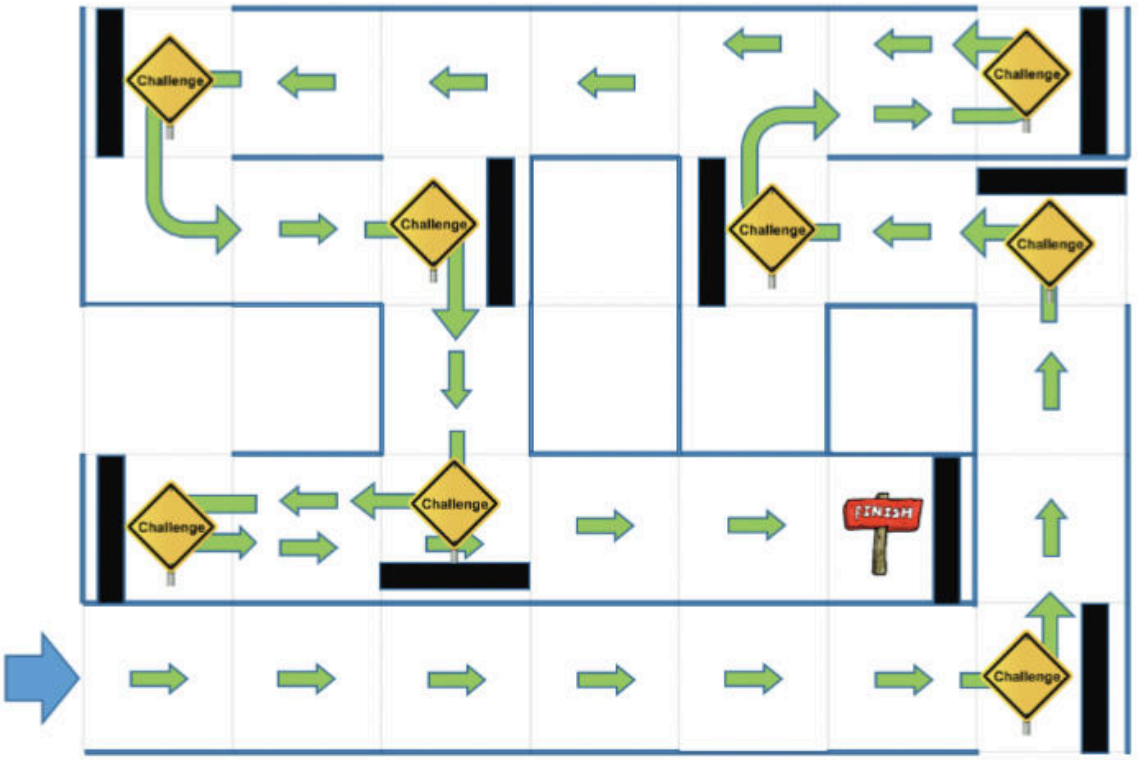


Figure 6 - Right View of MBot

2. Hardware Systems

Our objective is to navigate the mBot through a maze in the shortest possible time, executing all required waypoint actions while maintaining a stable trajectory between walls. Upon reaching the end of the maze, it should play a song to signal successful completion. A sample maze layout is as follows:

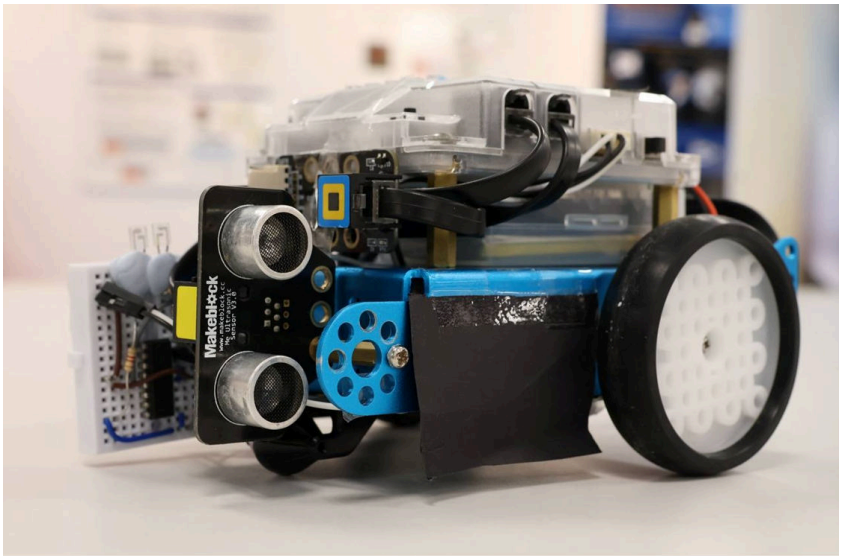


To achieve a workable solution, we identified several key hardware-related sub-problems that needed to be addressed:

1. Ultrasonic sensor setup and calibration
2. Infrared (IR) sensor setup and calibration
3. Light Dependent Resistor (LDR) sensor setup and calibration

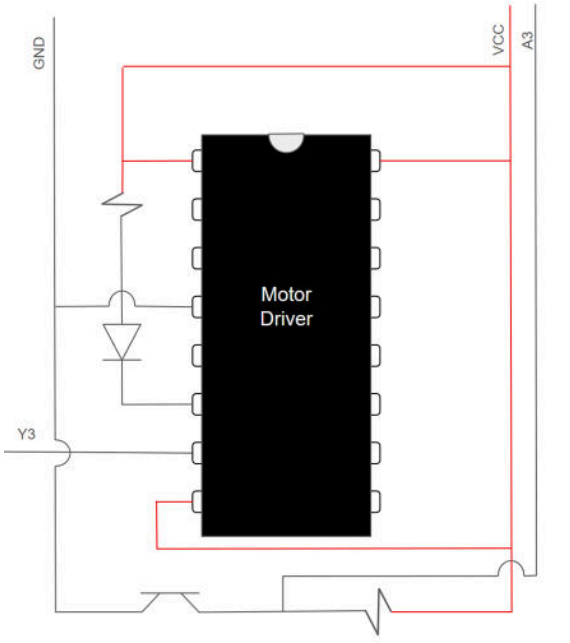
2.1 Ultrasonic Sensor Setup and Calibration

The ultrasonic sensor is directly plugged into the RJ25, facing towards the left side of the robot. The module is securely screwed onto the front of the robot in order to better detect collisions and respond faster to nearby walls. The straight positioning of the sensor ensures accurate distance readings throughout the maze and prevents ultrasonic waves from rebounding and giving false readings when the wall is not present.

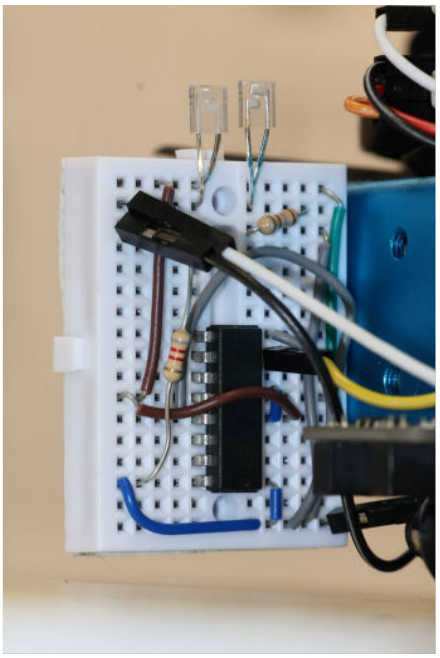


Ultrasonic Setup

2.2 IR Sensor Setup and Calibration



IR Circuit Diagram



Picture of IR Sensor Circuitry

On the right side of the robot, we have the IR emitter and sensor mounted on a breadboard, both facing the right wall. The IR emitter is powered via the Y3 output of the 2-to-4 chip from the other breadboard and is connected in series with a resistor to prevent it from blowing. The IR sensor is powered directly from the VCC and GND rails, and is also in series with a resistor. Analog pin A3 is connected at the junction between the sensor and resistor to measure the strength of the infrared pulse reflected from nearby surfaces.

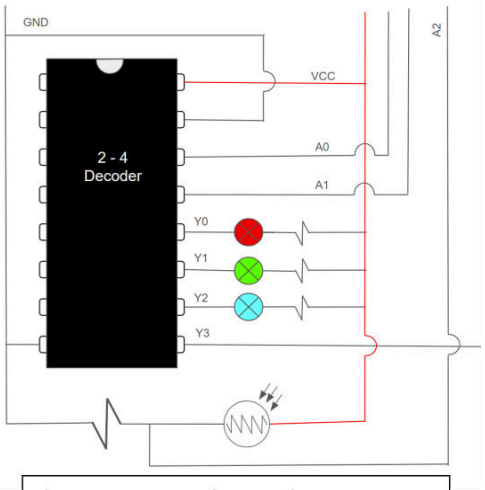
The breadboard itself is connected to the second RJ25 module, which supplies the VCC and GND lines for the IR circuit. During runtime, the Y3 pin is kept active while the robot is moving forward, ensuring that the IR emitter remains on to enable last-minute collision detection. The resistor for the IR emitter was calculated using a maximum

current of 50 mA, assuming a forward voltage drop of 1.5V. The resistor for the IR sensor was chosen based on Studio 12 reference values: 150Ω for the emitter, $8.2k\Omega$ for the detector

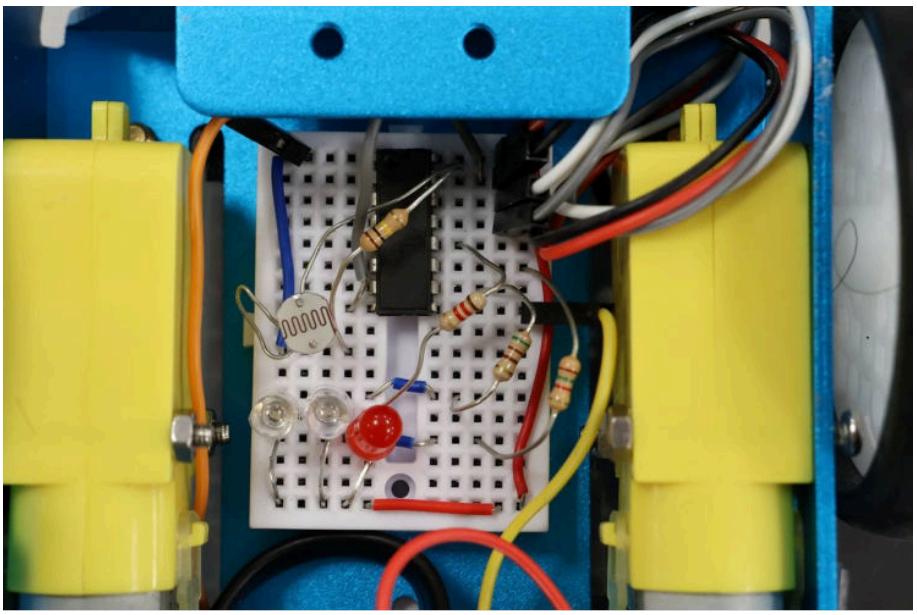
2.3 LDR Sensor Setup and Calibration

At the base of the robot, a breadboard houses the entire LDR-based color detection system. Red, green, and blue light emitting diodes (LEDs) are connected to three of the outputs (Y0 to Y2) of a 2-to-4 decoder, each in series with a resistor to prevent current overload. The decoder receives control signals from the mBot via analog pins A0 and A1 of the RJ25 interface, allowing direct digital control of the LED outputs.

On a separate track, the LDR is connected in series with a resistor, forming a voltage divider. A wire from the junction of this divider connects to analog pin A2 on the RJ25, enabling analog readings from the LDR. The VCC and GND rails on the breadboard are powered directly through the RJ25 interface. To minimize ambient light interference, both the LEDs and the LDR are enclosed within a black cover.



Colour Detection System Circuit Diagram



Picture of Colour Detection System Circuit

The LDR resistance was referenced from past studios at 100k ohms. The theoretical resistor values for the LEDs were calculated using data sheet specifications, assuming an average forward current of 20 mA and a supply voltage of 5V from the mBot:

$V_R = 1.9\,V\ at\ 20\ mA$

$Red: \frac{5}{20 \times 10^{-3}} \times \frac{5-1.9}{5} = 155\Omega$

$V_{B/G} = 3.2\,V\ at\ 20\ mA$

$Blue/Green: \frac{5}{20 \times 10^{-3}} \times \frac{5-3.2}{5} = 90\Omega$

However, later, we realised that our values were not providing enough range so, through a process of trial and error, our final resistance values were:

$V_R = 7.5k\Omega$

$V_G = 1.3k\Omega$

$V_B = 750\Omega$

The explanations for this are more detailed in our later sections on color detection.

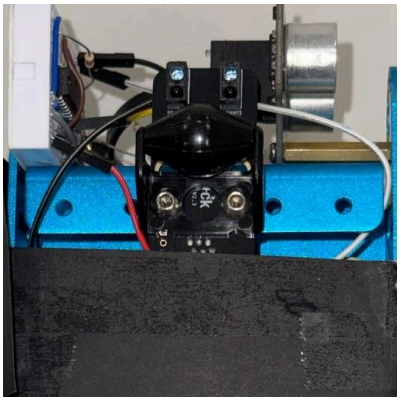
Table of 2-to-4 decoder mappings

Input 1 (Pin A0)	Input 2 (Pin A1)	Y0	Y1	Y2	Y3	Active
H	H	L	H	H	H	Green LED
H	L	H	L	H	H	Red LED
L	H	H	H	L	H	Blue LED
L	L	H	H	H	L	IR Emitter

This is our final 2-to-4 decoder mapping, to ensure the IR emitter is always on.

2.4 Line Sensor

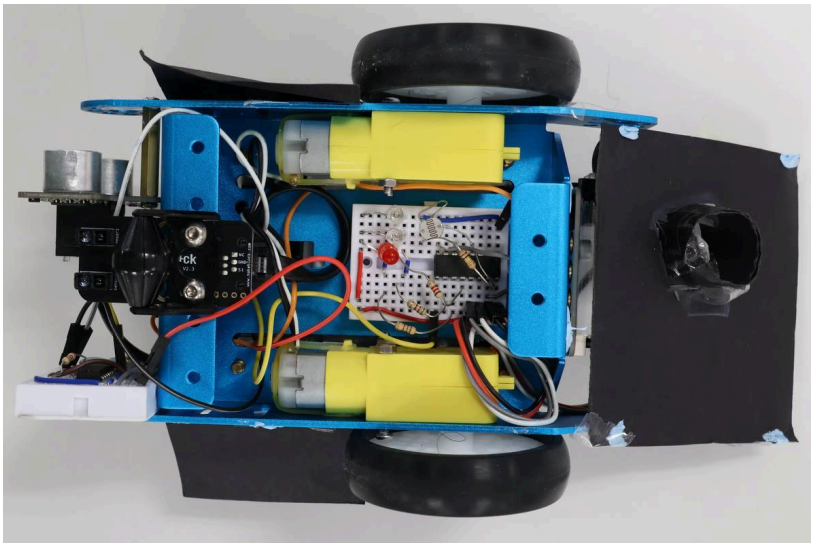
A black line sensor is placed near the front of the robot in order to detect the presence of a black line while the robot is moving forward. It is placed as far forward as possible on the robot in order to reduce the likelihood of collisions with walls in front of it.



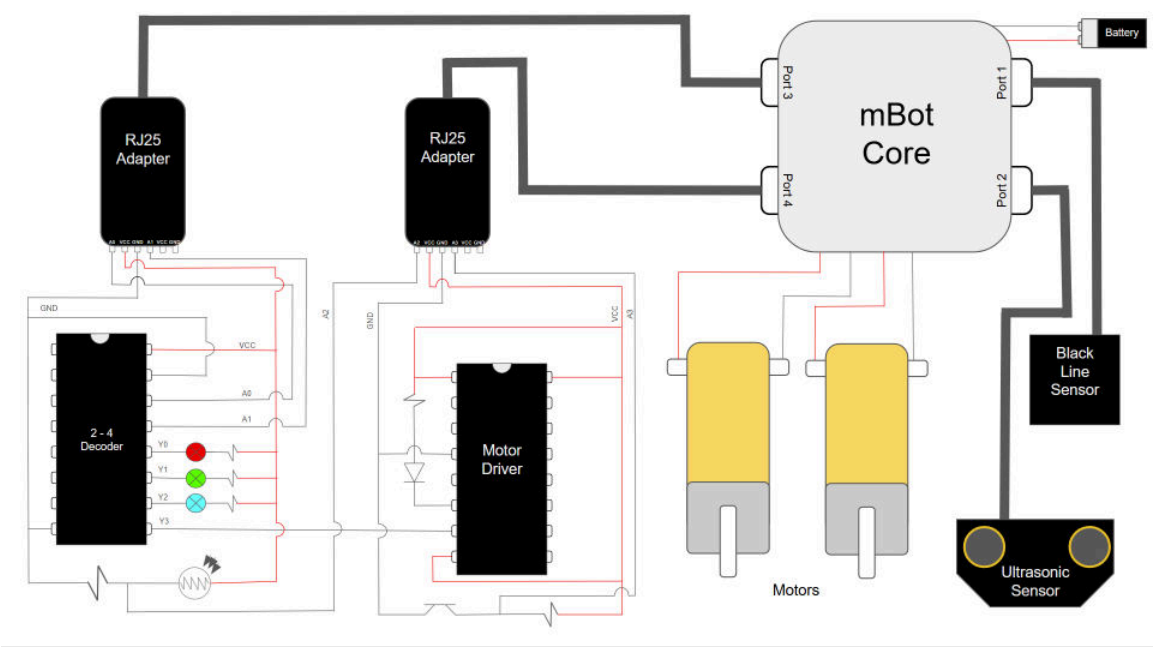
Picture of Black Line Sensor

2.5 Wire Management

As far as possible, we tried to ensure that most of the wires were contained within the mBot chassis as seen in the, tucking in and folding any excess wires into the chassis and rolling RJ25 cables under the mBot module to better conceal them.



Picture of Exposed Underside



Overview of Circuitry

3. Algorithm

3.1 Config.h

Our configuration file defines all global constants used across the program, ensuring consistent reference points and making system-wide adjustments easy. We separated it into the following sections:

Pin Definitions

```
#define DECODER_A A1
#define DECODER_B A0
#define LDR_PIN A2
#define IR_RECEIVER_PIN A3
```

Specifies how our mBot's ports are assigned to the decoders and sensors, where the RJ25 adapter lets us map six sensor ports onto the mBot's four available ports.

Speed & Time Constants

```
#define MOTOR_SPEED 255 // maxed out motor speed
#define RGB_TIME 100 // delay to allow the ldr readings to stabilise before flashing the led
#define TURN_TIME 290 // to turn left/right
#define FORWARD_TIME 655 // forward time before executing the second left/right
```

Speed constants: to maximise performance, we set MOTOR_SPEED to 255, which is the highest possible value on the mBot. Time constants: defined sampling rates and delays determined experimentally. We included a short RGB delay before each colour flash to let the LDR reading stabilise. Initially, our delay was 300 ms, but we reduced it to 100 ms as the color readings were not compromised, thereby allowing faster waypoint color detection.

Color Definitions

```
Colours mapped with value from 1 -> 6
long BLACK_ARR[3] = {463, 810, 835};
long WHITE_ARR[3] = {849, 971, 969};
long RED_ARR[3] = {819, 903, 908};
long ORANGE_ARR[3] = {844, 929, 915};
long GREEN_ARR[3] = {779, 950, 933};
long LIGHT_BLUE_ARR[3] = {741, 944, 957};
long LIGHT_PINK_ARR[3] = {878, 960, 961};
```

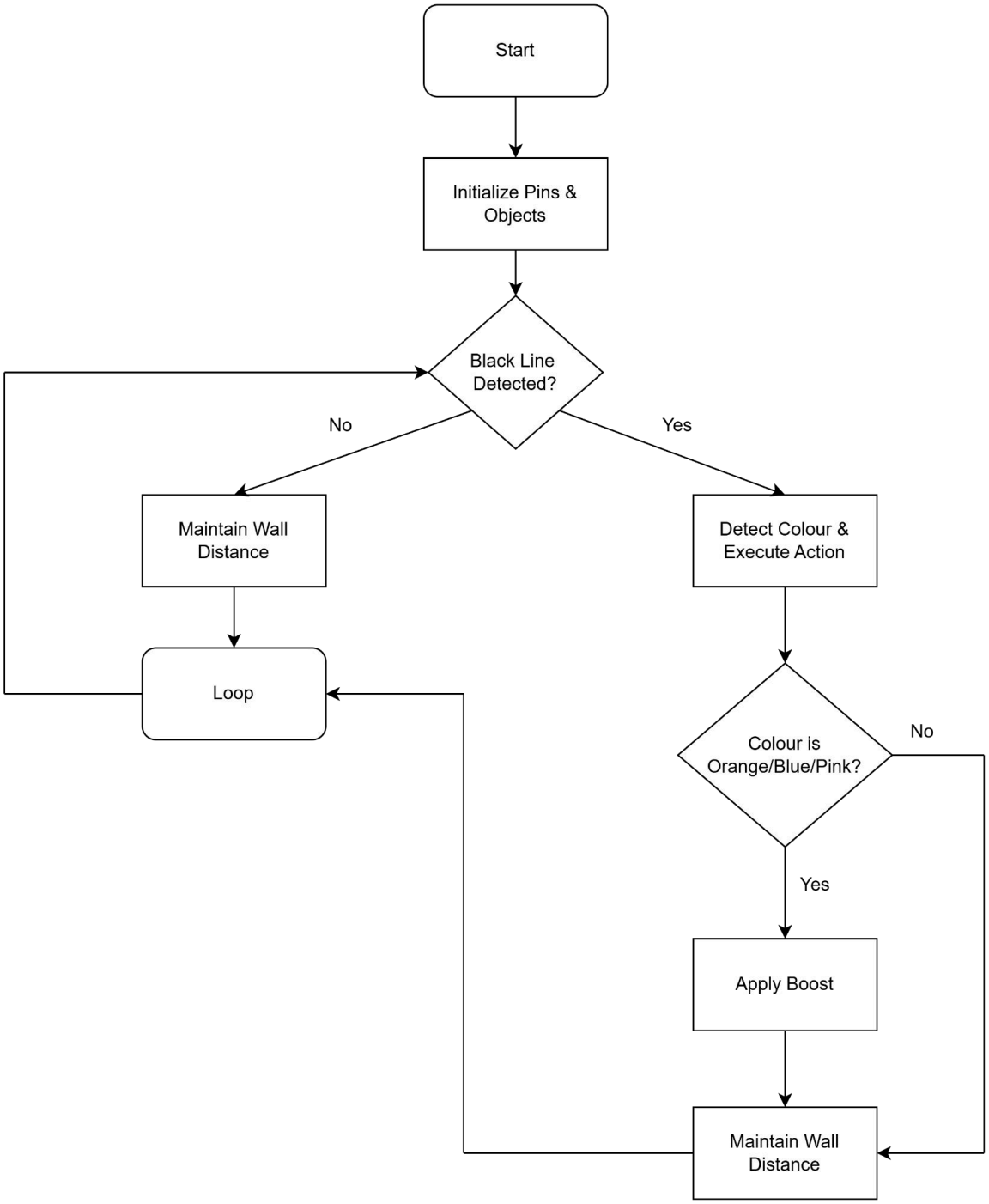
Maps each colour to an integer ID for use in the waypoint files and stores the un-normalised RGB calibration arrays recorded at the start of each session. Updating these values keeps colour detection adaptive to lighting changes. That being said, our skirting setup minimised ambient interference, so any calibration deviations were minor. In fact, for our final run, we used our calibration values from a day prior without facing any issues, meaning our circuit management and code were effective.

Ultrasonic & IR Parameters

```
// Ultrasonic Parameters
#define TOO_CLOSE 9.0 // too close to the left
#define CORR_LO 13.0
#define MID_CORR 14
#define CORR_HI 15.8
#define SMALL_CORR 40 // for small nudge
#define STRONG_CORR 68 // for strong nudge
// Ultrasonic Boost (After U-Turn & Double Turn)
#define BOOST_MS 2000 /
#define BOOST_ADD_STRONG 24
#define BOOST_ADD_SMALL 12
// Infrared Parameters
#define IR_DIST 850
#define extremeNudge 80
```

Ultrasonic parameters: set distance thresholds for too close, somewhat close and extremely close, preventing false positives when no wall is present. We only used one condition for the left wall since compensating for the IR sensor's limited sensitivity on the right wall was more important. IR parameters: if the IR reading exceeded 850 (near collision), we triggered an extreme nudge correction (fail-safe check). The boosting parameters are activated after u-turns/double-turns to ensure that the bot recenters itself without bumping into the walls, accounting for the fact that it tended to veer off-center more than in normal scenarios.

3.2 Main.h



This is a flowchart of how our code works. We start by initialising sensors and components, then entering a continuous loop where it first checks for a black line on the ground.

If no black line is detected, the robot maintains a safe distance from walls and continues detecting for black lines. When a black line is detected, the mBot identifies the color of an object and executes the corresponding waypoint action. If the detected color is orange, blue, or pink (u-turn and double turns), it applies a speed boost before resuming wall-distance maintenance; otherwise, it skips the boost and proceeds directly to maintaining wall distance.

Initialisation

```
// include the other files
#include <MeMCore.h> // mBot library
#include "Config.h" // defines key global values
#include "Proximity.h" // ultrasonic + ir sensor detection
#include "Buzzer.h" // plays celebratory tune
#include "Motion.h" // motor functions
#include "Waypoint.h" // color detection + waypoint execution

// defining mBot objects, and the ports they are connected to
MeBuzzer buzzer;
MeDCMotor leftMotor(M1);
MeDCMotor rightMotor(M2);
MeRGBLed  led(0, 30);
MeLineFollower lineFinder(PORT_1);
MeUltrasonicSensor us(PORT_2);
```

Initialisation defines the variables used in our main file. LAST_COLOR tracks the most recently detected colour to determine whether to apply the boosting correction, while nudgeBoostUntil records how long the boost should remain active.

Black Line Detection

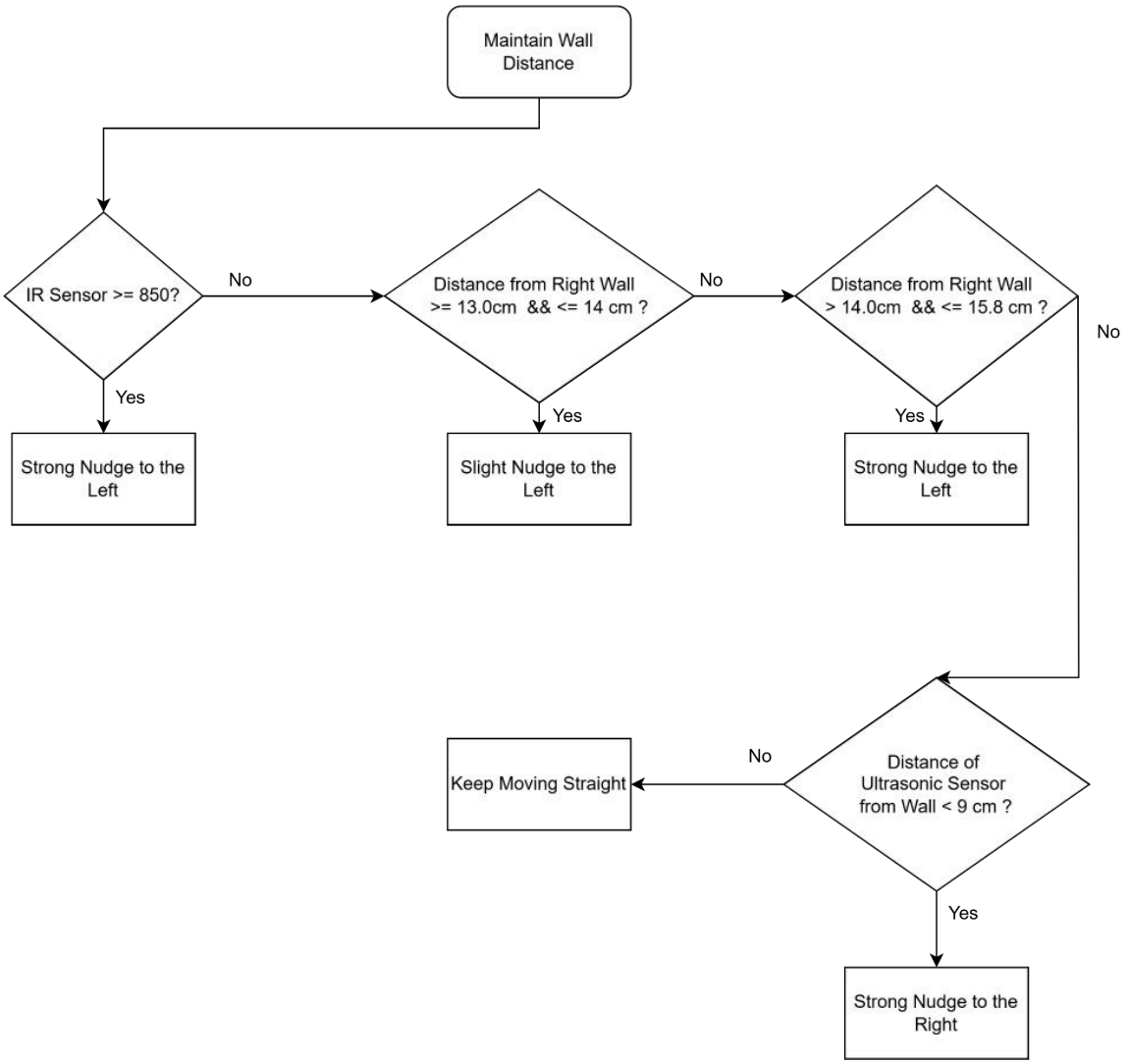
```
bool blackLine() {
  return lineFinder.readSensors() == S1_IN_S2_IN;
}
```

Calls the in-built functions in the mBot line follower to check if both the sensors detect black. Returns true only when both sensors detect black (i.e at the black line).

Maintain Wall Distance

```
void maintainWallDistance() {
  float dist = distance_left(); // get ultrasonic distance
  float irRight  = distance_right(); // get ir distance
  bool boosting   = (millis() < nudgeBoostUntil); // check if 'boosting' active or not
  // calculate steering nudges based on distance + additional boost (if
boosting==true)
  int  smallNudge = SMALL_CORR  + (boosting ? BOOST_ADD_SMALL  : 0);
  int  strongNudge= STRONG_CORR + (boosting ? BOOST_ADD_STRONG : 0);
  if (irRight >= IR_DIST) { // is the IR dangerously close to the left wall?
    nudgeLeft(extremeNudge); // extreme nudge to the left)
  }
```

```
  if (dist > 0 && dist < TOO_CLOSE) {  // ultrasonic distance too small?
    nudgeRight(strongNudge); // strong nudge to the right
  }
  else if (dist >= CORR_LO && dist <= MID_CORR) { // ultrasonic distance large?
    nudgeLeft(smallNudge); // small nudge to the left
  }
  else if (dist >= MID_CORR && dist <= CORR_HI) { // ultrasonic distance very large?
    nudgeLeft(strongNudge);  //  strong nudge to the left
  }
  else { moveForward(); } // if not too close to either wall, continue moving forward
  delay(30); // samples at 30 ms intervals so it doesn't act drunk
}
```



In our maintainWallDistance() function, we obtain the left and right distance measurements from the ultrasonic and infrared sensors respectively. If the robot is within the boosting period, the small and strong nudge values are increased to apply stronger corrective actions. We first check for the emergency correction condition, which is triggered when the IR sensor reading exceeds the threshold (EXTREME_CORR = 65), which applies the maximum corrective force to prevent imminent collision.

If the robot is not critically close to the right wall, we evaluate the ultrasonic distance and apply the corresponding correction:

- 0–9 cm: strong right nudge (too close to the left wall)
- 13–14 cm: small left nudge (somewhat close to the right wall)
- 14–15.8 cm: strong left nudge (too close to the right wall)

These adjustments ensure both rapid re-centering and smooth trajectory maintenance, minimising drift and oscillation. A temporary boost is applied to both the small (+4) and strong (+8) corrections immediately after double-turns and U-turns. These manoeuvres often leave the robot slightly misaligned, so the boosted corrections help it re-acquire the ideal wall distance faster and stabilise before entering straight sections. The boost values were determined empirically through a process of trial and error. Finally, a 30 ms delay is introduced to regulate the control loop frequency, ensuring the robot's motion is not erratic - if it sampled too frequently, the robot would act 'drunk'.

Setup

```
void setup() {
  Serial.begin(9600);
  pinMode(DECODER_A, OUTPUT);
  pinMode(DECODER_B, OUTPUT);
  pinMode(LDR_PIN, INPUT);
  led.setpin(13);
```

```
}
```

Sets the baud rate (for any debugging), connects the decoders to output (to send information) and LDR pin to input (to receive data).

Loop

```
void loop() {
  if (blackLine()) { // if we detect a black line, then...
    stopMotor(); // 1. stop the motor
    detect_and_execute_color(); // 2. detect the color and execute the appropriate waypoint action
    delay(100); // brief pause
    if (LAST_COLOR == ORANGE || LAST_COLOR == PINK || LAST_COLOR == LIGHT_BLUE) {
      nudgeBoostUntil = millis() + BOOST_MS; // 3. check if boosting constant should be applied
    }
  } else {
    maintainWallDistance(); // if no black line, continue maintaining the wall distance
  }
}
```

This is where the main logic happens as per our flowchart. The loop continuously checks for a black line using blackLine(). If detected, the robot stops, identifies the color, and executes the corresponding waypoint action within a 100 ms window, which we intentionally constrained to maintain speed. Based on the detected color, a temporary boost may be applied to improve post-turn stability. If no black line is detected, the robot instead calls maintainWallDistance() to continue stable navigation. The loop runs continuously until the robot encounters white, where our waypoint.h file will stop the motors and call our buzzer object.

We now explain into the sub-files we've neatly encapsulated to handle specific functionalities within the system: proximity.h, motion.h, waypoint.h, and buzzer.h.

3.3 Proximity.h

Proximity.h defines two core sensing functions used for wall-following: distance_left() for the ultrasonic sensor and distance_right() for the infrared receiver. These functions provide the left–right distance measurements needed to keep the robot within its wall-tracking bounds.

```
// Ultrasonic Sensor Reading
float distance_left() {
  return us.distanceCm();
}
```

```
// Infrared Sensor Reading
float distance_right() {
  digitalWrite(DECODER_A, HIGH);
  digitalWrite(DECODER_B, HIGH);
  return analogRead(IR_RECEIVER_PIN);
}
```

The ultrasonic reading is straightforward: since the ultrasonic module is an mBot object, we directly call us.distanceCm(), which handles pulse triggering and echo timing.

For the infrared reading, DECODER_A and DECODER_B are set high to activate the IR emitter via the decoder circuit, then the analog voltage at IR_RECEIVER_PIN is sampled. Standard practice involves subtracting ambient light levels by taking readings with the emitter off, but we found ambient IR remained stable throughout each lab session. Therefore, a single calibration at startup eliminates the need for repeated ambient sampling, avoiding the delays and computational overhead that would slow our control loop.

```
float ambient_IR() {
  digitalWrite(DECODER_A, LOW);
  digitalWrite(DECODER_B, LOW);
  delay(IR_DELAY);
  return analogRead(IR_PIN);
}
```

```
float distance_right() {
  digitalWrite(DECODER_A, HIGH);
  digitalWrite(DECODER_B, HIGH);
  delay(IR_DELAY);
  return analogRead(IR_RECEIVER_PIN);
}
```

Initially, this was our code to validate the ambient IR level. However, this approach introduced delays that slowed our ultrasonic sensor's update rate due to the additional sampling and computation required to verify threshold values. Since these readings remained unreliable regardless of the ambient subtraction, we removed this logic entirely to maintain faster response times in our control loop.

Similarly, we chose not to average multiple readings for either sensor. Empirically, we realised that adding averaging delays would slow the control loop and sometimes cause corrective nudges to occur too late, reducing the responsiveness of our wall-following logic.

3.4 Motion

Standard Motion:

```
// Standard Motion
Void stopMotor() {
  leftMotor.stop();
  rightMotor.stop();
}
void moveForward() {
  leftMotor.run(-MOTOR_SPEED);
  rightMotor.run(MOTOR_SPEED);
}
void turnRight(int speed) {
  leftMotor.run(-speed);
  rightMotor.run(-speed);
  delay(TURN_TIME);
  stopMotor();
}
void turnLeft(int speed) {
  leftMotor.run(speed);
  rightMotor.run(speed);
  delay(TURN_TIME);
  stopMotor();
```

```
void doubleLeftTurn() {
  turnLeft(MOTOR_SPEED);
  moveForward();
  delay(FORWARD_TIME);
  stopMotor();
  delay(80);
  turnLeft(MOTOR_SPEED);
}
void doubleRightTurn() {
  turnRight(MOTOR_SPEED);
  moveForward();
  delay(FORWARD_TIME);
  stopMotor();
  delay(80);
  turnRight(MOTOR_SPEED);
}
void uTurn() {
  leftMotor.run(-MOTOR_SPEED);
  rightMotor.run(-MOTOR_SPEED);
  delay(1.95*TURN_TIME);
```

```
}                                                    stopMotor();
                                                    }
```

Our standard motion functions provide deterministic, repeatable motion for reliable navigation. stopMotor(), moveForward(), turnRight(), and turnLeft() are implemented using a predefined MOTOR_SPEED, with turning durations calibrated through testing. For double-turn maneuvers at the blue and pink waypoints, we stop the motors fully between turns to ensure precise alignment and complete rotation.

In the u-turn, we found that turning approximately 97.5% of the nominal turning duration produced the most consistent results. Our current implementation always performs the u-turn to the right. Ideally, we would have liked to detect which wall was farther and turn in that direction. However, due to the unreliability of our IR sensor, which was reserved primarily as an emergency fallback, we were unable to do so. Future implementations incorporating this logic could be more robust.

That said, given our circumstances, we mitigated this in our Waypoint.h file by introducing a boosting correction after the blue and pink waypoints, ensuring the robot recentres itself before reaching the orange square. For all other paths leading to the orange, such as straight routes, we encountered no issues of centralisation so u-turning to the right was not an issue.

Stabilisation Motion

```
void nudgeRight(int speed) {
  int r = MOTOR_SPEED - speed;
  leftMotor.run(-MOTOR_SPEED);
  rightMotor.run(r);
}
```

```
void nudgeLeft(int speed) {
  int l = MOTOR_SPEED - speed;
  leftMotor.run(-l);
  rightMotor.run(MOTOR_SPEED);
}
```

Our stabilisation functions implement dynamic wall-following through variable-speed corrections (nudges). Allowing us to specify the speed enables the system to adaptively respond to different proximity conditions. To handle cases where the robot is veering towards the wall, moving close to it, or extremely near, we can simply adjust the correction speed.

Since we maxed out our motor_speed; the wheel cannot go faster. Thus, to maintain the speed differential, we make the other wheel go even slower based on which direction we are turning (right wheel slower to nudge right, left wheel slower to nudge left).

3.5 Waypoint.h

Colour Reading

```
long avgLDR() {
  long total = 0;
  total += analogRead(LDR_PIN); //
returns the LDR reading
  return total;
}
// Color Flashing
void shineRed() {
  digitalWrite(DECODER_A, LOW);
  digitalWrite(DECODER_B, HIGH);
  delay(RGB_TIME);
  COLOUR_ARRAY[0] = avgLDR();
}
void shineGreen() {
  digitalWrite(DECODER_A, LOW);
  digitalWrite(DECODER_B, LOW);
  delay(RGB_TIME);
  COLOUR_ARRAY[1] = avgLDR();
}
void shineBlue() {
  digitalWrite(DECODER_A, HIGH);
  digitalWrite(DECODER_B, LOW);
  delay(RGB_TIME);
  COLOUR_ARRAY[2] = avgLDR();
}
```

This code handles color detection once the mBot stops at a black line. Each shine function activates one LED color at a time by configuring the decoder pins, waits briefly for the LDR to stabilize, and then stores the averaged reading in a color array. Through testing, we found that one readings per color provided stable and reliable results without adding unnecessary delay. These were the same functions called to get the un-normalised colour arrays.

Colour Identification

```
void identifyColor() {
  const long* refs[] = { RED_ARR, GREEN_ARR, ORANGE_ARR, LIGHT_PINK_ARR,
LIGHT_BLUE_ARR, WHITE_ARR };
  long r = COLOUR_ARRAY[0], g = COLOUR_ARRAY[1], b = COLOUR_ARRAY[2];
  unsigned long bestd = 0xFFFFFFFFUL;
  unsigned long secondd = 0xFFFFFFFFUL;
  int bestColor = UNKNOWN;
  int secondColor = UNKNOWN;
  float w_r = 0.35;
  float w_g = 0.83;
  float w_b = 1.00;
  for (int i = 0; i < NUM_COLOURS; i++) {
    long dr = r - refs[i][0];
    long dg = g - refs[i][1];
    long db = b - refs[i][2];
    float d = w_r * dr * dr + w_g * dg * dg + w_b * db * db;
    if (d < bestd) {
      secondd = bestd;
      secondColor = bestColor;
      bestd = d;
      bestColor = i + 1;
    } else if (d < secondd) {
```

```
      secondd = d;
      secondColor = i + 1;
    }
  }
// Tie breakers between similar colours
  if ((bestColor == GREEN && secondColor == LIGHT_BLUE) || (bestColor ==
LIGHT_BLUE && secondColor == GREEN)) {
    long b_mid = (GREEN_ARR[2] + LIGHT_BLUE_ARR[2]) / 2;
    bestColor = (b >= b_mid) ? LIGHT_BLUE : GREEN;
  } else if ((bestColor == RED && secondColor == ORANGE) || (bestColor == ORANGE
&& secondColor == RED)) {
    long g_mid = (RED_ARR[1] + ORANGE_ARR[1]) / 2;
    bestColor = (g >= g_mid) ? ORANGE : RED;
  } else if ((bestColor == WHITE && secondColor == PINK) || (bestColor == PINK &&
secondColor == WHITE)) {
    long g_mid = (WHITE_ARR[1] + LIGHT_PINK_ARR[1]) / 2;
    bestColor = (g >= g_mid) ? WHITE : PINK;
  }
  execute(bestColor);
}
```

This is the most important code that we developed through several iterations. First, we got the RGB readings from the color sensor.

Then, we use weighted euclidean distance calculations to find both the closest and second-closest matching colors from our six reference arrays. We intentionally chose different resistor values for each RGB channel, which gives us different weightings (w_r = 0.35, w_g = 0.83, w_b = 1.00). This is because our sensor is most sensitive to red, then green, then blue, so we scaled the weightages accordingly.

We use the following formula to account for the varying sensitivities:

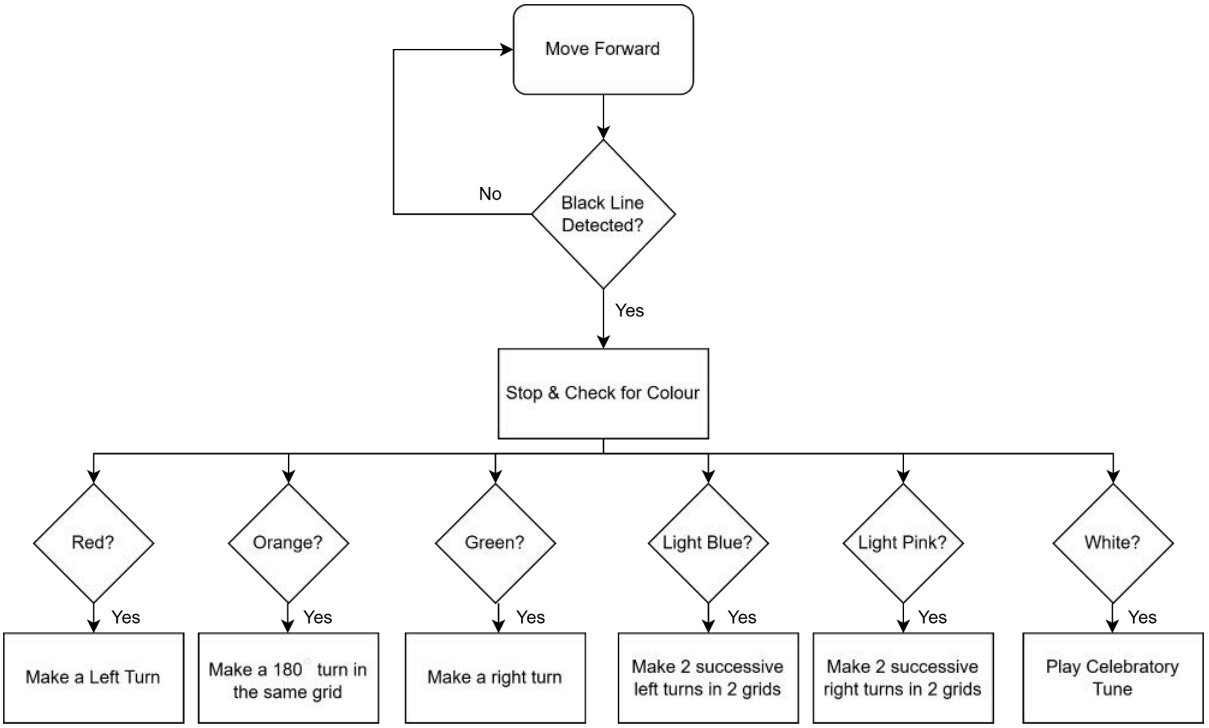
$$d_k^2 = w_R(R - R_k)^2 + w_G(G - G_k)^2 + w_B(B - B_k)^2$$

This gives less weight to the red channel (most sensitive) and more weight to the blue channel (least sensitive), ensuring all color differences are measured on an equal basis. The variation in RGB also makes debugging easier since the values spread across different ranges instead of clustering together where they're hard to distinguish.

Through testing, we discovered that certain color pairs kept getting confused with each other - specifically red/orange, green/light blue, and white/pink. To resolve this, we implemented tie-breaker logic that looks at the specific RGB channel where each pair differs most.

(1) For red versus orange, we compare the green channel against the midpoint between their reference values - if the measured green is higher, it's orange.
(2) For green versus light blue, we use the blue channel as the discriminator, choosing light blue when blue values are higher.
(3) For white versus pink, the green channel is the decision maker, with higher green indicating white.

This multi-stage approach ensures reliable color identification even with lighting variations. Once we've confidently identified the color, we pass it to the execute() function to trigger the appropriate robot action, which works as per this flowchart:



3.6 Buzzer.h

```
// define buzzer

// define note frequencies

// define song tempo + melody

// play song in celebrate()
```

Extracted from https://dragaosemchama.com/en/2019/02/songs-for-arduino/ and lightly modified to encapsulate the playback logic inside a single celebrate() function, allowing the main program to trigger the melody with a simple, centralised call. Our chosen song is “Imperial March”.

4. Calibration

4.1 Colour Sensor Calibration

1. Measured the LDR readings under black and white surfaces to define the lower and upper limits of the sensor response.
2. Captured 10 readings each for waypoint colours (red, orange, green, light blue, pink, and white) under identical lighting conditions.
3. Weighted each R, G, B value for the euclidean distance calculation based on the black and white limits and formula in section 3.5.

4.2 Infrared (IR) Distance Sensor

The IR sensor was manually calibrated using trial and error within the actual maze environment. Experimentally a value of around 850 was chosen as an emergency turn left.

1. Placed the mBot at known distances from the maze wall and recorded the detected IR output values. Used multiple readings and averaged them to reduce sensor noise.
2. Tuned the threshold value so that when the IR sensor on the right detected a value above this threshold, the mBot nudged left to maintain a safe distance.

4.3 Ultrasonic Distance Sensor

Calibration for the ultrasonic sensor was carried out using a ruler and measured distances.

1. Placed our mBot at different distances from the wall and averaged the values.
2. Defined our pre-emptive nudges distance and strong nudge distances.
    a. We chose to separate into different conditions so the robot doesn't over or understeer.
    b. We also considered a margin of error of ~3cm from the midpoint before it started nudging.

4.4 Movement Calibration

The robot's movement was tuned manually through iterative testing. We ensured the robot was fully charged before calibrating the motion to ensure our values were consistent.

1. Conducted multiple trials where the mBot moved forward, turned, and performed U-turns.
2. Adjusted motor speeds and time durations for each motion until the robot's trajectory matched the desired geometry within the maze (e.g., 90° turns, 180° U-turns).
3. Averaged timing over multiple trials to reduce random error.

5. Reflections

5.1 Work Division

Name	Work Done
Advait Bagri	● Responsible for the Source Code of the mbot ● Testing and Error Handling of the Algorithms ● Debugging of Source Code
Teammate 1	● Designed the Circuit Diagrams for the IR and Color Sensor ● Construction of the Color Sensor Circuit
Teammate 2	● Building of the Hardware of the mbot & Debugging the IR Circuit ● Wiring & Skirting Management
Teammate 3	● Construction & Debugging of the IR Circuit ● Wiring and Neatness Management

5.2 Significant Challenges

A major hurdle we faced in the initial stages was understanding how the various components worked. Due to the extensive documentation associated with each component, we had to carefully read through and extract the key information relevant to building our robot. Based on this understanding, we drew up schematics for the circuitry and calculated the required resistor values, allowing us to easily reference how the system should be wired.

Another issue was calibrating our IR sensor. After implementing the initial circuit design, we found that the IR sensor failed to detect walls reliably and was returning inconsistent values. To address this, we took two approaches. First, we reworked all the circuitry on our breadboards and recalculated the resistor values to ensure proper current flow throughout the system. We then unit tested both the IR emitter and sensor individually to confirm that the components themselves were functional. Second, we reviewed the code to ensure that the IR sensor was being read correctly, checking the logic carefully to ensure no variables or steps were missed. In the end, we replaced the resistors to stabilise the readings received by the mBot, and we modified the code so that the IR emitter was toggled on intermittently.

However, even after rigorous testing, one issue remained unresolved: the IR sensor only produced stable readings when pointed at darker coloured surfaces. As a result, it was volatile when detecting walls inside the maze, due to the white, reflective wall

surfaces. Hence, we decided not to implement any PID algorithm involving it, instead using it as a last resort due to its instability.

Another major challenge was ensuring consistent colour detection with our LDR-based sensor. During repeated tests, the mBot reported different colours even when run over the same coloured paper. This inconsistency was traced back to ambient light interference, as our initial shroud leaked considerable light, affecting the LDR readings. To resolve this, we redesigned the cover around the LEDs and LDR to block out ambient light more effectively. Additionally, we found that the LDR was positioned too far behind the LEDs, which caused it to saturate when measuring floor colour. We fixed this by adjusting the LDR's position so that it sat in front of the LEDs, detecting only the reflected light from the coloured surface.

The resistor value for the LDR circuit was initially chosen based on previous lab sessions to maintain a consistent level of detection accuracy. However, during testing, we observed that the LDR readings were too high, resulting in reduced variance between colors and lowered classification accuracy. As a result, we manually adjusted the LED resistor values to improve the system's color differentiation performance through trial and error.

Lastly, we noticed that the colour sensor frequently misclassified surfaces as either red or white, regardless of their actual colour. Upon reviewing our code, we found that the red channel had disproportionately high variance compared to the green and blue channels, causing it to dominate the Euclidean distance calculations. To fix this, we

reduced the weightage of the red component in the algorithm, which significantly improved the accuracy and consistency of our colour detection system.

5.3 Acknowledgements

We would like to thank our instructor Khoa for his invaluable advice and guidance throughout the course of the projects, as well as Bossman for helping us with swapping out some of our components and providing hardware support. We would also like to thank our TAs for helping to answer some of our doubts.