CG1111A

The A-maze-ing Race

B03-S3-T1

Loh Jay Hon, Giggs	A0273137E
Loh Yan Xun, Timothy	A0272149B
Loke Mun Kong Denzel	A0281142N
Low Beverly	A0282599H

Our mBot:

Ryan Jr.

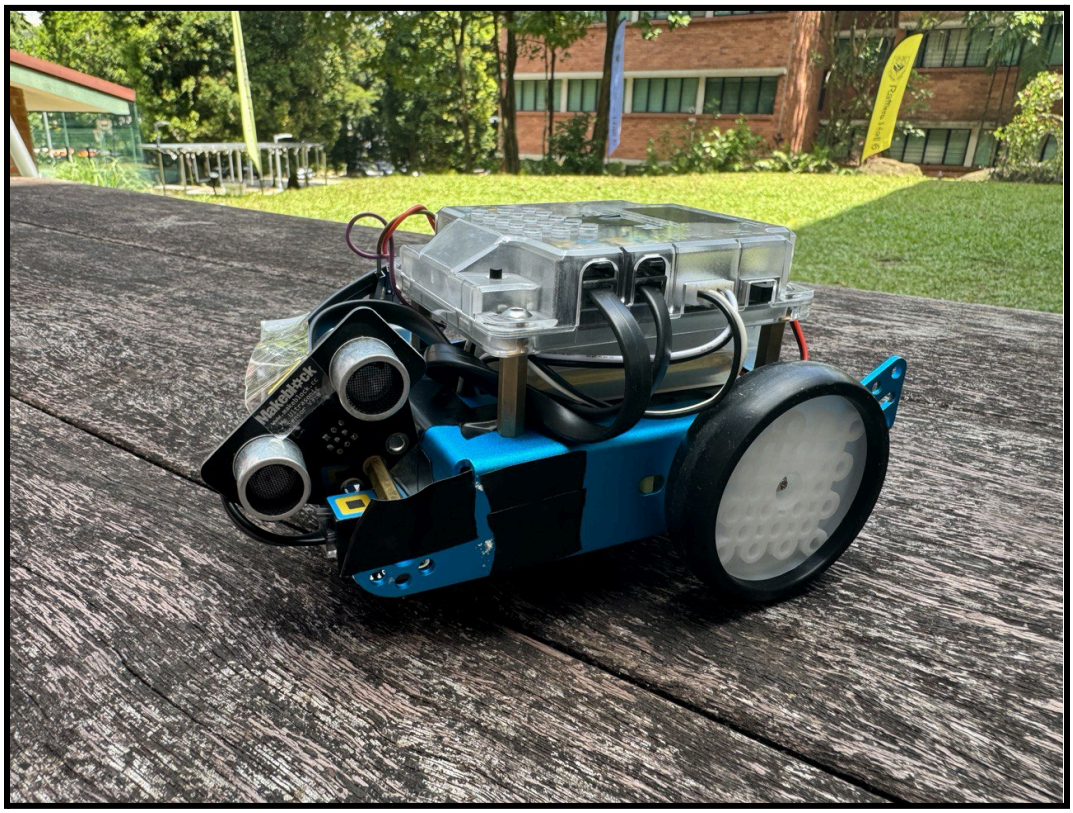


TABLE OF CONTENTS

GENERAL ALGORITHM

We built our algorithm systematically by incorporating 3 main sequences that reflect our mBot's 3 main functions (colour detection, left correction, and right correction). In this way, we were able to divide and conquer by working on each system simultaneously. This also enabled us to troubleshoot more efficiently by disabling each system to target the relevant matters at hand.

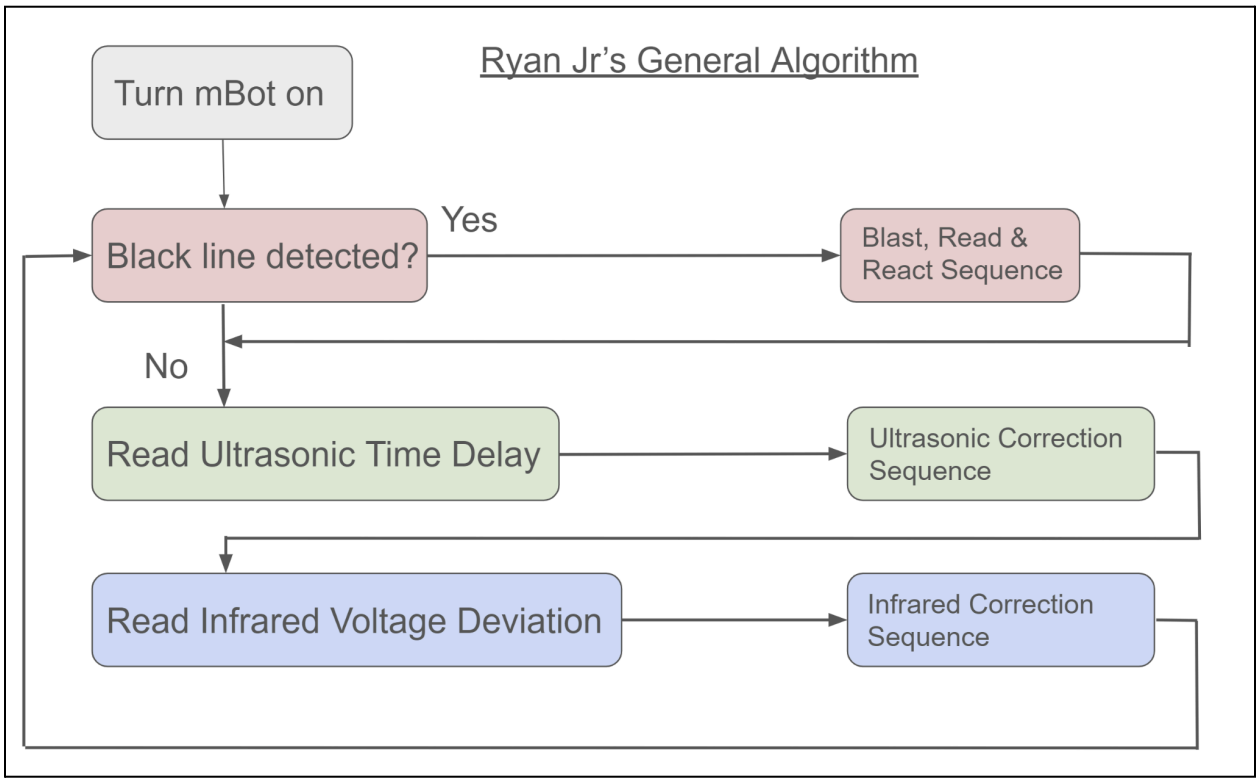


Figure 1: General Algorithm

Figure 1 shows the general algorithm that allowed our mBot to successfully navigate through the maze. When the mBot is turned on, it uses its meLine sensor to scan for the black stopline. If the stopline is detected, it will enter the "Blast, Read & React Sequence" (Refer to Figure 3).

As long as no black stopline is detected, the mBot will continue to advance while scanning for left wall proximity using the Ultrasonic Sensor, and right wall proximity using the Infrared Sensor.

While these two proximities are within an acceptable range, the mBot will continue moving forward. When either of these proximities become too close, the mBot will enter a Correction Sequence to correct his position to the centre of the lane to avoid bumping the walls.

SUBSYSTEMS

I. Lane Correction - Ultrasonic Sensor & IR Sensor

To keep the mBot in the centre of the pathway, we mounted the IR sensor on the right side of the mBot, to measure the distance between the sensor and right wall, as well as mounted the ultrasonic sensor on the bot's left side, to measure the distance between the sensor and left wall.

We measured the optimal distance detected by the ultrasonic sensor to be 11 centimetres from the left wall, which coincides with the centre of the maze's path. We allowed the mBot a margin of error of ±2 centimetres on either side of the intended path, before correcting its trajectory. Only when the ultrasonic sensor detects that the mBot is less than 9 centimetres, or more than 13 centimetres away from the left wall, will the mBot correct its trajectory.

To guard against potential reliability issues posed by the IR sensor (Refer to Page 11), we only used the IR sensor and emitter when the ultrasonic sensor detected that it was too far away from the left wall. Once the IR sensor also then detects that the change in IR is too high, which means that it is too close to the right wall, it will then correct the mBot's trajectory by veering left.

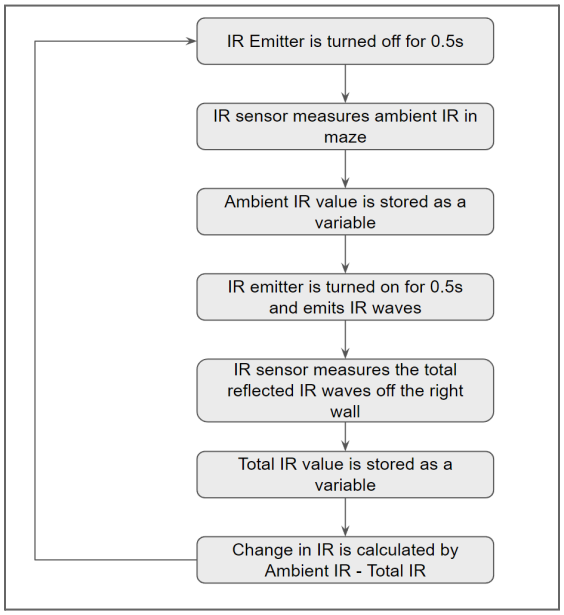


Figure 2: Algorithm used by IR sensor and emitter

Additionally, we wanted the bot to continue moving forward in the absence of walls. For the ultrasonic sensor, if there is no left wall detected, the distance measured will be much greater than 13 centimetres, in which case the ultrasensor will timeout and return 0 as the detected distance. We added a check to our code (if distance == 0) to ensure that the mBot would continue moving straight should this scenario take place.

II. Stopline Detection

The MeLine Follower continuously emits IR light at the ground. If the reflected IR is less than a certain amount, the status of the sensor will become "IN". Since the Maze floor is a glossy white material, and the stopline is a matte black tape, the amount of reflected IR would be high throughout the maze, and only drop below the threshold when the MeLine Follower hovers over the black stopline. As such, the mBot was configured to stop and enter its "Blast, Read & React Sequence" (Figure 3) once both the sensors on the MeLine Follower change to the "IN" status (Signalling low reflected IR).

III. Colour Identification

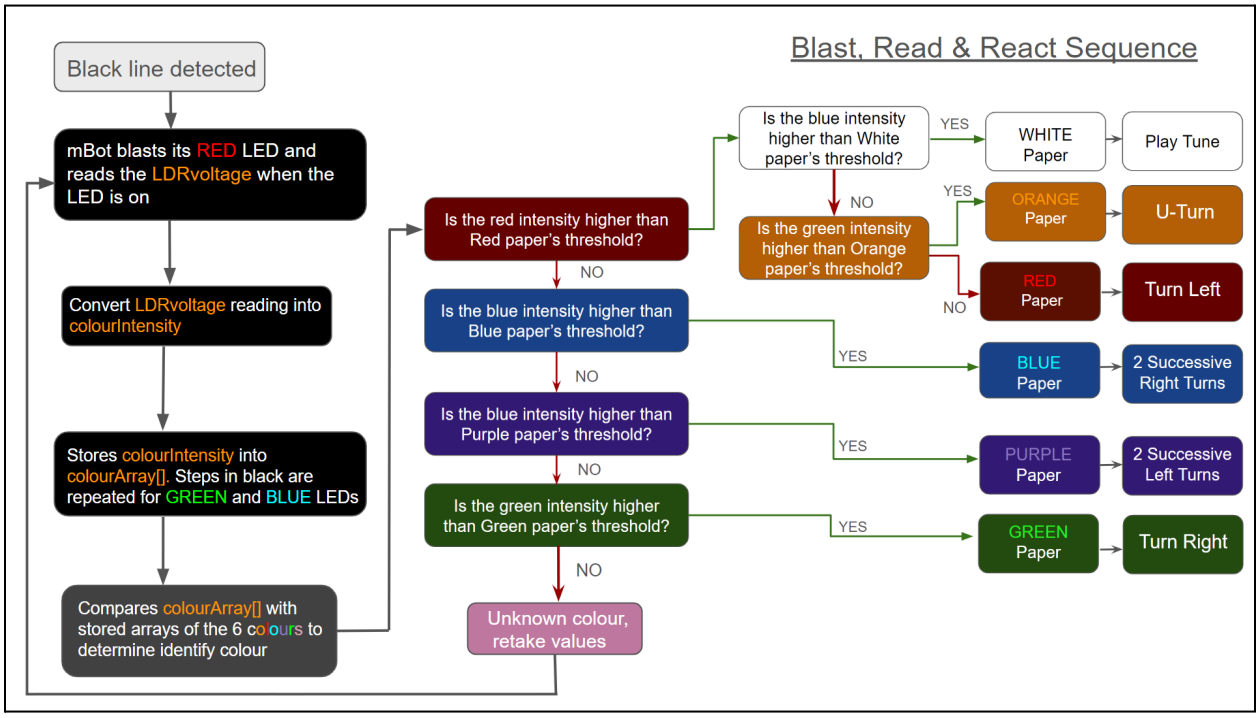


Figure 3: Blast, Read and React Sequence

Reading LDRvoltage

We wired the Colour Identifier circuit as shown in Appendix 8. This configuration allowed us to control the activation of the Red, Green and Blue LEDs (as well as the IR emitter) using only 2 output pins (A2 and A3) present on the mBot. The 2 wires were inputted into the HD74LS139P 2-4 converter, and by permuting logic configurations from HIGH-HIGH to LOW-LOW, we assigned each LED to a specific logic configuration.

This setup also allowed us to measure the potential difference across the LDR resistor using input pin A0. Since the LDR decreases in resistance as light intensity increases, the potential difference across the LDR resistor will increase as light intensity increases, by principle of potential divider. The potential difference across the LDR resistor is measured and stored in the variable `LDRvoltage`.

We can thus expect higher values of `LDRvoltage` to be measured when red LED is shone on red paper, or green LED is shone on green paper, as more light will be reflected off the paper into the LDR.

```
int colourIntensity = (LDRvoltage - blackArray[colour]) / greyDiff[colour] * 255;
```

Figure 4: Calculation of colourIntensity

Converting LDRvoltage into colourIntensity

Using this logic, we converted each `LDRvoltage` to `colourIntensity`. We took the `LDRvoltage` values for Red, Green and Blue LED shone on White Paper as 255, and the `LDRvoltage` values for Red, Green and Blue LED shone on Black Paper as 0 for each LED. `colourIntensity` is the expression of the `LDRvoltage`, with the White Paper values as maximum (255) and Black Paper values as minimum (0).

This `LDRvoltage` for each LED colour is then compared against arrays of stored values of the `LDRvoltages` captured for White paper and Black paper. `whiteArray` serves as the maximum `LDRvoltages` for each LED, since most light will be reflected from white paper. Conversely, `blackArray` serves as the minimum voltages for each LED. The calculation is shown in Figure 4. [colour] represents the LED on at that time.

Colour	Red LED `colourIntensity`	Green LED `colourIntensity`	Blue LED `colourIntensity`
`whiteColour`	255	255	255
`orangeColour`	247	185	152
`redColour`	244	150	142
`blueColour`	162	210	225
`greenColour`	93	163	104
`purpleColour`	187	166	188
`colourArray`	??	??	??

Figure 5: Calibrated RGB Colour Intensities for Different Colour Papers

Threshold of Error

Through repeated trial runs and calibrations, we discovered the need for a "threshold of error" for our `colourIntensity` values. The same colour paper measured at different positions gives slightly varying `colourIntensity` readings, and different sets of coloured paper may appear brighter or duller and reflect different amounts of light. In order to ensure that our mBot is able to differentiate colours in spite of all these factors, we deemed that a suitable threshold value will be 10 units less than the calibrated value. This allows the mBot some leeway, while preventing it from mixing up similar colours.

```
bool identifyColour() {
  //bracket 1 - high red intensity detected - either white orange or red
  if (colourArray[0] >= (redColour[0] - 10.0)) {
    if (colourArray[2] >= (whiteColour[2] - 10.0)) {
      Serial.println("WHITE");
      celebrate();  //play music
    } else if ((colourArray[1] >= (orangeColour[1] - 10))) {
      Serial.println("ORANGE");
      //180 turn within same grid
      kebelakangPusing();
    } else {
      Serial.println("RED");
      //left turn
      kekiriPusing(NINETY);
    }
```

Figure 6: Snippet of Code from Colour Identifier

Using colourArray to Identify Colour

We then stored the calculated `colourIntensity` in `colourArray`, `colourArray[0]` represents "`colourIntensity` of red LED shone on current paper", while green and blue LED are represented by `colourArray[1]` and `colourArray[2]` respectively.

We split the 6 colours into 3 baskets: "high red intensity", "high blue intensity", "low red & blue intensity". Through a series of logical conditions, we sifted the colour array into 1 of 3 baskets for further sorting.

First, we examined whether `colourArray[0]` was greater than the threshold value of `redColour[0]` (ie colourArray[0] >= redColour[0] - 10). If the red intensity was over the threshold, it was sure to be in basket 1 - either White, Orange, or Red. The colour is then sorted by its other properties. A high blue intensity means White paper, and a low blue intensity but higher green intensity indicates Orange. If blue and green intensities are both low, the paper is Red.

If `colourArray[0]` did not fall within the threshold to be in the first basket, it will then examine if `colourArray[2]` is greater than the threshold value of cyanColour[2] (ie colourArray[2] >= cyanColour[2] - 10). If it is, then the mBot identifies the paper as blue.

Lastly, if both of the conditions above fail, it will enter the last basket to determine if it is purple, green or unknown. At this point, the intensity of blue and red are lower than the prior colours mentioned. A higher blue intensity points to Purple paper while a higher green intensity points to Green paper. If the values do not fall within any of the threshold conditions, the mBot returns "unknown" and the mBot restarts the Blast, Read & React Sequence.

IV. Turns

```
void kekiriPusing(int turnTime) {  // Stationary left
  motor1.run(200);
  motor2.run(200);
  delay(turnTime);
  berhenti();
}
void kekananPusing(int turnTime) {  // stationary right
  motor1.run(-200);
  motor2.run(-200);
  delay(turnTime);
  berhenti();
}

void kebelakangPusing() { //3 point turn
  motor1.run(-250);  //right reverse
  motor2.run(0);     //left stops
  delay(650);
  motor1.run(0);     //right stop
  motor2.run(-250);  //left go forward
  delay(650);
}
```

Figure 7: Code for Turns

In order to reduce the likelihood of our robot hitting the wall, we went for an approach involving a stationary turn. This approach ensured that the turning point of the mBot lied at the centre of the mBot, allowing it to make a sharp turn with a low possibility of bumping the wall. To make a right turn, we programmed the left wheel to move forward while the right wheel moved backwards simultaneously, with the opposite configuration for left turn. `turnTime` determines how long the wheels will move for in milliseconds (Refer to Figure 7). This value was determined through trial and error and we found the value of 475 perfect to ensure that the mBot turns approximately 90 degrees.

For the U-turn, we noticed that a 180° stationary turn where both wheels move at the same speed in opposite directions tends to lead to a front collision with a wall. We deduced that this was because the axis of rotation in this case would be at the point between both wheels, which is relatively further back on the robot. This causes the mBot to swing its front face as it turns, likely ending in collision.

Thus, we took a page out of the drivers' manual and created a 1-point-turn inspired by the 3-point-turn. By immobilising one wheel at a time, the axis of rotation is now the stationary wheel. This allows us to have a safer and more predictable turn, so long as the mBot is around the centre of the lane.

V. Celebratory Music

We managed to procure a piece of sheet music for Qu Wanting's "我的歌声里". Using a chart we found online, relating each musical note and its frequency, we managed to list down all the frequencies needed by the mBot's MeBuzzer. The sheet music also provided the beats per minute (BPM) for each quarter note, and we were hence able to calculate the duration for each note in the song. Figure 8 shows the function to play our celebratory tune.

```
void celebrate() {
  // Each of the following "function calls" plays a single tone.
  // The numbers in the bracket specify the frequency and the duration (ms)
  buzzer.tone(622.25, 1166);    //D#5
  buzzer.tone(932.33, 583);     //A#5
  buzzer.tone(830.61, 1940.5);  //G#5
  buzzer.noTone();
  buzzer.tone(554.37, 291.5);  // C#5
  buzzer.tone(739.99, 291.5);  // F#5
  buzzer.tone(698.46, 583);    // E#5
  buzzer.tone(554.37, 291.5);  // C#5
  buzzer.tone(466.16, 291.5);  // A#4
  buzzer.tone(415.30, 291.5);  // G#4
  buzzer.tone(466.16, 1249);   // A#4
  buzzer.noTone();
  buzzer.tone(466.16, 666);   // A#4
  buzzer.tone(554.37, 291.5);  // C#5
  buzzer.tone(622.25, 583);    // D#5
  buzzer.tone(369.99, 583);    // F#4
  buzzer.tone(466.16, 666);   // A#4
  buzzer.tone(554.37, 291.5);  // C#5
  buzzer.tone(622.25, 583);    // D#5
  buzzer.tone(369.99, 583);    // F#4
  buzzer.noTone();
  buzzer.tone(369.99, 291.5);    // F#4
  buzzer.tone(415.30, 388.667);  // G#4
  buzzer.tone(466.16, 388.667);  // A#4
  buzzer.tone(415.30, 388.667);  // G#4
  buzzer.tone(466.16, 2000);     // A#4
  buzzer.noTone();
}
```

Figure 8: Buzzer Code for Qu Wanting's "我的歌声里"

CALIBRATING & OPTIMISING SENSORS

To make our mBot more reliable and be less dependent on the conditions of the environment to perform, we improved on the following components.

I. Ultrasonic Sensor

From observing several runs, we realised that in the absence of the left wall, the mBot would correct its trajectory incorrectly and veer to the left, meaning that the mBot would not be centred when it detects the following black stopline. This affected the subsequent turn/turns and could lead to possible collisions with the walls. To resolve this, we reduced the timeout in our code from 1500 milliseconds to 1250 milliseconds. By doing this, the ultrasonic sensor would then wait for a shorter period before timing out and returning a value of 0. This indicates quicker to the mBot that there is no wall on the left. The mBot would therefore move straight instead of veering left like before.

To add on, we found that the ultrasonic sensor was receiving some reflected sound waves from the ledge of the table. To solve this, we loosened the screw of the ultrasonic and used tape to make the ultrasonic tilt slightly upwards (Appendix 1).

As one markdown was the number of bumps the robot made with the wall, we ensured that there were no wires sticking out that could brush against walls or result in any faulty detection from the ultrasonic sensor. To do so, we organised most of the wires to be below the mBot where the colour detection system was, leaving no wire to stick outside the chassis of the mBot (Appendix 4 & 5).

II. Colour Sensor

To prevent surrounding light from causing fluctuations in the LDR's readings, we used black paper to skirt the bottom of the mBot (Refer to Appendix 6). This allowed us to ensure that there would be little to no ambient light that would affect the accuracy of colour detection. We also extended the skirt to have a chimney at the area containing the LDR and LEDs, making sure that the chimney would be as close to the floor as possible. This ensured that the calibrated values for each coloured paper did not differ greatly when the mBot was exposed to ambient light, increasing its accuracy.

As time was also a factor to take into consideration, we looked through our code and removed as many delays as we were able to. This made our mBot significantly faster in its colour detection process, making
the turn almost immediately after detecting the black stopline. However, we realised that while removing all the delays was optimal in making our mBot traverse the maze much faster, some delays were still needed to allow the LEDs time to light up and give the LDR significant exposure to each light. Hence, instead of including delays, we made our code more readable by adding delays through serial printing important information such as the colour intensity read by mBot and whether the mBot was centred. By doing all of this, our colour detection process not only became faster, but also became more coherent.

III. IR Sensor

When working with the IR sensor, we realised that the values returned by the IR sensor did not change significantly despite the changing distance between the sensor and the wall. This made it unreliable in determining whether the mBot was too close to the right wall. To increase the sensitivity of the IR sensor, we changed the resistor connected to the sensor from 10000 ohms to 12000 ohms. While we did see a slight increase in the variance of values returned, we decided that it was still not reliable enough and used it as a backup system that would only be activated if the ultrasensor determined that the mBot was too far from the left wall, and hence too close to the right wall. If the change in IR detected by the was above a certain threshold as well, then it would indicate to the mBot to veer left.

Furthermore, we also cut some short wires to fasten and secure the IR sensor and emitter firmly onto the breadboard to prevent them from shifting and adjusting when the mBot was in motion. This was done to prevent any possible inaccuracies or errors when the IR sensor was reading in IR.

TROUBLESHOOTING

I. Inconsistent RGB Readings -Ambient Light

The main difficulty we faced was the constant need to readjust the RGB values inside of our colour detector function. This had to be done through multiple recalibrations throughout every lab session and was extremely tedious and time-consuming. We hypothesised that the light values that the LDR was reading were inconsistent, due to ambient light being present within the LDR system. We then realised that light from the surroundings could reach the LDR through some small holes that were present on the chassis of the mBot. So, we covered these holes using black tape and black paper, ensuring that the only way light could reach the LDR was through the small chimney cutout we made on the skirting of the mBot. This significantly reduced the amount of times which we had to recalibrate the values for the RGB array in our code.

II. Inconsistency of IR sensor

Another difficulty which we faced was that the IR Sensor was inconsistent in detecting changes in IR and hence unreliable as a singular check as to whether the mBot was too close to the right walls. Despite measuring the ambient IR in the maze, before turning on the IR emitter periodically, we found that the change in IR detected did not stay consistent when the mBot was stationary whilst inside the maze. To deal with this, we decided to pair the use of the IR sensor and emitter with the ultrasonic sensor. Only when the ultrasonic sensor detected that the mBot was too far from the left wall, would the IR emitter and sensor be used to gauge whether the mBot was then close to the right wall. However, since the ultrasonic sensor was likely to keep the mBot within 11 centimetres from the left wall before going off the intended trajectory, we managed to reduce the dependency on the IR system, allowing us to spend our remaining time on improving other aspects of the mBot.

III. Jittery Advancement

When we ran our mBot in the maze, we realised that the mBot did not move straight after an initial correction, but instead would correct quickly in a zig-zag manner, almost appearing to jitter left and right as it advanced. With some help and advice from the dean, we took another look at our code and found a few causes for the mBot to jitter.

Firstly, we realised that the margin of error was too small, meaning that the mBot would correct its trajectory even when deviating slightly off the centre of the track. Secondly, was that the frequency of corrections was too high, and the amount of correction was too large. This meant that the mBot would overcompensate its correction, hence repeating the cycle and start to jitter. To fix this, we first increased the margin of error allowed by the ultrasonic sensor to detect. We increased the margin of error from 1 centimetre on each side of the centre path to 2 centimetres. Afterwards, we added delays in the loop of the code, to reduce the frequency of the ultrasonic sensor emitting and receiving reflected sound waves from the walls. Finally, we reduced the amount of correction done by the mBot by only slightly reducing the speed of the motor wheel of the side that it wanted to correct towards. These 3 changes allowed the mBot to correct its movement slightly and regularly, ensuring that no correction would be done unless necessary.

IV. Faulty components

Throughout the project, 3 of our mBot components became faulty at different times, and this frustrated the team greatly as we initially did not know where the issue was coming from, and had to spend our limited lab time debugging and resolving these issues.

Firstly, our 2 mBot motors each stopped functioning properly at separate times - the left wheel motor followed by the right wheel motor - causing the mBot to be unable to move or change directions correctly. When our wheel first started malfunctioning intermittently, it was unclear at first whether this was due to faulty motor functionality or incorrect coding, leading to arduous checks before eventually replacing the motor, which finally fixed the issues we were encountering.

Secondly, 2 days before the final evaluation, whilst working on colour calibration, the MeLine Follower was unable to consistently detect the black stoplines, greatly hindering the mBot's overall functionality. This required us to immediately figure out the root cause before being able to continue; we eventually dismantled the skirting and some wiring to replace the MeLine Follower, which fixed the issue at hand, much to our relief.

V. Battery voltage affecting mBot behaviour

As we tested our mBot, we realised that the voltage of the batteries significantly decreased throughout lab sessions, making our previous calibrations inaccurate and playing a part in us constantly recalibrating our colour array values. This was because the different voltages resulted in varying brightness levels of the LEDs, leading to inconsistent colour array values during each run. Unfortunately, we only realised this on the day of our final run where our mBot was behaving very differently from the previous session. After knowing this, we learnt that we should always ensure that the battery is charged before each lab and continuously charge it in between testings. This will ensure that we are always working with a 5 volt power source and nothing less, resulting in more consistent performances.

VI. Loop errors

In the first iteration of the mBot's code, we used a `while` loop to ensure that the mBot would correct itself to the centre of the lane before attempting any of its other functions.

```
void loop()
{
  ...
  leftDistance = measureLeftDistance();
  while (leftDistance <= tooClose) {
    veerRight();
    leftDistance = measureLeftDistance();
   }
  ...
}
```

This arrangement worked, until we began to implement the Colour Identification Sequence. We ran into a bug where the mBot constantly ran past the black stopline without stopping at all. After troubleshooting, we realised that the likely reason was because the mBot was stuck within the `while` loop (See Left), and unable to exit until it was perfectly aligned in the middle of the lane.

Our solution to this issue was 2-pronged. First, we increased the acceptable margin of error for lane correction. In doing so, the mBot would not trigger its correction sequences so readily. Secondly, we realised that since the `void` loop itself was also a loop, we could do away with the `while` loop altogether. Instead, we adopted a "fixed correction" approach. Upon sensing that the left wall is too close, the mBot would perform a fixed correction of `veerRight()` for 10ms. After which, it would move on to the next function in the loop regardless of its position in the lane, before repeating the loop again.

The rationale for such a process being as such: by cutting out enough redundant delays, we could make the frequency of the main loop high enough to cycle through a correction function multiple times. Even if the first fixed correction did not fully correct the mBot to the centre of the lane, the next one would be repeated in time, preventing the mBot from bumping into the walls.

However, the acceptable margin could not be too narrow and the correction could not be too drastic, otherwise each correction might immediately veer the mBot into the trigger zone of the opposing correction sequence, causing it to jitter as mentioned in part (III).

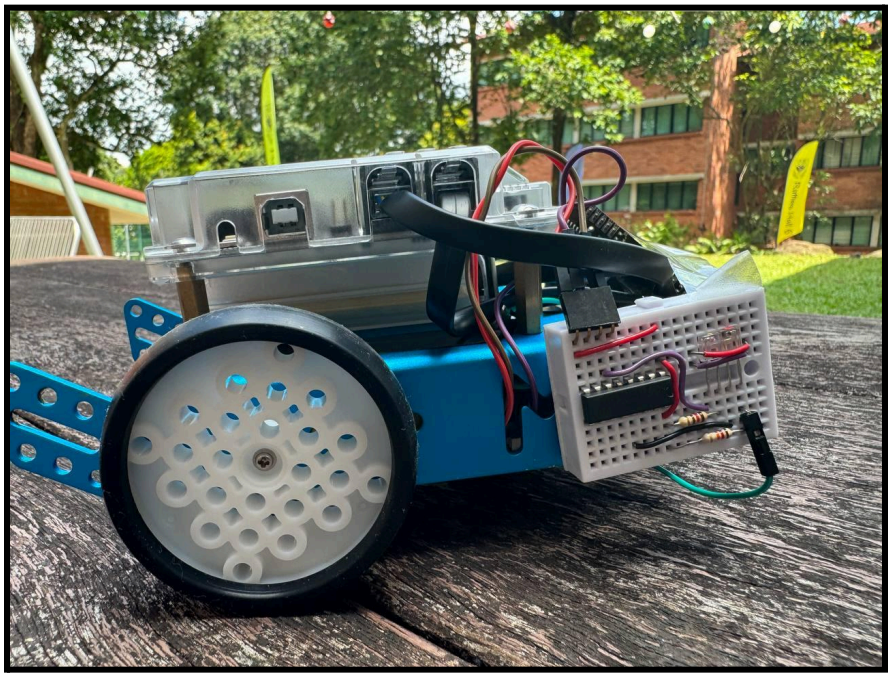
By trial and error, we eventually found the right harmony of these variables to ensure a smoothly correcting mBot that manages to avoid collision with walls, even at sharp angles.

WORK DIVISION

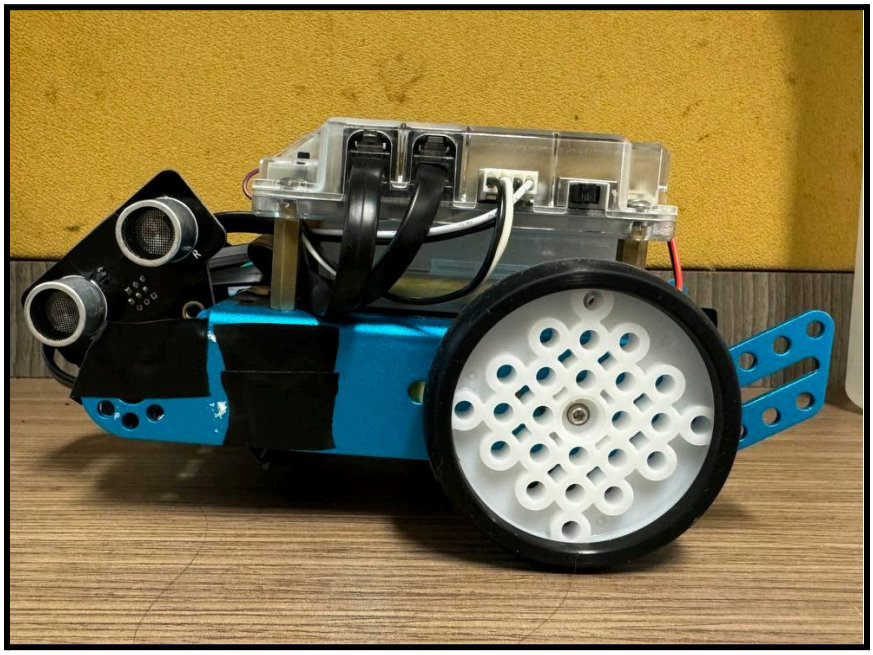
Member	Task Allocated
Giggs	- Wrote the code for Ultrasonic and IR sensor - Wrote the code for the MeBuzzer - Overall data collection and debugging
Timothy	- Wrote the code for Ultrasonic and IR sensor - Overall data collection and debugging
Denzel	- Wrote the code for colour sensor calibration and detector - In charge of calibration - Overall data collection and debugging
Beverly	- Set up the circuits for the IR and Ultrasonic sensors - Neatened the wires - Wrote the code for the MeLine Follower - Overall data collection and debugging

APPENDIX

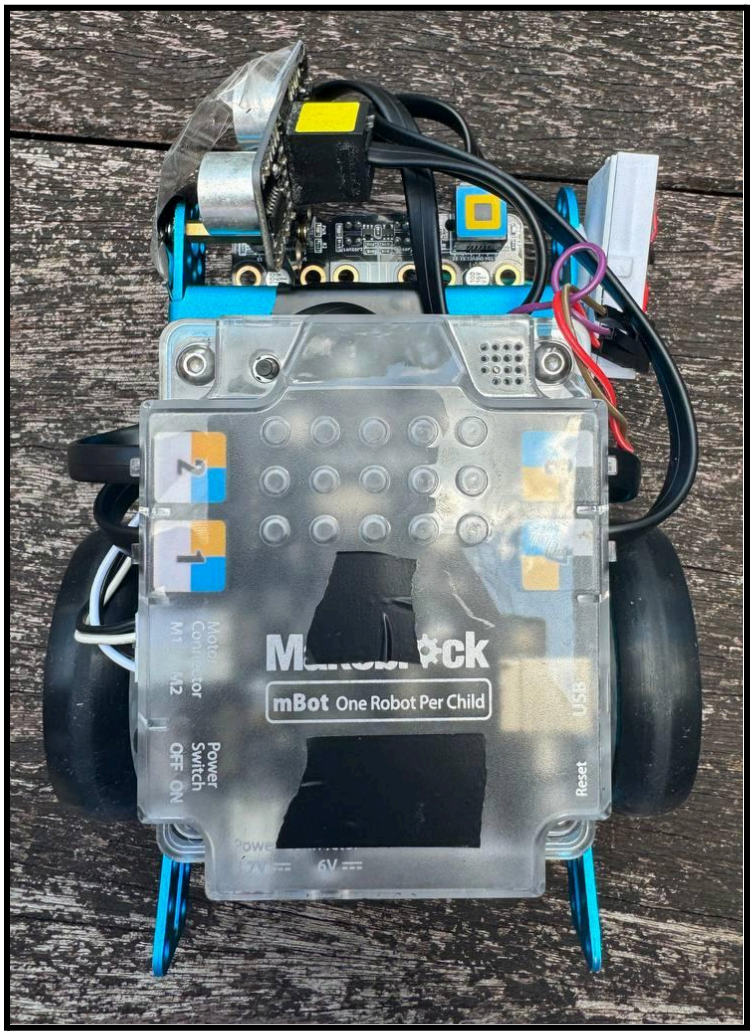
Appendix 1: Front view of mBot



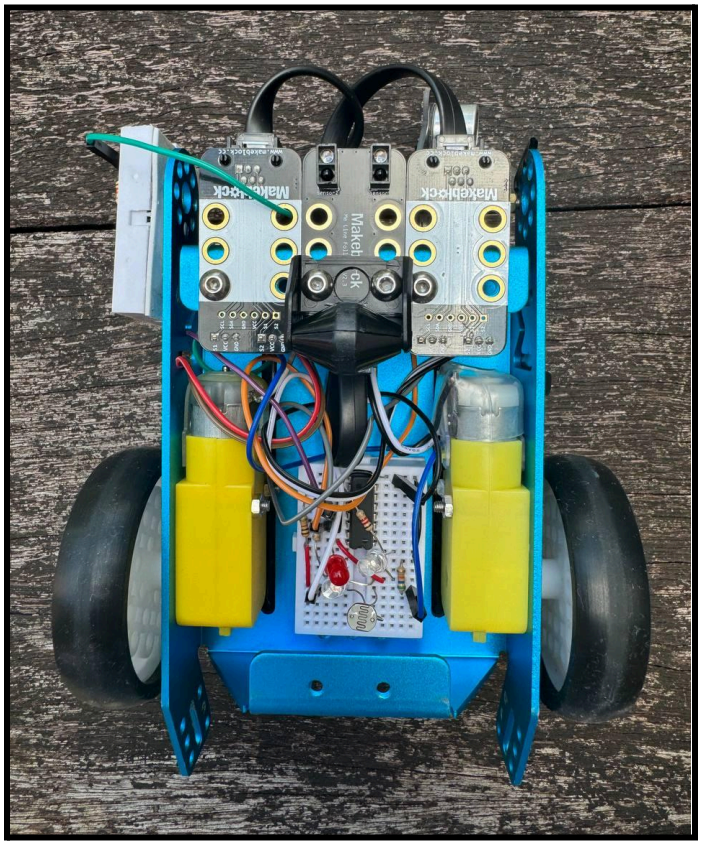
Appendix 2: Right view of mBot



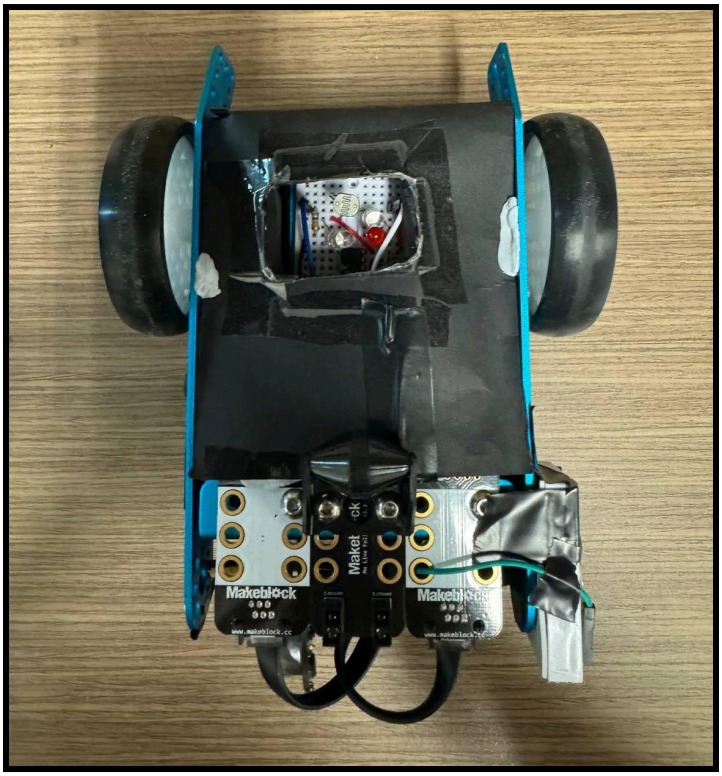
Appendix 3: Left view of mBot



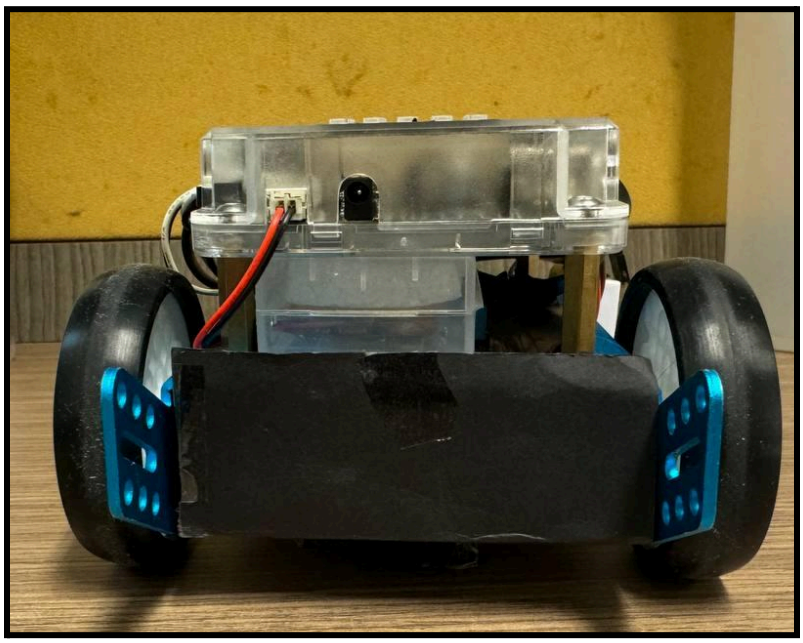
Appendix 4: Top view of mBot



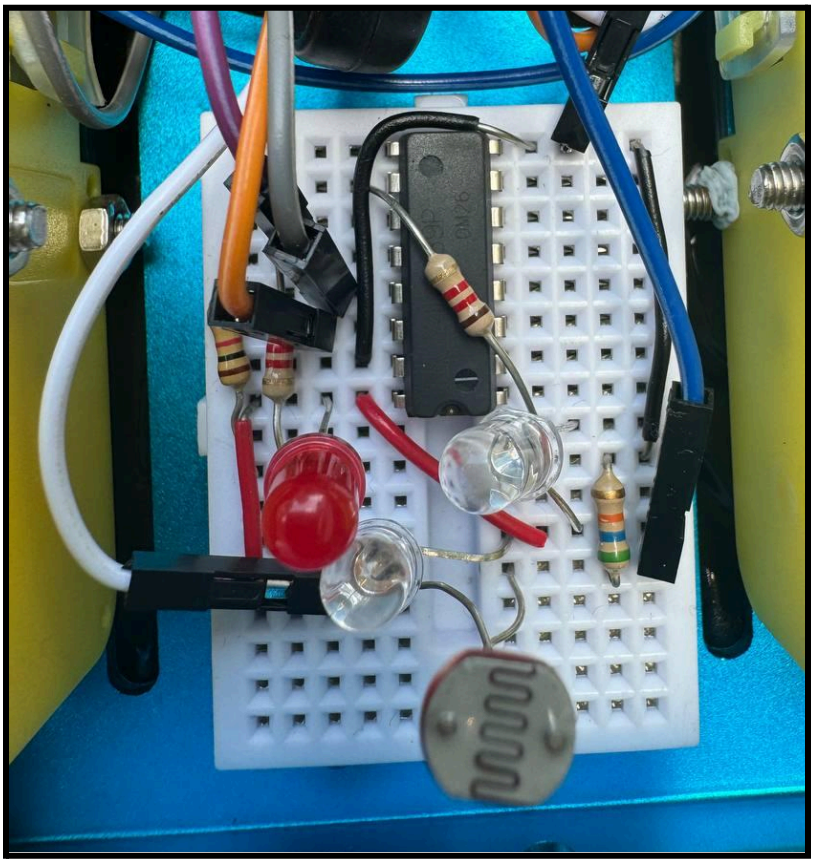
Appendix 5: Under view of mBot



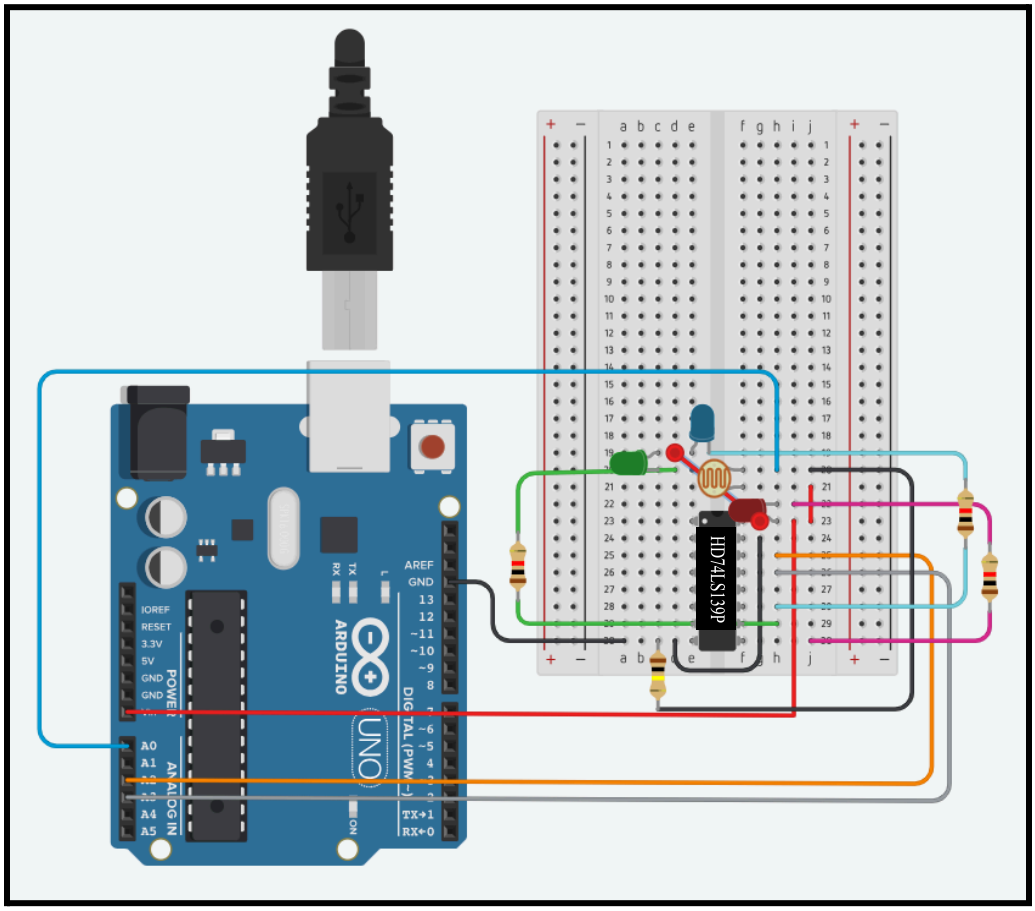
Appendix 6: Under view of mBot (With Skirting)



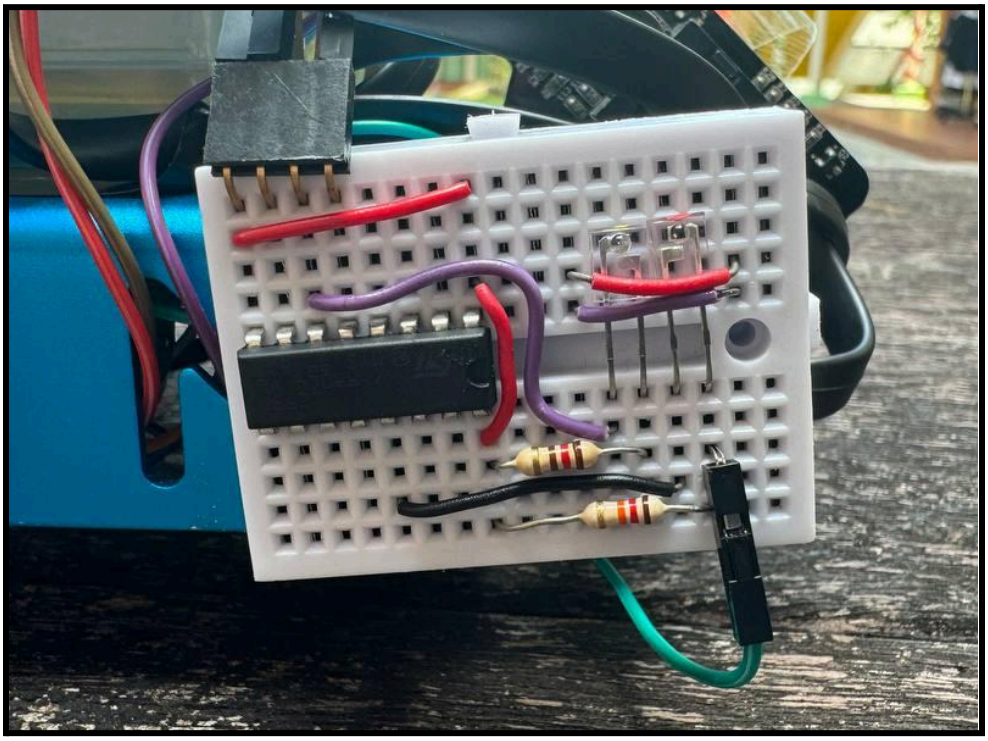
Appendix 7: Rear view of mBot



Appendix 8: Circuit for Colour Sensor



Appendix 9: Circuit Diagram for Colour Sensor



Appendix 10: Circuit for IR Sensor