CG1111A

ENGINEERING PRINCIPLES AND PRACTICE I

THE A-MAZE-ING RACE

B03-S4-T4

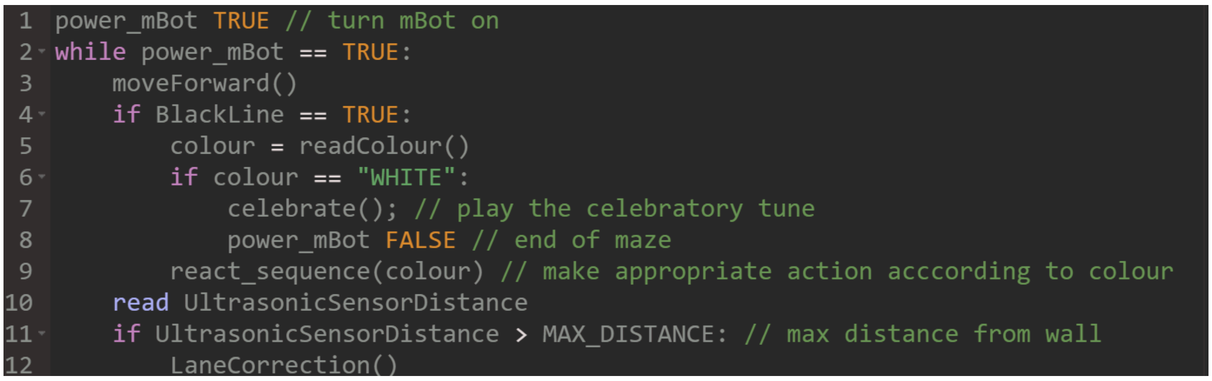
Prepared by:

Member	Student ID
Sim Chin Kai Alson	A0307409E
Tan Feng Yuan	A0305883X
Then Yi Ting Noelle	A0302930R

TABLE OF CONTENTS

1 GENERAL ALGORITHM

The bot was designed with two generalised sequences in mind: a colour reaction sequence and a distance correction sequence with the use of an ultrasonic sensor and an infra-red (IR) emitter and detector. By dividing them into subsystems, time wastage was minimised as we were able to better troubleshoot each subsystem more efficiently.



```
power_mBot TRUE // turn mBot on
while power_mBot == TRUE:
    moveForward()
    if BlackLine == TRUE:
        colour = readColour()
        if colour == "WHITE":
            celebrate(); // play the celebratory tune
            power_mBot FALSE // end of maze
        react_sequence(colour) // make appropriate action acccording to colour
    read UltrasonicSensorDistance
    if UltrasonicSensorDistance > MAX_DISTANCE: // max distance from wall
        LaneCorrection()
```

After the mBot is powered on, the line follower is activated and used to detect the presence of a black strip. If the mBot was indeed above a black strip, it would then stop and activate the colour reading algorithm, led by the appropriate colour reaction sequence. The table below details the possible actions the mBot would activate if a certain colour is read correctly.

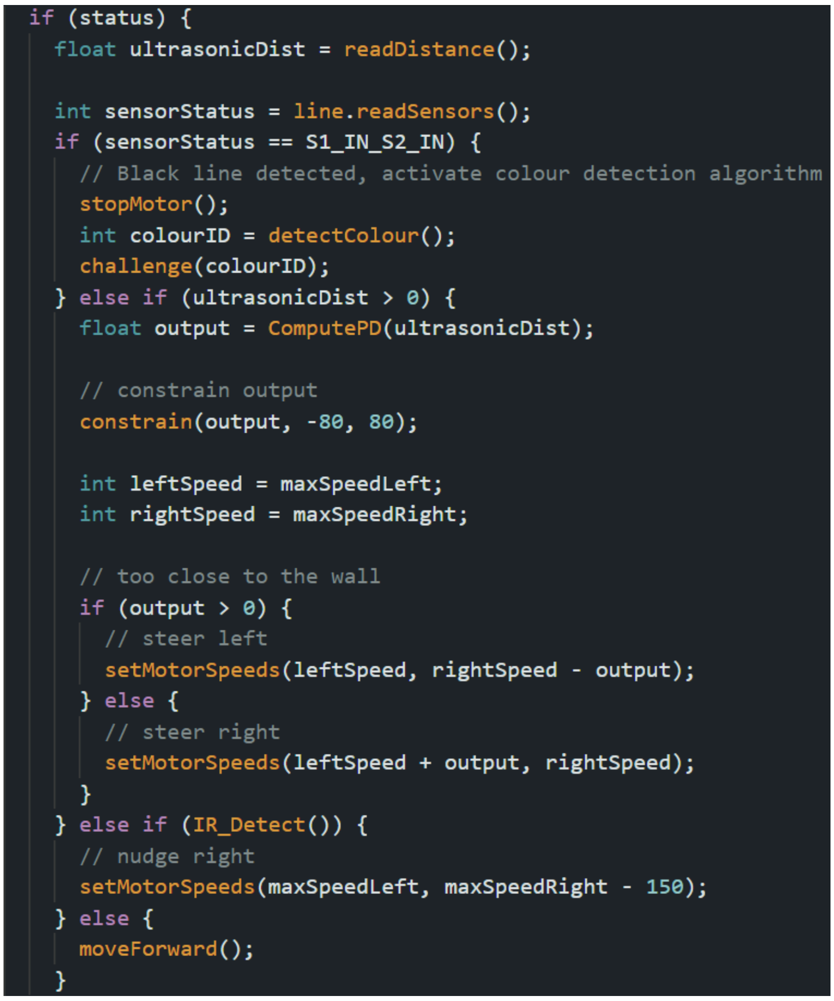
Colour	Interpretation
Red	Left-turn
Green	Right turn
Orange	180° turn within the same grid
Pink	Two successive left-turns in two grids
Light Blue	Two successive right-turns in two grids

In the absence of a black strip, the mBot would continue advancing, while reading proximity values from the ultrasonic and IR sensors attached to its sides. The ultrasonic and IR proximity correction sequences would be called if the proximity between the mBot and the barrier is not within the acceptable range.

2 SUBSYSTEMS

2.1 LANE CORRECTION

For the lane correction sequence, we took inspiration from a Proportional-Derivative (PD) controller, a feedback loop mechanism that would minimise the instability and jitteriness in the movement of our mBot. The ultrasonic sensor was the main instrument in enabling our PD loop to function, and it was mounted on the right side of the mBot.



```
if (status) {
  float ultrasonicDist = readDistance();

  int sensorStatus = line.readSensors();
  if (sensorStatus == S1_IN_S2_IN) {
    // Black line detected, activate colour detection algorithm
    stopMotor();
    int colourID = detectColour();
    challenge(colourID);
  } else if (ultrasonicDist > 0) {
    float output = ComputePD(ultrasonicDist);

    // constrain output
    constrain(output, -80, 80);

    int leftSpeed = maxSpeedLeft;
    int rightSpeed = maxSpeedRight;

    // too close to the wall
    if (output > 0) {
      // steer left
      setMotorSpeeds(leftSpeed, rightSpeed - output);
    } else {
      // steer right
      setMotorSpeeds(leftSpeed + output, rightSpeed);
    }
  } else if (IR_Detect()) {
    // nudge right
    setMotorSpeeds(maxSpeedLeft, maxSpeedRight - 150);
  } else {
    moveForward();
  }
```

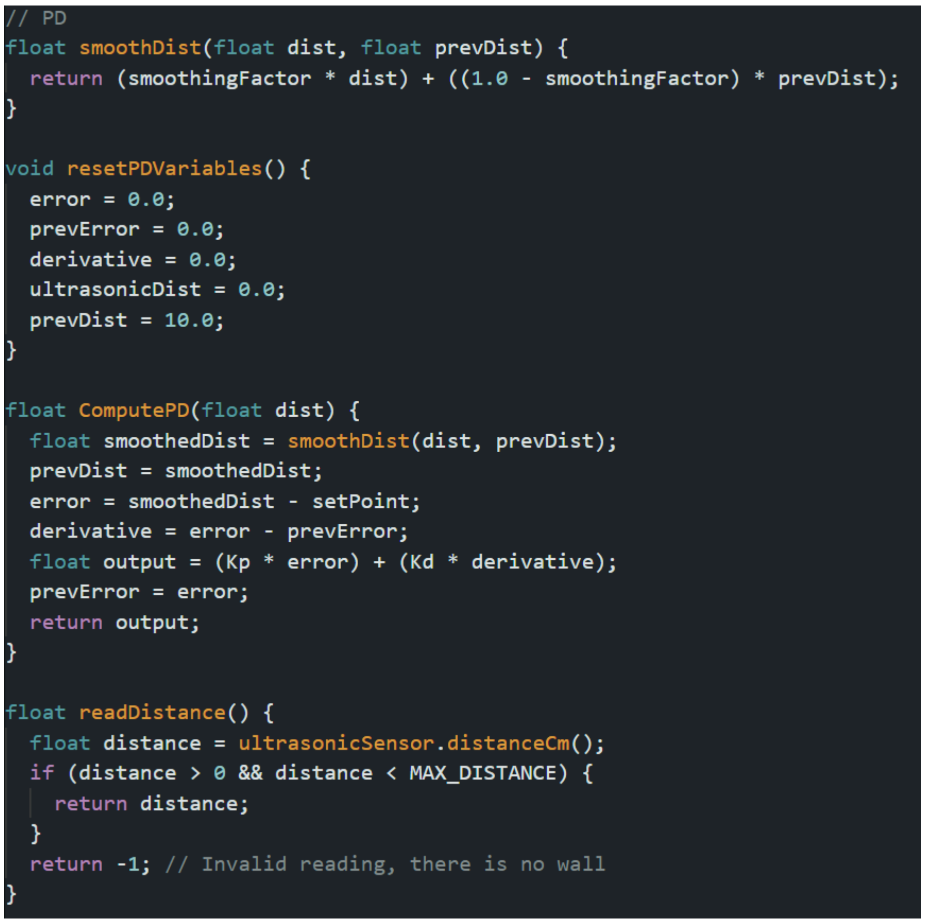
For our lane correction to start, the ultrasonic distance from the wall would be read and stored in the variable *ultrasonicDist.* If *ultrasonicDist* was more than 0, the PD system would start. This is so as *readDistance()* returns -1 if there is no wall for the sensor to detect, preventing the PD system from correcting itself unnecessarily when there is no wall detected, and instead activate the IR code or simply move forward. In ComputePD(), we took the input of *ultrasonicDist* and smoothen the distance with *smoothDistance()*. *prevDistance* and *smoothingFactor* which was previously set would be used to compute the smoothed distance with the formula:

$$smoothingFactor * ultrasonicDist + ((1.0 - smoothingFactor) * prevDist)$$

We found that a smoothing factor of 0.5 gave us the best version of the lane correction to reduce noise and prevent the bot from being too jittery. We would then set *prevDist* to the smoothed distance for the next iteration of the loop. *error* would then be calculated by using the smoothed distance subtracted by *setPoint*. *setPoint* is a variable set to be the desired distance from the wall in cm, which in this case is 10.0cm. *derivative* would then be calculated with *error* and *prevError*. Both *derivative* and *error* would act as the main variables that dictate the PD system. *output,* which is the adjustment our mBot should make, would be calculated with this formula:

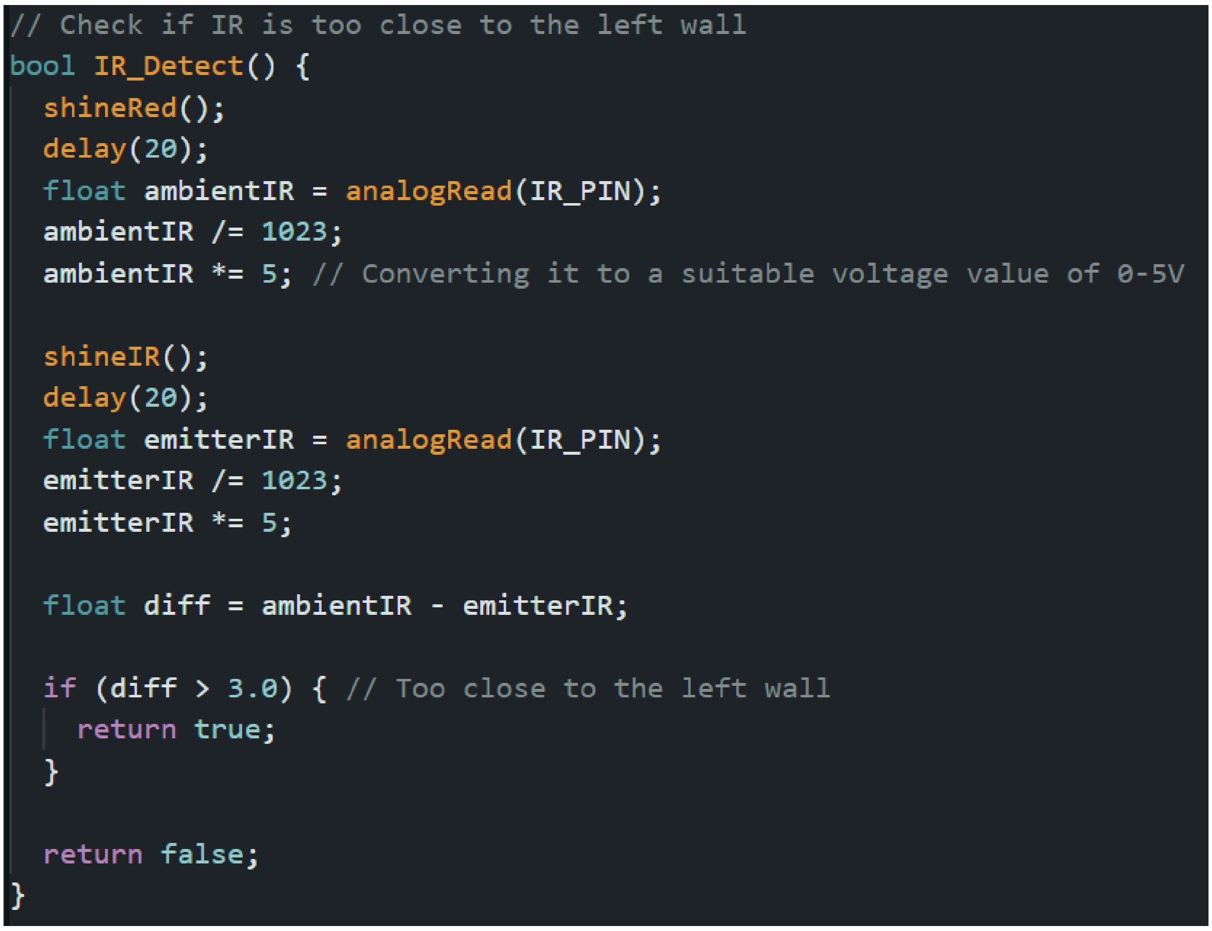
$$K_p * error + K_d * derivative$$

K_p and K_d which were set as the proportional and derivative constants would dictate the magnitude of the adjustment and we found that 60 and 20 respectively were suitable for the mBot. The movement of the left and right motors would be adjusted accordingly to *output* so the mBot would make the correct adjustments.

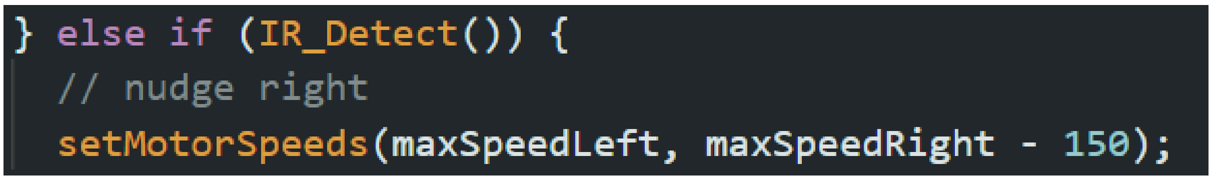


```
// PD
float smoothDist(float dist, float prevDist) {
  return (smoothingFactor * dist) + ((1.0 - smoothingFactor) * prevDist);
}

void resetPDVariables() {
  error = 0.0;
  prevError = 0.0;
  derivative = 0.0;
  ultrasonicDist = 0.0;
  prevDist = 10.0;
}

float ComputePD(float dist) {
  float smoothedDist = smoothDist(dist, prevDist);
  prevDist = smoothedDist;
  error = smoothedDist - setPoint;
  derivative = error - prevError;
  float output = (Kp * error) + (Kd * derivative);
  prevError = error;
  return output;
}

float readDistance() {
  float distance = ultrasonicSensor.distanceCm();
  if (distance > 0 && distance < MAX_DISTANCE) {
    return distance;
  }
  return -1; // Invalid reading, there is no wall
}
```

In case there were no walls for the ultrasonic sensor to read from, we produced the IR_Detect() function. In this function, we first read the ambient IR value and stored it in *ambientIR*. We then took this reading and converted it into a suitable voltage value with range 0-5V. The emitted IR value would then be read in the same fashion as the ambient IR. We then calculated *diff*, which was the difference between the emitted and ambient IR values. If *diff* exceeded the threshold (which is 3V in our case), this meant the mBot was too close to the left wall and the algorithm would return TRUE, so our mBot would swerve right and avoid the wall. Conversely, the algorithm would return FALSE, if *diff* was under the threshold, meaning that the mBot was far enough from the left wall.



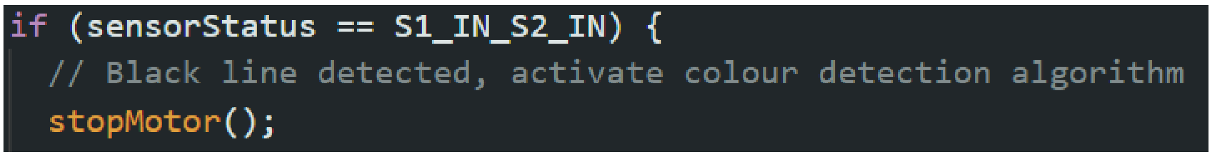
```
// Check if IR is too close to the left wall
bool IR_Detect() {
  shineRed();
  delay(20);
  float ambientIR = analogRead(IR_PIN);
  ambientIR /= 1023;
  ambientIR *= 5; // Converting it to a suitable voltage value of 0-5V

  shineIR();
  delay(20);
  float emitterIR = analogRead(IR_PIN);
  emitterIR /= 1023;
  emitterIR *= 5;

  float diff = ambientIR - emitterIR;

  if (diff > 3.0) { // Too close to the left wall
    return true;
  }

  return false;
}
```



```
} else if (IR_Detect()) {
  // nudge right
  setMotorSpeeds(maxSpeedLeft, maxSpeedRight - 150);
```

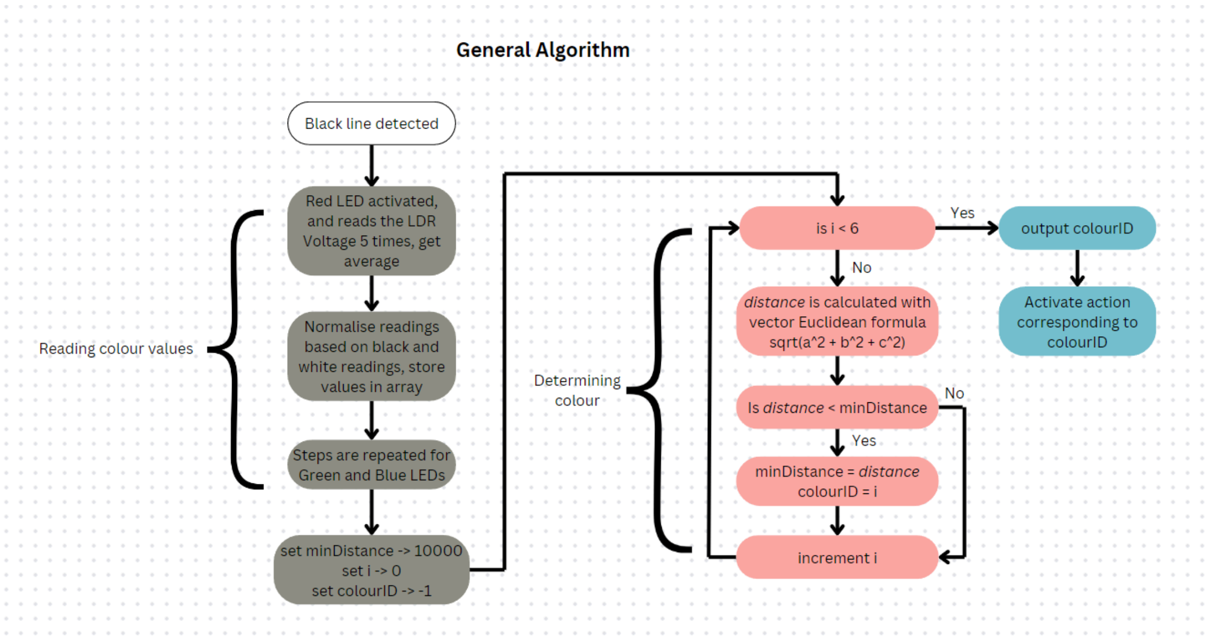
2.2 BLACK STRIP DETECTION

By following the information given in the project briefing, we made use of the Makeblock Line Sensor for the black strip detection. The sensor would return “IN” when the sensor is above the black strip, in which our mBot would proceed to read the colour below it with the LDR circuit.



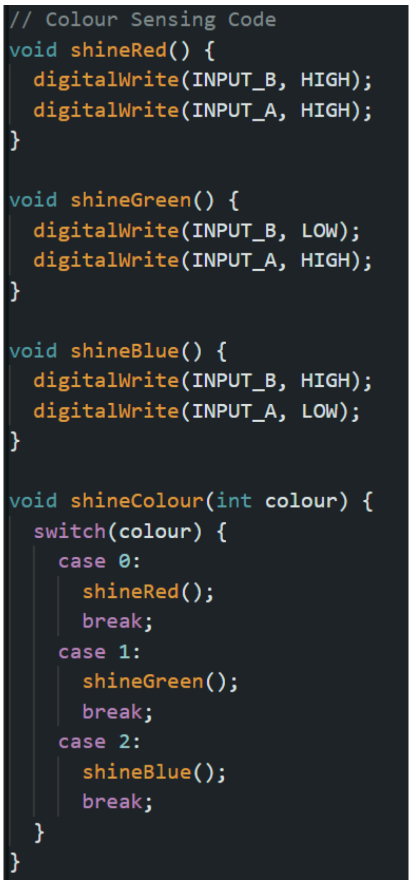
```
if (sensorStatus == S1_IN_S2_IN) {
  // Black line detected, activate colour detection algorithm
  stopMotor();
```

2.3 COLOUR INTERPRETATION SYSTEMS


General Algorithm
Black line detected
Red LED activated, and reads the LDR Voltage 5 times, get average
Normalise readings based on black and white readings, store values in array
Steps are repeated for Green and Blue LEDs
set minDistance -> 10000
set i -> 0
set colourID -> -1
Reading colour values
Determining colour
is i < 6
Yes
No
output colourID
Activate action corresponding to colourID
distance is calculated with vector Euclidean formula sqrt(a^2 + b^2 + c^2)
Is distance < minDistance
No
Yes
minDistance = distance
colourID = i
increment i

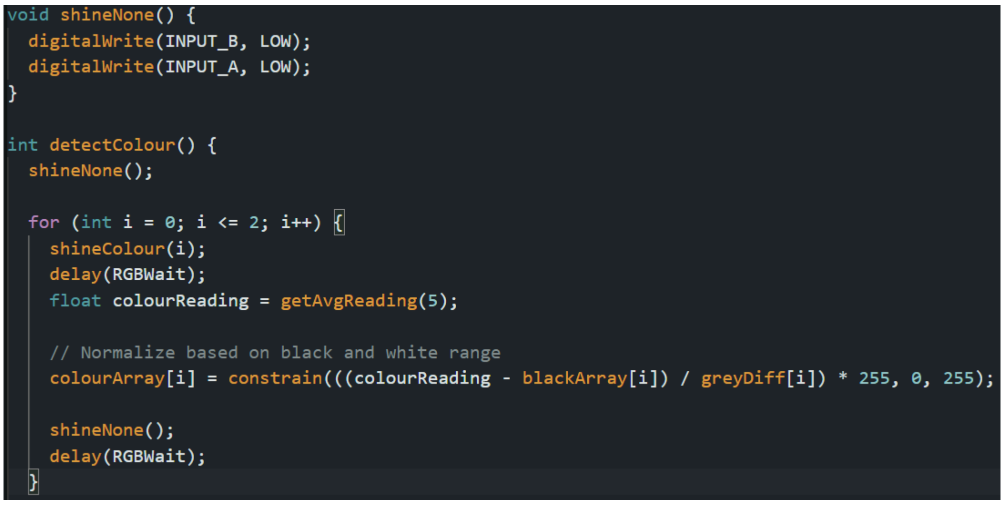
2.3.1 Red, Blue and Green LEDs

We configured the LEDs as per Appendix 4. Making use of the 2-to-4 decoder, this configuration specifically activates only the Red LED when both control pins A and B are set to HIGH, activates only the Green LED when control pin A is set to HIGH while pin B is set to LOW, and finally the Blue LED when control pin B is set to HIGH while pin A is set to LOW. By fitting code into a specific function to turn on each LED (called shineRed(), shineGreen() and shineBlue() respectively), we then nicely fit them into the shineColour() function as shown below.

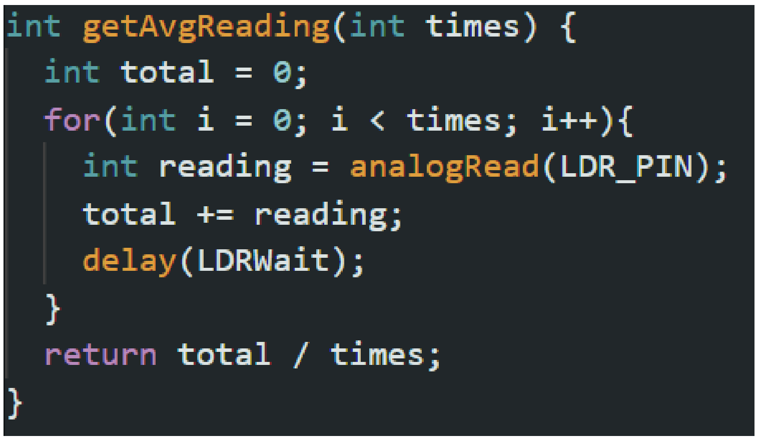


```
// Colour Sensing Code
void shineRed() {
  digitalWrite(INPUT_B, HIGH);
  digitalWrite(INPUT_A, HIGH);
}

void shineGreen() {
  digitalWrite(INPUT_B, LOW);
  digitalWrite(INPUT_A, HIGH);
}

void shineBlue() {
  digitalWrite(INPUT_B, HIGH);
  digitalWrite(INPUT_A, LOW);
}

void shineColour(int colour) {
  switch(colour) {
    case 0:
      shineRed();
      break;
    case 1:
      shineGreen();
      break;
    case 2:
      shineBlue();
      break;
  }
}
```

2.3.2 Detecting Colour



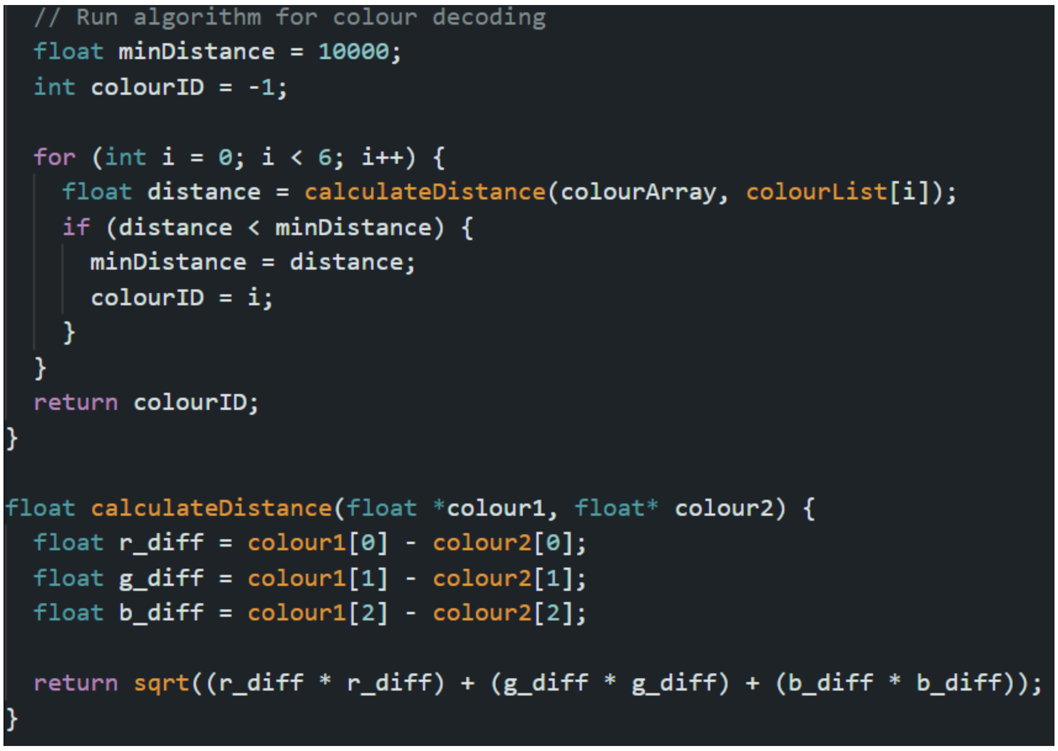
```
void shineNone() {
  digitalWrite(INPUT_B, LOW);
  digitalWrite(INPUT_A, LOW);
}

int detectColour() {
  shineNone();

  for (int i = 0; i <= 2; i++) {
    shineColour(i);
    delay(RGBWait);
    float colourReading = getAvgReading(5);

    // Normalize based on black and white range
    colourArray[i] = constrain(((colourReading - blackArray[i]) / greyDiff[i]) * 255, 0, 255);

    shineNone();
    delay(RGBWait);
  }
```



```
int getAvgReading(int times) {
  int total = 0;
  for(int i = 0; i < times; i++){
    int reading = analogRead(LDR_PIN);
    total += reading;
    delay(LDRWait);
  }
  return total / times;
}
```

We used the shineColour() function to activate the Red, Green and Blue LEDs to get a colour reading using the LDR, which is connected to pin A0. This colour reading would be calculated in the getAvgReading() function. The getAvgReading function() would take the colour reading as many times as requested, which would then be used to find the average. We decided to take the average of 5 readings for Red, Blue and Green, and the colour values would then be normalised based on the black and white colour values previously calibrated, and stored in the colourArray array, in the (Red, Green, Blue) order for each iteration of Red, Green and Blue. Before each iteration ends, we make sure to turn off all the LEDs to minimise any errors with the shineNone function(). The shineNone function() sets both control pins A and B to LOW, so that none of the LEDs would activate.

2.3.3 Identifying the Correct Colour



```
  // Run algorithm for colour decoding
  float minDistance = 10000;
  int colourID = -1;

  for (int i = 0; i < 6; i++) {
    float distance = calculateDistance(colourArray, colourList[i]);
    if (distance < minDistance) {
      minDistance = distance;
      colourID = i;
    }
  }
  return colourID;
}

float calculateDistance(float *colour1, float* colour2) {
  float r_diff = colour1[0] - colour2[0];
  float g_diff = colour1[1] - colour2[1];
  float b_diff = colour1[2] - colour2[2];

  return sqrt((r_diff * r_diff) + (g_diff * g_diff) + (b_diff * b_diff));
}
```

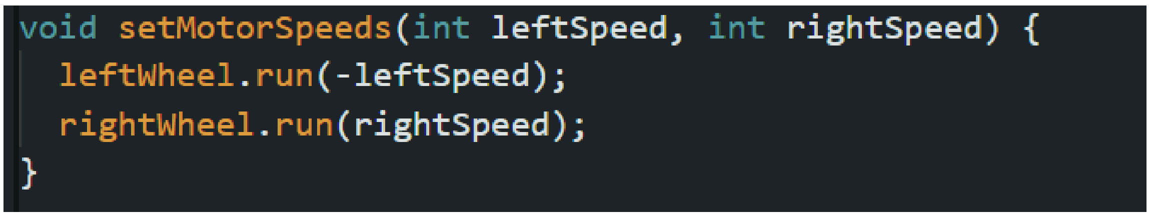
The general algorithm for colour detection calculates the euclidean distance between the RGB values stored in colourArray and the RGB values in the calibrated colourList. The algorithm will then detect a specific colour based on which of the distances calculated above is the least. This provides a wider threshold for error, as many different factors, such as the difference in ambient lighting or the colour strip simply being a different shade could affect the colour detection should we have followed closely with the initial calibrated values. Hence, we found that this method of colour detection worked best for our project and minimised errors as much as possible.

Thus, our mBot is able to successfully detect and identify each individual colour and proceed to its appropriate action corresponding with the respective colour.

2.4 MOVEMENT AND TURNING

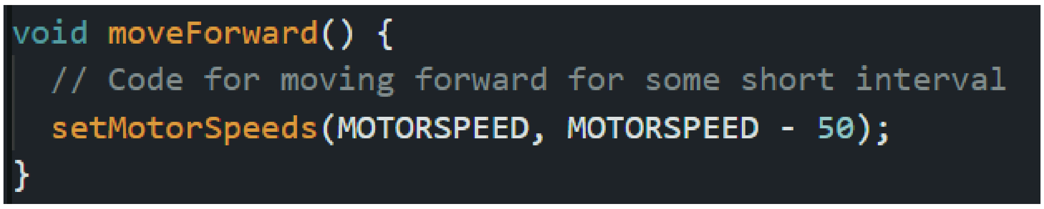
To make the writing of code easier, we wrote a general function setMotorSpeeds() that can be applied in general to other movement functions.

When the left motor takes on a negative value, the left wheel rotates anti-clockwise (when facing the exterior of the left wheel). On the other hand, when the right motor takes on a positive value, the right wheel rotates clockwise (when facing the exterior of the right wheel).



```
void setMotorSpeeds(int leftSpeed, int rightSpeed) {
  leftWheel.run(-leftSpeed);
  rightWheel.run(rightSpeed);
}
```

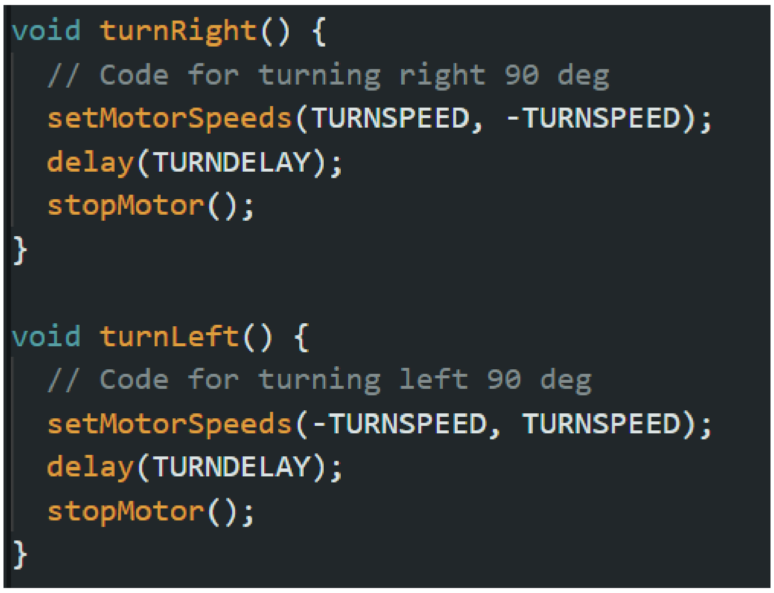
2.4.1 Moving Forward



```
void moveForward() {
  // Code for moving forward for some short interval
  setMotorSpeeds(MOTORSPEED, MOTORSPEED - 50);
}
```

We found that for some reason, the right motor was stronger than the left. Hence, we decreased the speed of the right motor to account for the difference.

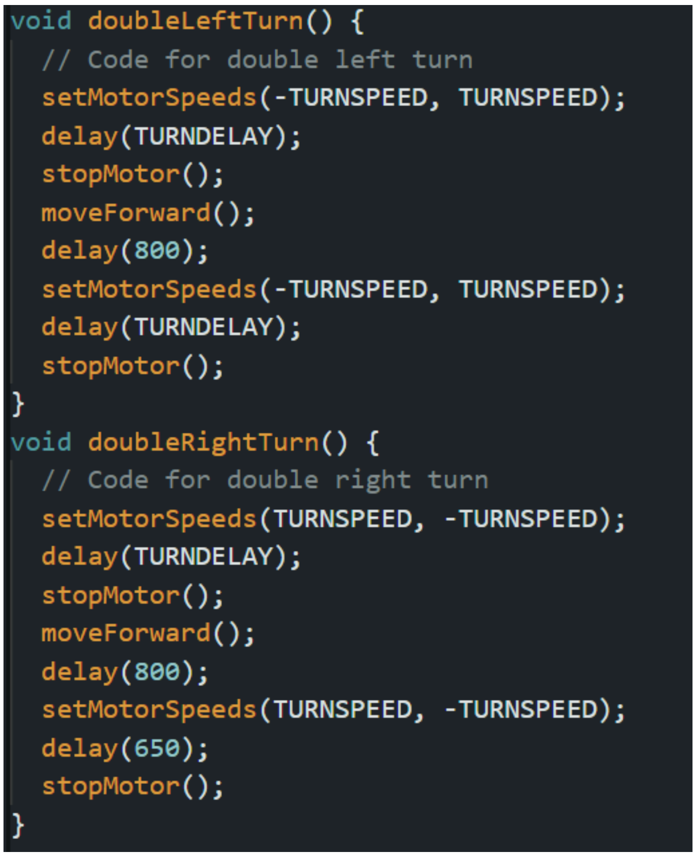
2.4.2 Turning Left and Right



```
void turnRight() {
  // Code for turning right 90 deg
  setMotorSpeeds(TURNSPEED, -TURNSPEED);
  delay(TURNDELAY);
  stopMotor();
}

void turnLeft() {
  // Code for turning left 90 deg
  setMotorSpeeds(-TURNSPEED, TURNSPEED);
  delay(TURNDELAY);
  stopMotor();
}
```

For turnRight(), we made both wheels turn anti-clockwise, which causes the mBot to turn right. On the other hand, for turnLeft(), we made both wheels turn clockwise, causing the mBot to turn left.

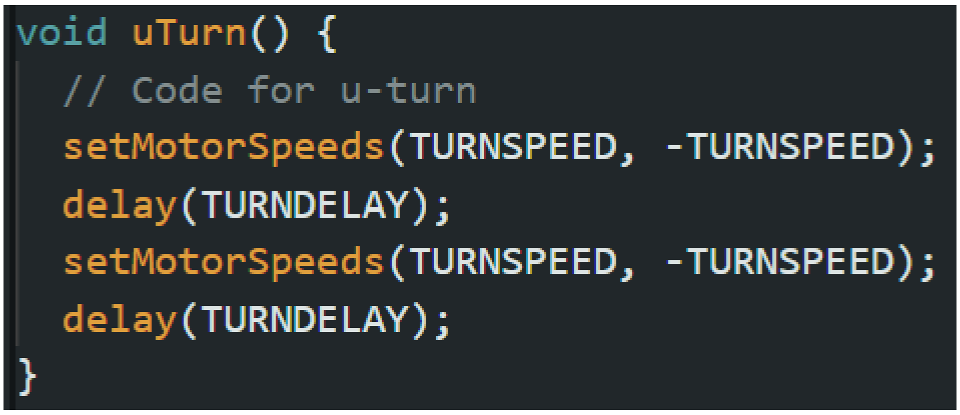
We found that a turning duration of 610ms would best allow our mBot to make a perfect 90-degree turn, hence initialised this value to TURNDELAY.



```
void doubleLeftTurn() {
  // Code for double left turn
  setMotorSpeeds(-TURNSPEED, TURNSPEED);
  delay(TURNDELAY);
  stopMotor();
  moveForward();
  delay(800);
  setMotorSpeeds(-TURNSPEED, TURNSPEED);
  delay(TURNDELAY);
  stopMotor();
}
void doubleRightTurn() {
  // Code for double right turn
  setMotorSpeeds(TURNSPEED, -TURNSPEED);
  delay(TURNDELAY);
  stopMotor();
  moveForward();
  delay(800);
  setMotorSpeeds(TURNSPEED, -TURNSPEED);
  delay(650);
  stopMotor();
}
```

For doubleLeftTurn(), we made the bot do a left turn, move forward for 800ms, then do a left turn again. In a similar fashion, for doubleRightTurn(), we made the bot do a right turn, move forward for 800ms, and do a right turn again.

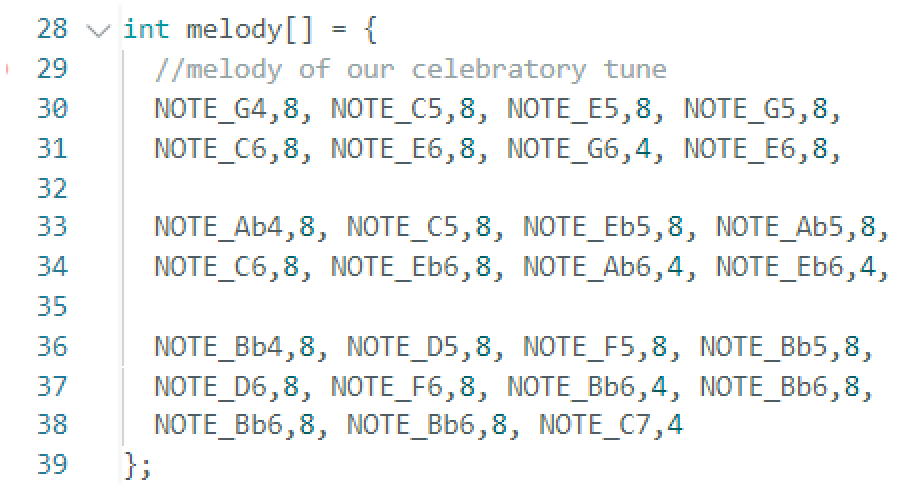
2.4.3 U-Turn



```
void uTurn() {
  // Code for u-turn
  setMotorSpeeds(TURNSPEED, -TURNSPEED);
  delay(TURNDELAY);
  setMotorSpeeds(TURNSPEED, -TURNSPEED);
  delay(TURNDELAY);
}
```

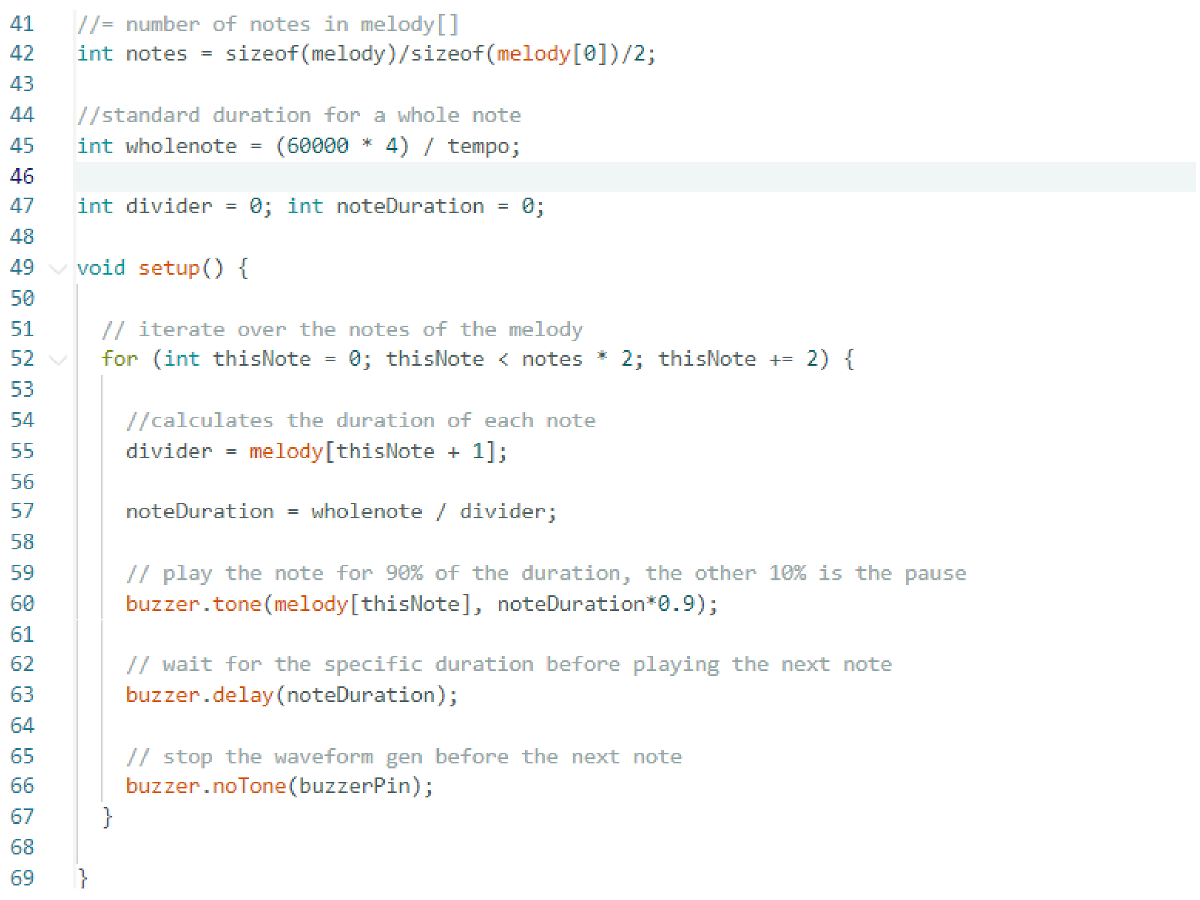
Our uTurn() function makes the mBot turn right twice, achieving a 180-degree turn effect.

2.5 CELEBRATORY TUNE



```
int melody[] = {
  //melody of our celebratory tune
  NOTE_G4,8, NOTE_C5,8, NOTE_E5,8, NOTE_G5,8,
  NOTE_C6,8, NOTE_E6,8, NOTE_G6,4, NOTE_E6,8,

  NOTE_Ab4,8, NOTE_C5,8, NOTE_Eb5,8, NOTE_Ab5,8,
  NOTE_C6,8, NOTE_Eb6,8, NOTE_Ab6,4, NOTE_Eb6,4,

  NOTE_Bb4,8, NOTE_D5,8, NOTE_F5,8, NOTE_Bb5,8,
  NOTE_D6,8, NOTE_F6,8, NOTE_Bb6,4, NOTE_Bb6,8,
  NOTE_Bb6,8, NOTE_Bb6,8, NOTE_C7,4
};
```

The melody array would store each note played in sequence. Each note frequency (e.g Note_G4) would be defined first, followed by a number which would represent a divider, which would be used to calculate how long the notes should be played comparatively to each other. For example, a note with 8 next to it would have a shorter beat than a note with 4 next to it. The code above is actually the "Mario Stage Win" tune!



```
//= number of notes in melody[]
int notes = sizeof(melody)/sizeof(melody[0])/2;

//standard duration for a whole note
int wholenote = (60000 * 4) / tempo;

int divider = 0; int noteDuration = 0;

void setup() {

  // iterate over the notes of the melody
  for (int thisNote = 0; thisNote < notes * 2; thisNote += 2) {

    //calculates the duration of each note
    divider = melody[thisNote + 1];

    noteDuration = wholenote / divider;

    // play the note for 90% of the duration, the other 10% is the pause
    buzzer.tone(melody[thisNote], noteDuration*0.9);

    // wait for the specific duration before playing the next note
    buzzer.delay(noteDuration);

    // stop the waveform gen before the next note
    buzzer.noTone(buzzerPin);
  }

}
```

The code shown above would calculate the total number of notes in the melody array, then find the general duration of a whole note (a note that lasts 4 beats for the respective tempo). Within the loop, noteDuration is set to find the duration of each note. Each note would be played for 90% of the calculated noteDuration, so there would be a small 10% pause, before the next note is played. By using our code above, if the tune is to be changed, only the melody array needs to be changed.

3 CALIBRATION

3.1 COLOUR SENSOR

In order to ensure that our colour sensor is able to detect colour accurately and consistently, we repeatedly calibrated the colour readings, as well as tested them rigorously throughout the entire process of the project. In addition, we tested and calibrated on different mazes as well to ensure that the colour detection algorithm was consistent.

In the colour detection algorithm itself, an average across 5 different readings is taken, which minimises error when detecting colour during a run.

Lastly, we found that charging the mBot to its full capacity is also extremely important in its colour detection process, hence we made it a point to charge the mBot every few runs to maintain its accuracy in detecting colour.

3.2 IR SENSOR

As the IR sensor depends heavily on the amount of ambient IR present, we had to go around different mazes to experiment and observe the different ambient IR values and figure out what sticks and what does not. After some trial and error, we found that making the mBot swerve when the voltage dip was higher than 3V was the most accurate for us as it yielded the most consistent results no matter which maze we tested the algorithm at.

4 TROUBLESHOOTING AND OPTIMISATION

4.1 RGB LED CIRCUIT BUILDING

Building the RGB LED circuit proved to be one of the main challenges for us. The circuit design was time-consuming and took the duration of two lab sessions for us to construct the first 'draft'. In the first circuit design 'draft', our main struggle was one, if not all, of the RGB lamps not lighting up, when powered with the test code given in Studio 12. The red RGB lamp would be lit even when not connected to any power, while the other two RGB lamps would not light up. This meant one of two things: either our circuit design was faulty; or the components were faulty. To overcome this, we referred to the Studio 12 briefing in order to guide our circuit design and replace each component every time we re-tested the circuit with the given code.

After a number of time-consuming replacements and trials and errors, we realised the breadboard was faulty. After refining the circuit design and replacing the breadboard, the RGB LED colour detection was fully functional.

4.2 INCONSISTENT TURNS

During the initial phases of programming, we ran into some trouble producing consistent 90 and 180 degree turns. During the first few trials and errors, our mBot would either turn too much or not turn enough.

In the pursuit of finding the best magnitude for the motor speed to facilitate a near-perfect 90 degree and 180 degree turn, we found that keeping the magnitude constant, while changing the time delay would yield better results. By changing just the time delay, we would only be focusing on just changing the angle in which our mBot would stop. By keeping the magnitude constant, we would not further complicate the process of finding the values to give a suitable turn, as the magnitude would also affect the speed of the turns of our mBot, thereby affecting the angle in which it ends at as well. We also realised some dirt would subtly clog up the wheels, which would affect the turns very slightly, so we made sure to clean every wheel every few trials.

5 REFLECTION AND CONCLUSION

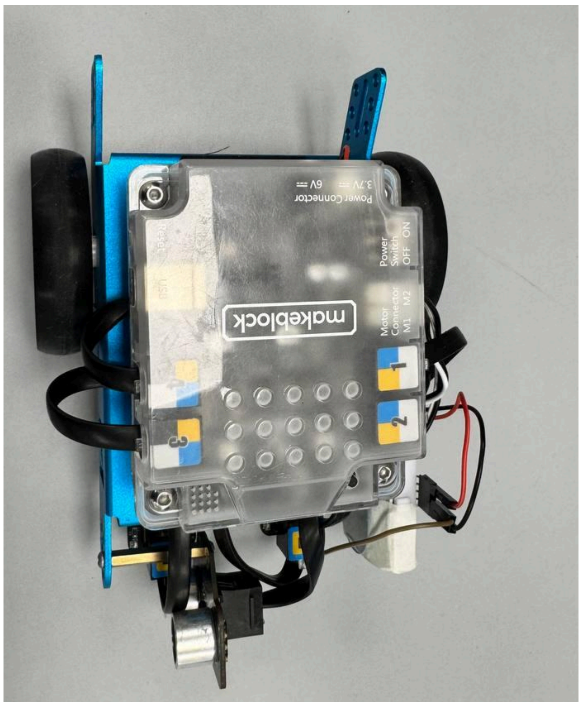
All in all, we feel that this project has allowed us to let all the skills we have acquired since the start of the EPP module come together and culminate into one tangible product, which is the mBot. Throughout this journey, we had a lot of fun, be it from figuring out how things worked in the studios to working together to build a functioning mBot despite being clueless and scared individuals having to tackle and adapt to university life for the first time. Although our mBot eventually went through some hiccups during the final run despite it undergoing perfect runs just before, we feel that we have done our very best and are satisfied with the outcome, as even though things did not go our way in the end, the process of getting to this point is what matters to us more.

In conclusion, working on this project has been a rewarding and fulfilling time for all three of us, and we look forward to what the future brings for us in EPP2.

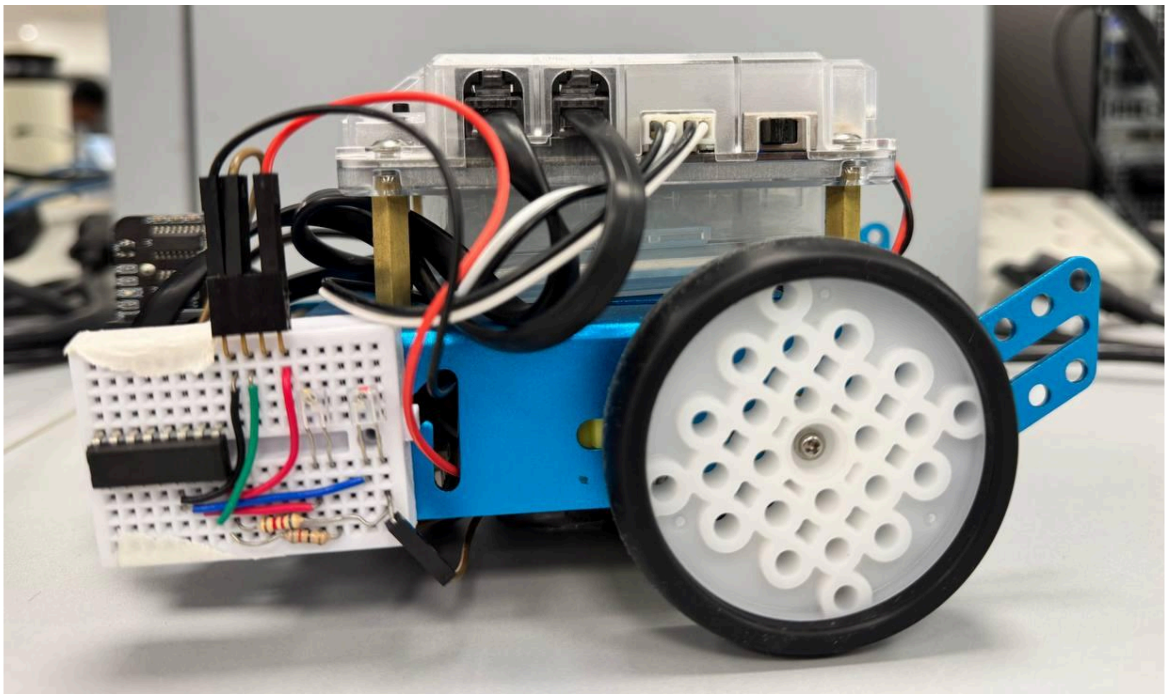
6 WORK DIVISION

Member	Work Allocated
Tan Feng Yuan	- Contributed to overall report - Contributed to overall design of mBot - Made flowcharts for the report - Worked on skirting - Worked on the Celebratory Tune code
Sim Chin Kai Alson	- Contributed to overall report - Contributed to overall design of mBot - Finalised design of skirting - Built ultrasonic sensor circuit - Worked on code
Then Yi Ting Noelle	- Contributed to overall report - Contributed to overall design of mBot - Worked on skirting - Built colour sensor and IR circuit - Worked on colour detection algorithm - Worked on movement algorithm

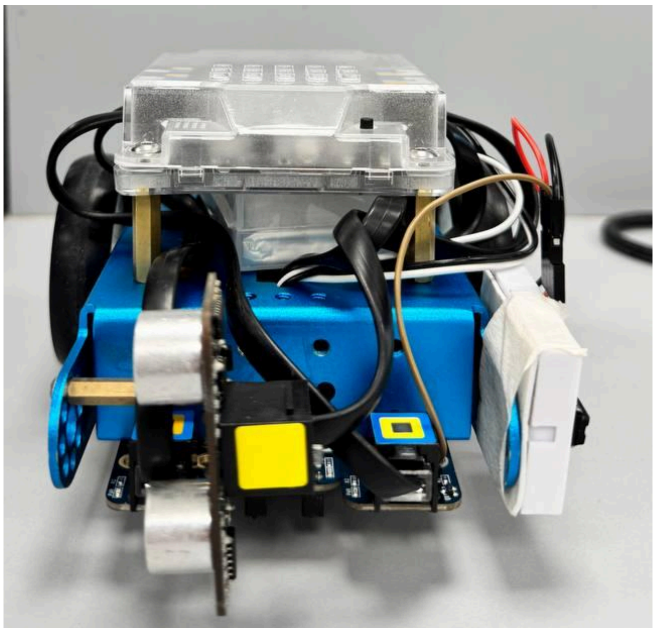
7 APPENDIX

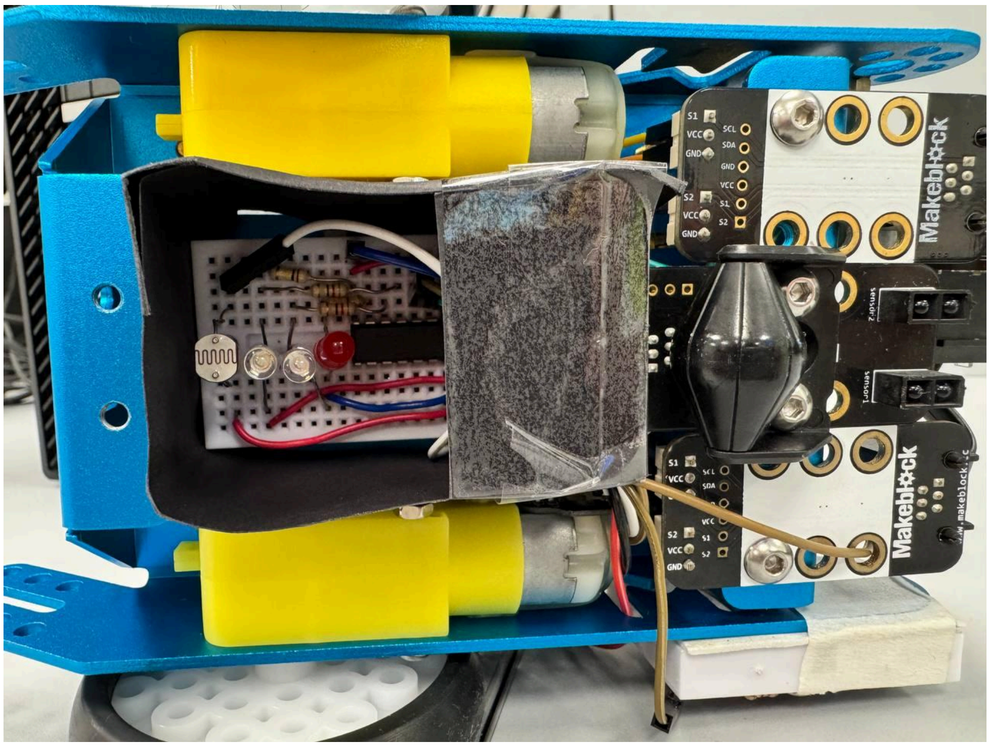


Appendix 1. Top view of our mBot.

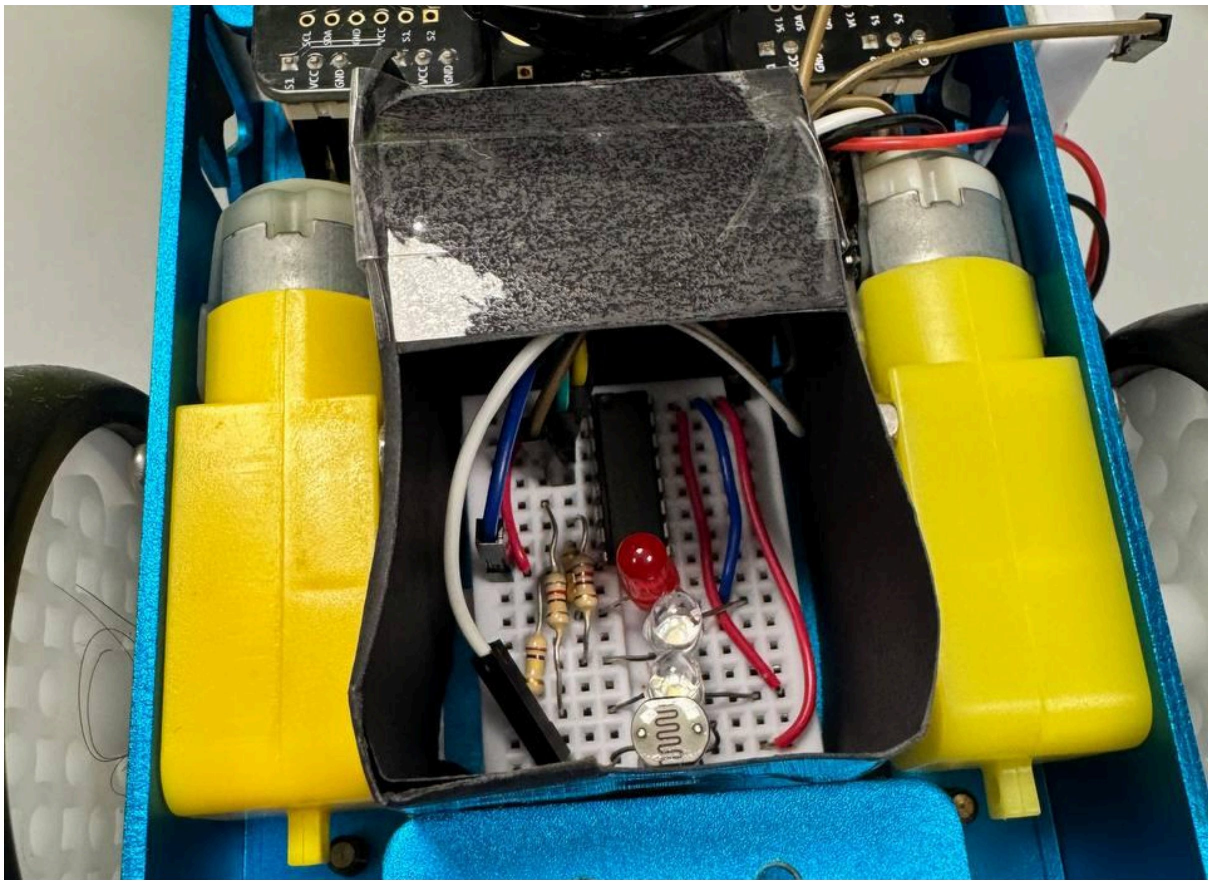


Appendix 2. Side view of our mBot, along with the IR Sensor


Appendix 3. Front view of our mBot. Ultrasonic sensor on the picture's left side, IR sensor on the right



Appendix 4. Bottom view of our mBot. LDR LED Circuit is attached to the bottom, along with the skirting



Appendix 5. A better look at the LDR LED Circuit.