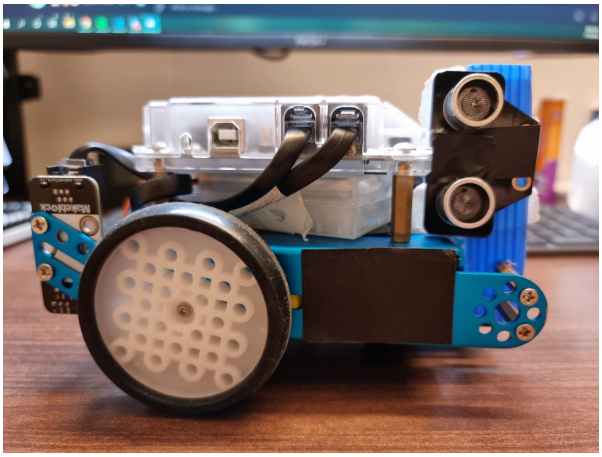
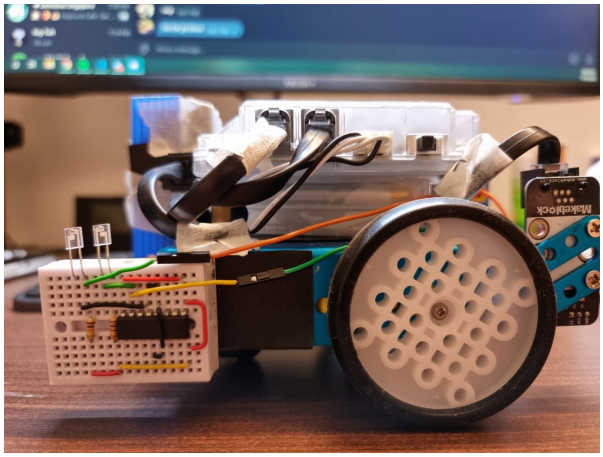
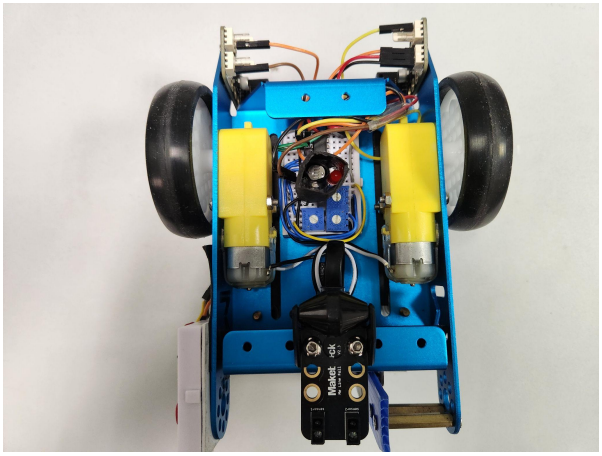
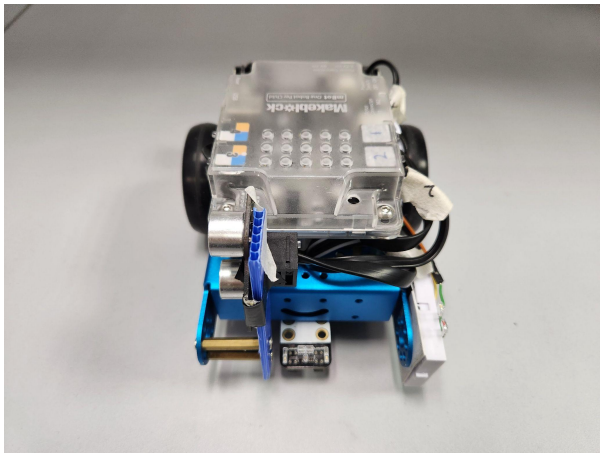
CG1111A A-maze-ing Race Project 2022

Studio Group No	B03
Section No	6B
Team No.	361
Team Members	Yeo Meng Han Tong Zheng Hong Stanley Wijaya Tang Jun Mei, Shanice

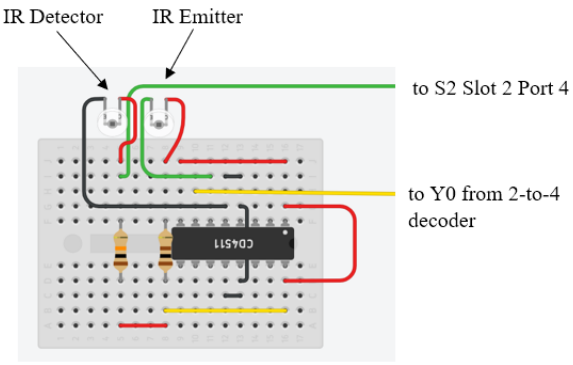
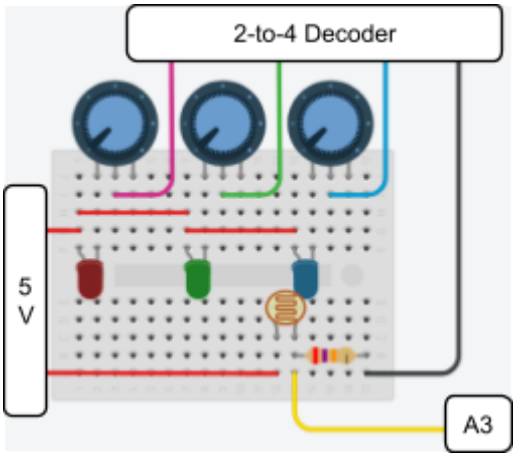
TABLE OF CONTENTS

1. Pictures

1.1. Pictures of the mBot

Right view	Left view
	
Bottom view	**Top view**
	

1.2. TinkerCAD model of circuits used

IR sensor circuit	Colour sensing circuit
	

2. Overall algorithm

Full code: https://github.com/FizzingForWurf/CG1111A

2.1. Flowchart of algorithm

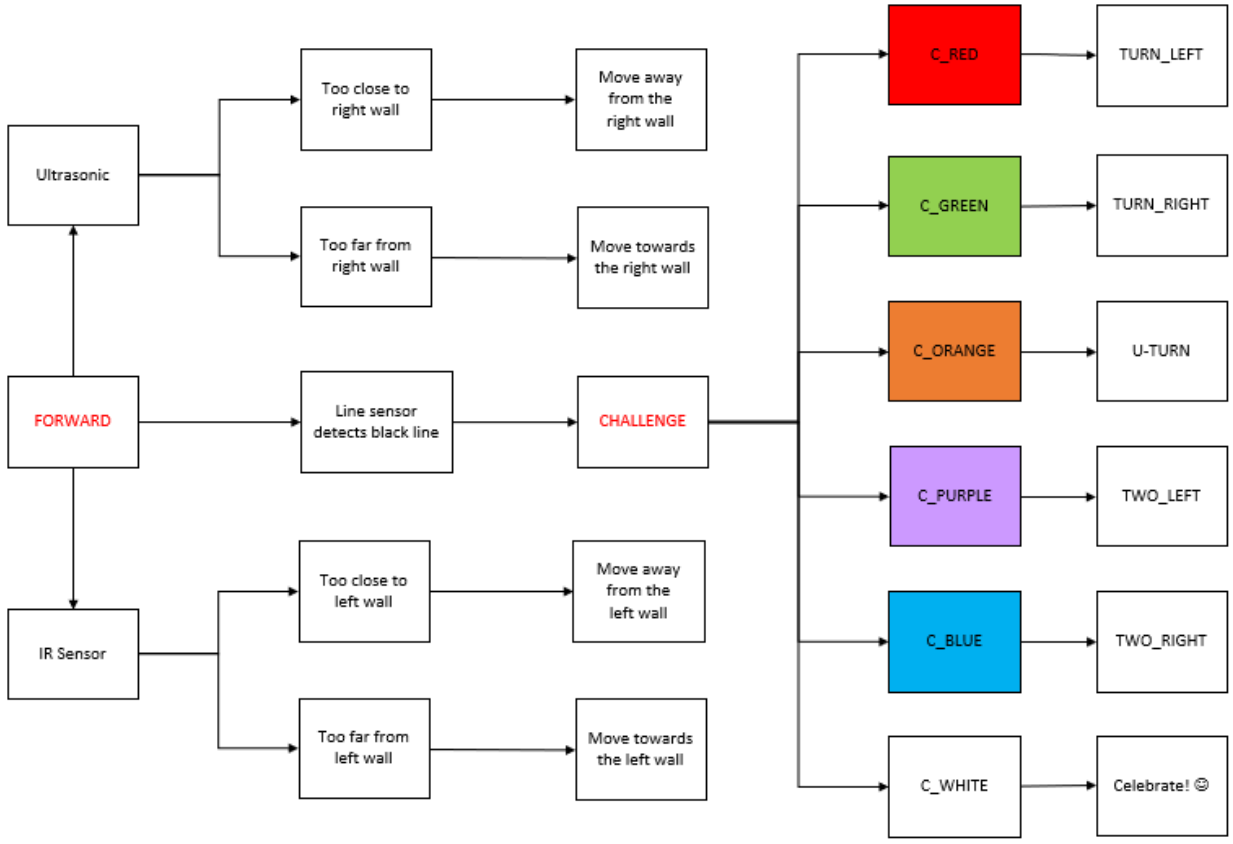


Figure 2.1: Overall Flow of the Algorithm

Before the mBot begins to navigate the maze, the colour sensor will have to be calibrated with white and black coloured paper.

While the mBot is in FORWARD state, the ultrasonic sensor and the IR sensor constantly checks the distance from both walls. If the ultrasonic sensor detects that the mBot is too close to the right wall, it will start correcting the mBot towards the left. Similarly, if the mBot is too far from the right wall, it will start correcting the mBot towards the right. The IR sensor on the left side has the same working principle as the ultrasonic sensor.

When the line detector senses a black line, the mBot will stop and its state will be changed to CHALLENGE, indicating that it has arrived at a waypoint challenge. The colour sensor will then switch on the LEDs one by one (red, green, blue) for 3 iterations and each time, the LDR will detect the amount of light reflected off the paper. Based on the readings, the colour will be determined based on the intensity values gained from the LDR and the mBot will move accordingly.

3. Navigation

3.1. Ultrasonic Sensor

3.1.1. Implementation

Measures the time taken for the ultrasonic waves to travel from the emitter to a surface and back into the receiver. The distance of the surface away from the sensor can be computed by finding the product of half the time taken and the speed of sound (around 340 m/s). The ultrasonic sensor is used with the PID algorithm to maintain a constant distance away from the wall.

3.1.2. Ultrasonic Sensor - Improving Robustness

As seen in section 1.1, we've built a custom vertical mount to elevate the position of the ultrasonic sensor. This achieves the following 2 effects:

1. Reduces the form factor of the mBot. Previously, our ultrasonic sensor was mounted at a 45° angle on the right side mount. It protrudes slightly outside from the front of the original form factor of the mBot chassis. With this custom mount, we're able to fit fully within the form factor. This facilitates manoeuvrability and reduces the likelihood of hitting walls while turning.

2. Raises above the height of the wooden side walls of the playfield, to prevent unwanted input for the PID algorithm. Previously, due to the low mounting of our ultrasonic sensor, we're constantly detecting the side walls and treating them the same as the white wall. This causes the mBot to jitter unpredictably when we encounter a no-wall scenario on the perimeters of the playfield. By raising the ultrasonic sensor, we can ignore this wall and instead focus on sensing the metallic pillars on the field to correct the PID when there's no wall present. The mount is specifically measured to be within the height of the metallic pillars.

3.2. PID algorithm - Keeping the mBot straight

3.2.1. The need for PID algorithm

While the mBot is travelling through the maze between the waypoint challenges, it has to travel a straight path and not bump into any walls. Simply setting both motors to the same value does not guarantee that the mBot will always travel in a straight line. This is because the motors spin with different speeds in real life due to manufacturing inaccuracies and other environmental factors. In our case, our mBot tends to deviate left when both motors are set to the same speed. Hence, our group implemented the PID control algorithm that utilises the ultrasound sensor's ability to detect distances to keep the mBot straight.

3.2.2. PID algorithm explained

The PID algorithm is a closed feedback control algorithm that aims to maintain a desired state of a system by making corrections based on the feedback it receives from a data source. It is made of 3 components:

- Setpoint: The desired state that the system should achieve and maintain
- Error: The difference between the current state of the system and the desired state (setpoint)
- Output: The correction that needs to be applied to the system to correct the error and ensure the setpoint is reached

In the case of the mBot, the system is the mBot itself and the desired state (setpoint) is the mBot positioned ideally 10 cm away from the right wall at all times. The error term would then be the difference between the current distance the mBot is from the right wall (read by the ultrasonic sensor) and the desired state of 10 cm. The output is the motor speed correction that would bring the mBot to just 10 cm away from the wall without significant over or under correction. In other words, the algorithm takes the distance measured by the ultrasonic sensor as input and outputs a motor correction that ensures that the mBot is 10 cm away from the wall regardless of other environmental influences like friction or unequal motor speeds.

3.2.3. Setting up main loop for PID

Given that the PID algorithm operates as a closed feedback loop, it must receive constant updates on the mBot's current distance away from the wall to make the corresponding adjustment to maintain the desired 10 cm distance. Consequently, the ultrasonic sensor must be polled/measured in each iteration of the main loop while the mBot is moving before the PID output can be computed and applied to the motors. This is achieved by keeping the main arduino 'loop' function **short** and only performing these steps repeatedly:

1. Measure distance to wall from the ultrasonic sensor
2. Calculate the PID output based on the mBot's distance away from the right wall
3. Apply the PID output to both the left and right motors

However, how can we handle other functions such as colour detection and turning if the main loop must be short and only consist of the above steps while moving straight? This is where the use of the global_state variable comes into play. The global_state variable will consist of all the possible states the mBot can be in, given by the enum called Motion. Initially, the state will be set to FORWARD to ensure the mBot moves from the start.

```
enum Motion {
  TURN_LEFT, TURN_RIGHT, U_TURN, TWO_LEFT, TWO_RIGHT, CHALLENGE, FORWARD, FINISH
};
Motion global_state = FORWARD;  // Default state of motion is FORWARD (with PID)
```

Next, the state will be checked in each iteration of the main loop and the corresponding logic will be executed based on the current state of the mBot. As such, when the global_state is set to FORWARD, the main loop will poll the ultrasonic sensor and the PID correction can be applied to the mBot repeatedly at a relatively high frequency. The code is as follows:

```
1.  void loop() {
2.    if (global_state == FORWARD) {
3.      float correction = calculate_pid();
4.      if (correction == -1) move_forward();
5.      else move_forward_correction((int) correction);
6.
7.      if (has_reached_waypoint()) {
8.        stop_moving();
9.        global_state = CHALLENGE;
10.     }
11.   } else if (global_state == CHALLENGE) {
12.     ... // Color detection algorithm here
13.   } else if (global_state == TURN_LEFT) {
14.
15.   } ... // Other turn logic below
16. }
```

Additionally, the mBot line sensor will also be polled in the main loop to check if the mBot has currently reached a waypoint challenge. This is implemented in has_reached_waypoint():

```
/* Check if mBot line sensor has detected black line on the floor */
bool has_reached_waypoint() {
  int sensor_state = line_finder.readSensors();
  return sensor_state == S1_IN_S2_IN;
}
```

It will read the line sensor and check if both IR detectors detect black, signifying the mBot is over a black line on the floor and is thus at a waypoint challenge. The mBot is stopped and the global_state variable is set to CHALLENGE to execute the colour detection logic in the next iteration of the main loop.

3.2.4. Implementing PID

The PID output is computed with the *calculate_pid()* function. Firstly, the distance away from the right wall is obtained from the ultrasonic sensor. If the returned value is 0 or greater than 15 cm, it is likely that there is no wall present on the right side. This will return -1 to signify no wall is present. Otherwise, if a wall is present, the function will continue to compute the PID output.

$$u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(\tau)\,\mathrm{d}\tau + K_\mathrm{d} \frac{\mathrm{d}e(t)}{\mathrm{d}t},$$

The algorithm consists of 3 parts, the P (proportional), I (integral) and D (derivative) controllers given by the equation above. Each controller has an individual gain K_p, K_i and K_d which needs to be tuned for the algorithm to converge to the setpoint quickly and without oscillation around the setpoint.

Proportional control is used to reduce the error by adjusting the output according to the magnitude of the error. When the error is large (i.e. mBot is too far away or too close from the wall), the output will also be relatively large. This helps to correct the error quickly and steer the mBot back to the 10 cm setpoint. As the mBot goes closer to the setpoint of 10 cm, the error will decrease, causing the output to decrease as well. As a result, when the mBot is not at the setpoint, the P controller will create an output that will correct its position over time by reducing its output as the error decreases.

However, the P controller has a tendency to over-correct which can lead to oscillations around the setpoint. For example, if the mBot is 14 cm away from the right wall, as it corrects to the right towards the 10 cm setpoint, it might go over 10 cm and become 9 cm away from the wall, thus requiring a leftward correction thereafter. This is because the mBot is not parallel to the wall when it is at the setpoint, causing additional unwanted correction over the setpoint. This can be addressed with the use of a small P gain as the mBot will correct more slowly and have a high chance of being parallel to the wall at the setpoint. However, a large P gain might be more favourable as it helps the mBot to converge quickly to the setpoint. The unwanted effect of over correction can be addressed with the help of the D controller.

Integral control is used to eliminate any steady-state error by summing the errors over time. This is used when there is residual error in the system that cannot be eliminated with a PD controller alone. However, in the case of our mBot, there is no steady-state error and thus the I controller can be ignored.

Derivative control is used to predict future errors by taking the derivative of the error. The derivative term helps to dampen the system and prevent it from overshooting the setpoint. The goal is to find the rate of change of error which can be achieved by first taking the difference between the current error and the previous error. This gives delta error while delta time which is simply the time to execute one main loop iteration (around 40 ms) will simply be part of the D_gain. If the error is rapidly decreasing, it means that the mBot is reaching the 10 cm setpoint very quickly and thus the D controller will output a large opposite output to reduce the overall output since the mBot is about to reach the desired state. As such, the dampening effect can be achieved.

Code implementation of the PID algorithm:

```
float pid_error = filtered_dist - PID_SETPOINT; // Calculate PID error

pid_i_mem += PID_I_GAIN * pid_error; // Accumulate error for I controller
float P_controller = PID_P_GAIN * pid_error;
float D_controller = PID_D_GAIN * (pid_error - prev_pid_error);

float pid_output = P_controller + pid_i_mem + D_controller; // Put them together

prev_pid_error = pid_error; // Update previous error
```

The PID output is then applied to the base motor speed using the *move_forward_correction(int correction)* function. If the output is positive, it signifies that the error is positive and hence the mBot is too far away from the wall (i.e. distance > setpoint 10 cm). Thus, the mBot should correct to the right with greater speed on the right wheel and slower speed on the left. The reverse is true if the output is negative. This is implemented as follows:

```
void move_forward_correction(int correction) {
  left_motor.run(-FORWARD_SPEED - correction);
  right_motor.run(FORWARD_SPEED - correction);
}
```

When there is no wall detected on the right, the mBot will simply move forward without any PID correction. This is implemented with the *move_forward()* function where individual deviation constants are applied to the base speed for each motor. The deviation factor will be discussed in a later section.

```
void move_forward() {
  left_motor.run(-FORWARD_SPEED + LEFT_DEVIATION);
  right_motor.run(FORWARD_SPEED - RIGHT_DEVIATION);
}
```

Global constant variables related to PID that we used:

	Variable	**Description**
1	PID_SETPOINT	The desired state of the mBot while moving straight. We want it to maintain 10 cm away from the right wall
2	PID_MAX_OUTPUT	Limit the maximum output of PID such that it would not exceed the motor's speed range of 255. The max output of PID is chosen 60 so the max speed of a motor is base speed 180 + 60 = 240
3	PID_MAX_I	Prevent I controller wind up but no relevant because I controller is not used in this case

3.2.5. Calibrating PID

To tune the PID controller, we set both the I and D gains to 0 and increased the P gain until there is constant oscillations around the setpoint as seen in figure 3.2.5a.

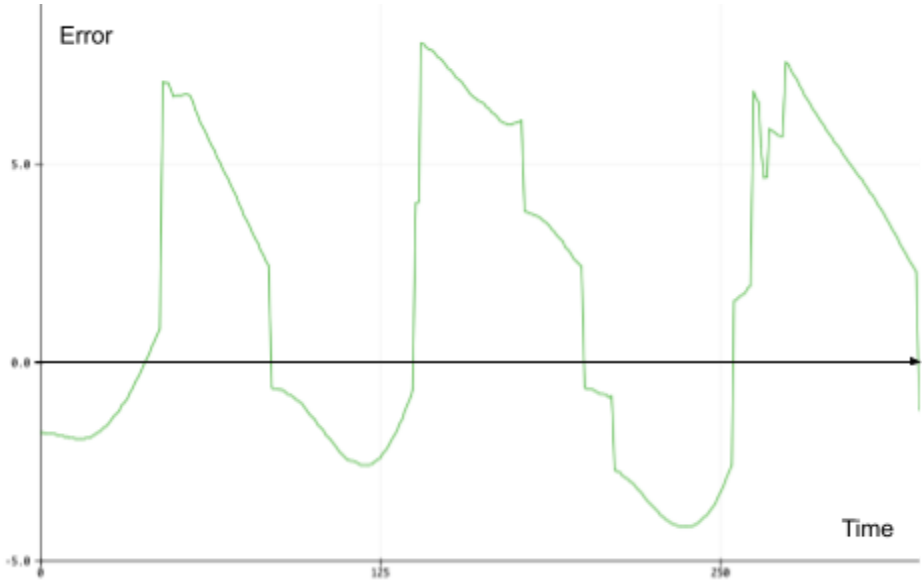


Figure 3.2.5a: P-controller only

Next, we leave the P-gain and start increasing the D gain. We can see that the introduction of the D controller will help the mBot reach the setpoint albeit with some oscillations around the setpoint.

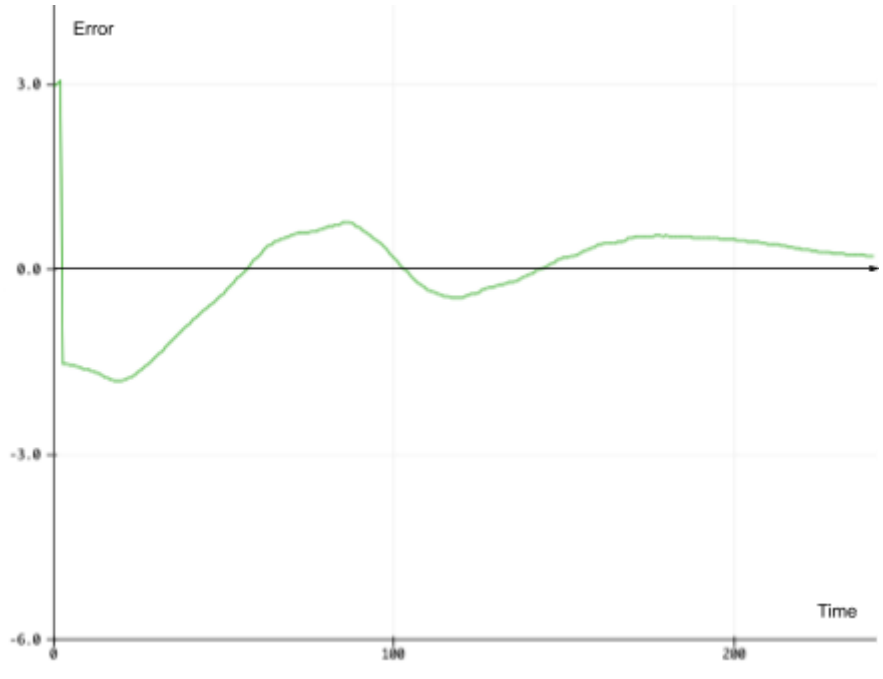


Figure 3.2.5b: PD controller with small D-gain

At the optimal D gain, we can see that there is no significant oscillations and overshooting of error. The mBot also converges to the setpoint relatively quickly.

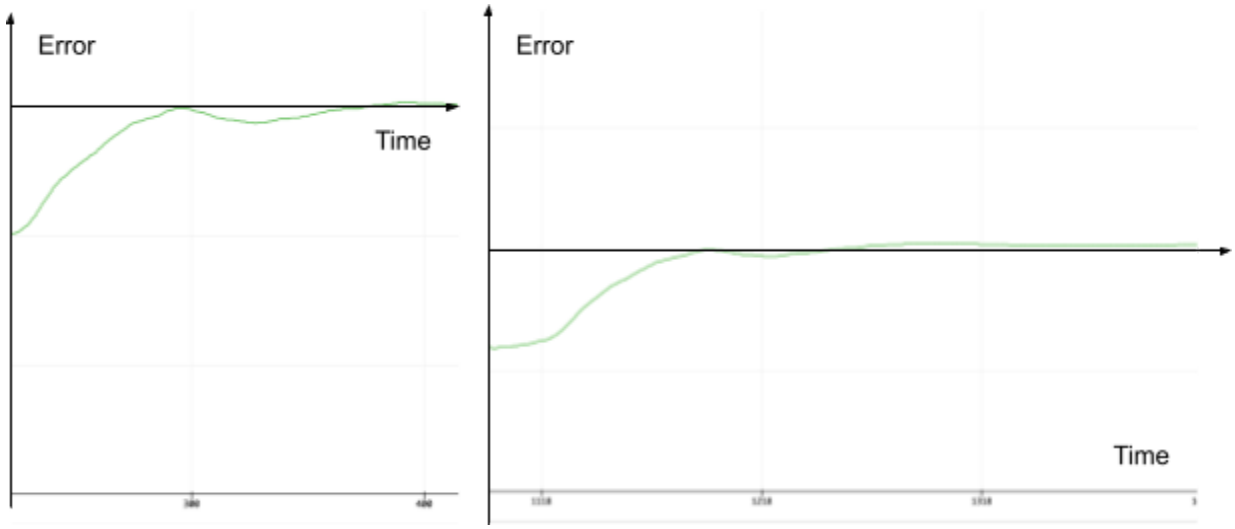


Figure 3.2.5c: PD controller with optimal P and D gains

Since there is no persistent error in our system, there is no need to accumulate the error over time and hence the I gain is left at zero.

The final P, I and D gains that we used for the graded runs are as follow:

PID_P_GAIN	PID_I_GAIN	PID_D_GAIN
30.0	0.0	800.0

3.2.6. Digital low-pass filter for PID to improve robustness

One issue with the PID controller is that noise from the input (ultrasonic) signal affects the output of the D (derivative) controller significantly enough to cause jitters in the mBot's movements.

The rate of change of a noisy/choppy signal is likely to fluctuate significantly as seen by how the gradient changes rapidly on a curve that increases and decreases relatively quickly over a small time step. Since the D controller computes the rate of change of error and applies it to the overall PID output, this would cause the output to be noisy/choppy as well. This effect is further amplified if a large D gain is used to dampen the change in error as this rapidly changing rate of change component would become a larger component of the overall output.

To address this issue, a digital low-pass filter is applied to the input distance signal to smooth the distance readings from the ultrasonic sensor. As seen in the figure 3.2.6a, the line in red shows the distance readings with the low pass filter applied which is much smoother than the original signal in green. There is a noticeable lag in the filtered red signal due to the phase shift of the low-pass filter but this lag is not significant enough to affect the PID algorithm.

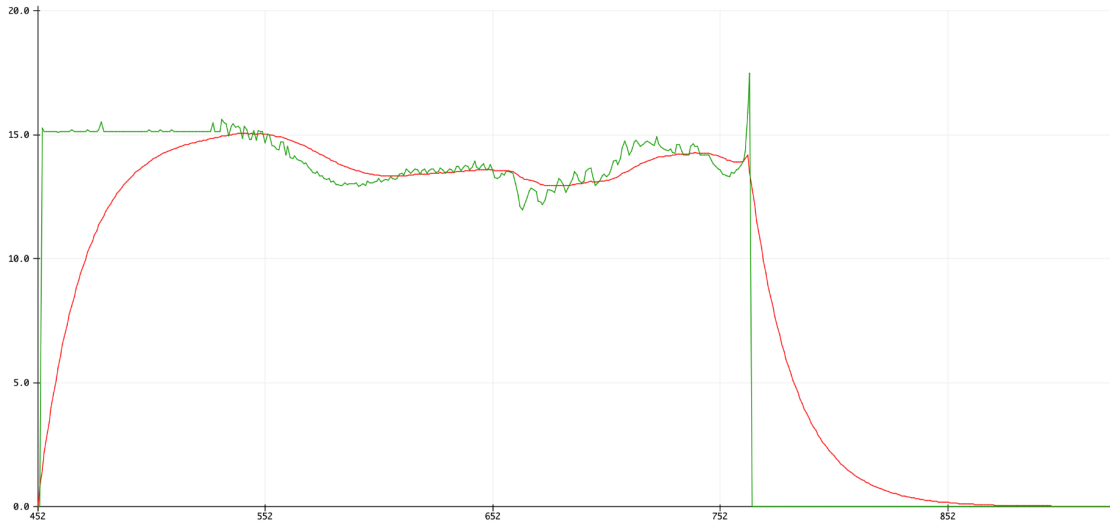


Figure 3.2.6a: Digital low pass filter (Red) applied to original distance reading from ultrasonic sensor (Green)

The implementation of the digital low pass filter is as follows:

```
1.  float low_pass_alpha = 0.05;    // ...
2.  float initial_threshold = 3.0;  // To account for extreme changes in value
3.  float prev_signal = 0.0;
4.
5.  float apply_low_pass_filter(float distance_cm) {
6.    float filtered_signal;
7.
8.    if (abs(distance_cm - prev_signal) > initial_threshold)
9.      filtered_signal = distance_cm;
10.
11.   filtered_signal = (1-low_pass_alpha) * prev_signal + (low_pass_alpha *
    distance_cm);
12.
13.   prev_signal = filtered_signal;
14.   return filtered_signal;
15. }
```

However, when there is a sudden change in the distance reading from the ultrasonic sensor, the low-pass filter will take a while to respond and reach the new distance reading. Sudden changes in distance readings occur when the mBot detects missing walls or detects a wall again. The delayed filtered signal will cause the mBot to think there is a wall slowly approaching or moving away from it, when in fact the wall is already present or not present. As a result, the PID algorithm will incorrectly adjust the mBot's position based on the delayed filtered signal.

As seen in figure 3.2.6a, when the raw distance reading (green) from the ultrasonic sensor decreases rapidly, for example from 10 cm to 0 in a short period of time, the filtered signal (red) will take some time to reach 0. This is because based on the code above (line 11), a large part of the filtered signal is based on its previous reading. Only a small portion of the new distance reading is taken into account, thus large changes would not affect the filtered significantly.

This is addressed by bringing the value of the filtered signal to the current distance reading immediately if the previous signal is very different from the actual distance reading. (line 8-9)

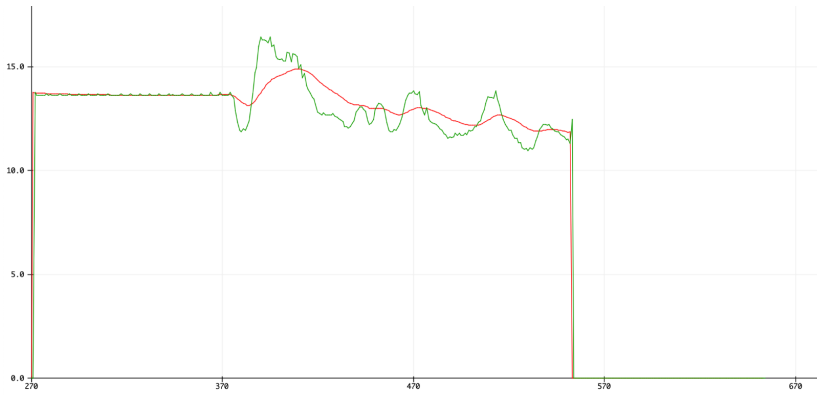


Figure 3.2.6b: Digital low-pass filter that brings filtered signal to raw reading if there is sudden change

3.2.7. Implementing navigation logic

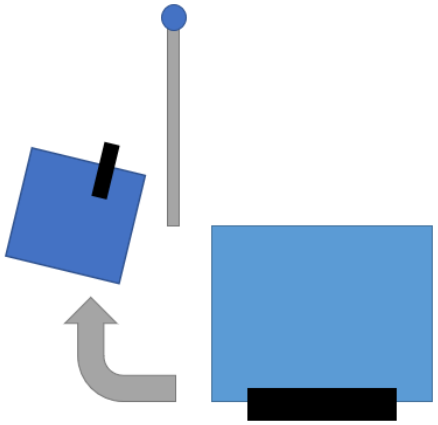
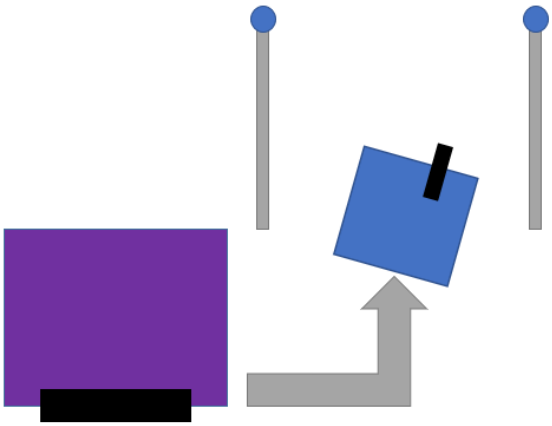
We defined some constants initially related to the motor:

	Constant	Description	
1	FORWARD_SPEED	● The speed when the mBot moves in a straight line ● Set to 180. While this could have been bumped up, we've calibrated our PID and subsequent timings to this and for the sake of time and precision, we've decided to forgo improved speed of the mBot.	
2	LEFT_DEVIATION	● Used for slowing down the **left** wheel. ● Set to 15 to deviate to left	Tendency to deviate left is further enhanced by intentionally slowing down the left wheel as the mBot turns right slightly when no wall is detected. Left deviation to correct this slight right deviation while the mBot is moving straight without PID correction.
3	RIGHT_DEVIATION	● Used for slowing down the **right** wheel. ● Set to 0.	
4	TURN_CORRECTION_TIME_MS	● The time the mBot will move forward for after each red or green 90° turn ● This is to ensure that after the 90° turn, the ultrasonic sensor will be approximately 10 cm away from the wall and no further adjustment is needed from the PID algorithm. This can reduce sideways correction to the mBot's orientation after each turn while it is moving forward with PID correction. ● Set to 100	
5	TURN_SPEED	● The speed when the mBot makes a turn. ● Set to 125.	
6	TURN_RIGHT_TIME_MS	● The time limit in milliseconds on how long the mBot makes a **right** turn. (Green tile) ● Set to 625.	
7	TURN_LEFT_TIME_MS	● The time limit in milliseconds on how long the mBot makes a **left** turn. (Red tile) ● Set to 625.	

8	UTURN_TIME_MS	● The time limit in milliseconds on how long the mBot makes a u-turn. (Orange tile) ● Set to 1125.
9	STRAIGHT_RIGHT_TIME_MS	● The time limit in milliseconds on how long the mBot goes straight during a **double right** compound turn. (Blue tile) ● Set it to 975.
10	STRAIGHT_LEFT_TIME_MS	● The time limit in milliseconds on how long the mBot goes straight during a **double left** compound turn. (Purple tile) ● Set it to 1050.
11	COMPOUND_TURN_LEFT	a. Used for the purple tile challenge b. compound_turn_left → straight → compound_turn_left c. We've made the mBot underturn such that the right_mounted ultrasonic sensor will be nearer to the wall after the compound turn. This is to allow the ultrasonic sensor to still be within range of the wall to allow the PID to function correctly. In fringe cases when there's no walls, we've noticed that this implementation works better compared to using the default left turn → go straight → default left turn method.
12	COMPOUND_TURN_RIGHT	a. Used for blue tile challenge b. compound_turn_right → straight → compound_turn_right c. Results in an overturn at the end to keep the right-mounted ultrasonic sensor nearer to the right wall to facilitate PID movement after turn is completed.

The motor default state always goes on a straight line. While the mBot is in this state, it always applies PID correction to ensure it does not deviate from the straight line. It works by having the right motor slow down for the ultrasonic to detect the wall for the PID. Regarding the turn speed, the motor will have a positive turn speed when turning left. Otherwise, the motor will have a negative turn speed when turning right.

We've implemented different turning timings for the compound turns (purple and blue tile challenges) to improve the functionality of PID after the 2nd turn is completed. The new compound turn sequence places the ultrasonic sensor nearer to the wall by underturning / overturning to ensure that the ultrasonic is within range such that the PID algorithm will kick in immediately. It is partly due to this implementation that our mBot is able to traverse through the maze even without any wall placements. This is illustrated in the diagrams below. Take the protruding black rectangle as the right-mounted ultrasonic sensor and the grey rectangle as walls.

Blue tile overturn	Purple tile underturn
	

For the red & green turns, we've also noticed that in fringe cases when there's no 3 walls consecutively, our mBot will hit the metallic pillar as the PID algorithm does not have sufficient time to correct the distance using the metallic pillar. The root cause is that the mBot is too far away from the wall after the turn. To compensate for this, we've added an additional forward motion before the turn to place the mBot near the 10 cm mark needed for the forward motion.

We've also added left deviation to further enhance the left-leaning bias of our mBot (mBot tends to move leftwards when programmed to go straight). This is to compensate for the slight right movements that the mBot will make just as it detects there is no wall on the right. This deviation will be applied when the mBot is programmed to move forward without PID during compound turns and when there is no wall on the right. The effect of left deviation will only become significant over a long distance. When traversing the maze, moving straight without PID only occurs over at most one tile where the effect won't be significant enough to cause major unwanted deviations.

4. Colour Sensor

4.1. Colour Sensor Implementation

All colours are made of varying amounts of the three primary colours - red, green and blue. A colour of an object can be determined by the amount of red, green and blue light reflected from a particular coloured surface. For example, a red surface will absorb green and blue light and most red light will be reflected for our eyes to detect the surface is red. Similarly, our colour sensor will measure the intensity of red, green and blue light reflected from the object and determine its colour based on the known characteristic RGB components of light reflected from each of the coloured tiles. (Red, Orange, Purple, Green, Blue and White)

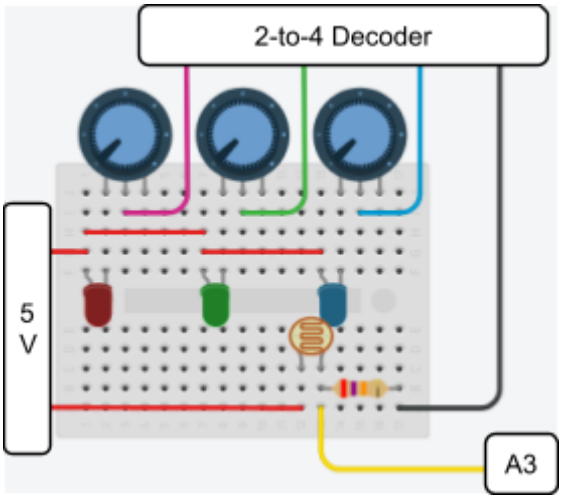


Figure 4.1: Colour Sensing Circuit

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

- As we have a limited number of pins to work with, we used the 2-to-4 decoder such that only 2 pins (A0 & A1) from the mBot are needed to control the 3 LEDs.
- At fixed intervals, we changed the state (HIGH or LOW) of pins A0 and A1 to switch on each LED and take the voltage divider reading of the LDR from pin A2.
- The program then calculates these values to match the RGB colour code, where each value is between 0 - 255

To calibrate the colour sensor, we measured the field's white and black colour. The white determines the readings when all light is reflected and black determines the readings when most light is absorbed. From the white and black readings, the greyscale can be determined by subtracting the black values from the white values.

In code, it look like this:

```
balanceArray[GREY][j] = balanceArray[WHITE][j] - balanceArray[BLACK][j];
```

As the values of the analog pin of the arduino ranges from 0 - 1023 (0V - 5V), we have to convert the values to resemble the RGB colour code (eg 255, 0, 0 for red). We did this by using the ratio of the difference between the measured value and the black value, and the greyscale, which can be represented with:

$$\frac{Avg\ LDR\ Readings - Black\ Values}{Grey\ Values} * 255$$

To put it in code:

```
finalColors[c][i] = ((float) (finalColors[c][i] - balanceArray[BLACK][i])) /
((float) balanceArray[GREY][i]) * 255.0;
```

At each waypoint challenge of the maze, the colour of the paper tells the mBot what action to perform next. These colours include: Red, Orange, Green, Blue, Purple and White. In order to differentiate the different colours, we used hard-coded threshold values. These values were obtained by observing the RGB values for each colour and determining the range of values each colour falls in.

At the start of the a_utiliy.ino code, we defined the following macros:

- WHITE_THRESHOLD
    - The threshold used to differentiate white from the other colours.
    - We've set this value to 200.
- RED_THRESHOLD
    - Raw R value used to differentiate red and orange from green, blue and purple.
    - We've set this value to 200.
- RED_ORANGE_THRESHOLD
    - Raw G value used to differentiate red from orange.
    - We've set this value to 120.
- PURPLE_GREENBLUE_THRESHOLD
    - Ratio of R and G, used to differentiate purple from green and blue.
    - We've set this value to 1.2.

The colour sensing algorithm consists of two functions: read_color_sensor() and match_color().

Within the read_color_sensor() function (found in d_sensor.ino):

1. The pins on the 2-to-4 decoder is set to switch on the LEDs in the following order
    a. HIGH, HIGH for Red LED
    b. LOW, HIGH for Green LED
    c. HIGH, LOW for Blue LED
2. When each LED switches on, the LDR value is read 5 times and the average is taken and stored in their respective positions (red, green & blue) in the int array currentColor[3].

Within the match_color() function (found in a_utility.ino):

1. The colours read from read_color_sensor() are stored in their respective positions (red, green & blue) in the int array currentColor[3].
2. First, we check if the coloured tile is white because white tends to reflect all colours of light and thus it is expected that all RGB values should be fairly high. Hence, we check if all RGB values are greater than WHITE_THRESHOLD, white is detected and the colour "white" is returned.
3. Next, we observed that the reflected red component is always above 200 for red and orange colored tiles. Hence, we check if red > RED_THRESHOLD (200), we can deduce that the colour of the paper is either red or orange.
    a. Between red and orange, we noticed that the orange tile reflected much more green light as compared to red tile. Thus, we attempt to differentiate red and orange by comparing the G value with the RED_ORANGE_THRESHOLD which through trial and error and observation, we've come to a value of 120.
    b. If green > RED_ORANGE_THRESHOLD, the colour "red' > is returned, else "orange" is returned.
4. Otherwise, the else statement will lead us to differentiate between purple, blue and green
    a. Two checks will be done to differentiate purple from blue and green.
        i. Since purple has more reflected red and blue light, we firstly check if red >= green. Hence, if the red value is greater or equals to green, the colour "purple" is returned.
        ii. In some cases, green is higher than red for when sensing purple (due to differences in ambient light). Another distinguishing factor is that the values for reflected red and green are close to each other for the purple tile. Hence we check if the percentage difference between green and red is smaller than PURPLE_BLUEGREEN_THRESHOLD. (20%) If the statement is true, the colour "purple" is returned.
    b. If both of the checks are false, the colour will either be green or blue. A blue tile will have more blue light reflected while a green tile will have more green light reflected. Thus, to differentiate between green from blue, we simply check if green > blue. If it is true, the colour "green" is returned, else "blue"is returned.

4.2. Flowchart of Colour Sensor

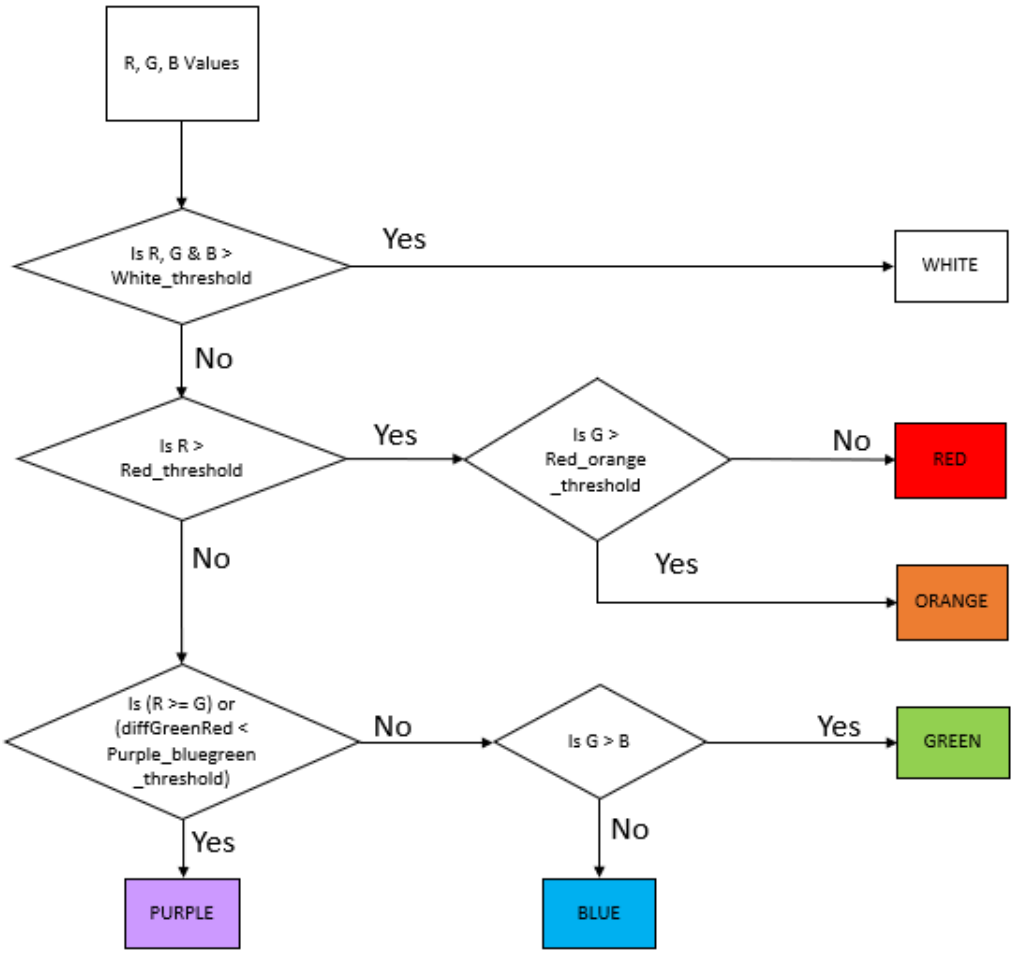


Figure 3: Flowchart of match_color()

4.3. Reading calibrated values into EEPROM

The calibrated balanceArray is stored on the EEPROM, which can be likened to the mBot's onboard memory. This allows us to store the black RGB values and the (white - black) range RGB values from colour calibration to the mBot. This eliminates the need to manually insert those calibrated values into the main project project code which is inefficient and prone to error. At the start of the main program, the values of balanceArray will be retrieved programmatically from the EEPROM.

4.4. Colour sensing improving robustness (hardware)

Version Control	Description
Version 1: Black box around breadboard	Initially we created a box directly around the breadboard to block out the ambient lighting. The box touches the floor directly and does it's job in ensuring that ambient light does not reach the LDR, We've also secured variable resistors to facilitate testing and calibration and the open black box concept allows for quick changes to the variable resistors. We've also tidied up the wires by using single core wires and ensuring that they're not in the pathway of the LEDs and LDR to prevent unwanted colours (from the wires) being bounced back to the LDR. We opted to use a 10k resistor for our LDR during this phase as we realised that anything above 50k causes our grey-scale to be single-digit (insensitive to the input of light absorbed) However, there are some drawbacks with this design 1. The drag induced by the box caused significant deviation for the forward movement of the mBot, especially when reaching the coloured tiles as it clips the edge. 2. The LED and LDR are not in place and can be shaky during the mBot run, affecting the colour sensing as the LED can potentially be blocked by the LDR.
Version 2: Side Skirts of mbot's underbelly	In this phase, we improved it with side skirts around the under-belly of the mBot (refer to chapter 1 for picture). This version manages to provide a stable reading for the LDR with minimal ambient lighting. However, there are drawbacks with this design: 1. The colour sensing circuit at the underbelly of the mBot can only be accessed if the black strip is pulled open. With each iteration, the adhesiveness of the double-sided tape is reduced, making it more likely for the structure to fail and allow ambient light in 2. There are certain gaps above the breadboard such as the holes which allow wires through to connect the breadboard to the arduino board. This allows unwanted ambient light in and reduces the reliability of this version 3. The LED and LDR continues to be shaky as nothing is holding them in place other than the breadboard

Version 3: Black cylinder around the LED & LDR	This is the final phase of our colour sensing circuit. We've reverted back to a smaller-scale curtain. This time, it's placed directly around the LEDs, holding them in place by double-side tape between the LED and curtain, and with an additional curtain separating the LED from the LDR to ensure that the input taken by the LDR is only restricted to light that bounced off the surface of the coloured tile. Furthermore, the colour sensor breadboard is easily accessible for changes to be made. The curtain is raised slightly above the floor to avoid grazing and repeating the same mistake as version 1. An observation made during this phase shows that the Red LED's placement is affecting the Green and Blue values as the Red LED is a diffused variation. To counter this, we mounted the Red LED slightly further away from the Green and Blue LED. We tinkered with the resistance values of variable resistors attached to each LED until we got the right resistance values combination for each of them to ensure that the colour sensor does not detect the wrong colour even with different ambient light settings. We've also changed the resistor of the LDR from 10k to 30k to increase the sensitivity and response time to a respectable level to reduce the amount of time spent on delay to take in the colour readings.

4.5. Colour sensing improving robustness (software)

1. On the software end, as mentioned above, we made use of 1 hard coded value initially and thereafter percentages to detect the colours. These values are obtained through trial and error.

2. Furthermore, during the runs, we made sure to take the average of the readings as we understand that the colour sensor may not have stabilised yet.

3. For troubleshooting, we've used the on-board arduino LED to display the colour matched by the mBot. This helps us to separate navigation and colour detection errors and remedy accordingly.

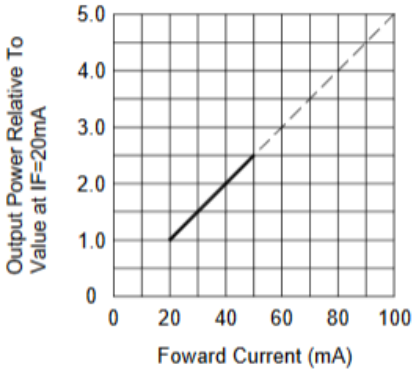
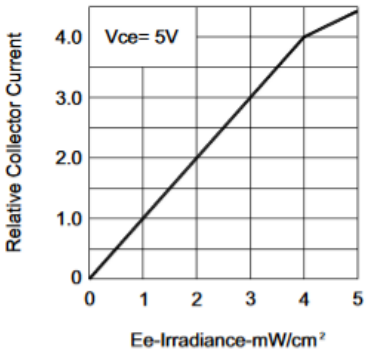
5. IR proximity sensing

5.1 Implementation details

Similar to the studio handout, we used the same circuit, but adapted it to include the motor driver chip for us to be able to switch the IR circuit on and off based on the voltage input into the motor driver, which is enabled by the 2-to-4 decoder when both pins A0 & A1 were set to LOW. We connected 5 V (Vcc) into pins 1, 8 and 16 into the L293D chip. Then, we connected Y0 (pin 4) of the 2-to-4 decoder IC chip into pin 7 of the L293D chip. Finally, we connect the IR Emitter to pin 6. We also ensure that the IR's emitter's current does not exceed 50 mA. To receive the IR detector readings to the mBot, we used one of our analog signal pins via RJ25 adapter to measure the voltage in the IR detector.

However, the sensitivity of the IR sensor using the studio handout resistance values was only 6 cm which we found to be too close to the wall. Therefore, we have decided to change the resistance values for both the IR emitter and the IR detector to make the IR sensor more sensitive and be able to detect longer distances than the IR sensor in the previous studio. With the modifications, our new IR sensor sensitivity increased to 8 cm.

To do this, we referred to the datasheet of the IR emitter and the hint given in the project brief provided to us:

IR Emitter	IR Detector
 FIG.5 RELATIVE RADIANT INTENSITY VS. FORWARD CURRENT According to this graph, if we increased the amount of current flowing through the IR emitter, theoretically the intensity of the IR being emitted should increase. Hence, instead of using 150 Ω as the handout suggested, we reduced it to 100 Ω.	From the datasheet:  FIG.4 RELATIVE COLLECTOR CURRENT VS IRRADIANCE From the project brief: "The 8.2 kΩ used in the studio may not be the best value for your setting. A larger receiver (IR detector) resistance makes the receiver voltage more responsive. However, it shouldn't be too large or it may result in saturation easily in the presence of even a little IR."

We were able to do this without exceeding the 50 mA limit because the IR emitter itself produces a forward voltage, which results in the actual current being lower than the calculated one.	Following this advice, we experimented with increasing values of resistances and found that 10 kΩ worked the best for our circuit.

Finally we built our circuit with the following configuration:

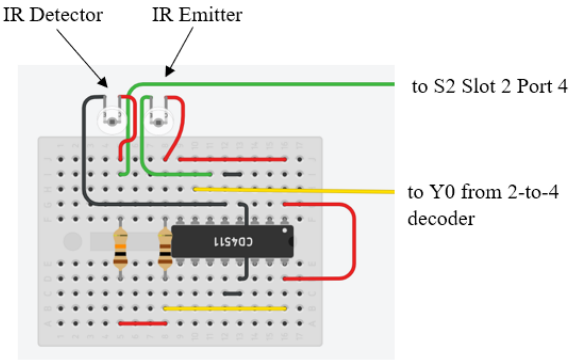


Figure 4: IR Sensor Circuit Breadboard

But before we mounted the IR sensor onto the mBot, we ensured it was working by testing it. The table below shows our tests for our IR sensor. Despite having different base voltages from each of these tests, it is noticeable that voltage increase slows down when the distance between the sensor and the wall is around 8 cm which applies for all three of these tests. After that, the voltage has a very small increase with each centimetre after the 8 cm point. This means that our IR sensor's optimal working range has indeed increased from 6 cm from the studio to 8 cm! :D

Emitter Resistance	Reciever Resistance	Distance	Vout			
			1	2	3	AVG
100	10k	1	0.2	0.18	0.16	0.18
		2	0.48	0.21	0.21	0.3
		3	1.35	0.65	0.71	0.903333
		4	1.88	1.13	1.21	1.406667
		5	2.12	1.44	1.44	1.666667
		6	2.3	1.63	1.65	1.86
		7	2.42	1.72	1.77	1.97
		8	2.47	1.82	1.82	2.036667
		9	2.55	1.86	1.86	2.09
		10	2.59	1.9	1.88	2.123333
		11	2.62	1.93	1.92	2.156667
		12	2.63	1.95	1.95	2.176667
		13	2.71	2	1.97	2.226667
		14	2.75	2	1.98	2.243333
		15	2.75	2	2	2.25

Table 1: The table readings about how much voltage the IR sensor receives based on the distance of the wall in centimetres

Using the results obtained, we plotted the characteristic graph for each test:

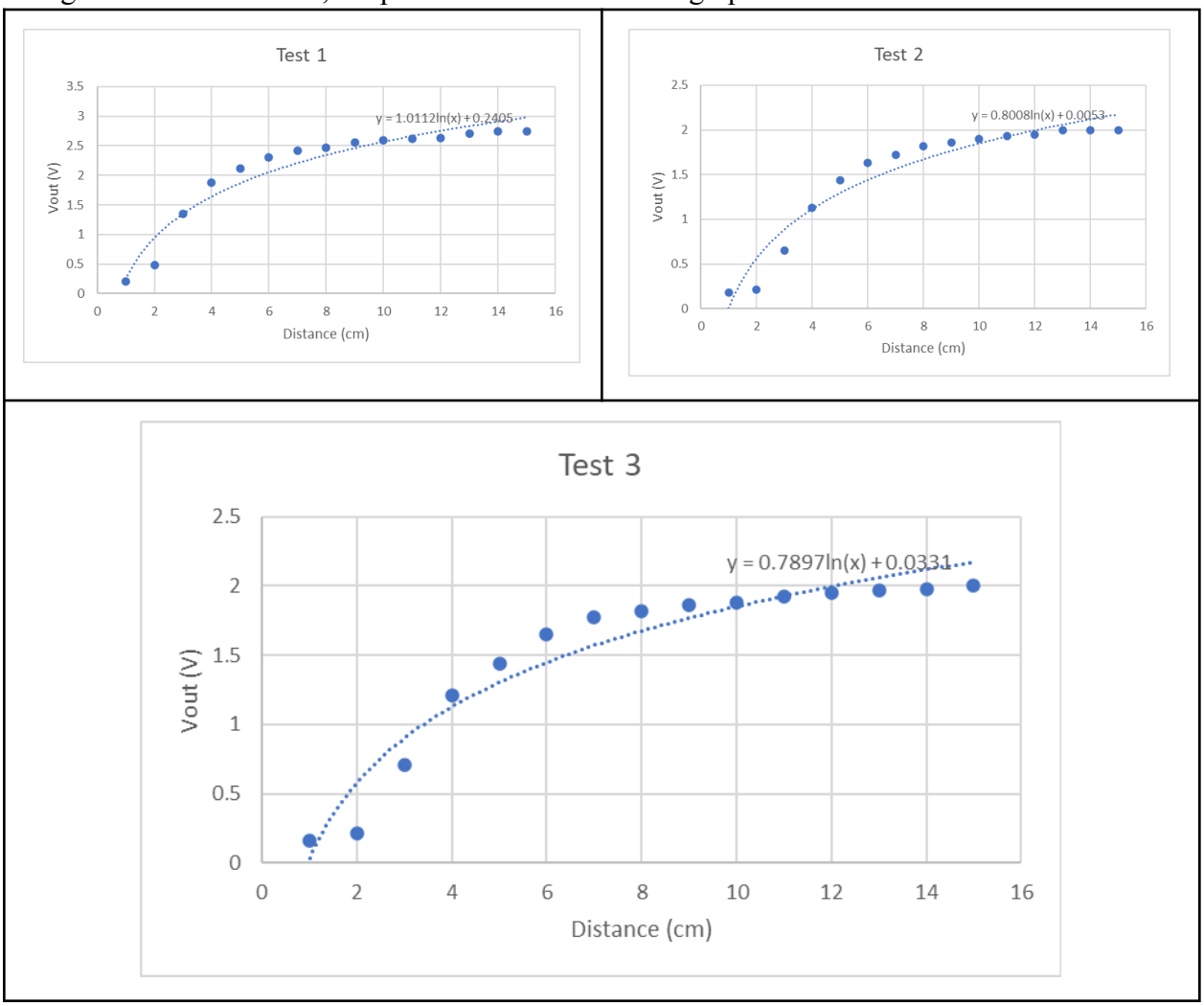


Figure 5: IR Sensor Output voltage vs Distance Graph

After constructing the IR sensor circuit, we mounted it on the mBot facing to the left to compensate for the lack of proximity sensing on the left side of the mBot because our ultrasonic sensor is facing to the right side of our mBot.

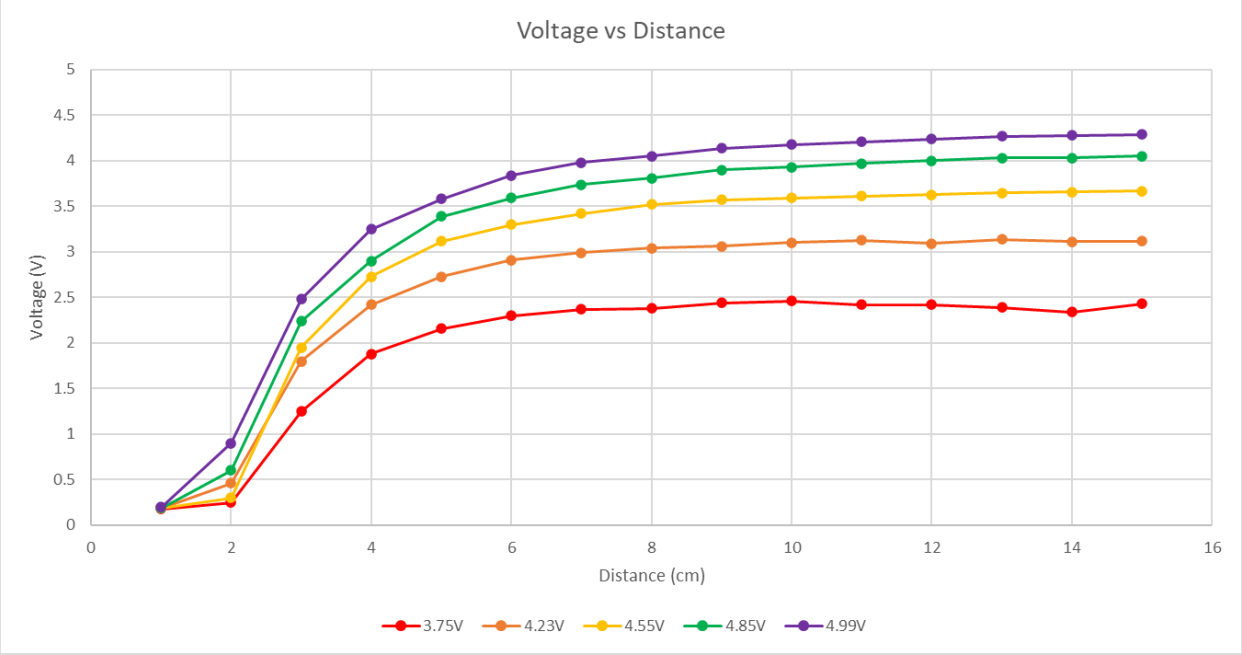
5.2 IR sensor - Improving Robustness

As we can see from the three tests that we have done, it is noticeable that the base and the maximum voltage is always different. This is due to subtle changes in the ambient light conditions when we do our tests, which causes the output voltage from our IR sensor to vary. To improve the robustness of our mBot, it is crucial to consider these environmental factors as the ambient light conditions in the maze are always changing.

Hence, to understand the relationship between the ambient light conditions and the output voltages, we tested the IR sensor in various different settings.

lighting condition	Baseline	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
desk lamp on (brightest)	3.75	0.18	0.25	1.25	1.88	2.16	2.3	2.37	2.38	2.44	2.46	2.42	2.42	2.39	2.34	2.43
desk lamp on (brighter)	4.23	0.19	0.46	1.8	2.42	2.73	2.91	2.99	3.04	3.06	3.1	3.13	3.09	3.14	3.11	3.12
desk lamp on (bright)	4.55	0.19	0.3	1.95	2.73	3.12	3.3	3.42	3.52	3.57	3.59	3.61	3.63	3.65	3.66	3.67
desk lamp off	4.85	0.19	0.6	2.24	2.9	3.39	3.59	3.74	3.81	3.9	3.93	3.97	4	4.03	4.03	4.05
darkness	4.99	0.2	0.9	2.48	3.25	3.58	3.84	3.98	4.05	4.14	4.18	4.21	4.24	4.27	4.28	4.29

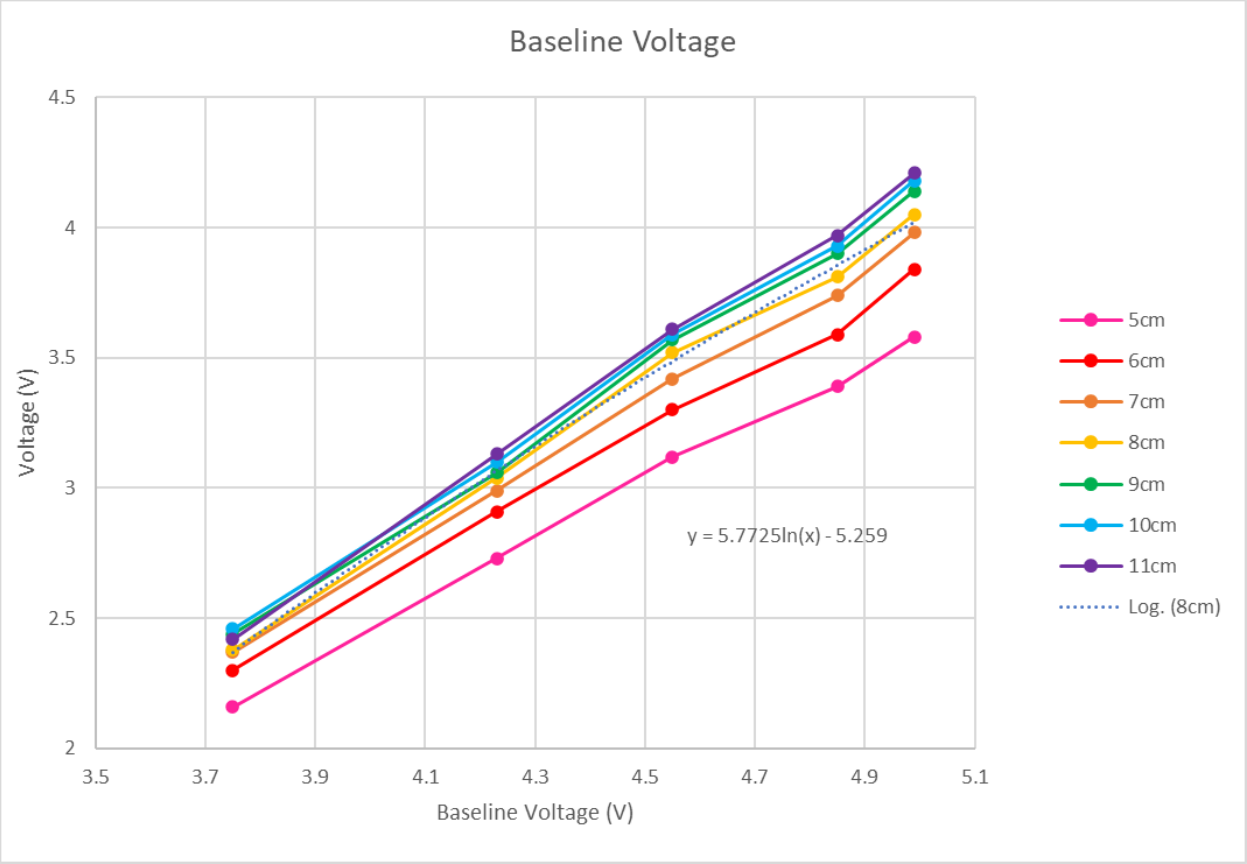
Table 2: This table shows how the ambient IR affects the base voltage and the output voltage



Graph 1: IR Sensor Output Voltage vs Distance for different lighting conditions

From the results above, we can see that the baseline voltage increases when there is less ambient IR in the surroundings. The greater the amount of IR exposed to the IR sensor, the lower the baseline voltage. However, this does not affect the optimal working range of the IR sensor as shown in the graph plotted.

Using the results obtained, we can observe how the output voltage is affected by the ambient light conditions for each distance. This gives us a graph resembling a linear relationship.



Graph 2: IR Sensor Baseline Voltage vs Output Voltage for each distance

Now that we have a general idea of how ambient light affects the IR sensor, we can improve the mBots's robustness by intermittently switching the IR emitter off to obtain the baseline voltage at that point in time to make our mBot adapt to the changing ambient IR. After reading the instantaneous baseline voltage, our IR sensor can roughly measure the right distances by adjusting accordingly to the baseline voltage.

When the baseline voltage is higher, the IR sensor will now expect a larger reading for the same distance. Otherwise, when the baseline voltage is lower, the IR sensor will now expect a smaller reading for the same distance too. So, we modify our code to adapt to these different ambient light settings with the help of the relationship shown in Graph 2, which is the gradient.

6. Work Division

Members	Roles
Zheng Hong	1. Coded the entire algorithm of the mBot. 2. Worked on the colour sensing circuit. 3. Contributed to the group report
Meng Han	1. Worked on the colour sensing circuit and ambient light blocker. 2. Mounted ultrasonic sensor stand & tidied wires 3. Contributed to the group report
Stanley Wijaya	1. Designed and connected the IR sensor circuit. 2. Contributed to the group report
Shanice	1. Designed and connected the IR sensor circuit. 2. Contributed to the group report

7. Challenges Faced

No.	Challenges	Solutions
1.	The RGB readings of the colour sensor are not differentiable between some colours.	We used variable resistors such that we are able to tune the resistances and the brightness of each LED accordingly. This ensures that the RGB values have a distinct between each colour such that the sensor is able to differentiate each colour on the waypoints challenge.
2.	The RGB readings of the colour sensor would change drastically when tested on different fields.	We suspected that this may be due to the fact that the colour sensing circuit was not properly secured and the LEDs and LDR would shift slightly when we brought the mBot around. To fix this, we made the LEDs shorter and built a cylinder around the LDR and then surrounded it with the 3 LEDs and another cylinder around everything (Version 3 of colour sensing hardware). The ambient light of each field was also slightly different, which affected the greyscale during calibration. So we calibrated the colour on every field and determined which one would cause the least offset to the RGB readings.
3.	The mBot would move slightly towards the right (ultrasonic sensor side) when some of the walls were missing as we were using PID to maintain a distance of 10cm from the wall. This is due to the sudden major increase in the ultrasonic sensor reading. The mBot will then apply the correction first before realising that the wall is missing and change to the move_forward() state.	We implemented a digital low-pass filter to filter out the noise so that the output will be smoother. Additionally, we changed the code such that the previous distance reading will be applied to the filtered signal. This allows the mBot to continue moving straight in the absence of a wall. However, we realised that the root cause was the mounting of the ultrasonic sensor. The ultrasonic sensor was detecting the side walls of the playfield when no wall was mounted. This distance from mBot to side wall is further away than that to the original wall, causing the mBot to compensate rightwards more than it needed to. To resolve this, we mounted the ultrasonic sensor above the height of the side walls to avoid false detection of a wall.

4.	The analog pin connected to the IR receiver circuit was unable to read the changes in voltage.	1. We suspected that our mBot board may have issues as the circuit was working fine with just an ordinary Arduino Uno board and changed to a different one. 2. We discovered that some wires connecting to the IR receiver circuit were faulty using the connectivity function in the DMM. So, we replaced those wires with new ones.
5.	Unpredictability of mBot runs, especially the turns.	From what we have noticed, there are some inconsistencies within our test runs. At first, we thought that our lack of power inside of the battery was the reason behind this. However, we noticed that our wheels are dirtier than the first time we do our test. So, we've decided to clean the wheels to ensure that the wheels do not deviate and slip from what the program intended. We also need to ensure metallic pillars are mounted properly before each run to ensure that our PID algorithm works properly since our algorithm also detects the pillars in the maze other than the walls. We also add some lines of code for the mBot to move slightly forward after detecting the line to the mBot and then turning either left or right. This is ensure that the mBot be more aligned when turning and not slightly facing left or right after finishing the turns.