International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Design and Development of a Line Following Robot Using the Proportional-Integral-Derivative (PID) Control System

Atharv Baluja

Student, Neerja Modi World School

Abstract
This paper explores the design and development of a line-following robot using the proportional-integral-derivative (PID) feedback control system. A line-following robot is a widely adopted autonomous system that can make a robot follow a predefined path based on sensor data and real-time adjustments. This paper details the components, construction, and programming needed to make the robot, with emphasis on the implementation and tuning of the PID system. The effect of tuning the PID parameters—proportional (Kp), integral (Ki), and derivative (Kd)—on the efficiency, stability, and path accuracy of the robot is thoroughly analyzed. This work contributes to understanding how PID systems can be optimized for applications in robotics, enabling precise and adaptive control.

Keywords: Control Systems, Line Follower, PID, PID Tuning, Proportional Integral Derivative, Robot Design, Robot Development, Robotics, Mechatronics

1. Introduction

The development of autonomous robotic systems has revolutionized modern technology, from industrial automation to consumer electronics. A line-following robot is an excellent example and learning project for understanding the principles of robotics and control systems. It also has real-world uses such as automating industrial tasks such as material handling, product assembly, and quality control. It uses sensors to detect a line and adjusts its movements in real-time, based on feedback mechanisms to stay on its trajectory. The Proportional, Integral, Derivative (PID) control system is a powerful approach for improving the accuracy and efficiency of such robots. By adjusting the PID parameters, engineers can optimize the robot's response to path deviations and enhance its stability. This paper discusses the design, development, and implementation of a line-following robot that employs a PID control system, including the tuning procedure and the challenges in achieving optimal performance.

2. Components Required

The components required to make the line-following robot are as follows:

A) Physical Components

i) Chassis - The chassis of the robot is 3D designed and shown in Figure (2.1) and Figure (2.2). It has been optimized for maximum performance. A longer chassis will be better than a shorter chassis since there will be more deviation of the IR sensor array resulting in a faster robot.

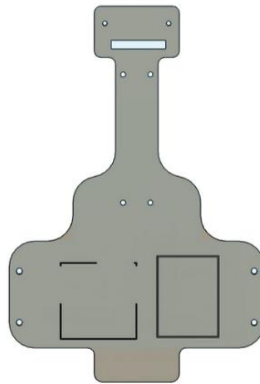

Figure (2.1): PID Line Following Robot Chassis Top View

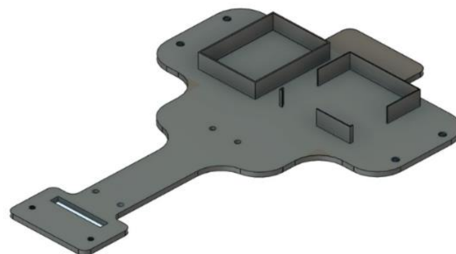

Figure (2.2): PID Line Following Robot Chassis Oblique View

ii) N20 Motor Mount - This will be used to attach the motor to the chassis. This will also be 3D printed and is shown in Figure (2.3).

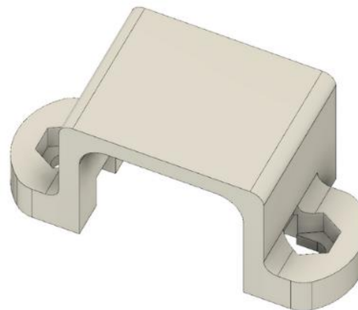

Figure (2.3): N20 Motor Mount

iii) 2x N20 Caster Wheel: This will be used to support the front part of the robot. Without it, the front part would scrape the ground as it moves. This part is shown in Figure (2.4)

Figure (2.4) - N20 Caster Wheel

B) Mechanical Components

i) N20 Metal Gear Motor: Motors are like the legs of the robot. Motors will make the robot be able to move around. N20 Metal Gear Motors, shown in Figure (2.5), are being used because of their small size and high speed. The motor operates at a speed of 20,000 RPM (unreduced) but with a gearbox reduction ratio of 100:1 it operates at a speed of 200 RPM. It generally runs on 6v but for this robot it will run on 7.4v

Figure (2.5) - N20 Metal Gear Motor

C) Electronic Components

i) 7.4V 1300mAh 25C 2S Lithium Polymer (LiPo) Battery Pack: A 2S battery, as shown in Figure (2.6) will be used to have an output voltage of 6.4V - 8.4V (Fully Discharged to Fully Charged). The motors are rated for 6V and the arduino nano runs on 7-12V so this battery is perfect.

Figure (2.6) - 7.4V 1300mAh LiPo Battery

ii) Arduino Nano: An Arduino Nano, as shown in Figure (2.7), will be used as the microcontroller for this robot because of its small size, lightweightness, clock speed (16MHz) and high number of GPIO pins. However, an ESP32 Development Board is also a viable option. It is not being used in this robot because it is mainly used in IoT applications since it has the ESP32 WiFi Module which is not needed for this robot and increases the power consumption.

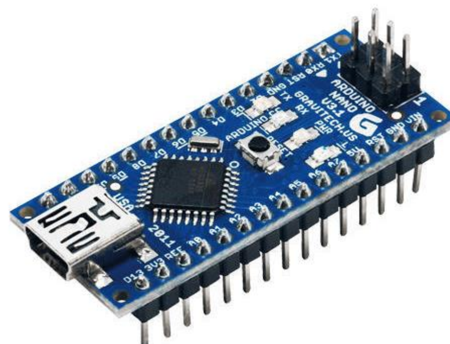

Figure (2.7) - Arduino Nano

iii) L298N Motor Driver: The L298N motor driver, as shown in Figure (2.8) is an H-Bridge motor driver with a supply range of 5V to 35V. However, to use the voltage regulator which converts it to 5v, the maximum to be supplied is 12V.

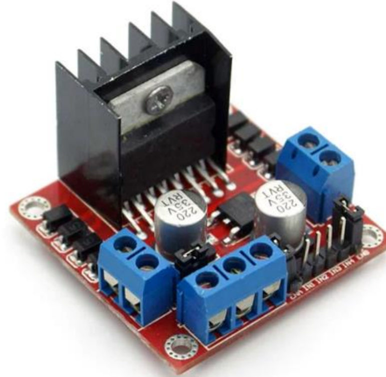

Figure (2.8) - L298N Motor Driver

iv) QTR-8RC 8 Reflectance Sensor Array: This is an array that consists of 8 reflectance sensors, as shown in Figure (2.9). Each one of the sensors measures the amount of light reflected from directly below them.

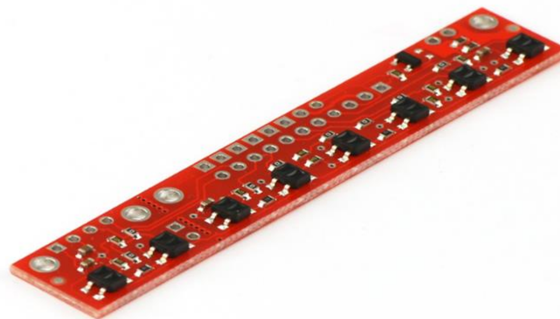

Figure (2.9) - QTR-8RC

v) Mini Breadboard: This will be used to supply power to multiple components from the same positive and negative terminal of the battery.

Figure (2.10) - Mini Breadboard

3. Robot Design and Development

3.1. Physical Construction

Firstly, all of these components need to be attached to the robot chassis. The QTR-8RC, N20 Motor Mounts and N20 Caster Wheels can be screwed into the holes on the chassis. Next, the L298N Motor Driver and breadboard will be attached to the chassis using double sided tape into their slots on the chassis. Finally, the battery and the arduino nano will be taped to the robot chassis since they will have to be removed from time to time to be charged and for new code to be uploaded.

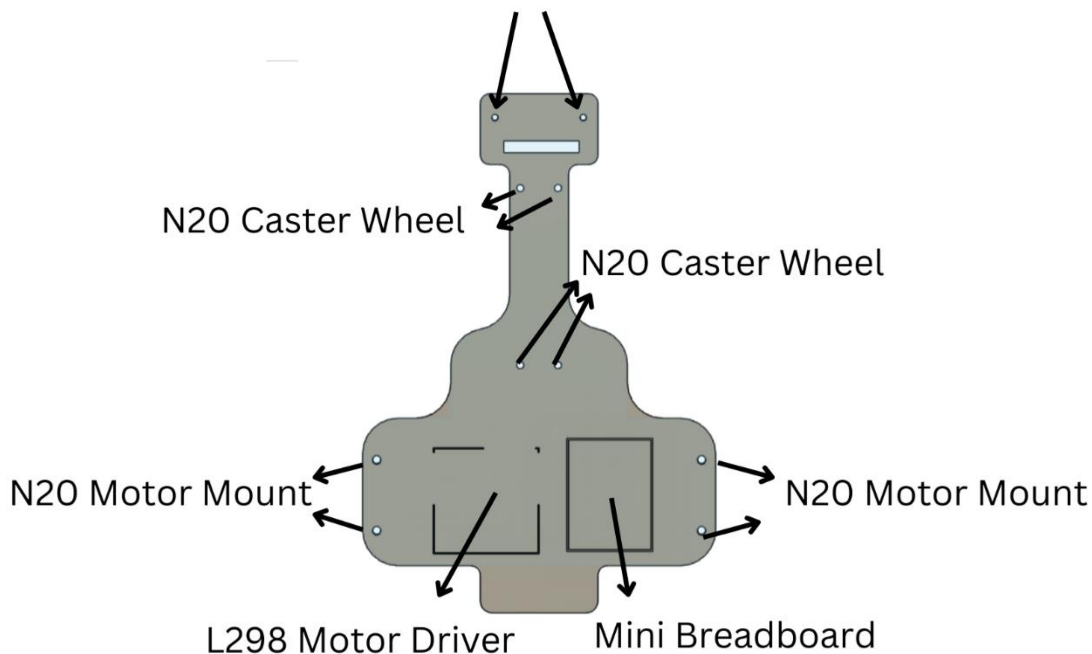

Figure (3.1)

3.2. Electronic Circuitry

Now it's time to make the electrical circuitry.

Firstly, the battery needs to be connected to give power to the L298N Motor Driver and the Arduino Nano. The **positive terminal** of the battery needs to be connected to both the **12V** terminal of the L298N Motor Driver and the **vin** pin of the Arduino Nano and the **negative** terminal of the battery needs to be connected to both the **GND** pin of the Arduino Nano and the **GND** terminal of the L298N. This can easily be done using a small breadboard, as shown in Figure (3.2)

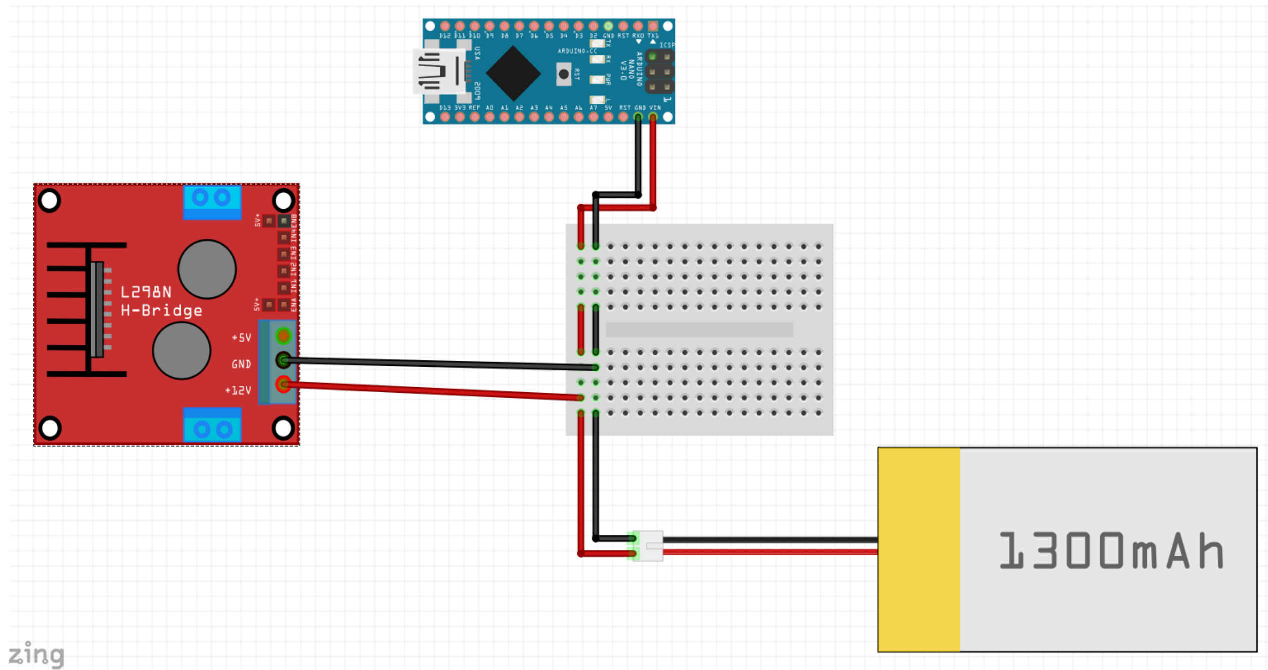

Figure (3.2) - Circuit Diagram Part 1

Next, the QTR-8RC Reflectance Sensor array needs to be powered. The 5V pin of the Arduino Nano needs to be connected to the VCC pin of the QTR-8RC and the GND pin of the Arduino Nano needs to be connected to the GND pin of the QTR-8RC. This is shown in Figure (3.3).

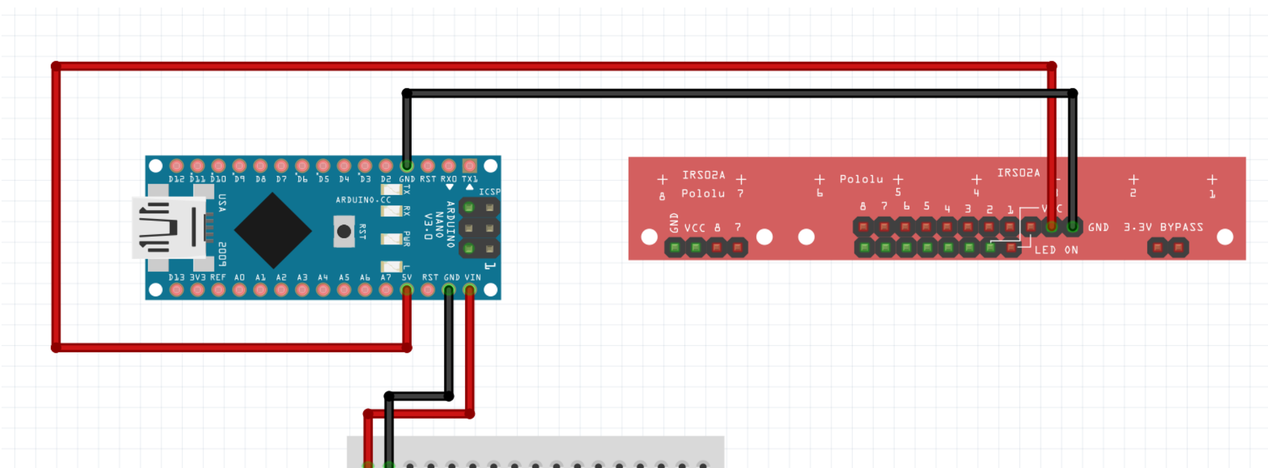

Figure (3.3) - Circuit Diagram Part 2

Now, the motors need to be connected to the L298N Motor Driver. The 2 terminals of the left motor will be connected to OUT1 and OUT2 of the L298N and the 2 terminals of the right motor will be connected to OUT3 and OUT4 of the L298N. This is shown in Figure (3.4).

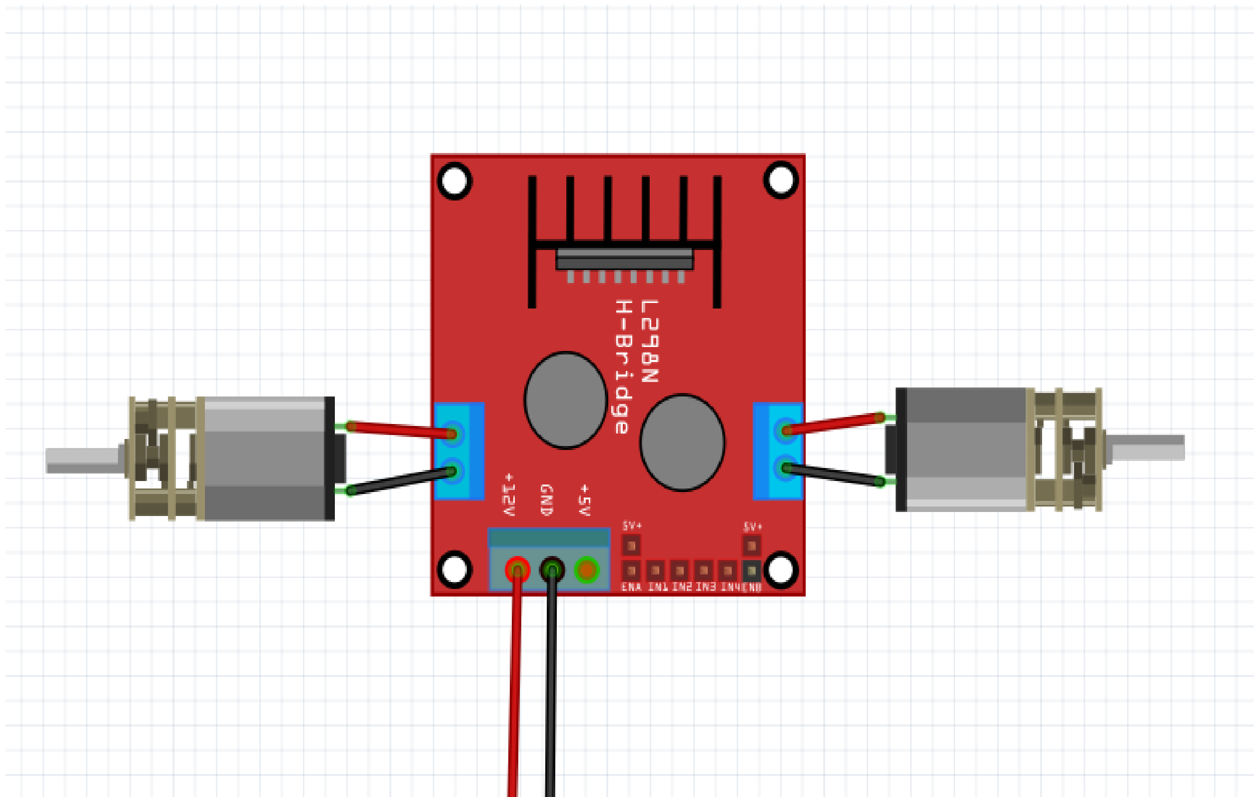

Figure (3.4) - Circuit Diagram Part 3

Now, the signal pins of the L298N Motor Driver need to be connected to the GPIO (General Purpose Input Output) pins of the Arduino Nano, The pins ENA, IN 1, IN2, IN3, IN4 and ENB will be connected to D5, D6, D7, D8, D9, and D10 respectively. This is shown in Figure (3.5).

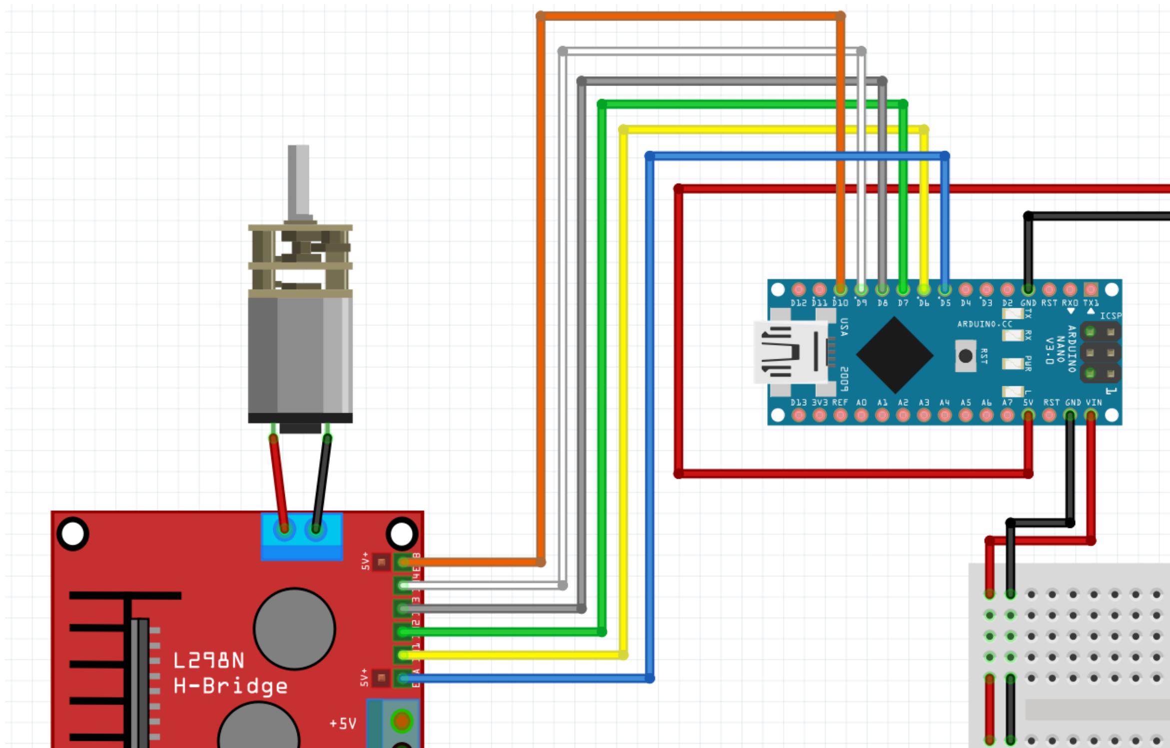

Figure (3.5) - Circuit Diagram Part 4

Next, the signal pins of the QTR-8RC Array need to be connected to the Analog pins of the Arduino Uno. Pin 1 - 8 of the QTR-8RC will be connected to A0 - A7 of the Arduino Nano. This is shown in Figure (3.6).

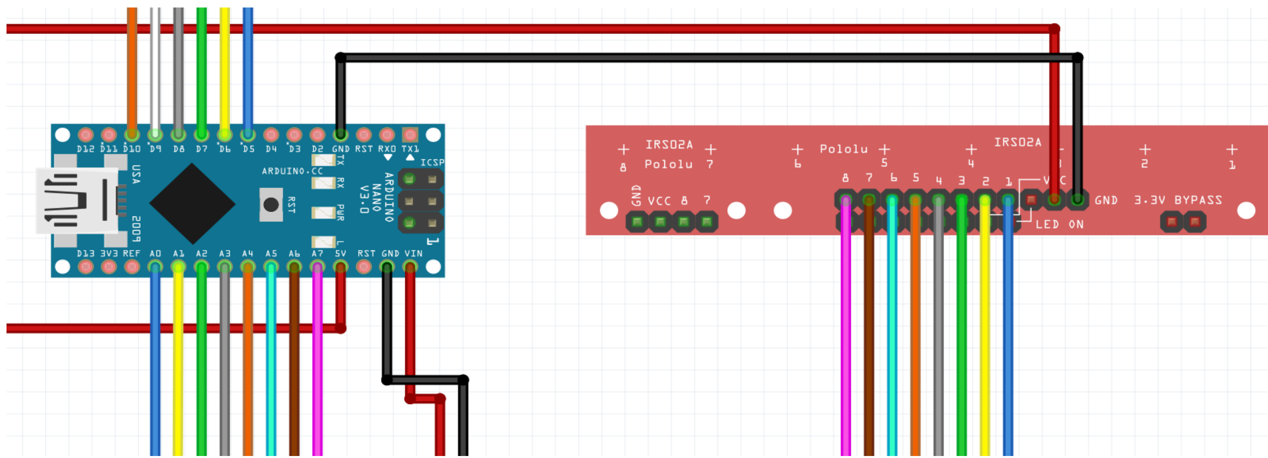

Figure (3.6) - Circuit Diagram Part 5

The finished robot should look like Figure (3.7) and Figure (3.8).

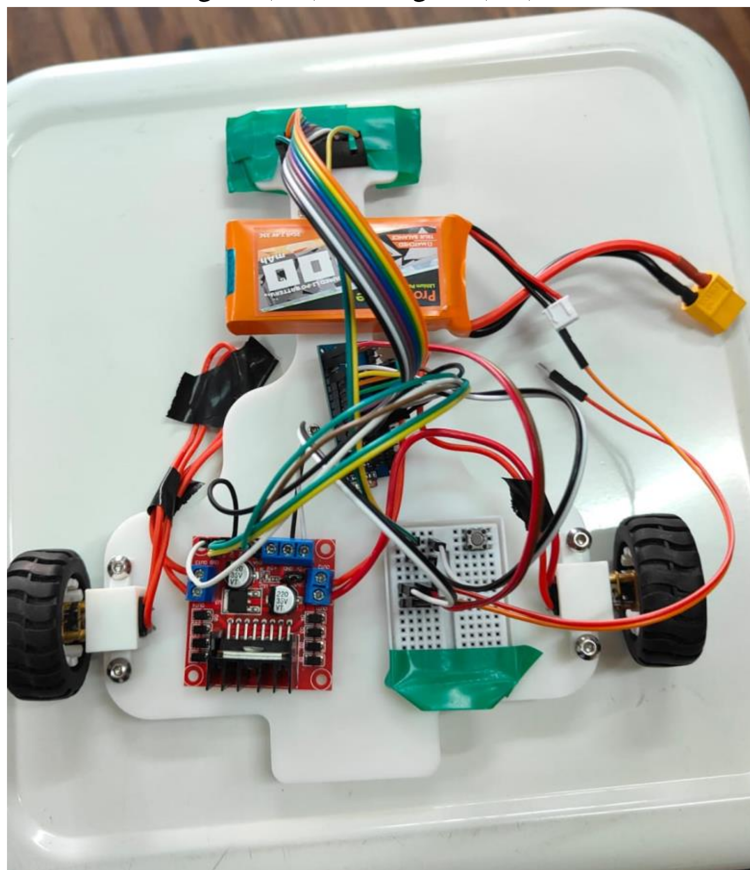

Figure (3.7) - Finished Robot Top View

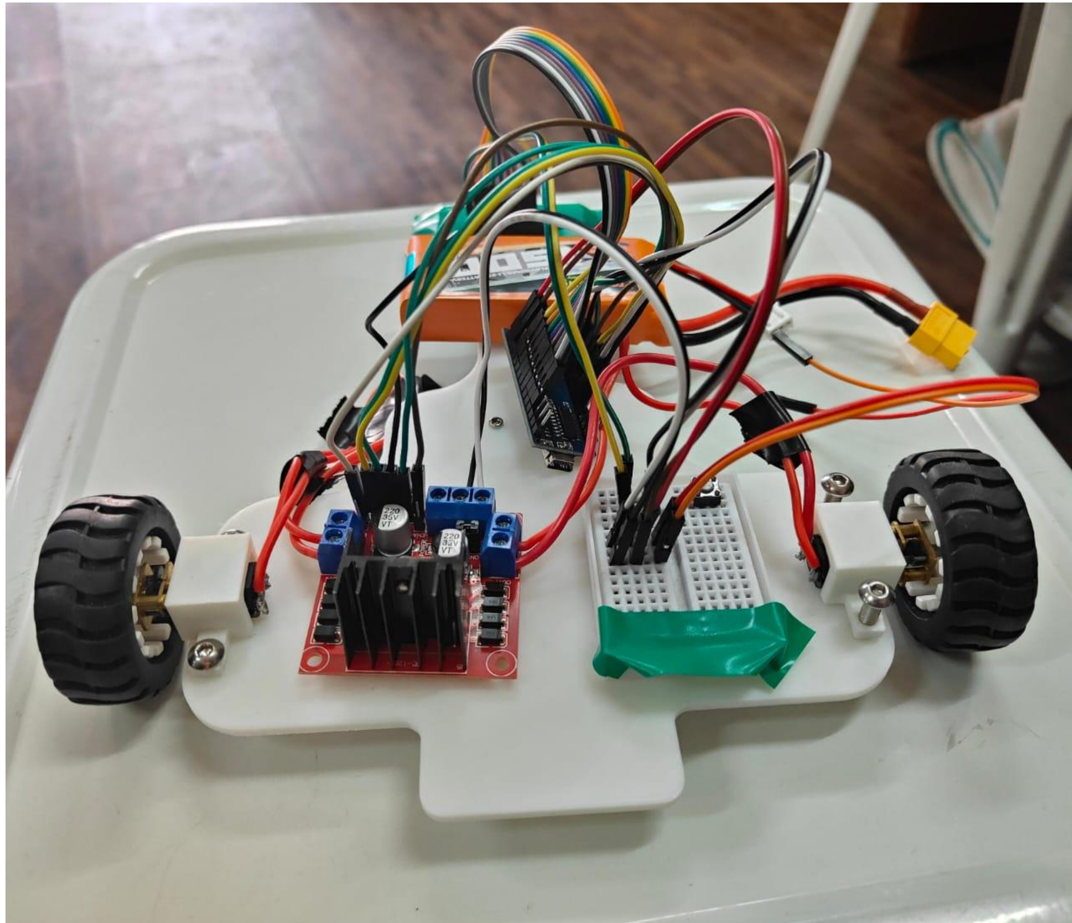

Figure (3.8) - Finished Robot

3.3. Basic Coding

First, a simple code needs to be made for getting the data from the QTR-8RC and sending signals to the L298N.

A function can be made to control a motor along with its speed.

```
void setMotorSpeed(int motorPin1, int motorPin2, int enablePin, int speed) {
  if (speed > 0) {
    // Forward direction
    digitalWrite(motorPin1, HIGH);
    digitalWrite(motorPin2, LOW);
  } else {
    // Reverse direction
    digitalWrite(motorPin1, HIGH);
    digitalWrite(motorPin2, LOW);
    speed = -speed;
  }
  analogWrite(enablePin, speed);
}
```

This code snippet contains a function that can be called to set the speed of the motors. It takes the two IN pins (IN1, IN2 for left motor and IN3, IN4 for right motor) as parameters along with the EN pin (ENA for left motor and ENB for right motor) and speed (int value between -255 and 255 (inclusive)). Since analogWrite can't be used with a negative value, if the speed value is negative it changes it into positive. Now the values from the QTR-8RC array need to be read. Since the output pins are connected to analog pins of the Arduino Nano, a threshold will need to be defined. If the output value is over 950, that means there is a line, and if it's below 950, it means there isn't a line. The threshold is 950.

```
const int sensorPins[8] = {A0, A1, A2, A3, A4, A5, A6, A7};
const int sensorValues[8];
...
void loop() {
  int sensorValues[8];
  int tval;

  for (int i = 0; i < 8; i++) {
    tval = analogRead(sensorPins[i]) ;
    if (tval >= 950) {
      tval = 1;
    }
    else {
      tval = 0;
    }
    sensorValues[i] = tval;
  }
}
```

This code snippet stores all the sensor values as binary values in the sensorValues array. A value of 1 means there is a line below that sensor and a value of 0 means there isn't.

4. Overview, Explanation, Implementation and Working of the PID Control System

The output values of the QTR-8RC can be computed to generate an 'error' value. The error value is how far the sensor is from the line. A positive error value means that the line is towards the right and a negative error value means the line is towards the left. This is shown in Figures (4.1), (4.2) and (4.3).

Error=(0)

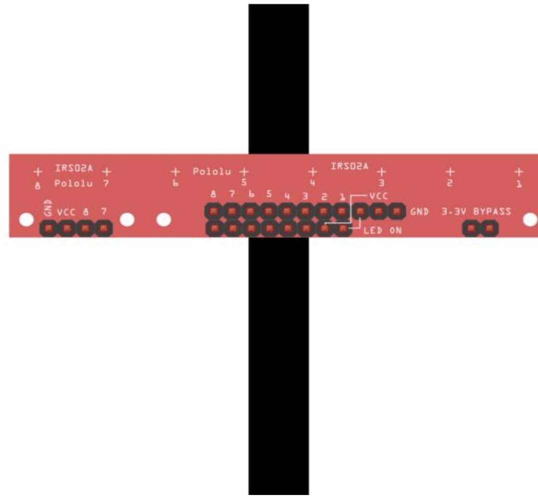

Figure (4.1) - Line in the Middle

Error=(4)

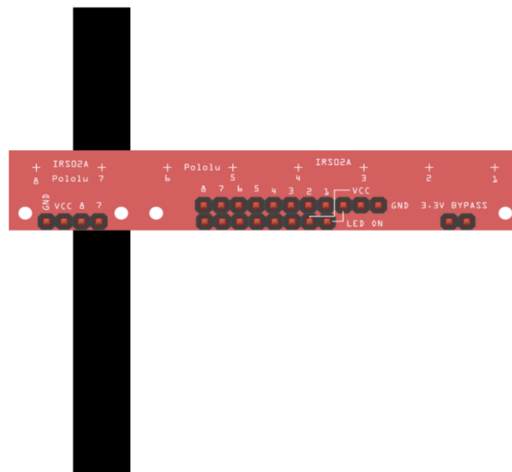

Figure (4.2) - Line towards the Left

Error=(4)

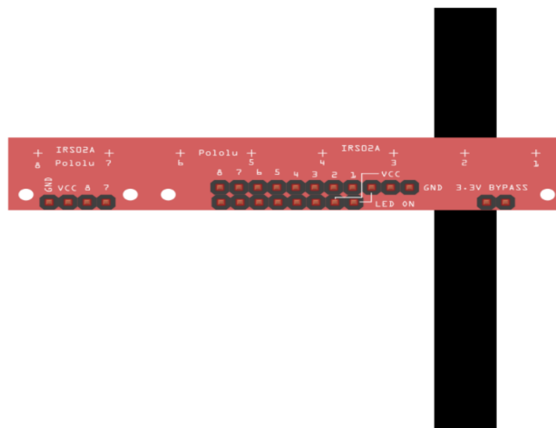

Figure (4.3) - Line towards the Right

This error value is crucial and will be used to control the bot.

```
const int sensorPins[8] = {A0, A1, A2, A3, A4, A5, A6, A7};
int sensorWeights[8] = {-3, -2, -1, 0, 0, 1, 2, 3};

void loop() {
 int sensorValues[8];
 int weightedSum = 0;
 int activeSensors = 0;
 int temp_val;
 int stop = 0;

 for (int i = 0; i < 8; i++) {
  temp_val = analogRead(sensorPins[i]) ;
  if (temp_val >= 950) {
   temp+val = 1;
  }
  else {
   temp_val = 0;
  }
  sensorValues[i] = temp_val;
  if (sensorValues[i] == 1) {
   weightedSum += sensorWeights[i];
   activeSensors++;
  }
 }

 // Calculate error if at least one sensor is active
 if (activeSensors > 0) {
  error = (float)weightedSum / activeSensors;
 } else {
  error = 0;
 }
}
```

This code snippet calculates the error value by having weights for every sensor in the array. From the error value, a ‘steer’ value can be derived. The steer value is the direction the robot goes in. A positive steer value corresponds to the robot moving towards the right and a negative steer value corresponds to the robot moving towards the left. There is an initial base speed value. The speed of the left motor is the base speed + the steer value and the speed of the right motor is the base speed - the steer value.

Now, the proportional part of PID needs to be integrated. The steer value will be set to the error e multiplied by a coefficient Kp. This means the greater the error, the greater the turning speed. A smaller Kp value will not be able to turn fast enough, and the turns will be laggy, whereas a larger Kp value will

overcompensate for the error and oscillate, resulting in a wavy movement. This value needs to be tuned using a trial and error method.
Next, how fast the error is changing, i.e. the derivative of the error, needs to be calculated. This derivative will be multiplied by a coefficient Kd and then added to the steer. So if the error is decreasing too fast, then the steering is slowed down so that it doesn't overshoot. If a higher Kd value is used, that will make the turns very smooth and slow. Whereas if a lower Kd value is used, the movement will be jerky.
After adding the proportional and derivative part, there still remains a small error which is too small for them to deal with. But if the small errors are added overtime, they become significant. So this error is integrated with respect to time and multiplied by a coefficient Ki and added this to the steer.

$$u = K_p e + K_i \int_0^t e\delta t + K_d \frac{de}{dt}$$

In this equation, u is the steer value.

After properly tuning the 3 coefficients - Kp, Ki, and Kd - a PID Controlled Line Follower Robot will be made.

5. PID Tuning

After the robot has been fully made and programmed, it is time to tune the parameters. To achieve the perfect line follower, the Kp, Ki, and Kd constants will need to be tuned. A trial-and-error method needs to be used for this.

5.1. Kp Tuning

The proportional gain (Kp) is the main parameter in a PID control system, having the most influence over the robot's response to the magnitude of the error. A higher Kp value makes the robot highly responsive, enabling it to make sharper turns and correct deviation quickly but results in oscillations rather than stability decreasing the overall speed. A lower Kp value results in a sluggish response, with the robot not being able to react to high errors, especially during sharp curves. If the robot has a wavy movement, the Kp value needs to be decreased, whereas if the robot isn't able to turn enough, the Kp value needs to be increased.

5.2. Ki Tuning

The integral gain (Ki) is the accumulation of the small errors overtime, correcting for biases that the proportional term cannot resolve alone. A higher Ki value accelerates the correction of residual errors, but leaves the robot susceptible to overshooting and oscillations, destabilizing the robot. A lower Ki value results in slow error correction, leaving the robot unable to solve path deviation effectively. If the robot takes too long to return to the line after drifting or exhibits a steady offset, the Ki value should be increased. On the other hand, if the robot oscillates even when deviations are small, the Ki value should be decreased.

5.3. Kd Tuning

The derivative gain (Kd) predicts and reduces the rate of change of error, allowing the robot to stabilize more effectively by combating overshoots. A higher Kd value smooths transitions and prevents oscillations, but it may make the robot overly cautious, which slows its response to rapid changes. A lower Kd value can lead to jerky movements and instability, especially when the robot encounters sharp turns.

If the robot's movement is too abrupt, the Kd value should be increased. On the other hand, if the robot becomes too slow and struggles to move quickly, the Kd value should be decreased.

7. References

1. Rajan K, Kishore. "The Role and Progression of Serving Robots in Hospitality and Service Industries." *IJFMR240527961*, vol. 6, no. 5, 2024, www.ijfmr.com/papers/2024/5/27961.pdf. Accessed 2 Dec. 2024.
2. Ravi, Shyam. "I Made a SUPER FAST Line Follower Robot Using PID!" YouTube, 14 July 2023, www.youtube.com/watch?v=QoNkpnpvEqc.