ENTHIRAN

—a line following robot—

Team Number: 02

Team Members:

E/21/013- T.ABISHANA
E/21/094- C.N.A. DIAS
E/21/120- S.DURGA
E/21/148- S.GANATHIPAN
E/21/188- K.JARSHIGAN
E/21/214- K.KARTHEEPAN
E/21/220- S.KAVISHANTHAN
E/21/386- J.SIVASUTHAN

INTRODUCTION

An autonomous mobile robot with the ability to recognise and follow a predetermined path typically indicated by a black line on a white surface is known as a line-following robot. This project aims to teach fundamental principles of sensor integration, motor control, and feedback mechanisms. It is a basic exercise in robotics and control systems. The automation uses a proportional-derivative (PD) controller to modify its movement in order to keep it on the path after using infrared (IR) sensors to detect the line.

OBJECTIVES

The objective of creating a line-following robot is to design and develop an autonomous system capable of detecting and following a predefined path, typically marked by a contrasting line on the ground. This project aims to integrate sensors, control algorithms, and mechanical components to enable the robot to navigate smoothly along the designated route, making adjustments to its speed and direction based on real-time input from sensors. The robot is designed to demonstrate key concepts in robotics, such as automation, sensor integration, and decision-making, while also providing a practical solution for applications in industrial automation, transportation, and smart systems. Additionally, this project serves as an educational tool to enhance understanding of robotics principles, programming, and hardware interfacing.

COMPONENTS

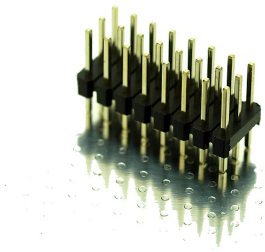
1. L29810 H bridge driver

The L29810 H bridge driver, which regulates the motors' speed and direction, is an essential part of the line-following robot. It controls the current going to the motors, which enables the robot to move forward, backwards, and turn. This driver is necessary for precise motor control in robotic applications since it can handle high currents.

2. 8 array IR sensors

Eight different sensors are attached with the robot but only 6 of them were used in this project to identify the route it follows. These sensors are placed in a sequence to track the contrast between the line and the background. The robot's motions are guided by the data from these sensors, which modify its path based on the position of the line to keep it on course.

3. Circuitboard soldered with male pins



The robot's circuit can be constructed and tested without the need for soldering using a breadboard, a prototyping tool. It makes connecting different electronic parts—like sensors, the Arduino board, and the H bridge driver—easy. The circuit may be modified and troubleshooted more easily during development.

4. Arduino uno

The microcontroller that powers the line-following robot's brain is called an Arduino Uno. It interprets sensor data and uses the H bridge driver to carry out commands to control the motors. The Arduino Uno is a great option for robotics projects since it can be programmed to carry out certain tasks.

5. Yellow motor and wheel x 2 Pcs

The wheels in conjunction with the yellow motors provide the robot the essential movement. Robust and effective, these DC motors provide sufficient torque to propel the robot forward and negotiate curves. These motors' wheels provide steady, smooth motion, which is essential for preserving the robot's equilibrium while it follows the line.

6. Universal Swivel Caster wheel

The robot's chassis is supported by the tiny, revolving caster wheel, which also facilitates smooth navigation. The caster wheel helps the robot to turn effortlessly and maintain balance rather than propelling it forward as the drive wheels do. Usually, it is positioned at the back of the robot to guarantee stability while it moves.

7. 3.7 volt battery (4200mAh) x 2 pcs

Two 3.7-volt batteries, each with a 4200mAh capacity, power the robot. The electrical energy required to power the motors, sensors, and Arduino board is supplied by these rechargeable batteries. They guarantee that the robot can work for prolonged periods of time without requiring frequent recharging due to their large capacity.

8. Battery holder

The two 3.7-volt batteries are held firmly in place by the battery holder, which also makes sure they stay connected to the circuit when it is operating. It offers a practical and secure method of mounting the batteries inside the robot, guarding against loose connections and possible component damage.

9. Jumper cable

Electrical connections are made between the Arduino Uno and the components on the breadboard using jumper cables. These pliable cables are necessary for power and signal routing throughout the robot's circuit, allowing the sensors, microprocessor, and motors to communicate with one another.

10. Pop stick

As a structural element of the robot, the pop stick is frequently used to mount or support different components, like the breadboard or sensors. It strengthens and stabilizes the robot's frame, guaranteeing that every part stays firmly in place while it moves.

11. Nut and Bolts

The robot's parts are fastened together with nuts and bolts to create a stable and secure construction. These tiny mechanical fasteners are essential to preserving the structural integrity of the robot because they guarantee that every component is perfectly aligned and securely fastened.

BODY AND THE CHASSIS

The robot's body is another crucial component. There are numerous types of chassis available, but it's important to remember that a chassis must be sturdy and able to hold all components. Glass, plastic, aluminum, or any other lightweight material can be used to make it. In the “Enthiran” robot, popsticks which are generally considered as a waste material for the body were used. Moreover, the length of the robot was decreased inorder to increase the compactness.

SCHEMATIC DIAGRAM

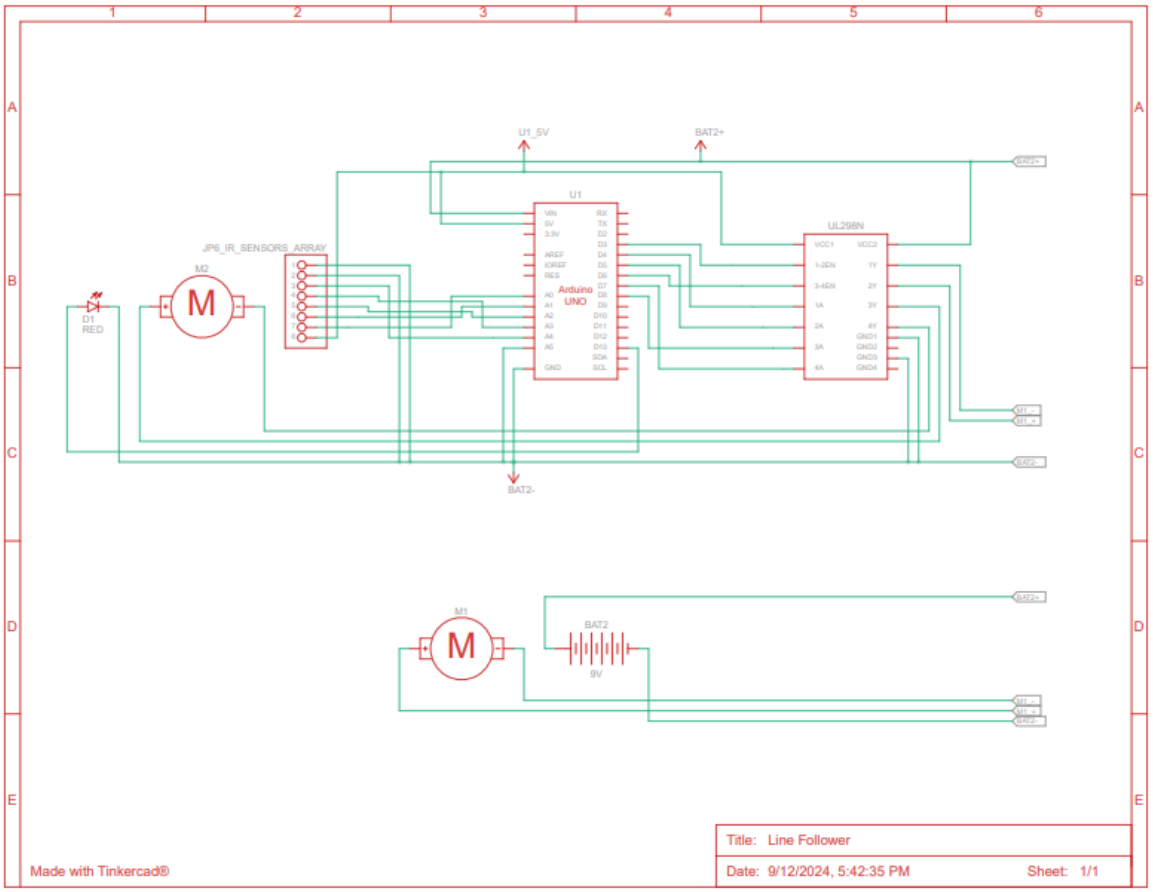


Figure 01 : Schematic of the Line Follower Circuit

CALIBRATION

Calibration is a crucial process to ensure the robot accurately detects the line. During calibration, the robot is manually moved over both the white and black surfaces while the IR sensor readings, which range from 0 to 1024, are recorded. The objective is to determine the maximum and minimum sensor values on these surfaces. A threshold is then set at the midpoint between the minimum and maximum readings. This threshold helps the robot differentiate between the line and the background, ensuring accurate line detection during operation.

PROTOTYPE

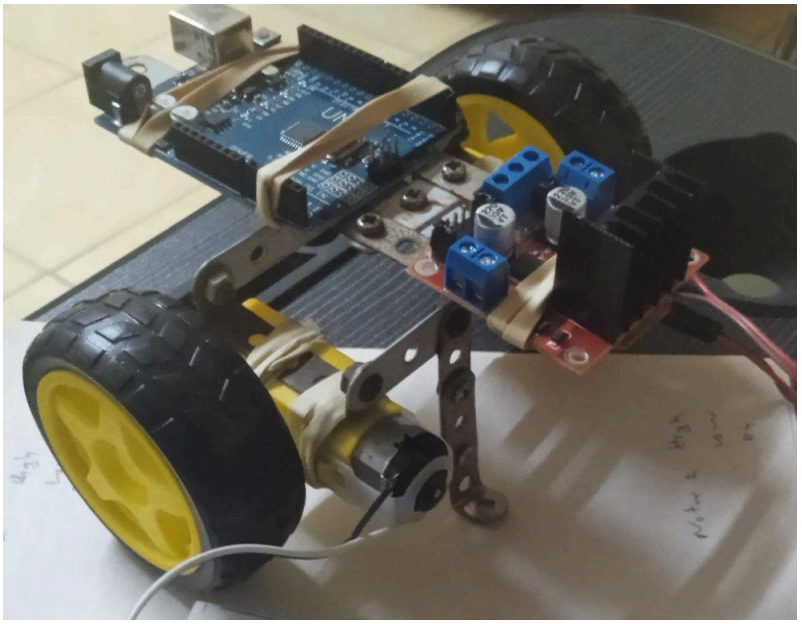


Figure 02 : Line Follower Prototype 01

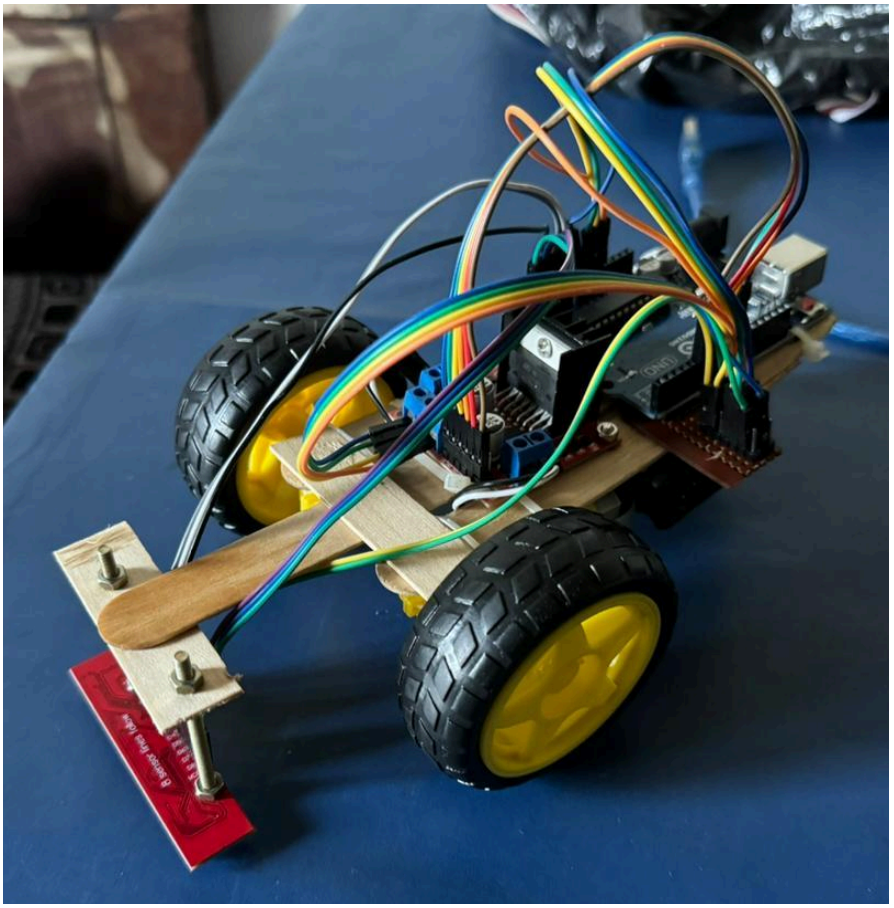


Figure 03 : Line Follower Prototype 02

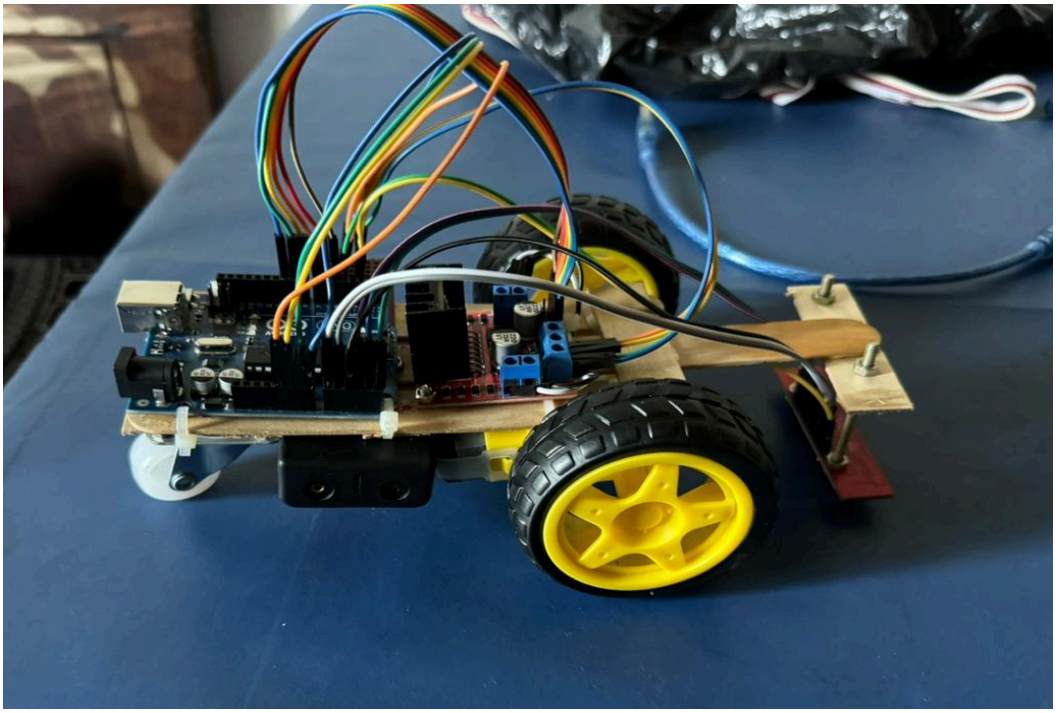


Figure 04 : Line Follower Prototype 02

FINAL MODEL

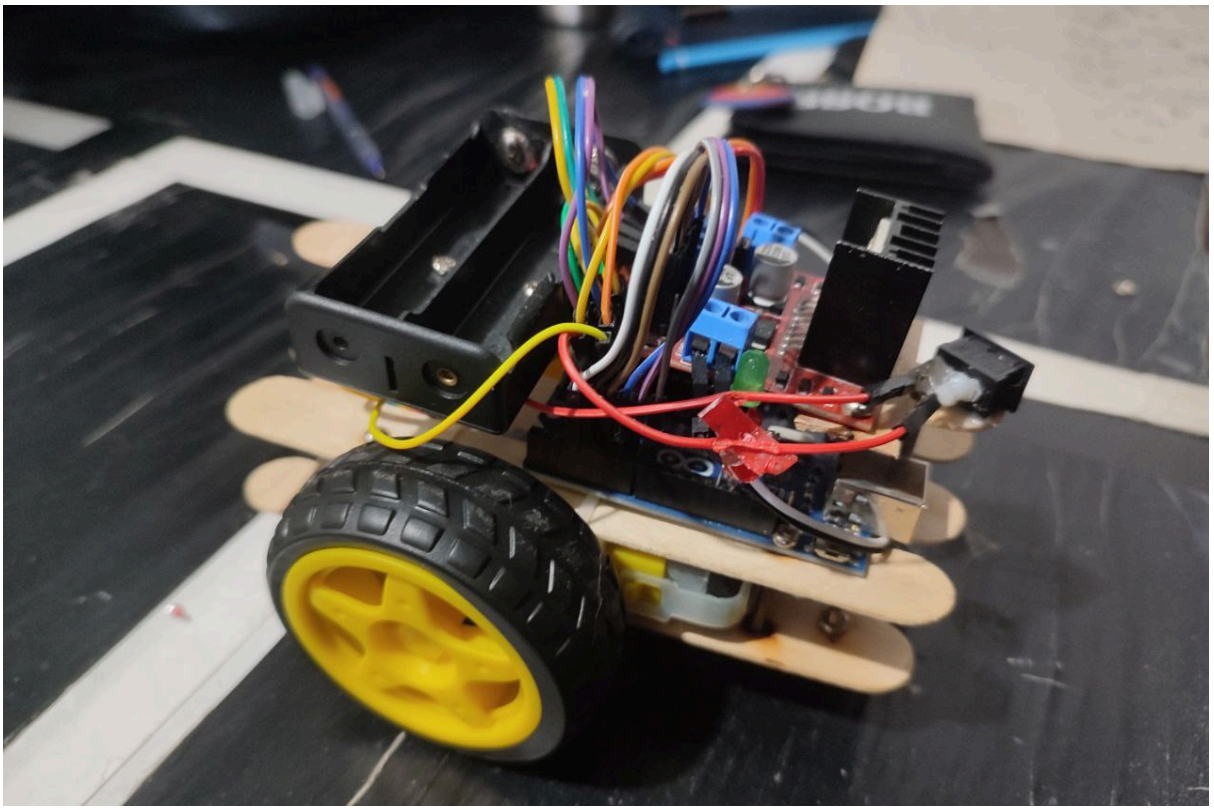


Figure 04 : Line Follower Final Model

SOURCE CODE

```
#include <Arduino.h>

// Defining Variables
  // Motor control pins
  const byte leftMotorEnablePin = 3;    // Enable pin for the left motor
3
  const byte leftMotorInput1 = 4;       // Input pin 1 for the left
motor 4
  const byte leftMotorInput2 = 5;       // Input pin 2 for the left
motor 5
  const byte rightMotorEnablePin = 6;   // Enable pin for the right
motor 6
  const byte rightMotorInput1 = 8;      // Input pin 1 for the right
motor 8
  const byte rightMotorInput2 = 7;      // Input pin 2 for the right
motor 7

  // Indicating LED
  const int LED = 13;

  // IR sensor Pins
  const byte sensorPins[] = {A0, A1, A2, A3, A4, A5};
  const int numSensors = sizeof(sensorPins) / sizeof(sensorPins[0]);
// Number of sensors
  float arena_threshold;

  // Motor regulation
  int motorSpeed = 120;

/**
 * @brief Blinks the LED according to the specified number of times.
 *
 * @param blinkCount Number of times the LED should blink.
 * @param pin The pin number to which the LED is connected.
 */
void blinkLED(int blinkCount, int pin) {
  for (int i = 0; i < blinkCount; i++) {
    digitalWrite(pin, HIGH);  // Turn the LED on
    delay(500);               // Wait for 500 milliseconds
    digitalWrite(pin, LOW);   // Turn the LED off
    delay(500);               // Wait for 500 milliseconds
  }
}

/**
 * @brief Collects the IR data from the surface and calculates the
average value.
 *
 * @param sensorPins Array of sensor pin numbers.
 * @param numSensors Number of sensors.
 * @param readings Number of readings to average.
 * @return The average value of the sensor readings.
 */
float surfaceData(const byte sensorPins[], int numSensors, int
readings) {
  float sum = 0.0;

  for (int j = 0; j < readings; j++) {
    for (int i = 0; i < numSensors; i++) {
```

```
      sum += analogRead(sensorPins[i]);
      delay(100);  // Small delay between sensor readings
    }
  }

  float average = sum / (numSensors * readings);
  return average;
}

/**
 * @brief Calculates the threshold value for IR sensor readings by
measuring
 *        the average IR values of white and black surfaces.
 *
 * @param sensorPins Array of sensor pin numbers.
 * @param numSensors Number of sensors.
 * @param readings Number of readings to average.
 * @return The calculated threshold value.
 */
float calculateThreshold(const byte sensorPins[], int numSensors, int
readings) {
  // Blink LED to indicate starting white surface measurement
  blinkLED(2, LED);
  delay(500);

  // Measure average IR value for white surface
  float white = surfaceData(sensorPins, numSensors, readings);

  // Blink LED to indicate starting black surface measurement
  blinkLED(2, LED);
  delay(500);

  // Measure average IR value for black surface
  blinkLED(4, LED);
  float black = surfaceData(sensorPins, numSensors, readings);

  // Calculate and return the threshold value
  float threshold = (white + black) / 2;

  return threshold;
}

/**
 * @brief Reads sensor values and updates a byte representing sensor
data based on a threshold.
 *
 * This function reads the values from multiple sensors, compares each
sensor's value to a given threshold,
 * and updates a byte where each bit represents whether the
corresponding sensor value exceeds the threshold.
 *
 * @param sensorPins Array of sensor pin numbers.
 * @param numSensors Number of sensors.
 * @param threshold Threshold value for sensor comparison.
 * @return A byte where each bit is set if the corresponding sensor's
value exceeds the threshold.
 */
byte sensorRead(const byte sensorPins[], int numSensors, float
threshold) {
  byte sensorData = 0;  // Initialize the byte to store sensor data
```

```
  float white[2] = {0.0f, 0.0f};
  float black[2] = {0.0f, 0.0f};

  // Iterate over each sensor
  for (int i = 0; i < numSensors; i++) {
    float sensorValue = analogRead(sensorPins[i]);  // Read the analog
value from the sensor

    // Check if the sensor value exceeds the threshold
    if (sensorValue > threshold) {
      // Set the corresponding bit in sensorData to 1 if sensor value
is above threshold
      sensorData |= (1 << i);
      black[0] += sensorValue;
      if (black[1] > sensorValue || black[1] == 0) {
        black[1] = sensorValue;
      }
    } else {
      white[0] += sensorValue;
      if (white[1] < sensorValue || white[1] == 0) {
        white[1] = sensorValue;
      }
    }
  }
  return sensorData;  // Return the byte with updated sensor data
}

/**
 * @brief Maps sensor data byte to error for line condition.
 *
 * @param Case The sensor data byte.
 * @return The error corresponding to the sensor data byte.
 */
float lineCondition(int Case) {
  float error;
  switch (Case) {
    case 32:
      error = 2;
      break;
    case 48:
      error = 1.25;
      break;
    case 56:
      error = 0.75;
      break;
    case 24:
      error = 0.3;
      break;
    case 12:
      error = 0;
      break;
    case 6:
      error = -0.3;
      break;
    case 7:
      error = -0.75;
      break;
    case 3:
      error = -1.25;
      break;
    case 1:
```

```
      error = -2;
      break;
    default:
      error = 0;
      break;
  }
  return error;
}

/**
 * @brief Controls the motors based on error.
 *
 * @param correction The correction for motor control.
 * @param motorSpeed The base motor speed.
 * @param difference The speed difference for turning.
 */
void motorControl(float correction, int motorSpeed, int difference) {
  int leftspeed = motorSpeed + correction * difference;
  int rightspeed = motorSpeed - correction * difference;

  leftspeed = constrain(leftspeed, 100, 255);
  rightspeed = constrain(rightspeed, 100, 255);

  if (leftspeed == rightspeed) {
    digitalWrite(leftMotorInput1, HIGH);
    digitalWrite(leftMotorInput2, LOW);

    digitalWrite(rightMotorInput1, HIGH);
    digitalWrite(rightMotorInput2, LOW);
  } else if (leftspeed < rightspeed) {
    digitalWrite(leftMotorInput1, HIGH);
    digitalWrite(leftMotorInput2, LOW);

    digitalWrite(rightMotorInput1, LOW);
    digitalWrite(rightMotorInput2, HIGH);
  } else {
    digitalWrite(leftMotorInput1, LOW);
    digitalWrite(leftMotorInput2, HIGH);

    digitalWrite(rightMotorInput1, HIGH);
    digitalWrite(rightMotorInput2, LOW);
  }

  // Set motor speeds using PWM
  analogWrite(leftMotorEnablePin, leftspeed);
  analogWrite(rightMotorEnablePin, rightspeed);
}

/**
 * @brief Calculates the PID correction value.
 *
 * @param error The current error value.
 * @param Kp The proportional gain.
 * @param Ki The integral gain.
 * @param Kd The derivative gain.
 * @return The calculated correction value.
 */
float errorCalculation(float error, float K1, float K2, float K3) {
  // PID Calculation
  static float previousError = 0;
  static signed long previousDerivative = 0;
```

```
  // To damp the oscillation speed
  signed long derivative1 = (error - previousError);
  signed long derivative2 = (derivative1 - previousDerivative);
  float correction = K1 * error + K2 * derivative1 + K3 * derivative2;

  previousError = error;
  previousDerivative = derivative1;
  return correction;
}

void setup() {
  Serial.begin(9600);
  // Indicator LED
  pinMode(LED, OUTPUT);

  // Left motor pins setup
  pinMode(leftMotorEnablePin, OUTPUT);
  pinMode(leftMotorInput1, OUTPUT);
  pinMode(leftMotorInput2, OUTPUT);

  digitalWrite(leftMotorEnablePin, LOW);
  digitalWrite(leftMotorInput1, LOW);
  digitalWrite(leftMotorInput2, LOW);

  // Right motor pins setup
  pinMode(rightMotorEnablePin, OUTPUT);
  pinMode(rightMotorInput1, OUTPUT);
  pinMode(rightMotorInput2, OUTPUT);

  digitalWrite(rightMotorEnablePin, LOW);
  digitalWrite(rightMotorInput1, LOW);
  digitalWrite(rightMotorInput2, LOW);

  // Sensor pins setup
  for (byte i = 0; i < numSensors; i++) {
    pinMode(sensorPins[i], INPUT);
  }

  // Calculate the threshold
  arena_threshold = calculateThreshold(sensorPins, numSensors, 20);
  delay(100);
}

void loop() {
  // Get sensor data
  byte sensorData = sensorRead(sensorPins, numSensors,
arena_threshold);

  // Determine the line condition (error)
  float error = lineCondition(sensorData);

  // Calculate correction based on the error
  float correction = errorCalculation(error, 6, 8, 3);

  // Control the motors based on the correction
  motorControl(correction, motorSpeed, 50);
}
```

CHALLENGES AND SOLUTIONS

1. Manual Threshold Detection

Initially, difficulties were faced in distinguishing between black and white surfaces using a manual threshold method. This method required setting a fixed threshold value, but since lighting and surface conditions varied in different environments, it was challenging to maintain consistent performance. This led to inaccuracies in line detection as the robot struggled to adapt to changes in the surroundings.

Solution:

To overcome this issue, a calibration method was implemented which allowed the robot to automatically set its own threshold value. The robot was programmed to take 20 sets of readings from both black and white surfaces during the calibration process. Based on these readings, it calculated the optimal threshold value on its own. This automated approach significantly improved the robot's adaptability to different lighting and surface conditions, making it much more reliable in various environments.

2. Sensor Calibration and Height Adjustment

Another challenge that was encountered was with the eight-channel IR sensor array, which lacked a calibration pin. This made it difficult to determine the correct height for optimal detection of the black-and-white line. The sensor's sensitivity was influenced by various factors, such as surface material and ambient lighting. Moreover, the sensor's performance diminished when the battery power decreased, further complicating accurate detection.

Solution:

To resolve this, multiple tests were conducted to determine the ideal height for the IR sensor. Through trial and error, it was ensured that the sensor was positioned at the right distance from the surface, allowing it to detect the lines and turns smoothly. By optimizing the sensor height, the issues caused by lighting and material variation were mitigated, ensuring consistent performance even in challenging conditions.

3. Difference in the speed of wheels.

The difference in speed of the wheels was observed even though PWM values were given as input. This was because of the difference in performance of the motor which results in oscillation of the wheels.

Solution:

Inorder to rectify this, the speed of the motor was decreased and as a result a stable motor with equal speeds was achieved. In addition, initially, a universal swivel caster wheel was used but it seemed to be restricting the free motion of the robot. Hence a steel ball swivel caster wheel was used instead providing a smooth run of the robot.

4. Stability of the design.

Initially 4 wheels were used and finding the center of gravity and stabilizing was hard especially while turning.

Solution:

By obtaining advice from the lecturer, the number of wheels were reduced from 4 to 2 and a caster wheel was added to increase efficiency.

5. High power supply damaged the components

A 9V battery was used which damaged the uno board and the IR array.

Solution:

2 batteries of 3.7V were used instead and power was maintained under 7.5V ensuring the safety of the components.

6. Budget Constraints

The project was also constrained by a limited budget, which presented significant challenges in acquiring premium components. With restricted resources, different ways were found to optimize the performance of the robot without compromising its functionality.

Solution:

To address this, cost-effective alternatives were focused and most of the parts were made using the available materials. Instead of using expensive acrylic boards to mount the robot's components, ice cream sticks were used as a support structure. This creative solution allowed to keep costs low while ensuring the robot remained functional and structurally sound. By carefully managing our budget and making practical choices, the project continued without exceeding the financial limitations.

CONCLUSION

Robotics plays a critical role in the global economy and everyday life, expanding applications in manufacturing, medical, service, defense, and consumer industries. Research in robotics focuses on competitiveness and designing patents that align with industry needs. A line follower robot, equipped with 6 IR sensors, an Arduino microcontroller board with other components serves as a prototype for industrial use, demonstrating how robots can improve efficiency and accuracy in manufacturing processes. While the setup cost is high due to expensive machinery, land, and skilled staff, such robots offer a viable alternative to skilled labor by handling more goods in less time with better precision and lower per capita costs, particularly in resource-scarce countries like Sri Lanka.