LINE FOLLOWING MAZE SOLVER

- Shivay Mendiratta

Abstract :

This report presents the development of a **Line Follower Maze Solver,** a robotic system designed to navigate a maze autonomously using a predefined algorithm. The project integrates **sensors, microcontrollers, and pathfinding logic** to achieve optimal performance. The robot follows a black line on a white surface (or a black line on white surface) and uses the **Left-Hand Rule Algorithm** to solve mazes efficiently.

Introduction :

Line Following Robots are widely used in Industrial automation, logistics, and autonomous navigation. This project aims to create a cost-effective, efficient, and scalable robotic system that can navigate mazes while following a set path. The scalable robotic system that can navigate mazes while following a set path. The implementation focuses on **sensor-based decision-making** and **motion control.**

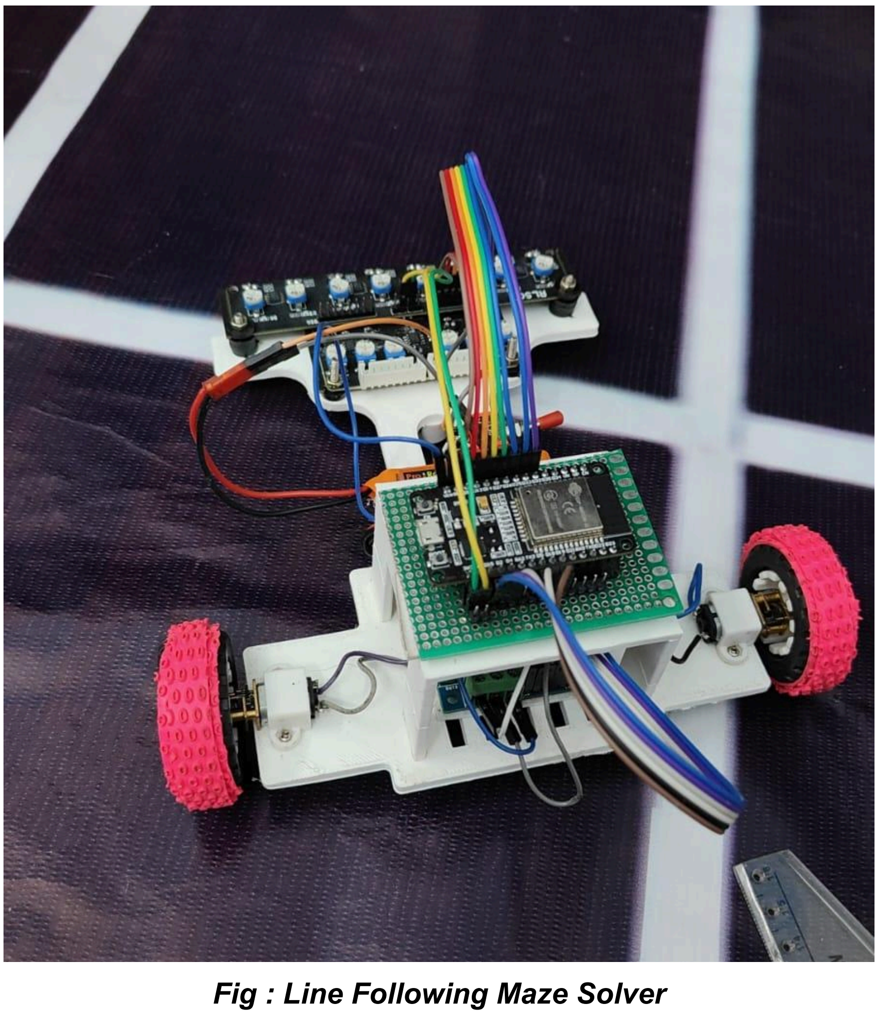


Fig : Line Following Maze Solver

Components Used :

1. Microcontroller - Arduino Nano, ESP32
2. Sensor - SmartElex RLS-08 Analog & Digital Line Follower Sensor Module
3. Motor - N20 Motors (600 rpm)
4. Power Supply - 7.4V Lipo battery
5. Chassis - 3D Printed
6. Motor Driver - L298
7. Jumper wires

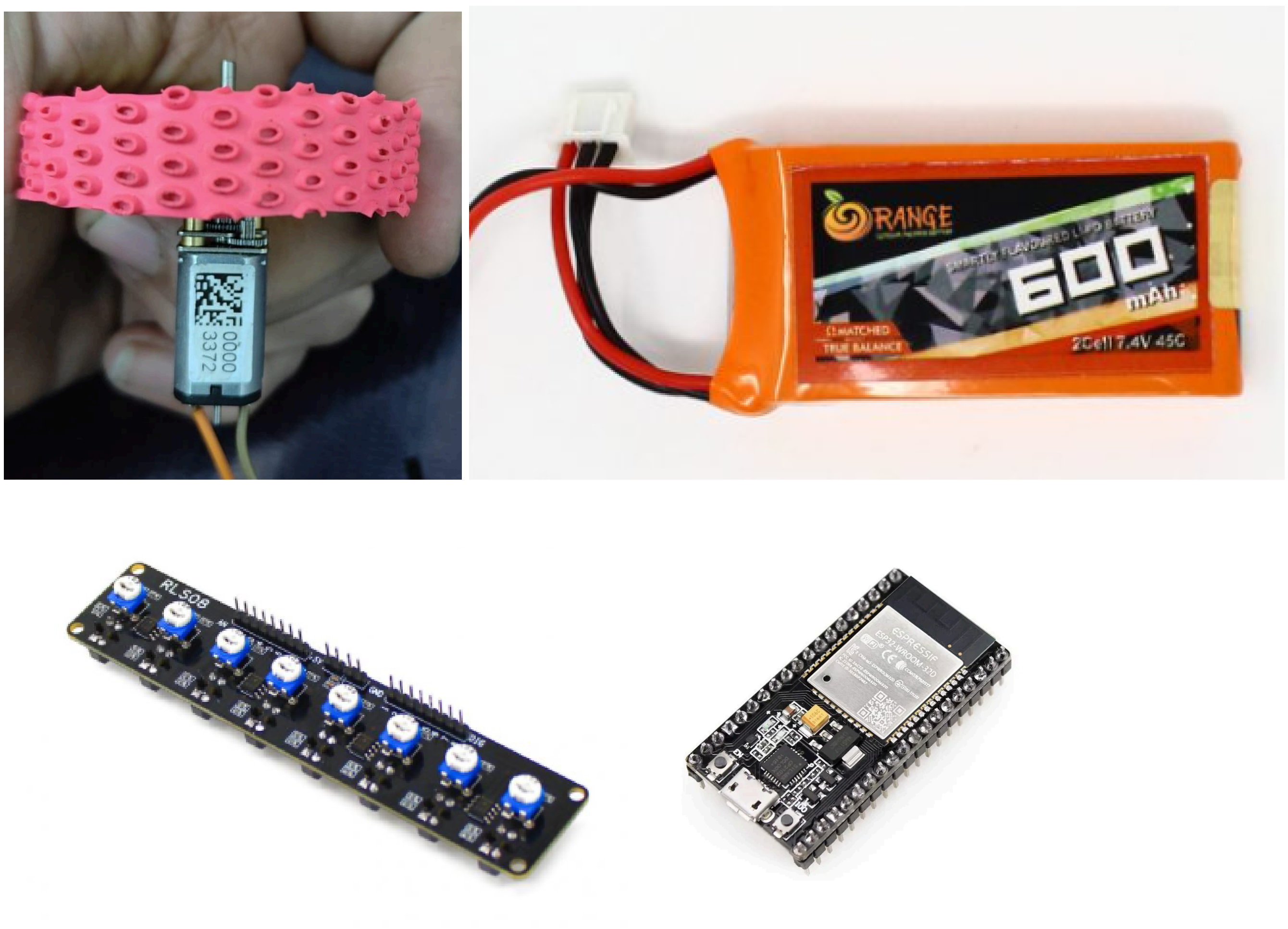


Fig : Motor, Battery, Sensor, Microcontroller

Working Principle :

1. **Line Detection -** IR sensors detect black and white surfaces to determine the robot's position on the track
2. **Decision Making -** The robot follows a predefined **Left-Hand Rule Algorithm,** where it turns left whenever possible to navigate the maze.
3. **Motor Control -** Based on sensor inputs, the microcontroller adjusts the motor speeds to ensure smooth movement.
4. **Path Optimization -** The system memorizes the path and eliminates unnecessary turns in future runs.

Algorithms :

1. **Left-Hand Rule -** The “left-hand rule” is a Maze Solving technique in which you place your left hand on the wall to your left, and keep it touching the wall as you move through the Maze. In other words, you continually take left turns. A parallel rule is the “right-hand rule” which is the same but you instead always take right turns.

   These wall following Maze solving techniques will always solve any 2D Maze, as long as the section of walls at the start is connected to the section of walls at the finish. This means wall following (although simple and easy to apply) won’t work to solve every possible Maze. In scenarios where wall following doesn’t work, you’ll go in a circle around the section of wall you’re connected with and return to where you started.

2. **Proportional Integral Derivative (PID) -** A PID (Proportional Integral Derivative) controller consists of three components that are adjusted based on the difference between a set point (SP) and a measured process variable (PV).

   $$e(t) = SP - PV$$

   The output of a PID controller (u(t)) is calculated using the sum of the Proportional, Integral, and Derivative terms where KP, KI, and KD are constants that can be adjusted to fine-tune the performance of the controller.

   $$u(t) = \mathrm{KP}\, e(t) + \mathrm{KI} \int_0^t e(t)dt + \mathrm{KD}\frac{de(t)}{dt}$$

   $$\mathrm{Output} = \mathrm{Proportional} + \mathrm{Integral} + \mathrm{Derivative}$$

   Proportional (P) control: This component adjusts the output of the process based on the current error between the setpoint and the process variable (PV). The larger the error, the larger the correction applied.

   $$\mathrm{Proportional} = \mathrm{KP}\, e(t)$$

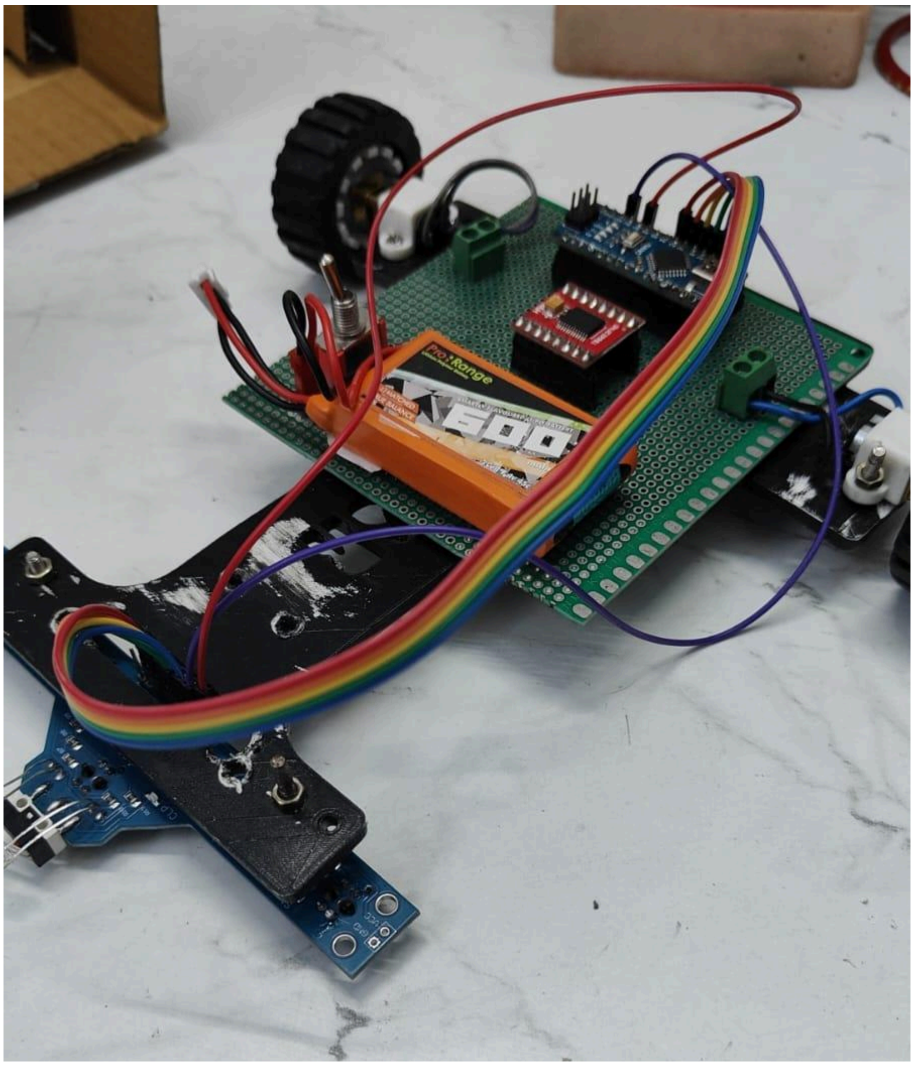
   Integral (I) control: This component adjusts the output based on the accumulated error over time. It helps eliminate steady-state error and can improve the stability of the control system.

$$\text{Integral} = \text{KI}\int_0^t e(t)dt$$

Derivative (D) control: This component adjusts the output based on the rate of change of the error. It helps to dampen oscillations and improve the stability of the control system but is often omitted because PI control is sufficient. The derivative term can amplify measurement noise (random fluctuations) and cause excessive output changes. Filters are important to get a better estimate of the process variable rate of change.

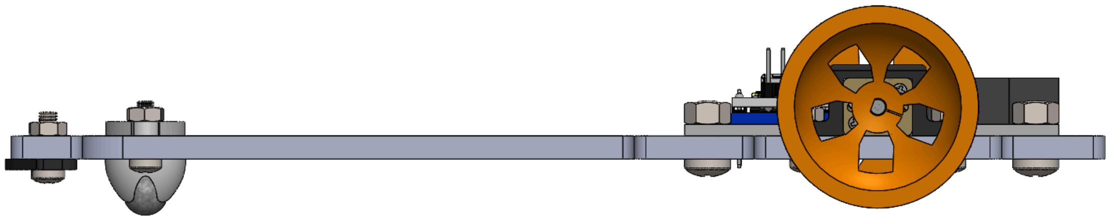
$$\text{Derivative} = \text{KD}\frac{de(t)}{dt} = -\text{KD}\frac{d(PV)}{dt}$$

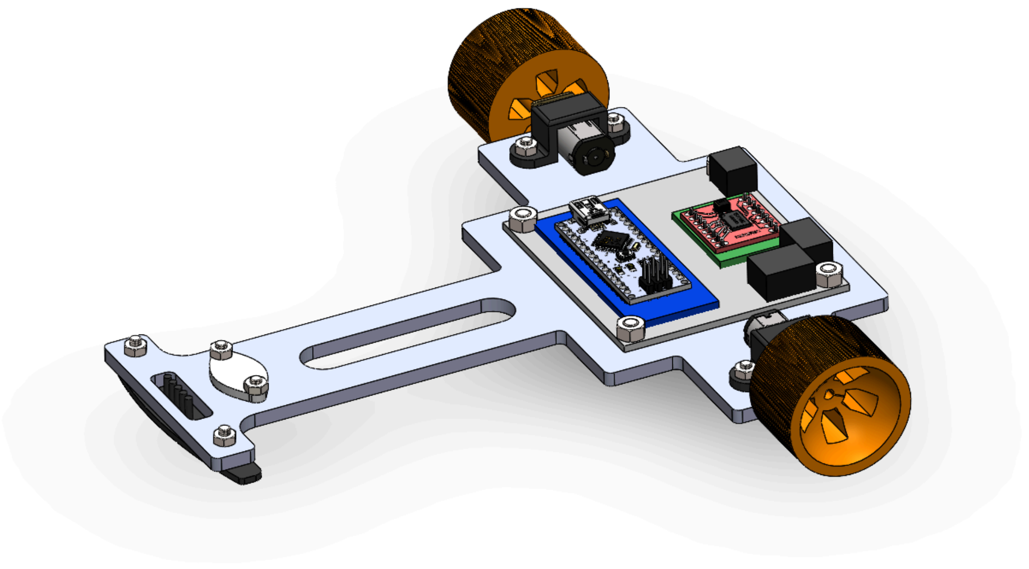
First Model -

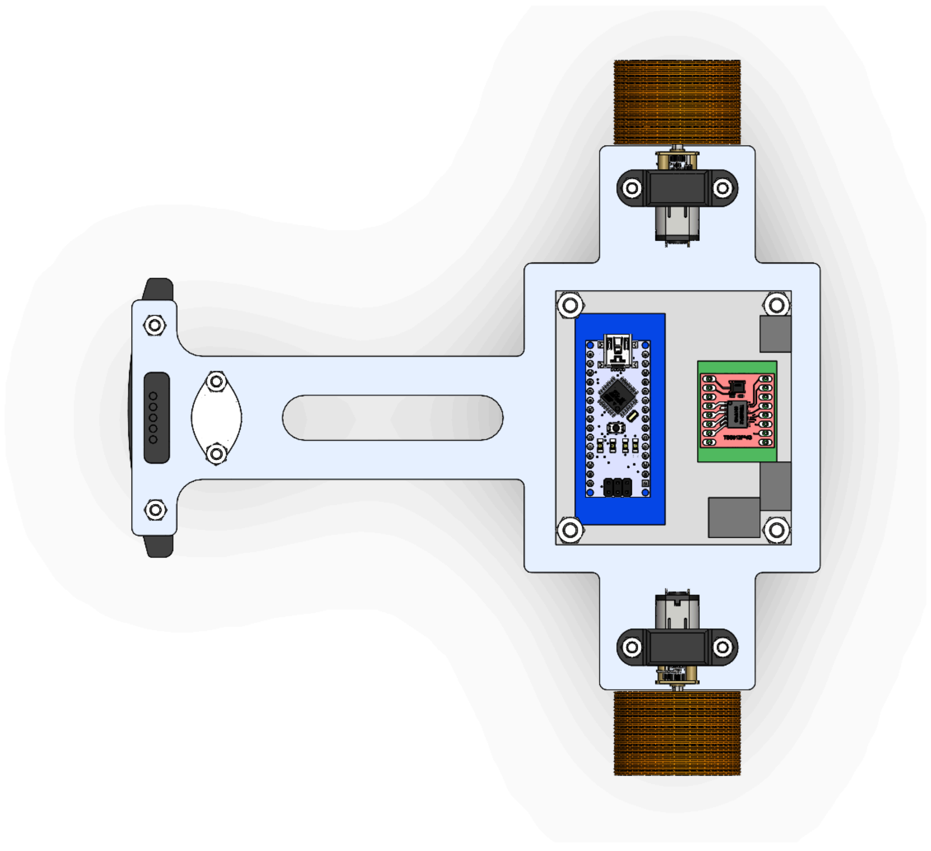


It uses Arduino Nano, which works on a 16MHz clock frequency. This frequency was not enough for calculating PID for motor speed control. That is, the PID worked on a low speed but at high speed it went out of control. The hardware of this model was also very heavy and the PCB was not optimized. The sensor used here was also a local sensor whose readings were not reliable.

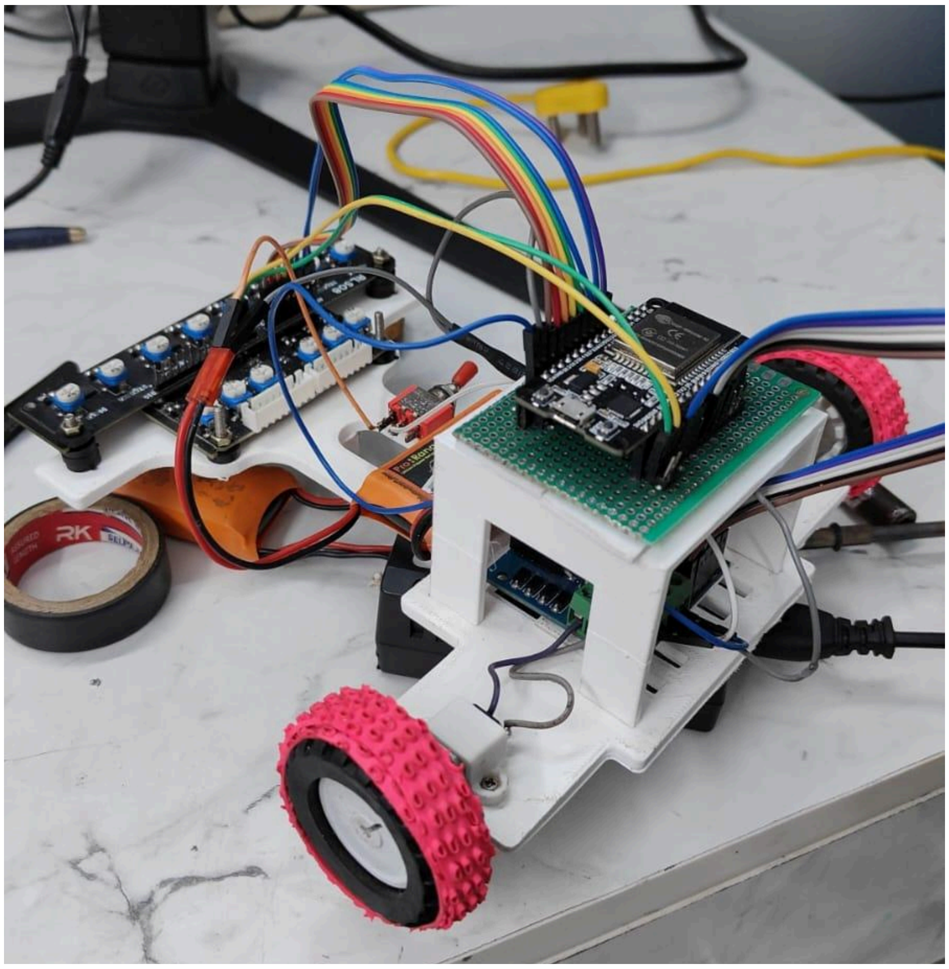
CAD MODEL :







Final Model :



This model uses ESP32, which works on a 240MHz clock frequency. This helped us to make the bot work at high speed. The sensor was also quite reliable and the difference in white and black readings was adequate. The PCB design is also optimized and the tires also have a lot of grip since this is lighter than the previous model, it required high friction to move else it slipped on the map.

Challenges & Optimizations :

- *Sensor Calibration :* Fine-tuning sensor sensitivity to avoid misreadings.
- *Speed Adjustment :* Optimizing motor speed for better stability.
- *Algorithm Enhancement :* Implementing path memory to avoid unnecessary turns.

CODE :

```
#include <QTRSensors.h>

#define LED_BUILTIN 21
// Line Sensor
QTRSensors qtr;
const uint8_t SensorCount = 8;
uint16_t sensorValues[SensorCount];

// Motor Driver
#define PWM1 19
#define AIN1 18// 4
#define AIN2 5// 5
#define PWM2 15 // 9
#define BIN1 2
#define BIN2 4

// PID Properties
const double KP = 0.075;
const double KD = 0.0;
const double KI = 0.0;
double lastError = 0;
const int GOAL = ((SensorCount-1) * 1000 / 2);
const unsigned char MAX_SPEED = 100;
const int turnSpeed = 80;

int intersections = 0;

char dryPath[100];

uint16_t whiteLimit = 200;

void setup() {
  // put your setup code here, to run once:
  // Configure the sensors
  qtr.setTypeAnalog();
  qtr.setSensorPins((const uint8_t[]){32,33,25,26,27,14,12,13}, SensorCount);
  qtr.setEmitterPin(23);

  pinMode(LED_BUILTIN, OUTPUT);


  // Motor
  pinMode(PWM1,OUTPUT);
  pinMode(AIN1,OUTPUT);
  pinMode(AIN2,OUTPUT);
  pinMode(PWM2,OUTPUT);
  pinMode(BIN1,OUTPUT);
  pinMode(BIN2,OUTPUT);

  // Initialize line sensor array
  calibrateLineSensor();


}

void loop() {
  // put your main code here, to run repeatedly:
  // readValues();

  int lineValue[8] = {analogRead(32), analogRead(33), analogRead(25), analogRead(26),
analogRead(27), analogRead(14), analogRead(12), analogRead(13)};

  if (lineValue[0] < whiteLimit && lineValue[1] < whiteLimit && lineValue[2] <
whiteLimit && lineValue[3] < whiteLimit && lineValue[4] < whiteLimit && lineValue[5]
< whiteLimit && lineValue[6] < whiteLimit && lineValue[7] < whiteLimit) {
    STOP();
    delay(400);
    moveForward(MAX_SPEED);
    delay(800);
    if (lineValue[0] < whiteLimit && lineValue[1] < whiteLimit && lineValue[2] <
whiteLimit && lineValue[3] < whiteLimit && lineValue[4] < whiteLimit && lineValue[5]
< whiteLimit && lineValue[6] < whiteLimit && lineValue[7] < whiteLimit) {
      Serial.println("Completed Maze");
      Serial.println(dryPath);
      delay(10000); // Stop until the button is pressed
    } else {
      while(lineValue[2] < whiteLimit) {
        moveLeft(turnSpeed);
      }
      followLine();
      dryPath[intersections] = 'L';
      intersections++;
      delay(500);
    }
  } else if (lineValue[4] < whiteLimit && lineValue[4] < whiteLimit && lineValue[5] <
whiteLimit && lineValue[6] < whiteLimit && lineValue[7] < whiteLimit) {
    STOP();
    delay(400);
    moveForward(MAX_SPEED);
    delay(800);
    STOP();
    delay(400);
    if (lineValue[3] < whiteLimit || lineValue[4] < whiteLimit) {
      followLine();
      dryPath[intersections] = 'S';
      intersections++;
      delay(500);
    } else {
      while(lineValue[2] > whiteLimit) {
        moveRight(turnSpeed);
      }
      followLine();
      dryPath[intersections] = 'R';
      intersections++;
      delay(500);
    }

  } else if (lineValue[0] > whiteLimit && lineValue[1] > whiteLimit && lineValue[2] >
whiteLimit && lineValue[3] > whiteLimit && lineValue[4] > whiteLimit && lineValue[5]
> whiteLimit && lineValue[6] > whiteLimit && lineValue[7] > whiteLimit) {
    STOP();
    delay(400);
    while(lineValue[3] < whiteLimit || lineValue[4] < whiteLimit) {
      uTurn();
    }

    followLine();
    dryPath[intersections] = 'U';
    intersections++;
    delay(500);
  } else if (lineValue[0] < whiteLimit && lineValue[1] < whiteLimit) {
    STOP();
    delay(400);
    while(lineValue[5] > whiteLimit) {
      moveLeft(turnSpeed);
    }
    followLine();
    dryPath[intersections] = 'L';
    intersections++;
    delay(500);
  } else {
    followLine();
  }

}


void readValues() {
  int lineValue[8] = {analogRead(32), analogRead(33), analogRead(25), analogRead(26),
analogRead(27), analogRead(14), analogRead(12), analogRead(13)};
}
```

```
void STOP() {
  digitalWrite(AIN1, LOW);
  digitalWrite(AIN2, LOW);
  analogWrite(PWM1, 0);
  digitalWrite(BIN1, LOW);
  digitalWrite(BIN2, LOW);
  analogWrite(PWM2, 0);
}

void moveForward(unsigned char maxSpeed) {
  digitalWrite(AIN1, HIGH);
  digitalWrite(AIN2, LOW);
  analogWrite(PWM1, maxSpeed);
  digitalWrite(BIN1, HIGH);
  digitalWrite(BIN2, LOW);
  analogWrite(PWM2, maxSpeed);
}

void moveRight(unsigned char maxSpeed) {
  digitalWrite(AIN1, HIGH);
  digitalWrite(AIN2, LOW);
  analogWrite(PWM1, maxSpeed);
  digitalWrite(BIN1, LOW);
  digitalWrite(BIN2, LOW);
  analogWrite(PWM2, 0);
}

void moveLeft(unsigned char maxSpeed) {
  digitalWrite(AIN1, LOW);
  digitalWrite(AIN2, LOW);
  analogWrite(PWM1, 0);
  digitalWrite(BIN1, HIGH);
  digitalWrite(BIN2, LOW);
  analogWrite(PWM2, maxSpeed);
}

void uTurn() {
  digitalWrite(AIN1, HIGH);
  digitalWrite(AIN2, LOW);
  analogWrite(PWM1, 100);
  digitalWrite(BIN1, LOW);
  digitalWrite(BIN2, HIGH);
  analogWrite(PWM2, 100);
}

void calibrateLineSensor() {
  // print the calibration minimum values measured when emitters were on


  Serial.begin(115200);
  delay(500);


  digitalWrite(AIN1, HIGH);
  digitalWrite(AIN2, LOW);
  analogWrite(PWM1, 100);
  digitalWrite(BIN1, LOW);
  digitalWrite(BIN2, HIGH);
  analogWrite(PWM2, 100);

  for (uint16_t i = 0; i <= 400; i++)
  {
    Serial.println(i);
    qtr.calibrate();
  }
  digitalWrite(LED_BUILTIN, LOW); // turn off Arduino's LED to indicate we are
through with calibration

  for (uint8_t i = 0; i < SensorCount; i++)
  {
    Serial.print(qtr.calibrationOn.minimum[i]);
    Serial.print(' ');
  }


  digitalWrite(AIN1, HIGH);
  digitalWrite(AIN2, LOW);
  analogWrite(PWM1, 0);
  digitalWrite(BIN1, LOW);
  digitalWrite(BIN2, HIGH);
  analogWrite(PWM2, 0);

  digitalWrite(LED_BUILTIN, HIGH); // turn on Arduino's LED to indicate we are in
calibration mode

  Serial.println();
}

void followLine(){

  // Get line position
    uint16_t position = qtr.readLineWhite(sensorValues);

    // Compute error from line
    int error = GOAL - position;

    // Compute motor adjustment
    int adjustment = KP*error + KD*(error - lastError);

    // Constraint on the speed of motor (0-255)
    int leftSpeed = constrain(MAX_SPEED + adjustment, 0, 255);
    int rightSpeed = constrain(MAX_SPEED - adjustment, 0, 255);


    // Adjust motors
    digitalWrite(AIN1, HIGH);
    digitalWrite(AIN2, LOW);
    analogWrite(PWM1, leftSpeed);
    digitalWrite(BIN1, HIGH);
    digitalWrite(BIN2, LOW);
    analogWrite(PWM2, rightSpeed);

}
```

CONCLUSION :

The Line Following Maze Solver effectively demonstrated autonomous navigation using simple yet efficient algorithms. The project can be expanded further by integrating **AI-based decision-making** and **real-time mapping** for improved performance.

FUTURE SCOPE :

- Use of Machine Learning for adaptive path optimization.
- Integration of wireless communication for remote monitoring.