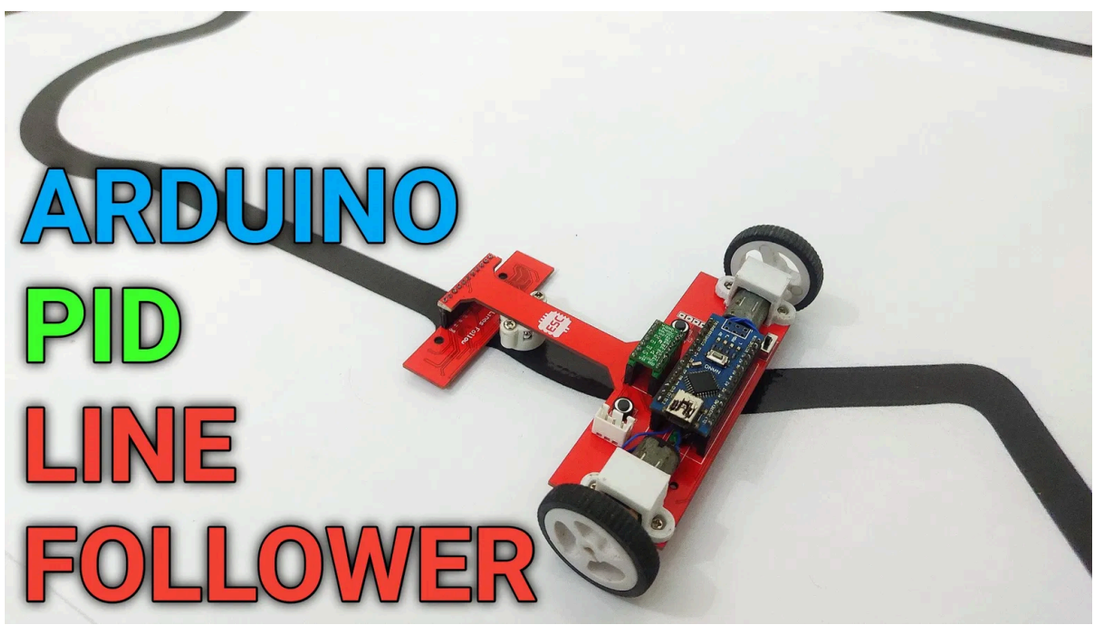
Competition-Level PID Line Follower Robot

Arduino Nano | RLS-08 Sensor Array | L298N Motor Driver

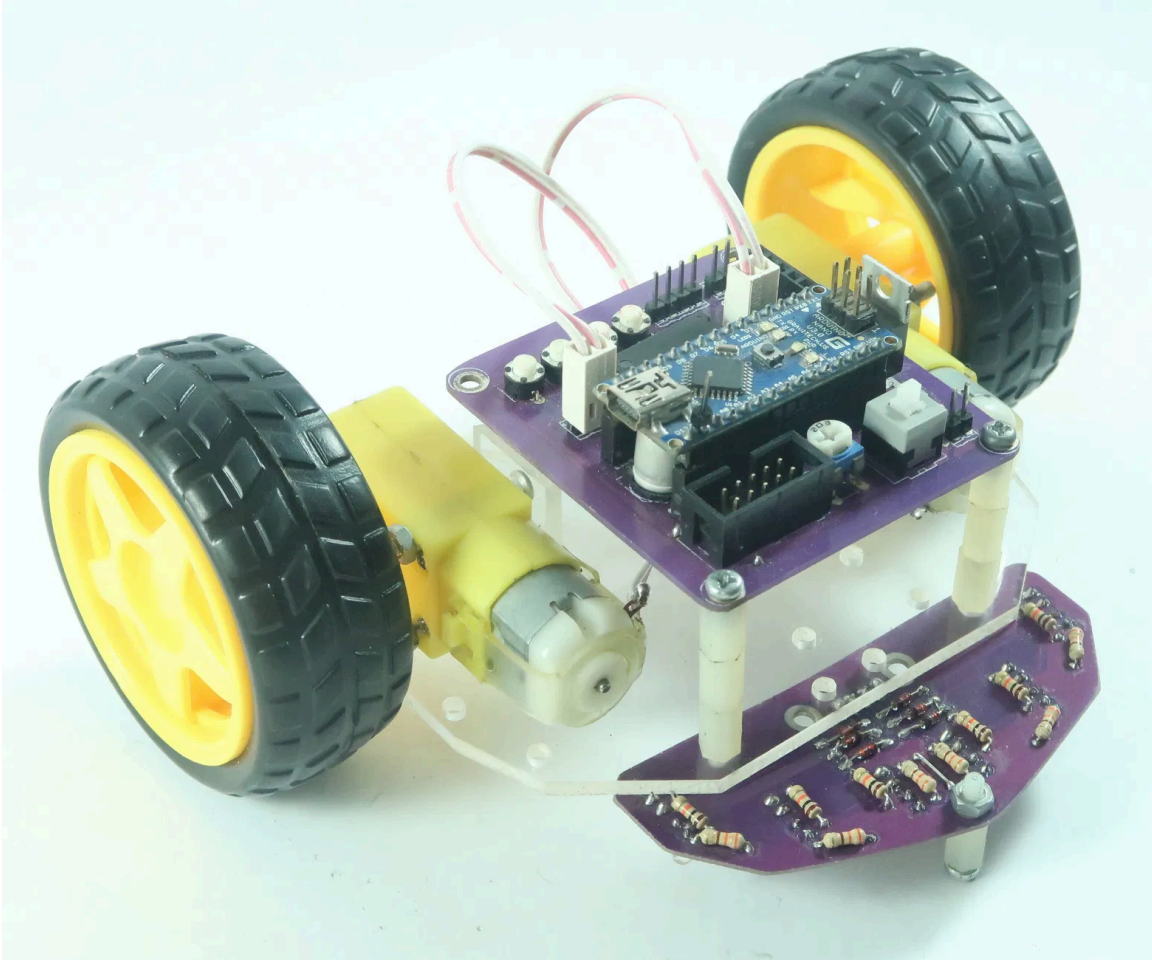


January 2025

Project Overview

Competition-Level Line Follower Robot

- High-performance PID-based line follower capable of navigating complex tracks with sharp turns and varying curves
- Utilizes Arduino Nano, RLS-08 8-sensor array, L298N motor driver, and N20 geared motors
- Advanced features include weighted position calculation, aggressive turning strategies, and line loss recovery
- Optimized for speed, accuracy, and adaptability in competitive environments
- Comprehensive PID tuning capabilities for optimal performance across different track conditions



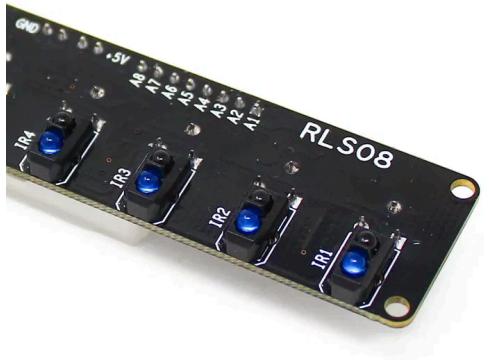
Hardware Components

Arduino Nano

Compact microcontroller with ATmega328P, 14 digital I/O pins (6 PWM), 8 analog inputs, 16MHz clock speed

RLS-08 Sensor Array

8 TCRT5000 IR sensors with 15mm spacing, analog/digital outputs, optimal sensing distance of 3mm



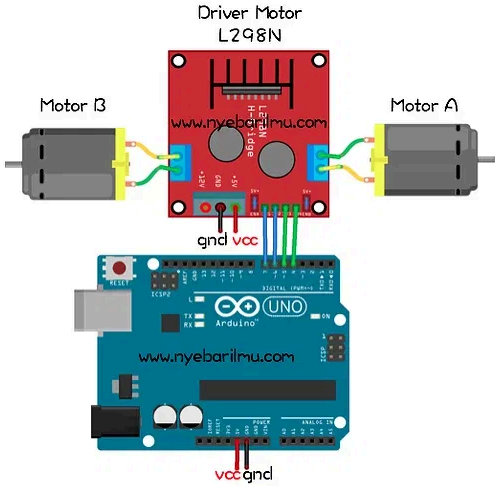
RLS-08 8-Channel Sensor Array

L298N Motor Driver

Dual H-bridge motor driver, 5-35V operating voltage, 2A max current per channel, PWM speed control

N20 Geared Motors

Compact DC motors with metal gearbox, 3-6V operating voltage, high torque at low speeds



L298N Motor Driver with N20 Motors

Power Supply

6V battery pack (4xAA) for motors, separate 5V for Arduino logic

Circuit Connections

L298N to Arduino Nano

ENA: Arduino Digital Pin 9 (PWM)

IN1: Arduino Digital Pin 8

IN2: Arduino Digital Pin 7

ENB: Arduino Digital Pin 10 (PWM)

IN3: Arduino Digital Pin 11

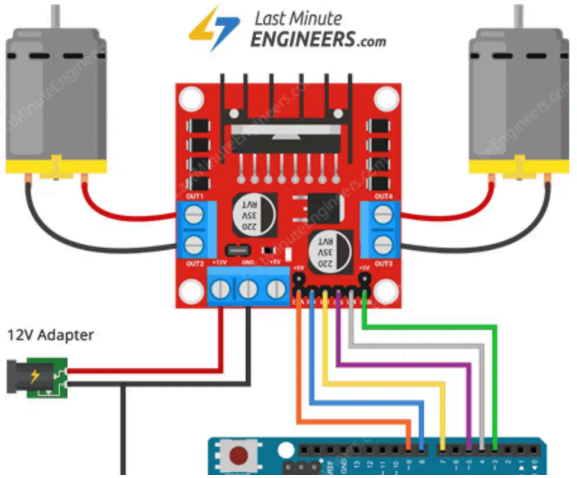
IN4: Arduino Digital Pin 12

RLS-08 to Arduino Nano

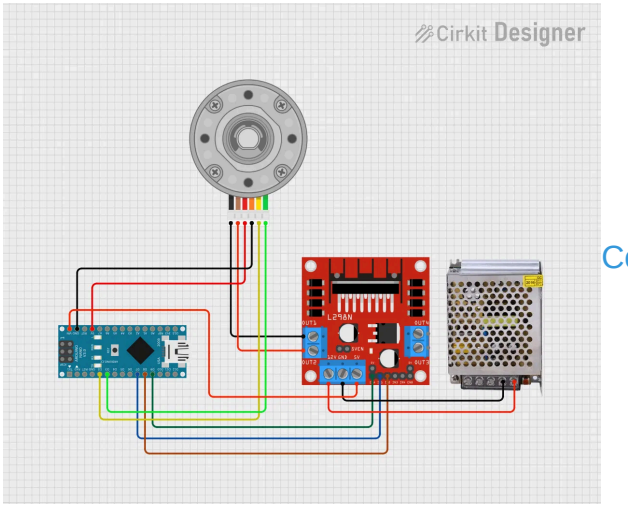
VCC: Arduino 5V

GND: Arduino GND

A0-A7: Arduino Analog Pins A0-A7



L298N Motor Driver Connections



Complete Circuit Connection Diagram

RLS-08 Sensor Array

Key Features

The RLS-08 is a specialized line follower sensor array featuring 8 TCRT5000 infrared emitter-receiver pairs arranged linearly.

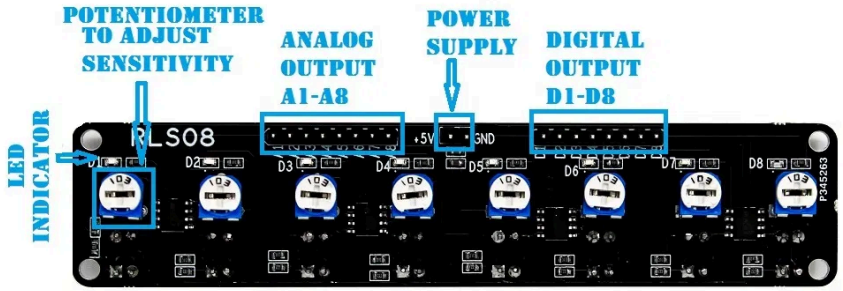
Sensor Type: 8 × TCRT5000 IR Reflective Sensors

Operating Voltage: 4.5V to 5.0V

Output Format: 8 Analog and 8 Digital outputs

Sensing Distance: 3mm optimal

Sensor Spacing: 15mm between each sensor



RLS-08 8-Channel Sensor Array

Advantages for PID Control

The 8-sensor array provides high-resolution line position detection, enabling precise error calculation for PID control. Analog outputs allow for weighted position calculation rather than simple binary detection.

RLS-08 with Adjustable Sensitivity Controls

PID Control Algorithm

PID Control for Line Following

The PID (Proportional-Integral-Derivative) controller calculates an error value as the difference between the desired position (center of line) and the measured position, then applies a correction based on three terms:

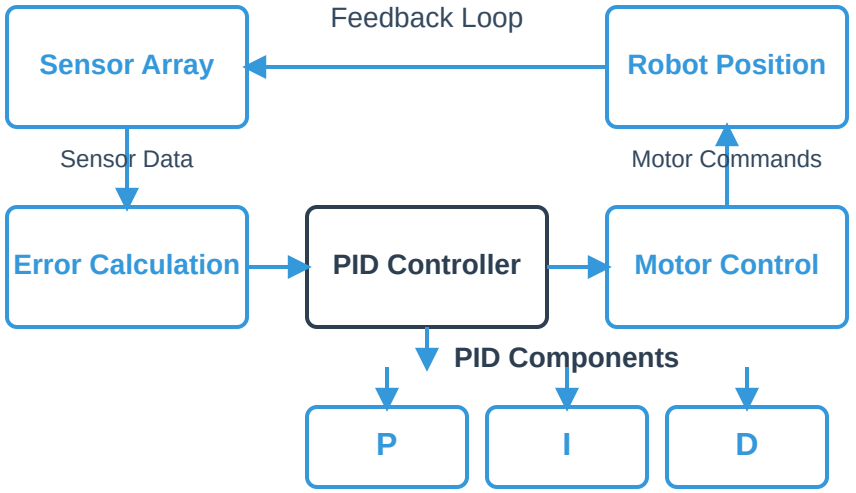
```
P-Term:  Kp × error
```

```
I-Term:  Ki × ∫error dt
```

```
D-Term:  Kd × d(error)/dt
```

```
Output:  P + I + D
```

The output is used to adjust the differential speed of the motors, allowing the robot to smoothly follow the line with minimal oscillation.



Error Calculation

Weighted Average Method

For an 8-sensor array, each sensor is assigned a position weight based on its distance from the center. This creates a continuous error value that precisely indicates the robot's position relative to the line.

> `Error = (Sum of (Sensor_Value * Position_Weight)) / (Sum of Sensor_Values)`

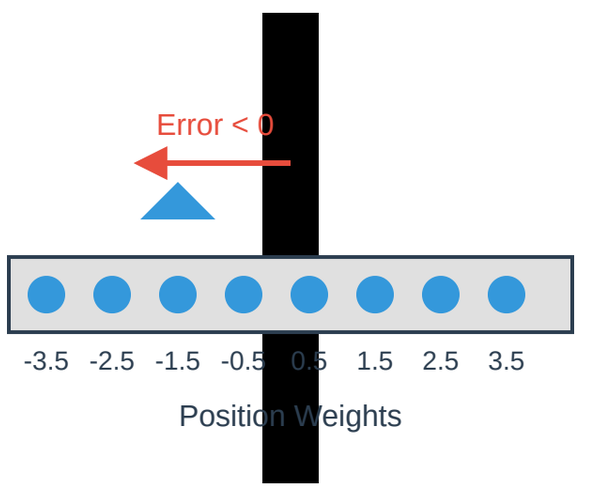
When the robot is centered on the line, the error will be close to zero. A negative error indicates the robot is too far left, while a positive error means it's too far right.

```
// Position weights for 8 sensors
float positions[8] = {-3.5, -2.5, -1.5, -0.5, 0.5, 1.5,
2.5, 3.5};

// Calculate weighted position
for (int i = 0; i < 8; i++) {
  weightedSum += sensorValues[i] * positions[i];
  totalActivation += sensorValues[i];
}

error = weightedSum / totalActivation;
```

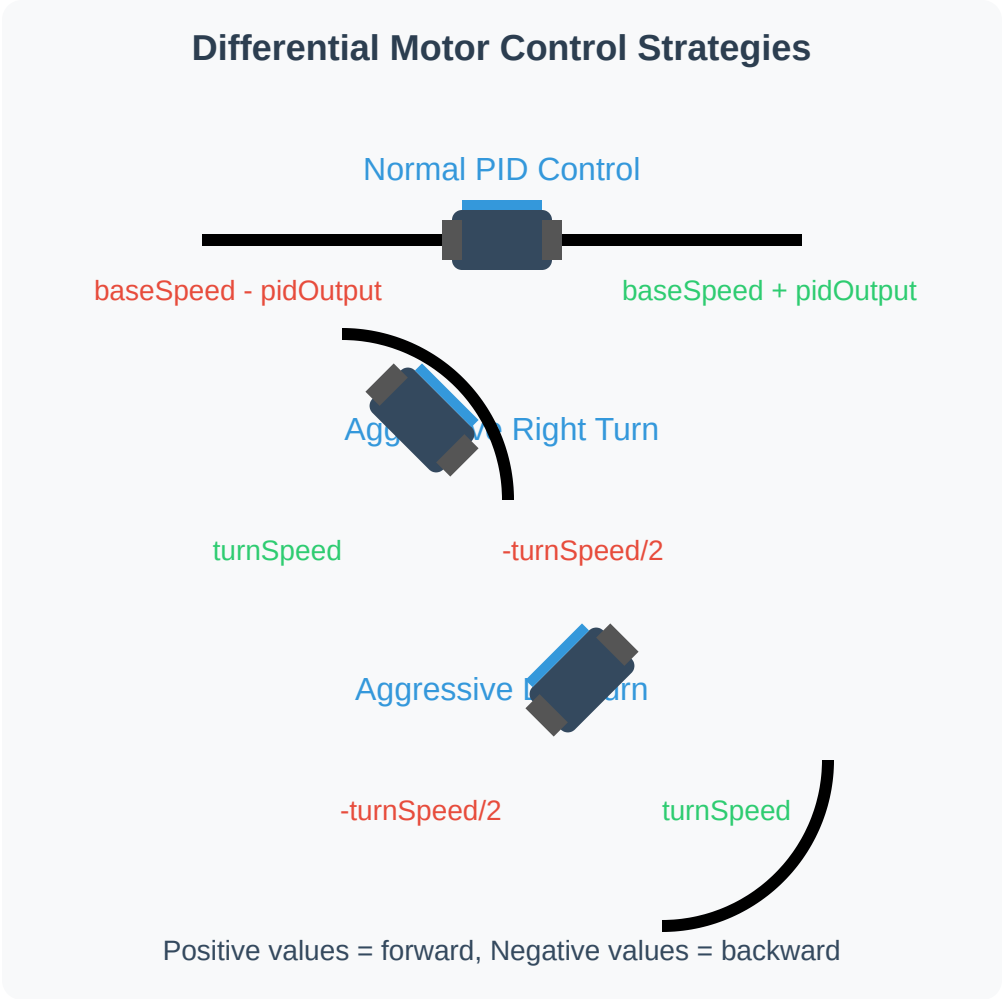
Weighted Sensor Position Calculation



Advanced Turning Strategies

Differential Speed Control

- Standard PID: Adjusts motor speeds proportionally to error (leftSpeed = baseSpeed - pidOutput, rightSpeed = baseSpeed + pidOutput)
- Aggressive Mode: Activates when |error| > threshold, uses pivot turning for sharp corners
- Sharp Right Turn: Left motor forward at turnSpeed, right motor backward at turnSpeed/2
- Sharp Left Turn: Right motor forward at turnSpeed, left motor backward at turnSpeed/2
- Line Loss Recovery: Continues turning in last known direction to search for the line



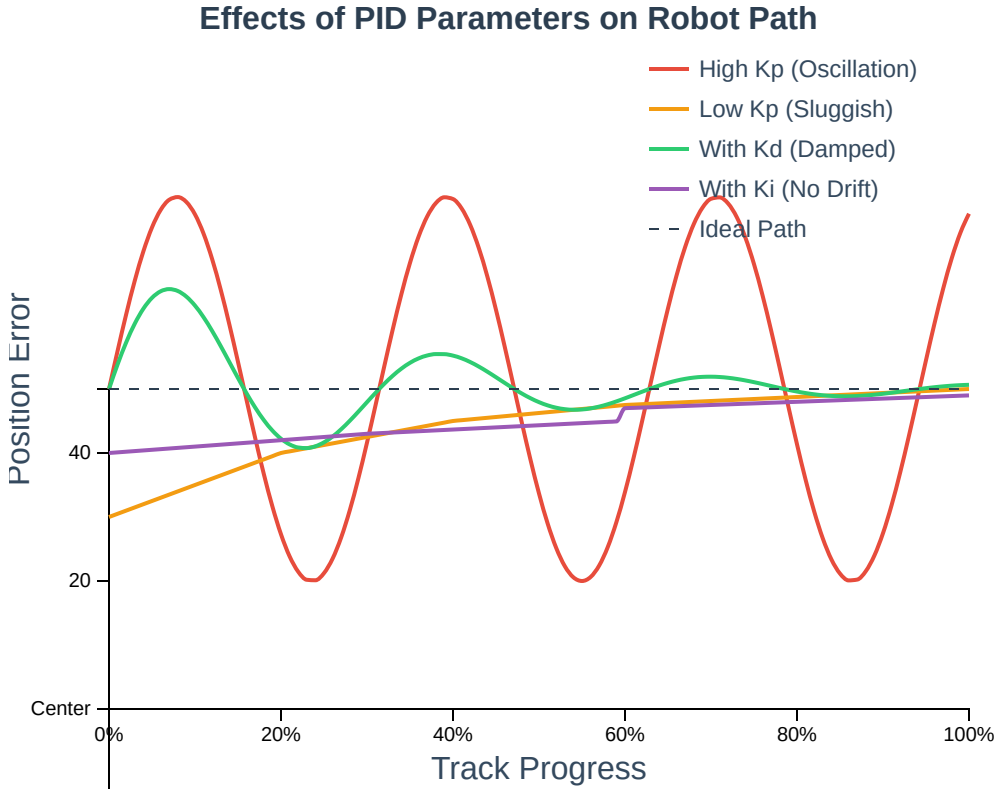
PID Tuning Guide

Systematic Tuning Approach

1. Start with **Kp** only (set Ki=0, Kd=0). Increase Kp gradually until the robot follows the line with slight oscillation.
2. Add **Kd** to dampen oscillations. Increase Kd until the robot moves smoothly without wobbling.
3. Finally, add small **Ki** to eliminate steady-state errors (consistent drift to one side).
4. Fine-tune all parameters together, then gradually increase base speed for optimal performance.

Common Issues

- ⚠ **Oscillation**: Kp too high or Kd too low
- ⚠ **Sluggish response**: Kp too low
- ⚠ **Consistent drift**: Ki too low or zero



Code Implementation

Key Code Components

- The code is structured into functional blocks: sensor reading, error calculation, PID computation, and motor control
- Sensor calibration ensures consistent readings across different lighting conditions and track surfaces
- Error calculation uses weighted position method for precise line detection with 8 sensors
- PID parameters (Kp, Ki, Kd) are tunable variables that control the robot's response characteristics
- Advanced features include aggressive turning mode, line loss recovery, and real-time parameter tuning via Serial Monitor

```
// PID variables
float Kp = 2.0;    // Proportional gain
float Ki = 0.0;    // Integral gain
float Kd = 1.0;    // Derivative gain

float error = 0;
float previousError = 0;
float integral = 0;
float derivative = 0;
float pidOutput = 0;
```

```
void calculateError() {
  // Weighted position calculation for 8 sensors
  float weightedSum = 0;
  float totalActivation = 0;

  float positions[8] = {-3.5, -2.5, -1.5, -0.5, 0.5, 1.5,

  for (int i = 0; i < 8; i++) {
    weightedSum += calibratedSensorValues[i] * positions[i
    totalActivation += calibratedSensorValues[i];
  }

  if (totalActivation > 0) {
    error = weightedSum / totalActivation;
  }
}
```

```
void calculatePID() {
  // Proportional term
  float proportional = error;

  // Integral term (with windup protection)
  integral += error;
  integral = constrain(integral, -100, 100);

  // Derivative term
  derivative = error - previousError;

  // Calculate PID output
  pidOutput = (Kp * proportional) + (Ki * integral) + (Kd

  // Store current error for next iteration
  previousError = error;
}
```

```
void applyMotorControl() {
  if (aggressiveMode && abs(error) > 2.0) {
    // Aggressive turning for sharp curves
    if (error > 0) {
      // Sharp right turn
      leftMotorSpeed = turnSpeed;
      rightMotorSpeed = -turnSpeed/2;
    } else {
      // Sharp left turn
      leftMotorSpeed = -turnSpeed/2;
      rightMotorSpeed = turnSpeed;
    }
  } else {
    // Normal PID control
    leftMotorSpeed = baseSpeed - pidOutput;
    rightMotorSpeed = baseSpeed + pidOutput;
  }
}
```