Control Systems Lab - Experiment No. 3: Line follower

Group No.: 11
Members:
Sanjay Meena(22B3978)
Devtanu Barman(22B3904)

October 21, 2024

Google Drive Link for Demo:
`https://drive.google.com/drive/folders/1EP4_pcv7_ygcM5xq2WvQcfCHcRlpCmow?usp=drive_link`

1 Aim

To design and implement a PID controller for the Spark V robot to make it follow a continuous track, using the IR sensors provided on the robot for this purpose.

2 Objectives

- To trace the track within 30 seconds.

3 Prerequisites

- AVR Programming
- PID Control Logic

4 Materials and Equipment Required

- Spark V bot

- Spark V charger
- ISP programmer
- A-B cable
- Screwdriver

5 Track

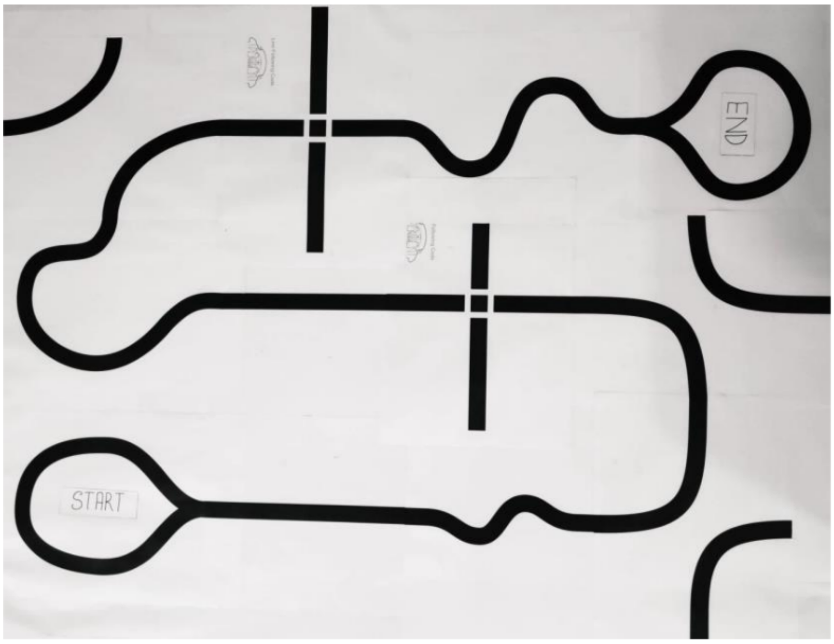

Figure 1: Design of the track to be traced

6 Explaination of the Code and Procedure

6.1 Code

The following C code explains the functionality and structure used for controlling the line-following robot using PID control. The robot uses ADC sensors to read input from the track, and PID control is used to adjust motor speeds to minimize deviations from the desired path. Below is the code we implemented:

```c
// AVR motor control code
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include "lcd.c"

// Global variables and functions
unsigned char ADC_Conversion(unsigned char);
unsigned char ADC_Value;
unsigned char l = 0;
unsigned char c = 0;
unsigned char r = 0;
int l_value = 0;
int c_value = 0;
int r_value = 0;
int l_max = 0;
int c_max = 0;
int r_max = 0;
float angle_error_value = 0;
float angle_error_guess = 0;
float angle_error_prev = 0;
unsigned char PortBRestore = 0;
float relative_error_l, relative_error_r;
float final_angle_error;
float actuation_CM;
float fb;
float l_speed;
float r_speed;
float kp = 0;
float ki = 0;
float kd = 0;
float angle_error_diff;
float angle_error_sum;
float mod = 0;
unsigned char direction_command = 0b00000000;

// Function to configure motion pins
void motion_pin_config (void) {
    DDRB = DDRB | 0b00001111; // Set direction of PORTB3 to PORTB0 pins as OUTPUT
    PORTB = PORTB & 0b11110000; // Set initial value of PORTB3 to PORTB0 pins to 0
    DDRD = DDRD | 0b00110000; // Setting PD4 and PD5 pins as OUTPUT for PWM
    PORTD = PORTD | 0b00110000; // PD4 and PD5 pins for VELOCITY CONTROL using PWM
}

```

```
// Function to set motor direction
void motion_set (unsigned char DirectionCommand) {
    unsigned char PortBRestore = 0;
    DirectionCommand &= 0b00001111; // Remove upper nibble from DirectionCommand
    PortBRestore = PORTB; // Read PORTB's original status
    PortBRestore &= 0b11110000; // Reset PB0,1,2,3 to 0 momentarily
    PortBRestore |= DirectionCommand;// Add direction command and restore PORTB status
    PORTB = PortBRestore; // Set the command to the port
}

// Motion control functions
void forward (void) { motion_set(0b00000110); }
void back (void) { motion_set(0b00001001); }
void left (void) { motion_set(0b00000101); }
void right (void) { motion_set(0b00001010); }
void soft_left (void) { motion_set(0b00000100); }
void soft_right (void) { motion_set(0b00000010); }
void soft_left_2 (void) { motion_set(0b00000001); }
void soft_right_2 (void) { motion_set(0b00001000); }
void hard_stop (void) { motion_set(0b00000000); }
void soft_stop (void) { motion_set(0b00000000); }

// Function to initialize ADC
void adc_init() {
    ADCSRA = 0x00;
    ADMUX = 0x20; // Vref=5V external --- ADLAR=1 --- MUX4:0 = 0000
    ACSR = 0x80;
    ADCSRA = 0x86; // ADEN=1 --- ADIE=1 --- ADPS2:0 = 110
}

// Function to initialize devices
void init_devices (void) {
    cli(); // Clears global interrupts
    port_init();
    adc_init();
    sei(); // Enables global interrupts
}

// Function to configure LCD port
void lcd_port_config (void) {
    DDRC = DDRC | 0xF7; // Set all LCD pin directions as output
    PORTC = PORTC & 0x80;// Set all LCD pins to logic 0 except PORTC7
}

// ADC pin configuration
void adc_pin_config (void) {
    DDRA = 0x00; // Set PORTA direction as input
```

```c
    PORTA = 0x00; // Set PORTA pins floating
}

// Function to initialize ports
void port_init() {
    lcd_port_config();
    adc_pin_config();
    motion_pin_config();
}

// Timer1 initialize - prescale: 64, desired value: 450Hz
void timer1_init(void) {
    TCCR1B = 0x00; // Stop
    TCNT1H = 0xFF; // Setup
    TCNT1L = 0x01;
    OCR1AH = 0x00;
    OCR1AL = 0xFF; // Left motor PWM
    OCR1BH = 0x00;
    OCR1BL = 0xFF; // Right motor PWM
    ICR1H = 0x00;
    ICR1L = 0xFF;
    TCCR1A = 0xA1;
    TCCR1B = 0x0D; // Start Timer
}

// Function to perform ADC conversion
unsigned char ADC_Conversion(unsigned char Ch) {
    unsigned char a;
    Ch = Ch & 0x07; // Select ADC channel
    ADMUX = 0x20 | Ch;
    ADCSRA = ADCSRA | 0x40; // Start conversion
    while ((ADCSRA & 0x10) == 0); // Wait for conversion
    a = ADCH;
    ADCSRA = ADCSRA | 0x10; // Clear ADIF (Interrupt Flag)
    return a;
}

// Main Function
int main(void) {
    // Initializations
    init_devices();
    timer1_init();
    lcd_set_4bit();
    lcd_init();

    while(1) {
        // Sensing part
        l = ADC_Conversion(3);
        c = ADC_Conversion(4);
```

```
        r = ADC_Conversion(5);
        lcd_print(1, 1, l, 3);
        lcd_print(1, 5, c, 3);
        lcd_print(1, 9, r, 3);

        // Angle error calculation
        l_max = 110;
        c_max = 113;
        r_max = 115;
        l_value = (l - 6) * 100 / l_max + 0.0001;
        c_value = (c - 6) * 100 / c_max + 0.0001;
        r_value = (r - 6) * 100 / r_max + 0.0001;

        if ((l_value >= 1) && (r_value >= 1)) {
            angle_error_value = 0;
            angle_error_prev = 0;
        } else if ((c_value <= 3) && (r_value >= 0.1)) {
            angle_error_value = 200 - r_value;
        } else if ((c_value >= 2) && (r_value >= l_value)) {
            angle_error_value = 100 - c_value;
        } else if ((c_value >= 2) && (l_value >= r_value)) {
            angle_error_value = c_value - 100;
        } else if ((c_value <= 3) && (l_value >= 0.1)) {
            angle_error_value = l_value - 200;
        } else {
            angle_error_value = angle_error_prev * 6;
        }

        angle_error_guess = angle_error_value / 6;

        // PID control
        kp = 10.0;
        ki = 0.0001;
        kd = 1;
        angle_error_sum += angle_error_guess;
        angle_error_diff = angle_error_guess - angle_error_prev;
        fb = (kp * angle_error_guess) + (ki * angle_error_sum) + (kd *
            angle_error_diff);

        // Actuation
        if (fb < 0) {
            l_speed = 255 + fb;
            r_speed = 255;
        } else {
            l_speed = 255;
            r_speed = 255 - fb;
        }

        if (l_speed > 255) l_speed = 255;
```

```
        if (l_speed < -255) l_speed = -255;
        if (r_speed > 255) r_speed = 255;
        if (r_speed < -255) r_speed = -255;

        // Set motor directions
        if (l_speed < 0) left();
        else if (r_speed < 0) right();
        else forward();

        if (l_speed < 0) l_speed = -l_speed;
        if (r_speed < 0) r_speed = -r_speed;

        OCR1AL = floor(l_speed);
        OCR1BL = floor(r_speed);

        angle_error_prev = angle_error_guess;
    }
}
```

6.2 Explaination

6.2.1 ADC Initialization and Conversion

- `adc_init()`: Initializes the ADC (Analog to Digital Converter) to read analog sensor values from the track. This function sets up the ADC registers.

```
    ADCSRA = 0x86; // Enable ADC and set prescaler
    ADMUX = 0x20; // Vref = 5V external, ADLAR = 1
```

- `ADC_Conversion(unsigned char Ch)`: Performs the ADC conversion for a specific channel (sensor) and returns the digital value.

```
    ADMUX = 0x20 | Ch; // Select ADC channel
    ADCSRA |= 0x40; // Start conversion
    while (!(ADCSRA & 0x10)); // Wait for conversion to complete
    return ADCH; // Return 8-bit result
```

6.2.2 PID Control Logic

PID (Proportional-Integral-Derivative) control is used to adjust the robot's motion based on the sensor readings. The following components of PID are used:

- **Proportional (K_p)**: Proportional control corrects the robot's motion based on the current error. The larger the error, the stronger the correction.

$$P = K_p \cdot \text{error} \tag{1}$$

- **Integral (K_i)**: Integral control accounts for past errors to ensure that small errors over time do not accumulate, helping to eliminate steady-state error.

$$I = K_i \cdot \sum \text{error} \tag{2}$$

- **Derivative (K_d)**: Derivative control predicts future errors based on the rate of change of the error, helping to dampen overshooting.

$$D = K_d \cdot \frac{d}{dt}(\text{error}) \tag{3}$$

- **Control Output**: The combined PID output determines the correction applied to the motors' speeds.

$$\text{output} = P + I + D \tag{4}$$

  This output is applied to the motor speeds to adjust the robot's direction.

- Proportional Control (K_p) in the context of the Track
  On straight sections of the track (as seen in Figure 1), the robot's error will be minimal, and proportional control will provide gentle corrections to keep the robot centered. On sharp curves, where the error might be large, K_p ensures a quick response to steer the robot back to the center of the line. However, too high a value for K_p could cause the robot to oscillate between extremes on the track.

- Integral Control (K_i) on Curves and Crossings
  In sections of the track where the robot crosses intersections or drifts due to minor sensor inaccuracies, the integral term will help compensate for cumulative small errors. The intersections on the track (Figure 1) might cause momentary loss of the line, and the integral term helps by gradually adjusting based on the history of errors.

- Derivative Control (K_d) to Avoid Overshooting
  On curves and at intersections, where the robot might face rapid changes in error, derivative control helps by predicting future errors. This prevents the robot from overshooting the line or making large, sudden corrections, particularly when approaching sharp curves or line crossings.

6.2.3 LCR Sensor Usage in Different Parts of the Track

- Straight Sections
  In straight sections, the C sensor detects the line, while both L and R sensors are low. The proportional term K_p adjusts for any minor drifts detected by the L or R sensors. If the L sensor detects the line, the robot slows down the left motor to steer back to the center.

- Curved Sections
  On a curve, the C sensor will detect the line along with either the L or R sensor. For left-hand curves, the L sensor and C sensor will detect the line, and the proportional term K_p reduces the left motor speed to follow the curve. The derivative term K_d helps prevent the robot from overshooting the curve by dampening the response.

- T-Intersections (T-Points)
  At a T-intersection, the robot detects both left and right lines using the L and R sensors while the C sensor loses the line. Based on the programmed logic, the robot will choose to turn left or right. For example, if both L and R sensors detect lines, the robot may be programmed to turn left, reducing the speed of the left motor and increasing the speed of the right motor.

- Crossing Points
  At crossing points, the robot may detect lines on all three sensors (L, C, and R). The robot continues following the C sensor while ignoring temporary inputs from the L and R sensors unless a turn is required.

6.2.4 Main Control Loop

In the main control loop, the LCR sensors (Left, Center, Right) provide real-time data on the robot's position relative to the line. The robot calculates the error based on these sensor values and adjusts the motor speeds using the PID controller.

- **ADC Conversion for Sensor Readings**
  The LCR sensors provide analog data, which is converted into digital values by the ADC (Analog-to-Digital Converter). These values represent the robot's position relative to the line on the track.

```
l = ADC_Conversion(3); // Left sensor
c = ADC_Conversion(4); // Center sensor
r = ADC_Conversion(5); // Right sensor
```

- The ADC Conversion function reads the sensor data and assigns the converted digital values to the variables ‘l‘, ‘c‘, and ‘r‘ for left, center, and right sensors, respectively.

- **Error Calculation Based on LCR Sensor Data**
  Once the sensor values (‘l‘, ‘c‘, and ‘r‘) are obtained, the robot calculates the error in its position relative to the line. The error represents how far the robot has deviated from the center of the line. This error is then used in the PID control algorithm to adjust the motor speeds.

```
// Calculate angle error and guess
    l_max = 110;
        c_max = 113;
        r_max = 115;
        l_value = (l - 6) * 100 / l_max + 0.0001;
        c_value = (c - 6) * 100 / c_max + 0.0001;
        r_value = (r - 6) * 100 / r_max + 0.0001;

        if ((l_value >= 1) && (r_value >= 1)) {
            angle_error_value = 0;
            angle_error_prev = 0;
        } else if ((c_value <= 3) && (r_value >= 0.1)) {
            angle_error_value = 200 - r_value;
        } else if ((c_value >= 2) && (r_value >= l_value)) {
            angle_error_value = 100 - c_value;
        } else if ((c_value >= 2) && (l_value >= r_value)) {
            angle_error_value = c_value - 100;
        } else if ((c_value <= 3) && (l_value >= 0.1)) {
            angle_error_value = l_value - 200;
        } else {
            angle_error_value = angle_error_prev * 6;
        }

        angle_error_guess = angle_error_value / 6;
```

- First, the sensor readings are normalized:

```
l_value = (l - 6) * 100 / l_max + 0.0001;
c_value = (c - 6) * 100 / c_max + 0.0001;
r_value = (r - 6) * 100 / r_max + 0.0001;
```

Here, 6 is subtracted from the raw values because the minimum sensor reading is 6 when the sensor detects a white surface, and then each value is scaled to fall within a 0-100 range.
These values are then used to estimate the error in the robot’s angle. Based on specific conditions:

– If both `l_value` and `r_value` are high, the robot moves straight (`angle_error_value = 0`).
– If `c_value` is low and `r_value` is high, the robot turns right.
– If `c_value` is low and `l_value` is high, the robot turns left.
– If none apply, the previous error is amplified.

The angle error guess is computed as:

```
angle_error_guess = angle_error_value / 6;
```

- **PID Control Based on the Error**

  The following part of code implements the PID control loop and motor actuation logic. It calculates the necessary adjustments to the motor speeds based on the proportional, integral, and derivative (PID) components of the angle error.

```
// PID control
kp = 10.0;
ki = 0.0001;
kd = 1;
angle_error_sum += angle_error_guess;
angle_error_diff = angle_error_guess - angle_error_prev;
fb = (kp * angle_error_guess) + (ki * angle_error_sum) + (kd *
    angle_error_diff);

// Actuation
if (fb < 0) {
    l_speed = 255 + fb;
    r_speed = 255;
} else {
    l_speed = 255;
    r_speed = 255 - fb;
}

if (l_speed > 255) l_speed = 255;
if (l_speed < -255) l_speed = -255;
if (r_speed > 255) r_speed = 255;
if (r_speed < -255) r_speed = -255;

// Set motor directions
if (l_speed < 0) left();
else if (r_speed < 0) right();
else forward();

if (l_speed < 0) l_speed = -l_speed;
```

```
if (r_speed < 0) r_speed = -r_speed;

OCR1AL = floor(l_speed);
OCR1BL = floor(r_speed);

angle_error_prev = angle_error_guess;
```

In this code:

- The PID control calculates feedback (`fb`) using the proportional (`kp`), integral (`ki`), and derivative (`kd`) terms, which help in adjusting the robot's motion.
- The motor speeds (`l_speed` and `r_speed`) are adjusted based on the feedback, ensuring that the motors remain within the range of -255 to 255.
- Depending on the speed values, the robot turns left, right, or moves forward.
- Finally, the motor speeds are set using the registers `OCR1AL` and `OCR1BL`.

- After calculating the correction value (`fb`), the motor speeds (`l_speed` and `r_speed`) are adjusted. If the robot is deviating to the left (negative error), the left motor is slowed down, and the right motor is sped up. If the robot is deviating to the right (positive error), the right motor is slowed down, and the left motor is sped up.

- Thus, This code implements a PID controller to adjust the robot's motor speeds based on real-time feedback from the sensors. By properly tuning the constants K_p, K_i, and K_d, the robot can accurately follow the track, correcting deviations dynamically. The combination of LCR sensor input, PID control, and motor actuation ensures that the robot stays aligned on the track and responds to changes in its environment, including intersections, curves, and straight paths, as shown in Figure 1.

7 Results

8 Observations and Inferences

- **Sensor Readings (LCR)**: The Left, Center, and Right (LCR) sensors effectively detected the line under standard lighting conditions. How-

Parameter	**Value Range**
K_p	10
K_i	0.0001
K_d	1
Lap Time	22 seconds

ever, variations in lighting or track surface affected sensor accuracy.

- **Proportional Control (K_p)**: Increasing K_p resulted in quicker responses to deviations from the line, but overly large values caused the robot to oscillate, especially around curves and intersections.
- **Integral Control (K_i)**: The integral term helped eliminate minor, accumulated errors over time. Low values of K_i resulted in steady motion, but high values caused the robot to overcorrect on straight paths.
- **Derivative Control (K_d)**: The derivative term was effective in preventing overshooting, particularly on sharp curves. Higher K_d values reduced oscillations but slowed the robot's response to sudden changes.
- **Lap Time**: The lap time improved as the PID constants were tuned more precisely. However, erratic behavior was observed when the constants were not optimized.

9 Problems Faced and solutions

9.1 Problems

Several challenges were encountered:

- **Sensor Inconsistency**: The LCR sensors occasionally provided inaccurate readings due to changes in ambient lighting or reflections from the track surface. This affected the accuracy of the robot's alignment with the line.
- **Over-correction with High K_p**: A high K_p value caused the robot to oscillate frequently around the line, leading to erratic motion.
- **Integral Windup (High K_i)**: When K_i was too high, the robot accumulated error too quickly, causing it to overshoot or take too long to stabilize.

- **Slow Response with High K_d**: While K_d helped reduce oscillations, a high K_d value made the robot sluggish in responding to sharp turns or sudden changes in the track.
- **Lap Time Variability**: Lap times varied significantly depending on the tuning of PID constants. Poor tuning resulted in longer lap times due to excessive corrections or oscillations.

9.2 Solutions

The following solutions were implemented to address the problems faced:

- **Sensor Inconsistency Solution**: The LCR sensors were recalibrated under different lighting conditions to improve accuracy. This involved adjusting the sensor threshold values to ensure proper detection of the line across varying track conditions.
- **Over-correction with High K_p**: The K_p value was reduced gradually to find a balance between responsiveness and stability. This eliminated oscillations while maintaining quick corrections for deviations.
- **Integral Windup (High K_i) Solution**: The K_i value was reduced to ensure that small accumulated errors were corrected without causing instability.
- **Slow Response with High K_d**: The K_d value was reduced slightly to balance responsiveness and damping. This allowed the robot to react more quickly to sharp turns while still preventing oscillations.
- **Lap Time Variability Solution**: By iteratively tuning the K_p, K_i, and K_d values, the robot's lap time was minimized, achieving a balance between speed and stability.

10 Conclusion

The implementation of PID control for the two-wheeled robot faced challenges in sensor calibration, motor control, and PID tuning. Issues like sensor inconsistency and over-correction were resolved by calibrating sensors and iteratively adjusting the PID constants. Optimizing K_p, K_i, and K_d improved robot motion and response times. Environmental factors were mitigated by modifying the track and shielding sensors, resulting in more reliable performance and highlighting the importance of tuning in autonomous systems.