EE324: Controls System Lab

Experiment 3: Line Follower

Team Members:
Aman Rishal C H (22B3914)
Hrishikesh S (22B4217)
Prajwal Nayak (22B4246)

Thursday Batch - Table 14

October 17, 2024

1 Aim

The aim of this experiment is to design and implement a PID controller for the Spark V robot to follow a continuous track using the IR sensors provided on the robot.

2 Objective of the Experiment

- To trace the designed track within 30 seconds.

3 Control Algorithm

In this section, we explain the control algorithm that the robot uses to follow the line on the track. The robot continuously reads sensor data, calculates how far it has drifted from the line (error), and makes corrections to stay on track. The algorithm uses a Proportional-Derivative (PD) controller to adjust the speed of the motors, allowing the robot to turn smoothly and avoid zigzagging.

3.1 Reading Sensor Data

The robot uses three infrared (IR) sensors:

- **Left sensor**: Detects if the robot is drifting to the right.
- **Right sensor**: Detects if the robot is drifting to the left.
- **Center sensor**: Checks if the robot is directly over the line.

The robot reads the sensor values using the following code:

```
left_sensor = ADC_Conversion(3);
right_sensor = ADC_Conversion(5);
center_sensor = ADC_Conversion(4);
```

This converts the analog signals from the sensors into digital values that the robot can process.

3.2 Calculating Error

The robot calculates the **error**, which tells it how far off it is from the center of the line. The error is calculated by comparing the left and right sensor readings:

```
right_left = right_sensor - left_sensor;
```

If the error is negative, the robot has drifted to the right and needs to turn left. If the error is positive, the robot has drifted to the left and needs to turn right.

3.3 Proportional-Derivative (PD) Control

The robot uses PD control to determine how strongly it should correct its movement. The PD control system combines two parts:

- **Proportional (P) control**: Corrects the robot's movement based on how far off it is. The larger the error, the stronger the correction.
- **Derivative (D) control**: Adjusts the correction based on how quickly the error is changing. This smooths out the robot's movement and prevents it from overcorrecting.

The PD control is implemented in the code as follows:

```
float pid_drive = Kp * right_left + Kd * (right_left - old_right_left);
```

This calculates the correction value (`pid_drive`) based on the error (`right_left`) and the change in error (`right_left - old_right_left`).

3.4 Motor Control

Based on the calculated `pid_drive`, the robot adjusts the speed of its motors. If the error is small, the robot moves forward. If the error is large, the robot turns left or right.

```
if (abs(pid_drive) < minThres) {
    forward();
} else if(center_sensor < 50) {
    if (pid_drive < 0) {
        soft_left();
    } else {
        soft_right();
    }
}
```

This code decides the robot's movement: - If the error is small, the robot moves forward. - If the center sensor detects that the robot is off the line, the robot makes a soft turn in the direction needed to get back on track.

The motor speed is adjusted based on the `pid_drive` value:

```
velocity(pid > v0 * 2 ? pid : v0 * 2, pid > v0 * 2 ? pid : v0 * 2);
```

This adjusts the speed of both the left and right motors based on how far off the robot is from the line.

The control algorithm continuously checks the sensor values, calculates the error, and adjusts the motor speeds to keep the robot on track. By combining proportional and derivative control, the robot can follow the line smoothly without overshooting or zigzagging. This system ensures that the robot stays on the line and can successfully navigate the track.

4 Code

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

unsigned char ADC_Conversion(unsigned char);
unsigned char ADC_Value;
unsigned char l = 0;
unsigned char c = 0;
unsigned char r = 0;
unsigned char PortBRestore = 0;
const float P = 7; // Changed to float
const float D = 1; // Changed to float

void motion_pin_config(void) {
DDRB = DDRB | 0x0F;   //set direction of the PORTB3 to PORTB0 pins as output
PORTB = PORTB & 0xF0; // set initial value of the PORTB3 to PORTB0 pins to logic 0
DDRD = DDRD | 0x30;   //Setting PD4 and PD5 pins as output for PWM generation
PORTD = PORTD | 0x30; //PD4 and PD5 pins are for velocity control using PWM
}

void velocity(unsigned char left_motor, unsigned char right_motor) {
OCR1AL = left_motor;
OCR1BL = right_motor;
}

//Function used for setting motor's direction
void motion_set(unsigned char Direction) {
unsigned char PortBRestore = 0;

Direction &= 0x0F; // removing upper nibbel as it is not needed
PortBRestore = PORTB; // reading the PORTB's original status
PortBRestore &= 0xF0; // setting lower direction nibbel to 0
PortBRestore |= Direction; // adding lower nibbel for direction command and restorin
PORTB = PortBRestore; // setting the command to the port
}

void forward(void) {
motion_set(0x06); //both wheels forward
}

void back(void) {
motion_set(0x09); //both wheels backward
}

void left(void) {
```

```
motion_set(0x05); //Left wheel backward, Right wheel forward
}

void right(void) {
motion_set(0x0A); //Left wheel forward, Right wheel backward
}

void soft_left(void) {
motion_set(0x04); //Left wheel stationary, Right wheel forward
}

void soft_right(void) {
motion_set(0x02); //Left wheel forward, Right wheel is stationary
}

void soft_left_2(void) {
motion_set(0x01); //Left wheel backward, right wheel stationary
}

void soft_right_2(void) {
motion_set(0x08); //Left wheel stationary, Right wheel backward
}

void hard_stop(void) {
motion_set(0x00); //hard stop(stop suddenly)
}

void soft_stop(void) {
motion_set(0x0F); //soft stop(stops slowly)
}

//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 = 1 1 0
}

void init_devices(void) {
cli(); //Clears the global interrupts
port_init();
adc_init();
sei(); //Enables the global interrupts
}

//Function to configure LCD port
void lcd_port_config(void) {
```

```
DDRC = DDRC | 0xF7; //all the LCD pin's direction set as output
PORTC = PORTC & 0x80; // all the LCD pins are set to logic 0 except PORTC 7
}

//ADC pin configuration
void adc_pin_config(void) {
DDRA = 0x00;   //set PORTF direction as input
PORTA = 0x00;  //set PORTF pins floating
}

//Function to Initialize PORTS
void port_init() {
lcd_port_config();
adc_pin_config();
motion_pin_config();
}

//TIMER1 initialize - prescale:64
// WGM: 5) PWM 8bit fast, TOP=0x00FF
// desired value: 450Hz
// actual value: 450.000Hz (0.0%)
void timer1_init(void) {
TCCR1B = 0x00; //stop
TCNT1H = 0xFF; //setup
TCNT1L = 0x01;
OCR1AH = 0x00;
OCR1AL = 0xFF;
OCR1BH = 0x00;
OCR1BL = 0xFF;
ICR1H  = 0x00;
ICR1L  = 0xFF;
TCCR1A = 0xA1;
TCCR1B = 0x0D; //start Timer
}

//This Function accepts the Channel Number and returns the corresponding Analog Val
unsigned char ADC_Conversion(unsigned char Ch) {
unsigned char a;
Ch = Ch & 0x07;
ADMUX = 0x20 | Ch;
ADCSRA = ADCSRA | 0x40; //Set start conversion bit
while ((ADCSRA & 0x10) == 0); //Wait for ADC conversion to complete
a = ADCH;
ADCSRA = ADCSRA | 0x10; //clear ADIF (ADC Interrupt Flag) by writing 1 to it
return a;
}

int main() {
```

```
timer1_init();
init_devices();
lcd_set_4bit();
lcd_init();

unsigned char left_sensor, right_sensor, center_sensor;
float Kp = 100, Kd = 5, minThres = 3, offset = 5;
int right_left = 0, old_right_left = 0;
int v0 = 100;

velocity(64, 64);

while (1) {
left_sensor = ADC_Conversion(3);
right_sensor = ADC_Conversion(5);
center_sensor = ADC_Conversion(4);

lcd_print(1, 1, left_sensor, 3);
lcd_print(1, 5, center_sensor, 3);
lcd_print(1, 9, right_sensor, 3);

forward();

float pid = 0.5 * center_sensor;

right_left = right_sensor - left_sensor;

float pid_drive = Kp * right_left + Kd * (right_left - old_right_left);
lcd_print(2, 1, pid_drive, 3);

velocity(pid > v0 * 2 ? pid : v0 * 2, pid > v0 * 2 ? pid : v0 * 2);

if (abs(pid_drive) < minThres) {
forward();
} else if (center_sensor < 50) {
if (pid_drive < 0) {
if (abs(pid_drive) < offset)
soft_left();
else
left();
velocity(pid > v0 ? pid : v0, v0);
} else {
if (abs(pid_drive) < offset)
soft_right();
else
right();
velocity(v0, pid > v0 ? pid : v0);
}
}

old_right_left = right_left;
_delay_ms(10);
}
}
```

5 Challenges Faced

Initially, we faced challenges in determining the correct threshold values for the left and right sensors to initiate a turn. After multiple attempts, we were able to identify an appropriate threshold that worked well for the robot's behavior.

In our first approach, we used a proportional controller, where the robot's speed was directly related to the center sensor's reading. However, this design caused issues during turns, as the center sensor value would drop too low, resulting in the robot stopping. To fix this, we introduced a minimum speed limit to ensure the robot continued moving, even when the controller calculated a lower speed, and this solved the problem.

Later, we encountered another issue with sharp turns on the track. The left and right sensor values were not changing quickly enough, making it hard for the robot to detect the turns. Additionally, the robot's speed was too high for it to react in time. To address this, we set a lower speed for turns, allowing the robot to slow down and navigate the turns more effectively.

6 Results

The control coefficients used by us are: $K_p = 2$, $K_d = 0.5$, $K_i = 0$ The traversal time averaged across 4 runs was 27 seconds, which is within the time constraint of 30 seconds.