EE324: Control Systems Lab

Experiment 3: Line Follower Bot

Thursday batch - Table 13

Jatin Kumar (22B3922)
Aayush Kumar (22B0769)
Archisman Bhattacharjee (22B2405)

October 23, 2024

1 Objective

The objective of this experiment is to design and implement a PID controller for the Spark V robot to follow a continuous track using the provided IR sensors. The system should achieve this within the constraint of completing the track in under 30 seconds. The controller needs to effectively manage the robot's movements and turning speed while ensuring it stays on course.

2 Control Algorithm

2.1 Introduction

In this Experiment, we designed and implemented a control system for a Spark V Robot using an AVR microcontroller (ATmega16) and sensors. The system enables the robot to navigate based on sensor inputs and react dynamically using a Proportional-Derivative (PD) control algorithm. The robot adjusts its direction and velocity in response to sensor feedback, ensuring smooth and efficient navigation.

2.2 Hardware Setup

The hardware configuration involved the following components:

- **Microcontroller:** ATmega16, which controls the motors and reads sensor data.
- **Sensors:** Three infrared sensors to detect obstacles or follow a path—left, right, and center sensors.
- **Motor Control:** PWM (Pulse Width Modulation) used to adjust motor speed through the microcontroller.
- **LCD Display:** Displays sensor values and system parameters.

The main components of the motion system include:

1. **Motor and Motion Control:** Direction and speed control using PORTB (for direction) and PORTD (for PWM).
2. **Sensors:** Sensor data is read via the ADC (Analog to Digital Converter) on PORTA, which provides inputs from the left, right, and center sensors.

2.3 Control Algorithm

We implemented a **Proportional-Derivative (PD) Control Algorithm** to adjust the robot's motion based on sensor inputs. The control method dynamically alters the robot's direction and velocity to minimize the error between the desired and actual positions.

2.3.1 Sensor Feedback

The three sensors provide real-time data to the microcontroller:

- **Left Sensor:** Detects objects or lines on the left side.
- **Right Sensor:** Detects objects or lines on the right side.
- **Center Sensor:** Detects the robot's alignment relative to a line or obstacle.

2.3.2 PD Controller

The PD control algorithm computes the required adjustments based on the difference in sensor readings between the left and right sides. The controller aims to balance the robot by minimizing this error.

The PD formula used is:

$$\text{pid\_drive} = K_p \cdot (\text{right\_left}) + K_d \cdot (\text{right\_left} - \text{old\_right\_left})$$

where:

- K_p: Proportional gain constant (set to 8).
- K_d: Derivative gain constant (set to 55).
- **right_left**: The difference between the right and left sensor readings.
- **old_right_left**: The previous sensor difference (used to calculate the derivative term).

2.3.3 Threshold Calculation

The threshold for decision-making is adjusted dynamically based on the sign of `right_left`.

- When `right_left` is positive (i.e., the robot needs to turn right), the threshold is calculated as:

$$\text{thresh} = \text{primary\_offset} - \text{pid\_drive}$$

- When `right_left` is negative (i.e., the robot needs to turn left), the threshold becomes:

$$\text{thresh} = -(\text{primary\_offset} + \text{pid\_drive})$$

Here, the `primary_offset` is set to 128, and `pid_drive` is adjusted based on the PD controller output.

2.3.4 Motion Control

The calculated `pid_drive` and threshold values are used to adjust the motor velocities. Depending on the error:

- For small corrections (error below threshold), we use **soft turns** (soft left or soft right).
- For larger deviations, **sharp turns** (left or right) are performed by adjusting the velocities of the wheels in opposite directions.

Motor control functions (`left()`, `right()`, `soft_left()`, `soft_right()`, etc.) are called with the velocity adjustments, ensuring smooth navigation.

2.4 Conclusion

In this experiment we successfully implemented a PD control algorithm to manage real-time robot navigation. The robot dynamically adjusts its direction and speed based on sensor feedback, using proportional and derivative control to ensure precise movement and smooth transitions, avoiding obstacles effectively.

3 Challenges Faced and Solutions

3.1 Implementation of PID Control

One of the main challenges we faced was the incorrect application of the PID control algorithm. Despite passing a zero velocity from the PID function, the robot continued to move at a default speed, indicating that the PID loop wasn't fully integrated with the hardware.

Solution: This issue was resolved by properly initializing `Timer1` at the appropriate point in the code. This allowed the robot to effectively run through the PID control algorithm, particularly during turns, where adjusting speed and direction was crucial.

3.2 Sensor Threshold Calibration

Another challenge was accurately calibrating the sensor thresholds for detecting black and white surfaces. Initially, all sensors detected white for values below 005 and black above it. However, discrepancies in sensor readings led the robot to make sudden stops or unexpected turns.

Solution: We observed that the right sensor detected white at a threshold slightly higher than the other sensors. To correct this, we calibrated the thresholds by reducing the left and center sensor values by 5, and the right sensor by 6. This adjustment improved the uniformity of sensor detection and minimized erratic behavior.

3.3 Handling Track Intersections

When encountering intersections on the track, the robot would mistakenly trigger a complete stop, as all sensors detected white, which was interpreted by the control logic as being off-track.

Solution: To resolve this, we introduced a brief delay before executing the hard stop logic, ensuring the robot only halted when truly off the track. This prevented unnecessary stops at intersections and allowed smoother navigation through the course.

3.4 Fine-tuning Control Parameters (K_p, K_d)

Achieving smooth tracking and handling soft turns required fine-tuning the proportional gain (K_p). If K_p was set too low, the robot struggled to follow the line, while a higher K_p caused aggressive corrections, often leading to U-turns.

Solution: After extensive testing, we found that a K_p value of 8 provided optimal tracking performance, offering a good balance between stability and responsiveness. We also experimented with adding a derivative term (K_d) to smooth out the control during turns, and found that $K_d = 55$ improved turn handling, enhancing the overall performance of the control system.

4 Code

The following is the code we flashed onto the Bot through ISP Programmer.

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

#define RS 0
#define RW 1
#define EN 2
#define lcd_port PORTC
#define sbit(reg,bit)    reg |= (1<<bit)
#define cbit(reg,bit)    reg &= ~(1<<bit)
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 my_velocity(float val,float scale, int offset, float drive)
{
  unsigned char velocity = scale*drive + offset;

  if(val>0)
  {
    if(val<scale+offset)
    soft_right(velocity,0);
    else
    right(velocity,-velocity);
  }
  else
  {
    if(val>-(scale+offset))
    soft_left(0,velocity);
    else
    left(-velocity,velocity);
  }
}

unsigned char min(int a, int b)
{
  if( a> b)
  {
    return (unsigned char)b;
  }

  return (unsigned char)a;
}

int abs(int a)
{
  if( a > 0)
  return a;
  else
  return -a;
}

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

//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
      restoring the PORTB status
  PORTB = PortBRestore;        // setting the command to the port
}

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

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

void left (unsigned char v1, unsigned char v2)             //Left wheel backward, Right
    wheel forward
{
  motion_set(0x05);
}

void right (unsigned char v1, unsigned char v2)            //Left wheel forward, Right
    wheel backward
{
  motion_set(0x0A);
}

void soft_left (unsigned char v1, unsigned char v2)        //Left wheel stationary,
    Right wheel forward
{
  motion_set(0x04);
}

void soft_right (unsigned char v1, unsigned char v2)       //Left wheel forward, Right
    wheel is stationary
{
  motion_set(0x02);
}

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

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

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

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

void init_ports();
void lcd_set_4bit();
void lcd_init();
void lcd_wr_command(unsigned char);
void lcd_wr_char(char);
void lcd_home();
void lcd_cursor(char, char);
void lcd_print(char, char, unsigned int, int);
void lcd_string(char*);
```

```
void init_devices (void)
{
  cli(); //Clears the global interrupts
  port_init();
  adc_init();
  lcd_init();
  lcd_set_4bit();

  // timer1_init();
  sei(); //Enables the global interrupts
}
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
}

void lcd_set_4bit()
{
  _delay_ms(1);

  cbit(lcd_port,RS); //RS=0 --- Command Input
  cbit(lcd_port,RW); //RW=0 --- Writing to LCD
  lcd_port = 0x30; //Sending 3 in the upper nibble
  sbit(lcd_port,EN); //Set Enable Pin
  _delay_ms(5); //delay
  cbit(lcd_port,EN); //Clear Enable Pin
  _delay_ms(1);

  cbit(lcd_port,RS); //RS=0 --- Command Input
  cbit(lcd_port,RW); //RW=0 --- Writing to LCD
  lcd_port = 0x30; //Sending 3 in the upper nibble
  sbit(lcd_port,EN); //Set Enable Pin
  _delay_ms(5); //delay
  cbit(lcd_port,EN); //Clear Enable Pin

  _delay_ms(1);

  cbit(lcd_port,RS); //RS=0 --- Command Input
  cbit(lcd_port,RW); //RW=0 --- Writing to LCD
  lcd_port = 0x30; //Sending 3 in the upper nibble
  sbit(lcd_port,EN); //Set Enable Pin
  _delay_ms(5); //delay
  cbit(lcd_port,EN); //Clear Enable Pin

  _delay_ms(1);

  cbit(lcd_port,RS); //RS=0 --- Command Input
  cbit(lcd_port,RW); //RW=0 --- Writing to LCD
  lcd_port = 0x20; //Sending 2 in the upper nibble to initialize LCD 4-bit mode
  sbit(lcd_port,EN); //Set Enable Pin
  _delay_ms(5); //delay
  cbit(lcd_port,EN); //Clear Enable Pin
}

//Function to Initialize LCD
void lcd_init()
{
  _delay_ms(1);
  lcd_wr_command(0x28); //4-bit mode and 5x8 dot character font
  lcd_wr_command(0x01); //Clear LCD display
  lcd_wr_command(0x06); //Auto increment cursor position
  lcd_wr_command(0x0E); //Turn on LCD and cursor
  lcd_wr_command(0x80); //Set cursor position
}

//Function to write command on LCD
void lcd_wr_command(unsigned char cmd)
{
  unsigned char temp;
  temp = cmd;
  temp = temp & 0xF0;
```

```c
  lcd_port &= 0x0F;
  lcd_port |= temp;
  cbit(lcd_port,RS);
  cbit(lcd_port,RW);
  sbit(lcd_port,EN);
  _delay_ms(5);
  cbit(lcd_port,EN);

  cmd = cmd & 0x0F;
  cmd = cmd<<4;
  lcd_port &= 0x0F;
  lcd_port |= cmd;
  cbit(lcd_port,RS);
  cbit(lcd_port,RW);
  sbit(lcd_port,EN);
  _delay_ms(5);
  cbit(lcd_port,EN);
}

//Function to write data on LCD
void lcd_wr_char(char letter)
{
  char temp;

  temp = letter;
  temp = (temp & 0xF0);
  lcd_port &= 0x0F;
  lcd_port |= temp;
  sbit(lcd_port,RS);
  cbit(lcd_port,RW);
  sbit(lcd_port,EN);
  _delay_ms(5);
  cbit(lcd_port,EN);

  letter = letter & 0x0F;
  letter = letter<<4;
  lcd_port &= 0x0F;
  lcd_port |= letter;
  sbit(lcd_port,RS);
  cbit(lcd_port,RW);
  sbit(lcd_port,EN);
  _delay_ms(5);
  cbit(lcd_port,EN);
}

void lcd_home()
{
  lcd_wr_command(0x80);
}

void lcd_string(char *str)
{
  while(*str != '\0')
  {
    lcd_wr_char(*str);
    str++;
  }
}

//Position the LCD cursor at "row", "column"
void lcd_cursor (char row, char column)
{
  switch (row) {
    case 1: lcd_wr_command (0x80 + column - 1); break;
    case 2: lcd_wr_command (0xc0 + column - 1); break;
    case 3: lcd_wr_command (0x94 + column - 1); break;
    case 4: lcd_wr_command (0xd4 + column - 1); break;
    default: break;
  }
}
unsigned int temp;
unsigned int unit;
unsigned int tens;
```

```
unsigned int hundred;
unsigned int thousand;
unsigned int million;
// Function to print any input value up to the desired digit on LCD
void lcd_print (char row, char coloumn, unsigned int value, int digits)
{
  unsigned char flag=0;
  if(row==0||coloumn==0)
  {
    lcd_home();
  }
  else
  {
    lcd_cursor(row,coloumn);
  }
  if(digits==5 || flag==1)
  {
    million=value/10000+48;
    lcd_wr_char(million);
    flag=1;
  }
  if(digits==4 || flag==1)
  {
    temp = value/1000;
    thousand = temp%10 + 48;
    lcd_wr_char(thousand);
    flag=1;
  }
  if(digits==3 || flag==1)
  {
    temp = value/100;
    hundred = temp%10 + 48;
    lcd_wr_char(hundred);
    flag=1;
  }
  if(digits==2 || flag==1)
  {
    temp = value/10;
    tens = temp%10 + 48;
    lcd_wr_char(tens);
    flag=1;
  }
  if(digits==1 || flag==1)
  {
    unit = value%10 + 48;
    lcd_wr_char(unit);
  }
  if(digits>5)
  {
    lcd_wr_char('E');
  }
}

//ADC pin configuration
void adc_pin_config (void)
{

  DDRA = 0x00; //set PORTA direction as input
  PORTA = 0x00; //set PORTA pins floating
}

//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
}


//This Function accepts the Channel Number and returns the corresponding Analog Value
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
  // ADCSRB = 0x00;
  return a;
}

int main()
{
  init_devices();
  unsigned char left_sensor   =6;
  unsigned char right_sensor  =6;
  unsigned char center_sensor =6;
  unsigned char old_left_center=0;
  unsigned char old_right_center=0;

  float thresh                =0;
  float Kp                    =8;

  int left_center             =0;
  int right_center            =0;
  int Kd                      =55;
  int right_left              =0;
  int old_right_left          =0;
  int primary_offset          =128;
  int secondary_offset        =26;

  while(1)
  {

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

    left_center  = left_sensor - center_sensor;
    right_center = right_sensor - center_sensor;
    right_left   = right_sensor - left_sensor;

    unsigned int left_sensor_disp = left_sensor;
    unsigned int right_sensor_disp = right_sensor;
    unsigned int center_sensor_disp = center_sensor;


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

    if(right_left>0)
    thresh = (primary_offset - pid_drive);
    else
    thresh = -(primary_offset + pid_drive);

    if(right_left>(-thresh) && right_left<(thresh))
    {
      forward();
    }
    else if(right_left<(-thresh) || right_left>(thresh))
    {
      if(right_left>0)
      {
        my_velocity(right_left,thresh,secondary_offset,pid_drive);
      }
      else
      {
        my_velocity(right_left,thresh,secondary_offset,pid_drive);
      }
    }
    else
    {
      forward();
```

```
    }
    old_right_left=right_left;
  }
}
```

Listing 1: Code for Line Follower Bot

5 Results

The final parameters for the bot were as follows:

Parameter	Value
Time taken	27 seconds
Kp	8
Kd	55
Ki	0

Table 1: Parameters for the bot

The bot successfully completed the track within the desired time of 30 seconds, demonstrating smooth and stable turns with minimal jitter.

6 Observations and Inferences

- After addressing the initial issues, the robot performed reliably, achieving smooth tracking and successfully navigating turns. The proportional control (K_p) played a crucial role in maintaining stability during straight segments and responding to directional changes, while the derivative term (K_d) helped dampen any overshoot during sharp turns.
- One key observation was the importance of precise sensor calibration. By fine-tuning the sensor thresholds, the robot could consistently differentiate between black and white surfaces, minimizing erratic behavior. This adjustment significantly improved its overall navigation, especially at intersections and during soft turns.
- Through iterative tuning, we found that setting $K_p = 8$ and $K_d = 55$ provided optimal control, allowing the robot to follow the track smoothly without aggressive oscillations. The inclusion of a small derivative gain (K_d) helped improve the robot's performance, particularly during quick directional changes.
- As a result, the robot met all performance requirements, completing the track with a total time of 27 seconds. These adjustments ensured reliable behavior and consistent performance across various track configurations.

7 Link to Results

Below is the QR code for the demonstration video. In case the QR code does not work, you can also access the video through the link provided below.

Video documentation of the project can be found at the following link:
https://drive.google.com/drive/folders/1RbfcOecZHOhfqq1LEMUFcQBp4f1rT9HZ?usp=sharing