EE324: Control Systems Lab
Experiment 3: Line Follower Robot
Group 1 - Thursday

Harsh S Roniyar 22B3942 | Pranav Prakash 22B3945 | Aman Verma 22B3929

October 1, 2024

1 Objective

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.

The specific objectives were:

- To trace the track within 30 seconds.

2 Control Algorithm

The control algorithm used in this experiment is Proportional-Integral-Derivative (PID) controller.
The output of the PID controller is given by the equation -

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\mathrm{d}\tau + K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{1}$$

where $e(t)$ is the error signal, K_p, K_i, and K_d are the proportional, integral, and derivative gains respectively.
The PID controller is implemented in the AVR microcontroller to control the motion of the robot.

3 Implementation

Some important code snippets showing the implementation for the line follower robot SPARK V.
We make use of the custom function `speed_control()`, to update the motion as well as PWM of the motors on the basis of two control signals provided for each the left and right motor. The PORTB register is accordingly varied to adjust the motion.

Code for `speed_control()` function

```
void speed_control(int left_motor, int right_motor){
    unsigned char l_val, r_val;
    unsigned char newPORTB = PORTB;
    newPORTB &= 0xF0;

    if(left_motor < 0){
      //take absolute value
      l_val = -left_motor;
      //perform soft left 2
      newPORTB |= 0x01;
    }
    else{
      l_val = left_motor;
      //perform soft right
      newPORTB |= 0x02;
    }

    if(right_motor < 0){
      //take absolute value
      r_val = -right_motor;
      //perform soft right 2
      newPORTB |= 0x08;
    }
    else{
      r_val = right_motor;
      //perform soft left
      newPORTB |= 0x04;
    }

    l_val = l_val > 255 ? 255 : l_val;
    r_val = r_val > 255 ? 255 : r_val;
    PORTB = newPORTB;
    velocity(l_val, r_val);
}
```

We also change the PWM using the custom velocity() function to adjust the speed of left and right motors using PWM by setting values ranging from 0 to 255 to the registers OCR1AL and OCR1BL.

Code for PWM `velocity()` function

```
void velocity(unsigned char left_motor, unsigned char right_motor){
    //PWM control
    OCR1AL = left_motor;
    OCR1BL = right_motor;
}
```

To decide on how to adjust the left and right sensor values, we create a signal called rotate which will depend on control_sig, which we keep updating every iteration from its prev value. We have bound this in between -1 to 1 for the sake of simplicity. Division by (cc + offset) is used to get an appropriately scaled version of control signal as control_sig in itself can take a very large range of values.

```
//set the value of rotate
double rotate = prev_rotate + (control_sig/(cc + offset));

//check if its in range of -1 to 1
rotate = (rotate < -1) ? -1 : rotate;
rotate = (rotate > 1) ? 1 : rotate;
```

Now using the rotate and translate signals produced we need to correct the PWM values to the motors accordingly. For this we make use of the custom function correction(). We now check the value of the

control signal rotate. If it is negative we increase the left control signal to be more than right control signal by a value of 2*translate*rotate, if it is positive we make right control signal less than left by a value of 2*translate*rotate and accordingly adjust the motion of bot.

Code for PWM `correction()` function

```
void correction(double translate, double rotate){
    int l_val, r_val;

    if(rotate < 0){
      l_val = translate + 2*translate*rotate;
      r_val = translate;
    }
    else{
      l_val = translate;
      r_val = translate - 2*translate*rotate;
    }
    speed_control(l_val, r_val);
}
```

Code for `main()` function

```
int main(void){
    init_devices();
    lcd_set_4bit();
    lcd_init();

    // control value constants
    double kp = 1.05;
    double ki = 0.00000028;
    double kd = 0.49;
    double dt = 1;

    double prev_err = 0;
    double prev_rotate = 0;
    double integral = 0;

    while(1){
        double cl, cc, cr;
        double offset = 0.5;

        l=ADC_Conversion(3);
        c=ADC_Conversion(4);
        r=ADC_Conversion(5);

        //Display The Sensor Values
        // lcd_print(1, 1, l, 3);
        // lcd_print(1, 5, c, 3);
        // lcd_print(1, 9, r, 3);

        cl = l;
        cc = c;
        cr = r;

        double average = (cl + cc + cr)/3;
        double err = (cr - cl);
        double control_sig = kp * err + ki * integral + kd * (err - prev_err)/dt;

        integral += err * dt;
        prev_err = err;

        //set the translate
        double translate = 45 + average*0.85;
        //check if its in range of 255
        translate = (translate > 255) ? 255 : translate;

        //set the value of rotate
        double rotate = prev_rotate + (control_sig/(cc + offset));
        //check if its in range of -1 to 1
        rotate = (rotate < -1) ? -1 : rotate;
        rotate = (rotate > 1) ? 1 : rotate;

        if((cl > 50 && cr > 50) && cc < 120) {
            rotate = 0;
            translate /= 8;
            correction(translate, rotate);
            _delay_ms(100);
        }

        correction(translate, rotate);
        prev_rotate = 0.2*rotate;
    }
}
```

4 Challenges Faced and Solutions

Problem 1: **Issue with Initial PID Parameters:** The code did not perform as expected with the initially set values of k_p, k_i, and k_d, resulting in unsatisfactory motion on the track.
Solution: After multiple iterations of tuning these parameters, we were able to achieve smooth turning around the curves.

Problem 2: **Unbounded Control Signals:** Initially, control signals were not bounded, which caused no visible variations even when we changed parameters.
Solution: We solved this by bounding the rotate signal between -1 and 1. Additionally, introducing checks to limit PWM values to 255 resolved the issue.

Problem 3: **Failure on Discontinuous Track Segments:** The system failed to handle discontinuous square blocks in the track.
Solution: By varying the threshold values (e.g., 50, 120) through multiple iterations, we were able to handle this track segment more effectively.

5 Results

The bot successfully completed the track in 26 seconds, consistently staying well under the 30-second mark. After incorporating the PID controller, the bot executed smooth turns throughout the course, as opposed to the sharp and jerky turns observed with the initial thresholding-based approach.
The final PID controller constants used were:

- $k_p = 1.05$
- $k_i = 2.8 \times 10^{-7}$
- $k_d = 0.49$

6 Observations and Inference

- The implementation of the PID controller significantly improved the bot's ability to handle curves on the track, leading to smoother turns compared to the initial thresholding method.
- Fine-tuning of the PID constants (*kp*, *ki*, *kd*) allowed for an optimal balance between stability and responsiveness, leading to faster track completion times.
- The marginal contribution of *ki* (2.8×10^{-7}) indicates that integral control had minimal impact, likely due to the relatively short duration of the course and the real-time nature of the control adjustments.
- The overall improvement in performance demonstrates that the PID approach is far more efficient and adaptable for handling dynamic changes in the track.