Experiment 2

Line Follower Bot

Shounak Das, 21D070068
Prajapati Kishan K, 21D070048
Vinay Sutar, 21D070078

September 2023

Contents

1 Aim

Design and implement a PID controller for the Spark V robot, utilizing its built-in IR sensors to enable it to accurately track a continuous black path. The goal is to fine-tune the controller's parameters to ensure that the robot consistently follows the path within a 30-second timeframe.

2 Introduction

- The robot is built around the ATMEGA16 microcontroller. It is programmed using the AVR Programmer tool in Embedded C/C++.
- One of the standout features of Spark V is its ability to move in eight different directions, all of which can be easily adjusted by modifying the values of the relevant registers. These 8 are Forward, Reverse, Right, Left, Soft Right, Soft Left, Soft Right 2 & Soft Left 2.
- Furthermore, Spark V offers precise speed control using Pulse Width Modulation (PWM). This feature allows users to fine-tune the robot's speed, enabling it to move at different velocities for various tasks and applications.
- In terms of sensing capabilities, Spark V is equipped with three Infrared (IR) sensors located under its panel. Each sensor provides readings ranging from 0 to 255, which correspond to the reflectivity of the surface it is detecting. These sensors enhance the robot's ability to navigate its environment by providing data on nearby objects and obstacles.

3 Method

- Initially the bot needed to be tuned to read the correct values from the 3 Infrared sensors attached at the base corresponding to the left, right and center directions.
- After the bot read correct sensors values for the white and black regions on the board, these values were used to implement the PID algorithm for follwing the line.

4 Getting Started

4.1 Track

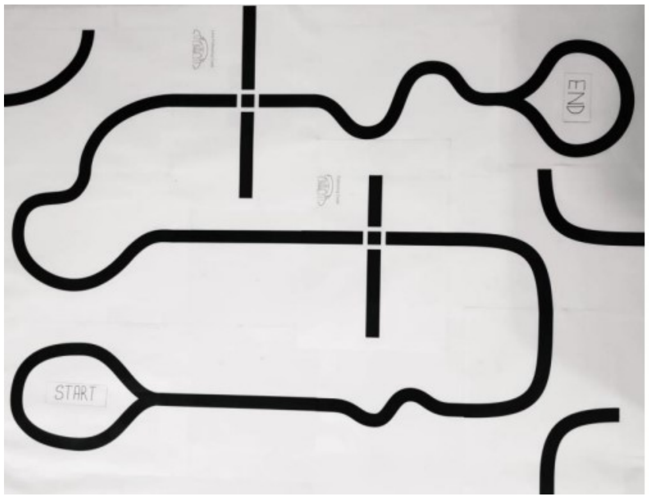

Figure 1: Design of the track to be traced

4.2 Components

1. *Spark V bot*
2. *Spark V charger*
3. *ISP programmer*
4. *A-B cable*

5 Experiment

5.1 Sensor Input

We used following code for reading the left, centre & right sensor values :

```
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);
```

- The code interfaces with three analog sensors connected to pins 3 (left sensor), 4 (centre sensor), and 5 (right sensor) of PORT A. The `ADC_Conversion` function reads analog values from these sensors.

5.2 PID Implementation for Motion Control

We used following code to implement PID in for direction & velocity control:

```
void velocity (unsigned char left_motor, unsigned char right_motor)
{
OCR1AL = left_motor;
OCR1BL = right_motor;
}
//Main Function
int main(void)
{
init_devices();

lcd_set_4bit();
lcd_init();

double kp = 0.9;
double kd = 0.05;
double ki = 0.0004;

double pre_err=0;
double err=0;
double int_err=0;
double err_der;
double control;
int count = 0;

while(1)
{
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);
```

```
err = r-l;
err_der = err-pre_err;
int_err += err;
control = kp*err+kd*err_der + ki*int_err;
pre_err = err;

double k;
if(control>100){
k = 100;
}
else if(control<-100){
k = 100;
}
else{
k = abs(control);
}
if(control > 1.1){
if(control<=100*kp){
velocity(k,k);
right();
}
else{
velocity(k, k);
soft_right();
}
}
else if(control< -1.1){
if(control>=-100*kp){
velocity(k,k);
left();
}
else{
velocity(k, k);
soft_left();
}
}
else{
velocity(150, 150);
forward();
int_err  = 0;
}
```

```
}
}
```

Other code snippets that like those of LCD interfacing, motion_set, ADC interfacing etc. were given previously and are not added here.

- The code implements a **PID (Proportional-Integral-Derivative)** controller to control the robot's movement based on sensor inputs. The error at any instant of time is the **difference in sensor values** corresponding to the left and right directions.
- The simple idea behind this selection is that if a right turn arrives then the sensor value corresponding to right IR sensor if higher than the left one. Thus the error is **right value - left value** > 0 thus the value of velocity is adjust to turn right at that position.
- The PID parameters are defined as follows:

  Proportional Error:, as the term suggests this response is proportional to the current error, i.e. it multiplies the error(e(t)) with a constant also known as the Proportional Gain (K_p). The proportional term is responsible for providing an **immediate** response to the current error, which is the difference between the robot's current position (off the line) and the desired position (on the line). if the K_p gain is too **high**, it can lead to oscillations and instability as the robot might **overshoot** the line.

  Integral Error, the integral term focuses on the accumulation of past errors and aims to eliminate any steady-state errors that may persist. It basically adds up the past errors and multiplies them by a constant (K_i) to produce a correction term. This correction would help the robot deal with biases like uneven motor response and wheel imperfections, it basically would help us **counter small and consistent errors**.

  Derivative Error, this type of response predicts how the error will change in the future, i.e. it is an estimate of the future trend of the error based on its current rate of change (K_d * rate of change of error). It acts as a **dampening** mechanism to reduce overshooting and oscillations caused by rapid changes in error. It basically prevents the robot from making **sudden** and **excessive** corrections.

  Thus every response has its own significance and can be used as a combination to build a stable control system.

The PID controller calculates the `control` based on the error (`err`), its derivative (`err_der`), and the integral of the error (`int_err`).

Direction Control

- The code uses the `motion_set` function to set the direction of the robot's motors. Several direction control functions are provided, including `forward`, `back`, `left`, `right`, `soft_left`, `soft_right`, `soft_left_2`, `soft_right_2`, `hard_stop`, and `soft_stop`, which configure the motors for different movements.
  The controller's output also determines the motor's direction. The controller's output needs to be greater than 1.1 in order for the motor to rotate to the right. Tuning led to the selection of this value. We now maintain a threshold condition (1.1 here) to determine whether to do a soft or normal right turn. In other words, when the controller's output is between 4 and 100*kp, we will turn right normally, and when it is larger than 100*kp, we will turn right softly.
  The controller's output must be less than -1.1 for the motor to turn left. The threshold value for the right turn should be comparable to this value. Once more, we maintain a threshold for reasonable and gentle left turns. That is, we will do a hard left turn when the controller's output is less than -1.1 and larger than -100*kp, and a conventional left turn when the controller's output is less than -100*kp. The motor will move directly in the track's direction if the controller's output is between -1.1 and 1.1.

Velocity Control

- The `velocity` function sets the PWM (Pulse Width Modulation) values for the left and right motors, allowing independent speed control. The code calculates PWM values based on the `control` signal and sets motor velocities accordingly.

6 Results, Observations and Inferences

6.1 Results

Our line-following robot completed the maze in **2.59**s, which is within the required specification of 30s.

- K_p: **0.9**
- K_i: **0.05**
- K_d: **0.0004**

The bot completed the track with no errors, with no problems faced at the crossroads and sharp turns.

7 Challenges Faced

Calibrating the IR sensor

We had to adjust the potentiometers attached to our IR sensors because they were providing readings in a variety of ranges. To prevent favoring one side and affecting the motion, it was essential to equalize the reading ranges of the right and left sensors. The objective was to ensure that the general ranges of readings from each sensor were similar. To overcome this difficulty, we continued to fine-tune our IR sensors using the potentiometer (essentially changing its resolution) in order to achieve ideal sensing and proper PID controller operation.

Tuning the hyperparameters

The main challenge always revolves around tuning hyperparameters to achieve accuracy while meeting all constraints. This task becomes even more daunting when dealing with inaccuracies in the sensing of IR sensors.

In order to get around this problem, we gradually increased and improved our hyperparameters while making the required corrections to the IR sensor potentiometers during multiple test runs on the track. Additionally, we adjusted the thresholds, K_p and K_d, thanks to our newfound understanding of the bot's turning behavior. Surprisingly, the value of K_i didn't need to be changed because it already performed satisfactorily.

Sharp Turns and Junction Turning

We discovered that the sensing process was not perfect and that some time was needed for detecting when our robot occasionally failed sharp bends. This issue was resolved by slowing down the turns' base speed, which gave the bot enough time to recognize turns. We also resolved this issue by adjusting the turn and K_p thresholds.
Another difficulty our bot ran into was junction turning. To overcome this problem, we first noticed that on straight tracks, the center IR sensor produced somewhat higher numbers when aligned exactly with the line than it did at turns. Keeping this in mind, we set a threshold for the reading from the center IR sensor below which we would drive turns and otherwise forward motion.