Chapter 1

PID control of a line following robot (SPARK V)

1.1 Problem statement:

To design and implement a PID controller for the Spark V robot to make it follow continuous lines (Black lines on White paper).

1.2 About SPARK V

SPARK V is a robot which is based on ATmega16A microcontroller. SPARK V is jointly designed by NEX Robotics with Department of Computer Science and Engineering, IIT Bombay. It has 7.2 V NiMH battery, 3 analog white line sensors, 3 analog InfraRed proximity sensors and a 2x16 alphanumeric LCD, indicator LEDs, Buzzer, an L293D motor driver circuit and many other sensors and features for different purposes. It has Two DC geared motors and caster wheel as support. SPARK V supports GUI based control and programming can be done on WinAVR or AVR studio. SPARK V includes USB interface for PC Connectivity. It comes with Bootloader Utility from NEX Robotics which uploads the firmware into the robot.

1.3 Line Following Robot

A line following robot follows a path made of black lines on white surface or vice versa. Its objective is to sense the line and stay on the line while constantly correcting its position by means of a feedback mechanism. A feedback system makes the it as a closed loop system.

1.3.1 Motion control using PWM

SPARK V's motion control involves direction control and velocity control. Its direction control is made by its DC motors differential drive. DC motors support differential drive allows us to move the robot in forward, backward, sharp right turn, sharp left turn, soft right turn and soft left turn. Sharp turns are zero radius turns making one motor run in clockwise direction and other in anticlockwise direction. Soft turns are made by making one motor completely stop while other run in clockwise or anticlockwise direction. This is done by changing the appropriate logic levels on L293D motor driver IC's direction control pins.

Velocity control of the robot is used by Pulse width modulation technique. Pulse Width Modulation is a technique by which power is delivered to the drive by means of switching the switch between power source and drive at very high rates. This very high switching reduces power losses in the drives occurs significantly in other resistive control methods. The average power supplied to the drive is proportional to the duty cycle which is ON period over total period of the pulse Fig.(1.1).

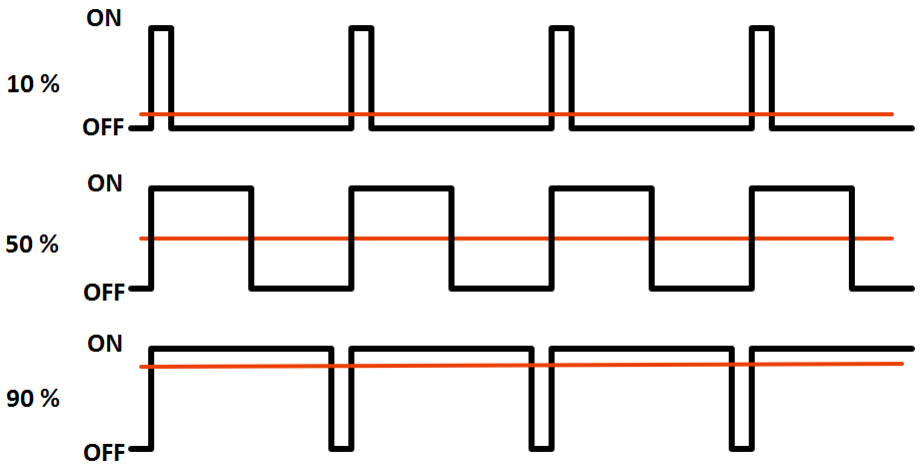

Figure 1.1: Pulse Width Modulation.

There are two parts of motion control.
1. Direction control is done using pins of PORTB0 to PORTB3 by giving appropriate logic levels.
2. Velocity control by PWM technique is done by setting PD4 and PD5 as output pins using OC1A and OC1B of timer 1.

1.3.2 Direction control working principle

The analog white sensors whenever encounters a black surface the output of these sensors is an analog signal of low voltage which is sensed by Analog to Digital Converter (ADC) of microcontroller and consider it as LOW. similarly whenever sensor is over white surface, its output analog signal is of high voltage which ADC consider it as a HIGH. Output of the white sensor always gives an analog signal which depends on the amount of the light reflected by the surface. The direction control is implemented with the help of above principle.

Appropriate logic is defined for the robot to move forward whenever a straight line is present. The middle sensor will always face on black line and rest two will be on white surface and it is made to move forward. Similarly if all three sensors are on white surface implies absence of black line, then it is made to move forward again until it reaches the black line (Fig. 1.2).

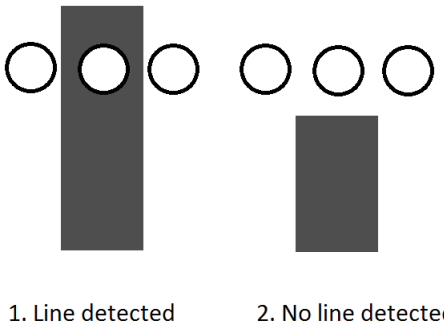

Figure 1.2: Line following robot on a straight line.

On the curved lines, if left sensor senses black line and right sensor is on white surface, it implies a left turn is detected and a logic is given to make robot turn soft left. Similarly if left sensor is on white surface and right sensor detects black line, it means a right turn is detected and a logic is given to make robot turn soft right (Fig. 1.3).

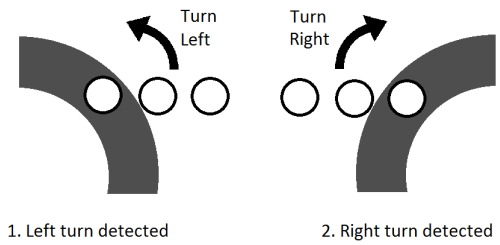

Figure 1.3: Line following robot on a curved line.

On 90 degrees turns, the middle sensor and either of the left or right sensor is over the black line and one remaining is on white line. Such turns requires sharp turns to take. Appropriate logic is created to face such turns.

The robot also encounters a crossroad, so an appropriate logic is given to make robot pass that crossover and move forward. It happens when all three white sensor detects black surface. The following Figure 1.4 gives insight of the turns.

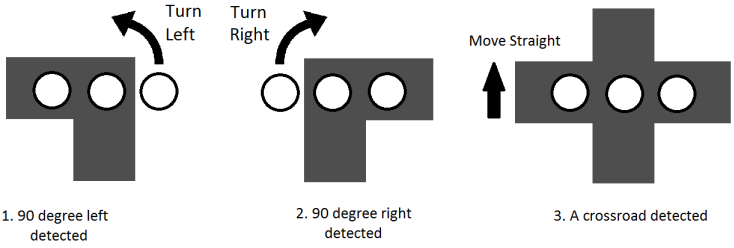

Figure 1.4: Line following robot on a 90 degrees turn and crossroad.

1.4 Implementation of Proportional-Integral-Derivative (PID) controller

PID is a generic control loop feedback mechanism which continuously determines error values as a difference between present state and desired state and it attempts to minimize the error by adjusting a measured process variable. It involves three separate correction methods that are proportional to error, accumulated error and real time difference of the error. If $u(t)$ is control variable and $e(t)$ is the error value then

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d(\tau) + K_d \frac{de(t)}{dt}$$

where K_p, K_i, K_d are all non-negative, denote the coefficients for the proportional, integral, and derivative terms.

The proportional term produces an output value that is proportional to the value of error. It is regulated by multiplying the error term by K_p, which is also called the proportional gain constant. The contribution from the integral term is proportional to both magnitude of the error and the duration for which the error is accumulated. The integral term gives the sum of instantaneous error over time. It is multiplied by the K_i and is added to the controller. This K_i is called integral gain constant. The derivative term gives the rate of error over time, which is multiplied by the K_d and it is added to the controller. K_i is called the derivative gain constant of the controller.

The effects of these parameters changing independently are seen from the following table.

Effects of changing the parameters independently					
Parameter	Rise time	Overshoot	Settling time	Steady state error	Stability
K_p	Decrease	Increase	Small change	Decrease	Degrade
K_i	Decrease	Increase	Increase	Eliminate	Degrade
K_d	Minor change	Decrease	Decrease	No effect in theory	Improves if K_d is small

Table 1.1: Effects of change in parameters

1.5 Manual Tuning of PID controller

Initially all control constants should be set to zero. We start tuning our controller by increasing K_p first. With increase in K_p the robots overshoot increases and starts oscillating. Proportional tuning requires to take the major change of output of controller and reach to the setpoint. Too high proportional constant can make the robot unstable. It is tweaked till the robot reaches the setpoint and is at brink of oscillating around the setpoint. The following Figure 1.5 shows the effect of increase in K_p.

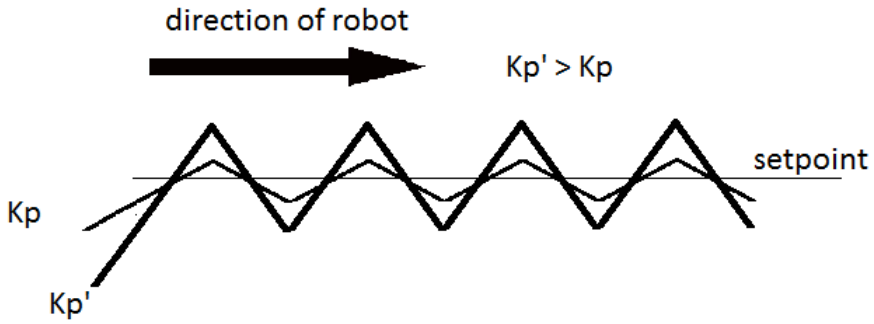

Figure 1.5: Effect of increasing K_p on line follower robot.

After K_p is tuned, K_i is increased until steady state error is eliminated in sufficient amount of time. The integral term accelerates the robot to reach setpoint, and can cause increase in overshoot due to responding to accumulated errors. Therefore too much increase

in K_i can also cause robot to become unstable. After K_i is tuned such that steady state error in the robot is eliminated, then K_d is increased and tuned such that the objective is reached. Too much increase in K_d will lead the robot to respond very fast and will cause overshooting again. The following Figure 1.6 shows the effect of tuning of K_i and K_d.

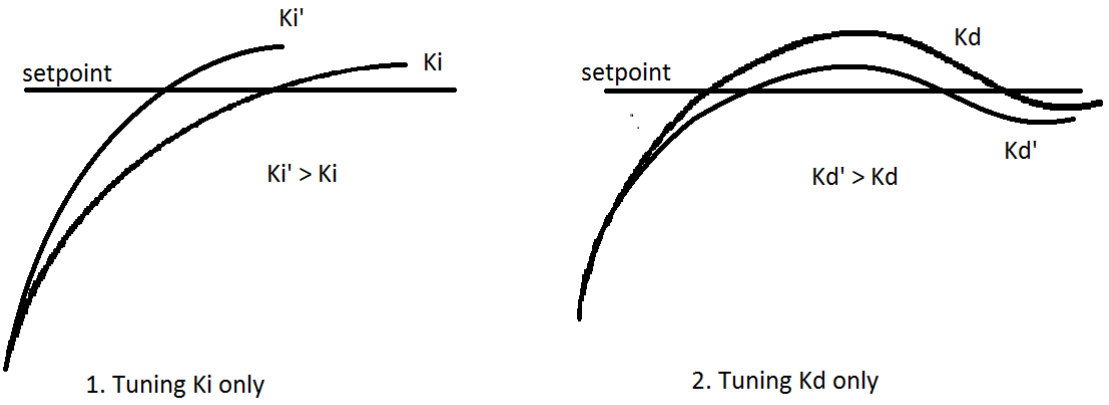

Figure 1.6: Effect of increasing K_i and K_d on line follower robot.

It can be seen that increasing K_i improves rise time and reaches to setpoint, but it also causes the robot to overshoot more. The effect of increasing K_d can be seen by observing it improves settling time, and overshoot. It can also be observed that it has negligible effect on rise time.

1.6 Source Code with explainations

The source code is written in AVRstudio. Different sections of the main source code are explained here as follows:

1.6.1 Timer and counter configuration: Fast PWM mode

In this section we have initialized TIMER 1 in Fast PWM mode for velocity control of the robot. Timer set point value is set to TOP=0x00FF. TCNT1H and TCNT1L registers denotes counter's highest and lowest 8 bit value to which OCR1AH and OCR1AL will compare with. OCR1A and OCR1B registers are the Output Compare Registers for Left Motor and Right Motor respectively. TCCR is Timer/Counter Control Register has is responsible for setting up the Fast PWM mode, with normal mode waveform generator. TCCR1A defines control output mode of channel, and TCCR1B defines Fast PWM in 8 bit mode..

```
void pwm_init(void)
{
    TCCR1B = 0x00;
    TCNT1H = 0xFF; //setup
    TCNT1L = 0x01;
    OCR1AH = 0x00;
    OCR1AL = 0xFF;
    OCR1BH = 0x00;
    OCR1BL = 0xFF;
    ICR1H  = 0x00;
    ICR1L  = 0xFF;
    TCCR1A = 0xA1;
    TCCR1B = 0x0D;
}
```

1.6.2 Pin configuration for Motion and PWM

Function motion_pin_config is for pin configuration of motion and pwm. DDRB sets direction of the PORTB3 to PORTB0 pins as output, PORTB sets initial value of the PORTB3 to PORTB0 pins to logic0. DDRD sets PD4 and PD5 pins as output for PWM generation. PORTD sets PD4 and PD5 pins for velocity control using PWM.

```
void motion_pin_config (void)
{
    DDRB = DDRB | 0x0F;
    PORTB = PORTB & 0xF0;
    DDRD = DDRD | 0x30;
    PORTD = PORTD | 0x30;
}
```

1.6.3 Pin configuration and Initialization for ADC

Function adc_pin_config configures the ADC pins on PORT A of the microcontroller. DDRA sets PORTA direction as input and PORTA sets PORTA pins floating.

```
void adc_pin_config (void)
{
    DDRA = 0x00;
    PORTA = 0x00;  }
```

Function adc_init initializes the ADC on microcontroller. ADCSR is the ADC Control and Status register. ADMUX is ADC Multiplexer Selection register has ADLAR register which is set to HIGH to enable left adjustment in ADC. ADEN = 1 enables ADC, ADPS is the ADC Prescaler Select register and by setting ADPS2 = 1, ADPS1 = 1, ADPS0 = 0, its prescaler selection is set to 64. This determine the division factor between the XTAL frequency and the input clock to the ADC.

```
void adc_init()
{
    ADCSRA = 0x00;
    ADMUX = 0x20;              //ADC CONVERSION
    ACSR = 0x80;
    ADCSRA = 0x86;
}
```

Function ADC_Conversion accepts the Channel Number and returns the corresponding Analog Value to the program. ADCSRA = ADCSRA | 0x40 Sets the start conversion bit. While loop ensures the ADC conversion is complete. ADCSRA = ADCSRA|0x10 clears ADIF (ADC Interrupt Flag) by writing 1 to it.

```
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;
    return a;
}
```

1.6.4 Functions for Motor velocity control

The function speed() takes velocity for left motor and right motor as input parameter and assigns them to output compare register OCR1A and OCR1B. Channel A is used for left motor and channel B is used for right motor. Since we are using PWM in 8 bit resolution we only load lower byte of the OCR1A and OCR1B registers. The if else condition makes 250 as maximum threshold for both motors. The OCR1 register for motor gets its values from the final output of the PID routine which is discussed later.

```
void speed(int l_motor,int r_motor)
{


        if (l_motor>250){
                l_motor = 250;
                }
        if (l_motor<0){
                l_motor = 0;
                }
        if (r_motor>250){
                r_motor = 250;
                }
        if (r_motor<0){
                r_motor = 0;
                }
    OCR1AH = 0x00;
    OCR1AL = l_motor;
    OCR1BH = 0x00;
    OCR1BL = r_motor;
}
```

1.6.5 Error Calculation and PID functions

The function calc_error takes three parameters s1, s2 and s3 which are ADC values of Left sensor, Centre Sensor and Right Sensor, and calculates their weighted error. This is the instantaneous position error of the robot from the centre of the line. After calculating the error, the control of the program is transferred to the PID routine.

```
void calc_error(char s1, char s2, char s3)
    {
            float s;
        perror = error;
        s = (s1+s2+s3);
        error = ((1*s1 + 2*s2 +3*s3)/s);
            error = error - 1.9;
        }
```

After the program control is transferred to the pid function, it performs the PID algorithm over the calculated error and gives out the correction term. This correction term is used as feedback value to the OCR1 registers for motor speed control which makes appro-

priate speeds to both the motors so that the robot always follows the line with optimal control.

```
void pid(void)
{
    P = error * Kp;
    Iprev += error;
    I = Iprev * Ki;
    D = 0;
    D = Kd*( error - perror);
    correction = P + I + D;
}
```

A sample code snippet of right turn shows how this correction term is taken into account inside the speed function. The code snippet also shows how logic for right turn detection is written. Here THRESHOLD defines the analog equivalent value of ADC from which a sensor changeover from black line to the white surface. It is typically around 30.

```
if ((Center_sensor < THRESHOLD) && (Left_sensor < THRESHOLD)
&& (Right_sensor > THRESHOLD))
{
    speed(50 + correction,50 + correction);
    right();
    flag =2;
    _delay_ms(150);
}
```

1.7 Source code

```
//  Line following Robot
//  Created by Ayan Sengupta and Sarthak Sharma on 25/07/16.

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

#define THRESHOLD  30
#define BASESPEED 80

unsigned char ADC_Conversion(unsigned char);
```

```
unsigned char ADC_Value;
unsigned char Left_sensor = 0 ;
unsigned char Right_sensor = 0;
unsigned char Center_sensor = 0;

// Define pid parameters
float Kp = 120;
float Ki = 0.3;
float Kd = 50;
float error = 0, perror = 0, Iprev = 0;
float P, I, D, correction;

//PWM
void pwm_init(void)
{
    TCCR1B = 0x00;
    TCNT1H = 0xFF;
    TCNT1L = 0x01;
    OCR1AH = 0x00;
    OCR1AL = 0xFF;
    OCR1BH = 0x00;
    OCR1BL = 0xFF;
    ICR1H  = 0x00;
    ICR1L  = 0xFF;
    TCCR1A = 0xA1;
    TCCR1B = 0x0D;
}


// ALL PIN CONFIGS
void motion_pin_config (void)
{
    DDRB = DDRB | 0x0F;
    PORTB = PORTB & 0xF0;
    DDRD = DDRD | 0x30;
    PORTD = PORTD | 0x30;
}

void lcd_port_config (void) {
```

```
    DDRC = DDRC | 0xF7;
    PORTC = PORTC & 0x80;

}

//All the ADC pins must be configured as input and floating
void adc_pin_config (void){

    DDRA = 0x00;
    PORTA = 0x00;

}

void port_init()
{
    motion_pin_config();
    lcd_port_config();
    adc_pin_config();
}

void adc_init() {
    ADCSRA = 0x00;
    ADMUX = 0x20;
    ACSR = 0x80;
    ADCSRA = 0x86;
}

//ADC CONVERSION
unsigned char ADC_Conversion(unsigned char Ch)
{
    unsigned char a;
    Ch = Ch & 0x07;
    ADMUX= 0x20| Ch;
    ADCSRA = ADCSRA | 0x40;
    while((ADCSRA&0x10)==0);
    a=ADCH;
    ADCSRA = ADCSRA|0x10;
    return a;
}
```

```
void sensor_value(char row, char coloumn, char channel)
{
    ADC_Value = ADC_Conversion(channel);
    lcd_print(row, coloumn, ADC_Value, 3);

}

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;
    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 (void) //Left backward, Right forward
{
    motion_set(0x05);
}

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

void soft_left (void)  //Left stationary, Right forward
{
```

```
    motion_set(0x04);
}

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

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

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

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

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

//Function defining speed of motors
void speed(int l_motor,int r_motor)
{
        if (l_motor>250){
                l_motor = 250;
                }
        if (l_motor<0){
                l_motor = 0;
                }
        if (r_motor>250){
                r_motor = 250;
                }
```

```
        if (r_motor<0){
                r_motor = 0;
                }
    OCR1AH = 0x00;
    OCR1AL = l_motor;
    OCR1BH = 0x00;
    OCR1BL = r_motor;
}

void init_devices (void)
{
    cli();
    port_init();
    pwm_init();
    adc_init();
    sei();
}

// calculate error for pid implementation
void calc_error(char s1, char s2, char s3){

        float s;
        perror = error;
        s = (s1+s2+s3);
        error = ((1*s1 + 2*s2 +3*s3)/s);
        error = error - 1.9;
        }

//pid
void pid(void){

    P = error * Kp;
    Iprev += error;
    I = Iprev * Ki;
    D = 0;
    D = Kd*( error - perror);
    correction = P + I + D;
}

// MAIN CODE
```

```
int main(void) {

    init_devices();
    lcd_init();
    lcd_set_4bit();
        int flag =0;

    while(1){

        Left_sensor = ADC_Conversion(3); //ADC3 for left sensor
        Right_sensor = ADC_Conversion(5); //ADC5 for right sensor
        Center_sensor = ADC_Conversion(4); //ADC4 for center sensor
                _delay_ms(5);
                calc_error(Left_sensor,Center_sensor,Right_sensor);
                pid();
                _delay_ms(5);

                sensor_value(1,1,3); //Prints value of Left sensor
        sensor_value(1,5,4); //Prints value of Center sensor
        sensor_value(1,9,5); //Prints value of Right sensor
                _delay_ms(10);

                if (Center_sensor > THRESHOLD) {
            speed(255,255);
                forward();
                flag =1;
        }

        if ((Center_sensor < THRESHOLD) &&
        (Left_sensor < THRESHOLD) &&
        (Right_sensor > THRESHOLD)){

            speed(50 + correction,50 + correction);
            right();
            flag =2;
            _delay_ms(150);
        }

        if ((Center_sensor < THRESHOLD) &&
        (Left_sensor > THRESHOLD) &&
```

```
(Right_sensor < THRESHOLD)){

    speed(50 - correction,50 - correction);
    left();
    flag = 3;
    _delay_ms(150);
}

if ((Center_sensor > THRESHOLD) &&
(Left_sensor > THRESHOLD) &&
(Right_sensor > THRESHOLD)){

    speed(200,200);
    forward();
    _delay_ms(1000);
        }

        if ((Center_sensor < THRESHOLD) &&
        (Left_sensor < THRESHOLD) &&
        (Right_sensor < THRESHOLD)){

            if (flag ==2) {
            speed(50 + correction,50 + correction);
            right();
            _delay_ms(250);
                }

                else if (flag ==3){
                speed(50 - correction,50 - correction);
                left();
                _delay_ms(250);
                }

                else {
                speed(100,100);
                back();
                _delay_ms(250);
                }

        }
```

```
    }
}
```

1.8 Challenges

The experiment took almost three weeks to complete. Some major challenges were faced while performing the experiment are discussed below:

1.8.1 LCD module

LCD module played an important role in determining the threshold value of analog sensors. The threshold value for sensors is different for different robots. So incorporating LCD became one of the main sub-task of the experiment. The LCD module displayed the real time ADC values of sensors.

Including LCD header file was a tough task. It consistently gave error each time we included lcd.h in our program. So we had to write the whole code for lcd.h inside our source code. This worked out for us.

1.8.2 Implementation of PID

More the number of sensors, more we get a wide range of error to work on. With only three sensors present, implementation of PID algorithm was a bit inefficient. The error range from three sensors would be limited. A psuedo PID algorithm was used that required PID to work along with threshold conditions and correction terms from PID were added to the speed function of corresponding motor direction function. It made the robot fast on track but wobbling was a bit increased.

1.8.3 Overcoming Shadows

Shadows generate random errors in line follower robot. The sensors would give garbage random values whenever they would face shadows of surrounding objects due to improper lighting of the surface. Sometimes the robot's own shadow would degrade its performance. To overcome this problem, we calibrated the our threshold values in the dark by turning off the lights. This reduced our threshold of sensors from previous threshold value, that can distinguished between shadow and the dark line.