12/21/2017

ADVANCE LINE FOLLOWING ROBOT

Applied Mechatronic Construction Project-EEEE-1031

Vigneshwaren Sunder
STUDENT ID: 20021065

1.0 INTRODUCTION:

1.1 OVERVIEW:

Robots are widely used in industries as they became more affordable and efficient. However, robots still tend to make some errors and their task won't be perfectly accurate. To overcome this a better controller can be implemented which allows the robot to perform efficiently and make fewer errors. A line following robot is an autonomous vehicle which is capable of following a given path. We made a traditional line following robot which consists of a chassis, IR sensors, microcontrollers, DC motors and motor driver. This project tries to implement PID controller on the prebuilt line following robot and thus trying to reduce the errors made by the line following robot while moving. The robot consists of a sunfounder line following module which consists of array of 8 IR sensors through which the robot detects the black line and follows the path. The PID is implemented so that the robot will be able to follow the given black line effectively and smoothly.
The control system used here senses the black line and maneuvers the robot to stay on course while constantly correcting the wrong moves using PID control.

1.2 OBJECTIVES:

- Learn how to use the I2C interface and process data from sensors to achieve simple navigation.
- Design an advance line following robot which works on PID control system.
- Learn how to use the MPU-6050 gyro accelerometer sensor module and allow the robot to navigate a simple path that leads to the ramp and measure its gradient and also, to make a 360 degrees turn over the ramp using the yow value of the MPU-6050 value after a 3 seconds delay after reaching the top of the ramp.
- Implement a weighted average algorithm and demonstrate its operating principle for a range of reference points by recording the reflectivity value profiles as the sensor is moved over a black line on a white background.

- Learn to tune the values of PID constants to get the best possible performance.

2.0 MPU-6050 Gyro Accelerometer Sensor Module:

It is the world's first integrated 6-axis MotionTracking device.

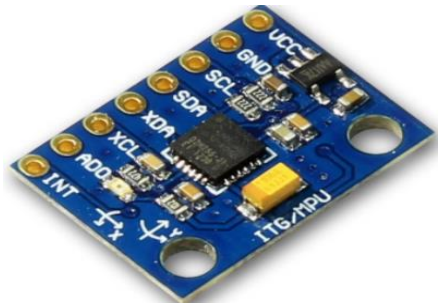


Fig: 1.0

The InvenSense MPU-6050 sensor contains a MEMS accelerometer and a MEMS gyro in a single chip which is processed by the Digital Motion Processor. It is very accurate, as it contains 16-bits analog to digital conversion hardware for each channel. Therefore, it captures the x, y, and z channel at the same time. The sensor uses the standard I2C bus for data transmission. The MPU-6050 always acts as a slave to the Arduino with the SDA and SCL pins connected to the I2C-bus. The connection layout of MPU-6050 to the Arduino is given below.

Pin on Arduino UNO	Pin on MPU-6050
5V	VCC
GND	GND
A4	**SDA**
A5	**SCL**
2	INT

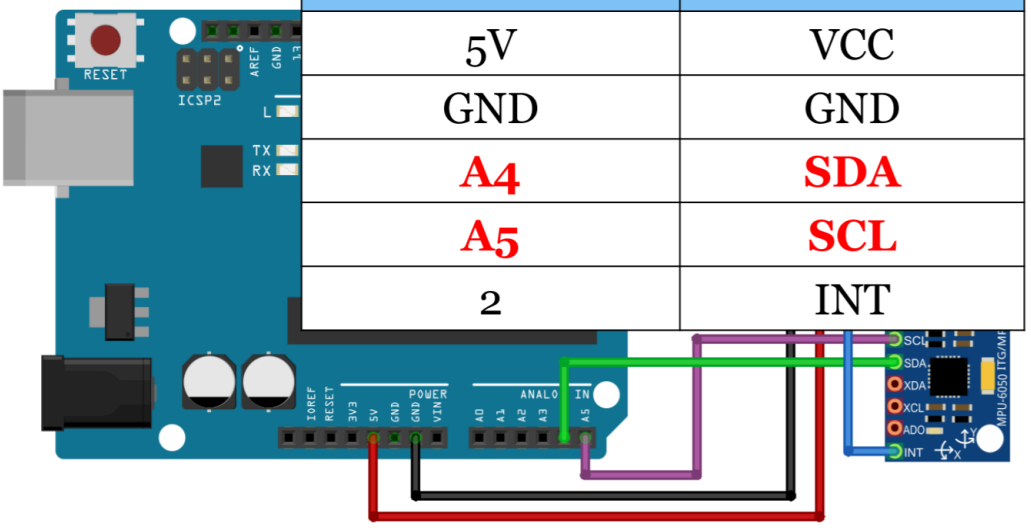


Fig: 1.2

The primary use of the MPU-6050 module in our project is to measure the angle of the ramp on either side and print the output on the LCD display. We have also used this module to make a 360 degrees turn by measuring the yaw value. When the robot climbs the ramp, the value is processed by the DMP(Digital Motion Processor). The maximum value processed is then displayed on the LCD screen as ramp angle.

2.1 APPLICATIONS OF MPU-6050 GYROSCOPE:

- Motion-enabled game and application framework
- Location-based services, points of interest
- Handset and portable gaming
- Motion-based game controllers
- Wearable sensors for health, fitness, and sports
- Toys

3.0 INTER-INTEGRATED COMMUNICATION (I2C BUS):

3.1 Overview:

The I2C bus is a very popular mode of communication. Through I2C bus, the communication can be easily implemented between the master and slave devices. The fact that a multiple numbers of master and slave modules can be connected with just two wires makes it more efficient and easy to use. Master is the device that generates the clock, starts communication, sends I2C commands and stops communication. The slave is the device that listens to the bus and is addressed by the master Multi-master I2C can have more than one master and each can send command.

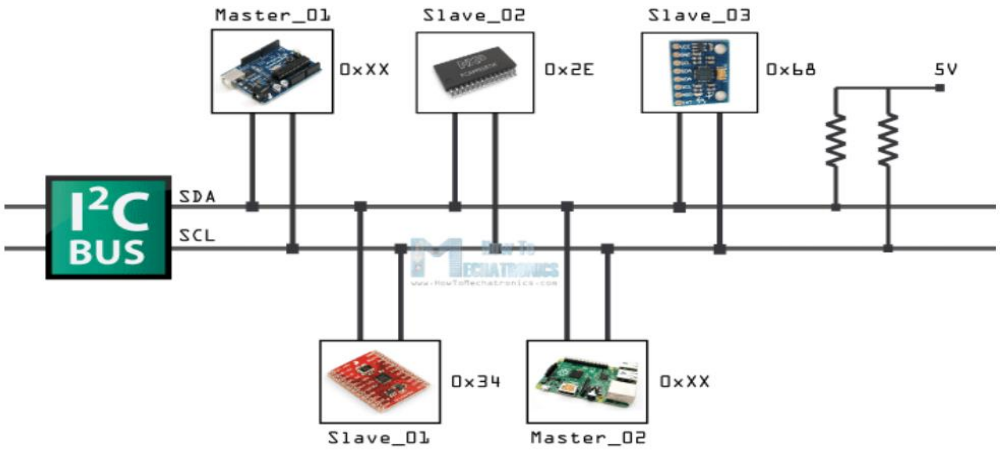
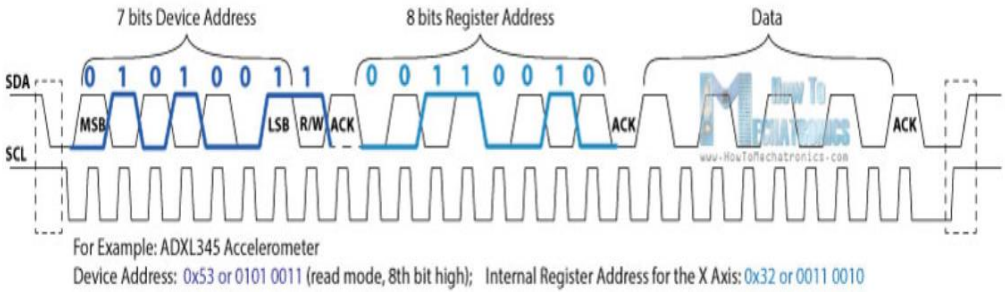


Fig: 1.3

3.2 Woking of the I2C Bus:

The two wires are called Serial Cloak Line(SCL) and Serial Data Line(SDA). The SCL will synchronize the data transfer between the devices on the I2C bus and it is generated by the master device. The SDA line carries the data. Data are transferred in a sequence of 8-bits which are sent with the most significant bits first. They can have up to 128 devices. Now to communicate with the relevant devices, we first need to find the unique address of that device. Each Slave device has to have its own unique address and both master and slave devices need to take turns communicating over the same data line.

Now we have to create the relevant code to initiate the I2C bus communication. We can use the Arduino Wire Library. Firstly, we need to define the sensor address and internal register addresses. Then, the Wire.begin() function is used to initiate the Arduino wire library and the serial communication.



After the addressing, the data transfer sequences begin either from the master or the slave depending on the selected mode at the R/W bit. After the data is completely sent, the transfer will end with a stop condition which occurs when the SDA line goes from low to high while the SCL line is high.

4.0 SUNFOUNDER IR SENSORS:

The robot uses Sunfounder IR sensor module to sense the line. It consists of an array of 8 IR sensors faced towards the ground. The communication between the module and the Arduino is done by I2C bus. The sensor module is fixed to the robot by using a 3-d printed IR sensor holder which is designed using SolidWorks software.

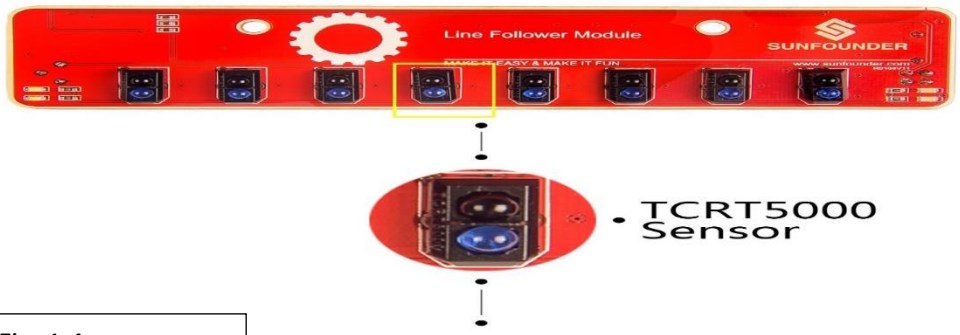


Fig: 1.4

4.1 CALIBRATION OF THE LINE FOLLOWER SENSOR MODULE:

We need to calibrate the output from the sensor array since not every sensor give the same value. Therefore, individual sensors are calibrated to return a maximum and minimum value ranging from 0 to 1000, so that it can find which sensor detects the black line and thus moves and corrects the position of the robot. The calibration process can be done easily, refer the flowchart below to know how calibration process works.

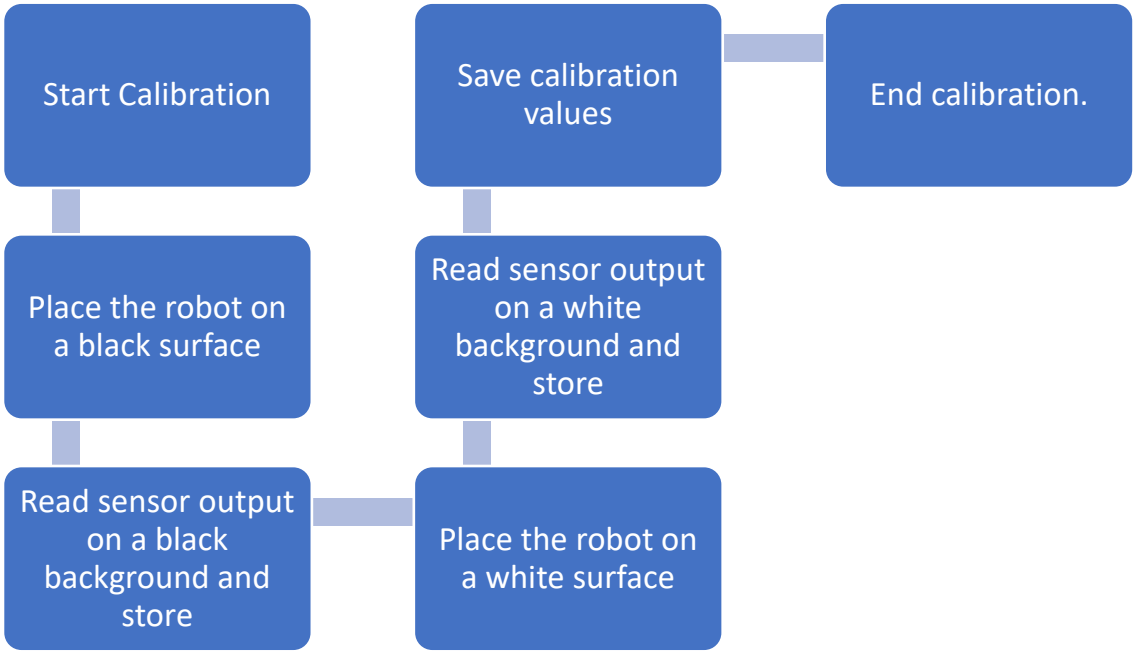


Fig: 1.5

Once the values have been noted down, the individual values from the sensors for black and white readings need to be mapped to a common range. The flowchart below shows how to map the acquired readings and get the calibrated values.

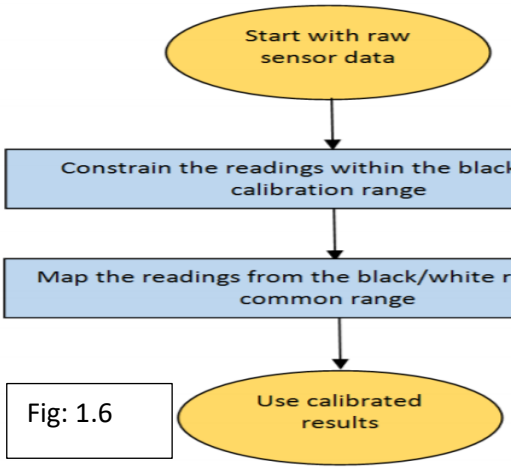


Fig: 1.6

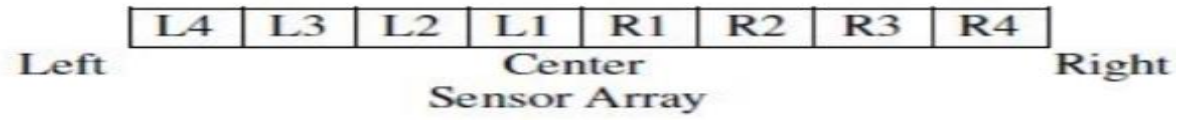
An analog signal is obtained in the output based on the amount of reflected light and is fed to the comparator and the digital signal is sent to the microcontroller. Here, the analog signal is converted into digital by the array.

4.2 Weighted Average Algorithm for Line Following:

A weighted average algorithm can provide a measure of error. Each sensor is calibrated to return a maximum and minimum value. The center point of the sensor module forms the reference point. Each sensor point is given a weighting. i.e., distance from the reference point which is given by,

$$d = \frac{\sum_{i=0}^{8} sensor\_value_i \times weight_i}{sensor\_value_i}$$

Consider the following schematic of the sensor module,



Let's assume that when the sensor senses the black line, it reads 0 and when it is off the black line, it reads 1. The microcontroller corresponds to the algorithm and executes the next movement in such a way that the center most sensors(L1 and R1) reads 0 and the rest of the sensors read 1. In this way, we need to do the calibration process and implement the weighted average algorithm.

5.0 The PID CONTROL SYSTEM:

A PID controller is a feedback loop system.

The basic idea is to steer the robot towards the line, and the farther off it is, the more it corrects itself.

The PID algorithm can be described as the following equation.

$$out(t) = K_p \times error(t) + K_i \times \sum_{n=0}^{n=t} error(n) + K_d \times (error(t) - error(t-1))$$

The output from the PID controller is used to change the speed of the motors.

The basic terminology of the PID control system is as follows.

Proportional(P)- The proportional constant directly multiplies the error and should, therefore, set a reasonable motor speed change given the error magnitude.

Integral(I)- The integral component multiplies a running sum of the error, therefore, its effect increases the longer the robot is not on track.

Differential(D)- The differential component multiplies the change in error, therefore, its effect depends on how quickly the line is changing compared to the course of the robot.

The figure below gives an overview of the PID constant and its individual contribution to the control system.

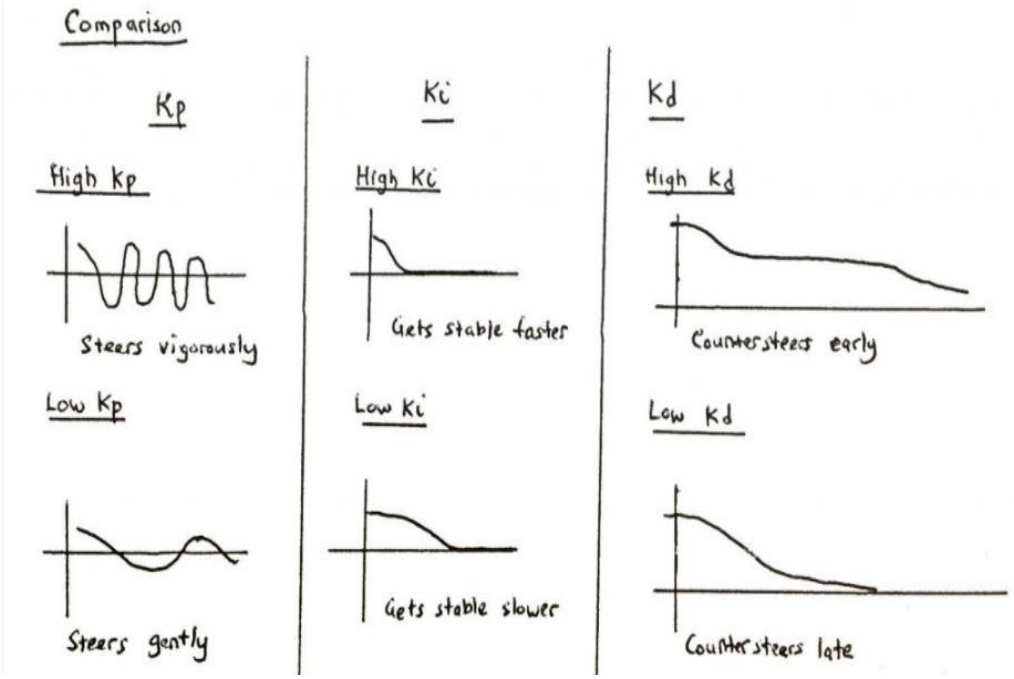


Fig: 1.7

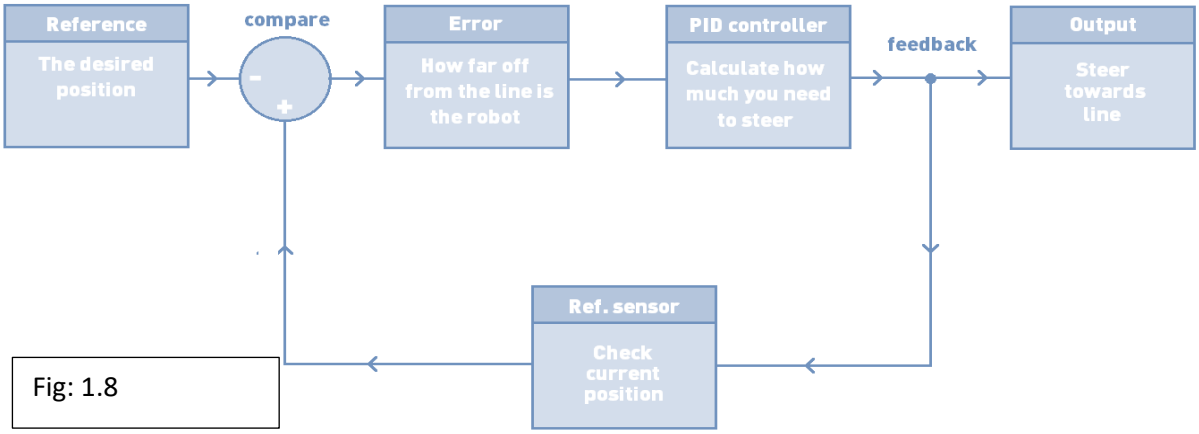


Fig: 1.8

The process is a feedback loop as shown in the figure 1.8. Such a design allows the maneuver robot on the track with the implementation of PID controller.

5.1 An overview of how PID Control System works in a line following robot:

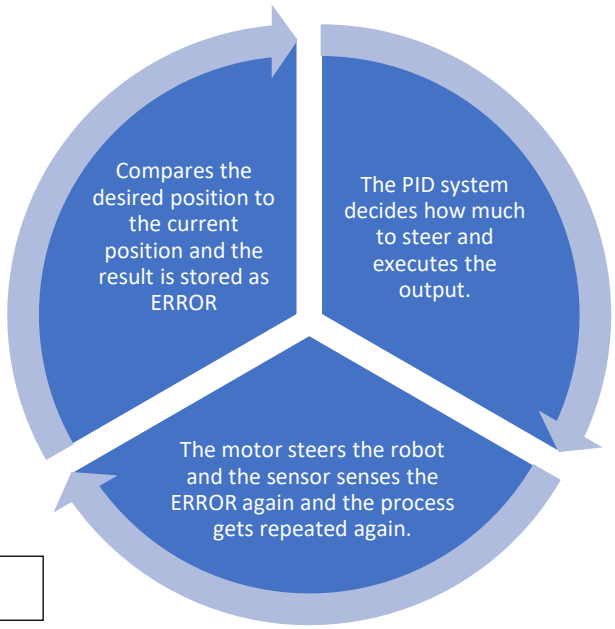


Fig: 1.9

5.2 TUNING THE PID CONSTANTS:

In this project, we used trial and error approach to design the PID controller.

The first step of creating a PID feedback loop is to define our desired position. We need to calibrate the values by simply moving the robot across the line and note its readings

We need to add the error value to the output by measuring the center point deviation of the line. If the robot deviates

from the black line, a proportional constant is given to bring the robot back to the black line as quick as possible. But, to prevent the robot from overshooting, a derivative constant is given. This corresponds to the rate of change of error. The integral constant is used to improve the accuracy and smoothness of the movement of the robot. It corresponds to the sum of the recent errors.

The terms in the PID equation namely Kp-pfactor, Ki-ifactor, Kd-dfactor is a constant value used to increase or decrease the impact of the proportion, integral and derivative respectively.

CONTROL VARIANT	DESCRIPTION	FINAL COEFFICIENT VALUE
PROPORTIONAL	Kp*error: The constant of proportionality is the gain of Kp. The proportional value determines the reaction to the current error. It reduces the large part of the error based on present time error.	20
INTEGRAL	Ki*integral: The integral improves the steady state performance of the robot. It determines the reaction based on the sum of the recent errors. It eliminates the steady state error. It reduces the final error in the system based on the log of the previous errors over time.	10
DERIVATIVE	Kd*(error-lasterror): The derivative corresponds to the rate of change of error. Has an element of prediction of future errors. React to rapid rate of change before error grows to big. It stabilizes the system and thus prevents overshoot. It conteracts	100

	the Kp and Ki values quickly when the output changes	

5.3 The overall program flow:

6.0 Conclusion:

In our project we implemented the PID control system and used the microcontroller to build a line following robot. Our expectation for this project is to make the car move with PID controller and use the MPU-6050 module to display the angle of the ramp kept in the track, climb down the track and move for certain distance and show the angle of the other side of the ramp. Though we achieved the ultimate goal of the project, we faced many challenges during the making of the bot.

One main challenge was the MPU-6050 library crashed whenever new codes are added to it. To overcome that issue, we edited the library code so that it gets executed in a loop. Therefore, the code gets looped back to be reset until the desired sensor value is obtained.

There are many flaws in the design and operation of our robot. The movement of our robot was not smooth as there was high degree of oscillation in the movement of the robot. That was because the PID constants that we used were not so accurate. Also, the initial placement of the robot has to be at the center of the black line. If not, the initial error will not be accurate and the robot may fail to follow the track smoothly.

The output on the LCD screen is given below.

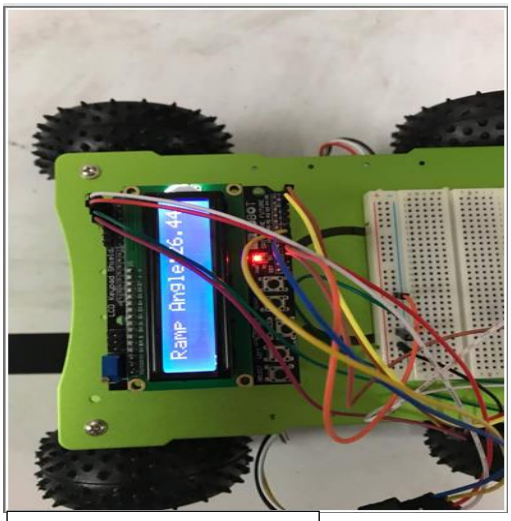


Fig: 2.0

Fig: 2.1

Here are some methods that can improve the performance of our line following robot.

- Fine tune of Kp and Kd parameters used in the PID controller.
- Implementing PID constants by using Zeigler-Nichols tuning method as this would give better accuracy.
- Better placement of sunfounder line follower sensor module.
- Use of fully charged battery to get a better output in the motors.

Thus, our advance line following robot was able to complete the track more smoothly and accurately, unlike the last project week where we used normally if statements to move the.

7.0 References:

1. Jamie. PID Control. Arduino Platforms. [Online][cited: 1yr ago] https:// car www.mvcode.com/lessons/pid-control-jamie.
2. Arduino official. MPU-6050 Arduino platforms.[Online] https://playground.arduino.cc/Main/MPU-6050.

3. MTaylor. Line Follower Array Hookup Guide. [Online] https://learn.sparkfun.com/tutorials/sparkfun-line-follower-array-hookup-guide/introduction.
4. Peter Harison. Simpler Line follower sensors. [Online][Cited: April 15, 2011] http://www.micromouseonline.com/2011/04/15/simpler-line-follower-sensors/.
5. Dejen Nedelkovski. Arduino Tutorial. [Online] http://howtomechatronics.com/tutorials/arduino/how-i2c-communication-works-and-how-to-use-it-with-arduino/.
6. Enigmerald. PID Tutorials for Line Following. [Online] [cited: January 6, 2014] https://www.robotshop.com/letsmakerobots/pid-tutorials-line-following.

Appendix:

What is a control system?

A control system is an interconnection of components forming a system configuration that will provide the desired system response. It can be generally classified into two types.

- Open loop control system
- Closed loop control system

In this project, we used closed loop control system. The characteristics of closed loop control system are:

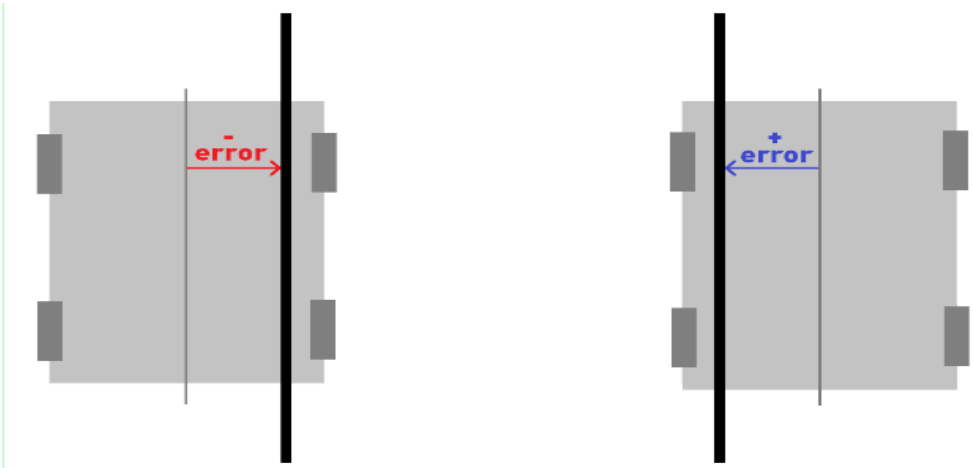
- Needs a measurement element at plant output.
- Control is error driven.
- Responds to unmeasured disturbances.

A key element of the closed loop system is that the controller works based on an error signal – the difference between the desired output (setpoint) and the actual output. It attempts to minimise the error.

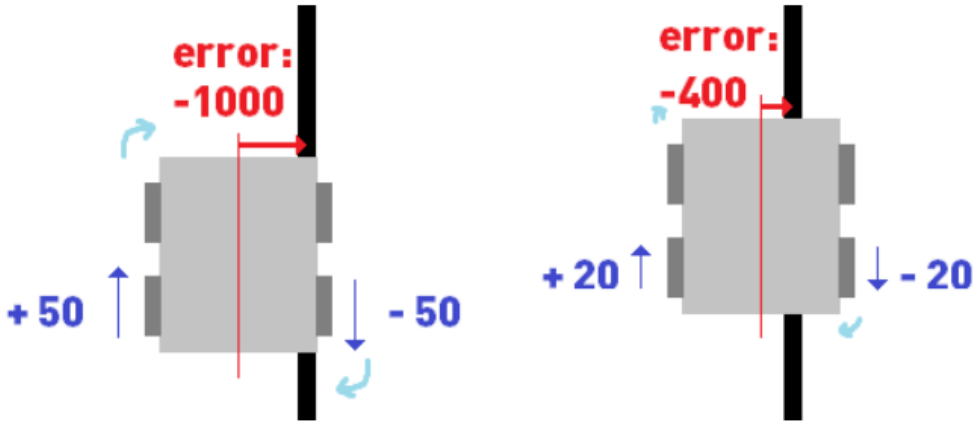
A detailed explanation of how PID control system works.

The error is a measure of how far the cart is from the track. The further the car is off the track, the larger the error.

More specifically we can say that error is the difference between the current position and the target position.



The error will be negative when the car is on the right side of the black line. Similarly, the error will be positive when the car in on the left side.



The Fig. shows how the speed of the wheel is changed based on the error value obtained

The codes used in this project is given below.

CODES:

This code is uploaded to Arduino duemilanove.

```
#include "motordriver_4wd.h"
#include <seeed_pwm.h>

#include <Wire.h>
#include "motordriver_4wd.h"
#include <seeed_pwm.h>

#define leftMotorBaseSpeed 15
#define rightMotorBaseSpeed 15

#define min_speed -50
#define max_speed 50

int leftMotorSpeed;
int rightMotorSpeed;
```

```
unsigned char data[16];
unsigned int sensorData[8];
unsigned calData [8];

float Kp, Ki, Kd;
int i,n;
int previousTime;
float lastError;

char input;

void setup() {
  // put your setup code here, to run once:
  Wire.begin();
  Serial.begin (115200);
  MOTOR.init();

  PIDconstants();
  readSensorData();
  lastError = weightedAverage(); // set the first detected error as the reference error to calculate differential error when Hercules starts to move
  previousTime = millis(); // set 0 as the initial time before Hercules starts to move
}

void loop() {
  // put your main code here, to run repeatedly:
  if (Serial.available () > 0)
  {
```

```
input =  Serial.read (); // read message sent from Uno
switch (input)
{
  case 'a': // move forward if Uno sends an 'a'
    MOTOR.setSpeedDir (20, DIRF);
    break;

  case 'b': // turn left if Uno sends a 'b'
    MOTOR.setStop1();
    MOTOR.setStop2();
    MOTOR.setSpeedDir2 (60, DIRF);
    MOTOR.setSpeedDir1 (60, DIRR);
    break;

  case 'c': // turn right if Uno sends a 'c'
    MOTOR.setStop2();
    MOTOR.setStop1();
    MOTOR.setSpeedDir1 (60, DIRF);
    MOTOR.setSpeedDir2 (60, DIRR);
    break;

  case 'd': // stop if Uno sends a 'd'
  MOTOR.setStop1();
  MOTOR.setStop2();
  break;

  case 'e': // follow line with PID if Uno sends a 'e'
  PID ();
```

```
      default:
        break;
    }
  }
 }

void PID (void)
{
  readSensorData ();
  calibrateSensorData ();
  double error = weightedAverage();
  int output = PID(error);

  leftMotorSpeed = leftMotorBaseSpeed + output;     // Calculate the modified motor speed
  rightMotorSpeed = rightMotorBaseSpeed - output;

  int condition;

  if (leftMotorSpeed > 0 && rightMotorSpeed > 0)
  {
    leftMotorSpeed = constrain(leftMotorSpeed, 0, max_speed);
    rightMotorSpeed = constrain(rightMotorSpeed, 0, max_speed);

    condition = 1;
  }
  else if (leftMotorSpeed < 0 && rightMotorSpeed > 0) // set negative leftMotorSpeed to positive
  {
```

```
    leftMotorSpeed = constrain(leftMotorSpeed, min_speed,
0);
    leftMotorSpeed = leftMotorSpeed * -1;


    rightMotorSpeed = constrain(rightMotorSpeed, 0,
max_speed);


    condition = 2;
  }
  else if (leftMotorSpeed > 0 && rightMotorSpeed < 0) // set
negative rightMotorSpeed to positive
  {
    rightMotorSpeed = constrain(rightMotorSpeed, min_speed,
0);
    rightMotorSpeed = rightMotorSpeed * -1;


    leftMotorSpeed = constrain(leftMotorSpeed, 0,
max_speed);


    condition = 3;
  }


  switch (condition)
  {
    case 1:  // Move forward
    MOTOR.setSpeedDir1 (rightMotorSpeed, DIRF);
    MOTOR.setSpeedDir2 (leftMotorSpeed, DIRF);
    break;


    case 2:  //Turn right
    MOTOR.setSpeedDir1 (rightMotorSpeed, DIRF);
```

```
    MOTOR.setSpeedDir2 (leftMotorSpeed, DIRR);
    break;

    case 3:  //Turn left
    MOTOR.setSpeedDir1 (rightMotorSpeed, DIRR);
    MOTOR.setSpeedDir2 (leftMotorSpeed, DIRF);
    break;

    default:
    break;
  }
}

void PIDconstants (void)
{
  Kp = 20;
  Ki = 10;
  Kd = 100;
}

void readSensorData (void)
{
Wire.requestFrom(9, 16);    // request 16 bytes from slave device #9 which is the Line Follower Module
  while (Wire.available())   // slave may send less than requested
  {
    data[i] = Wire.read(); // receive a byte as character
    if (i < 15) // save the data into an array
    {
```

```
    i++;
   }
   else
   {
    i = 0;
   }
  }
  for(n=0;n<8;n++)
  {
   sensorData[n] = data[n*2]; //convert 16-bit array into 8-bit array
  }
}

void calibrateSensorData (void) // offset each value into a fixed average value so that each sensor
{                               // will give an approximately same value when sensing black and white line
  calData[0] = sensorData[0] + 14;
  calData[1] = sensorData[1] - 2;
  calData[2] = sensorData[2] + 7;
  calData[3] = sensorData[3] - 2;
  calData[4] = sensorData[4] - 50;
  calData[5] = sensorData[5] + 20;
  calData[6] = sensorData[6] - 4;
  calData[7] = sensorData[7] + 16;
}

double weightedAverage (void)
{
```

```
  double weightedSum;
  double dataSum;
  double calibratedValue;
  double weightage[8] = {4.41,3.15,1.89,0.65,-0.65,-1.89,-3.15,-4.41}; // weightage fixed according to the distance between the midpoint and the sensor
  weightedSum = 0;
  dataSum = 0;
  for (n=0;n<8;n++)
  {
    weightedSum += calData[n] * weightage[n];
  }
  for (n=0;n<8;n++)
  {
    dataSum += sensorData[n];
  }
  double weightedAverage;
  weightedAverage = (weightedSum/dataSum);
  calibratedValue = (weightedAverage + 0.03 ) * 10; // offset weighted average value to 0 when the black line is at the middle
  return constrain (calibratedValue, -2.5, 2.5); // to prevent anomaly from sensor 7 from disrupting the symmetry of error
}


double PID (double error)
{
  double errorSum;
  double differentialError;


  errorSum += error; // calculatr sum of error
```

```
  float currentError = weightedAverage();
  int currentTime = millis();
  differentialError = (error - lastError)/(currentTime -
previousTime); // calculate rate of change of error


  lastError = error;
  previousTime = currentTime;


  if (abs(error) > 1) // prevent too much error is carry forward
over course correction
  {
    errorSum = 0;
  }


  float proportional = error * Kp;  // Calculate the components
of the PID


  float integral = errorSum * Ki;


  float differential = differentialError * Kd;


  double output = proportional + integral + differential;  //
Calculate the result


  return output;
}
```

This code is uploaded to the Arduino UNO.

```
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"
#include <LiquidCrystal.h>

#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
#include "Wire.h"
#endif

MPU6050 mpu;

#define OUTPUT_READABLE_YAWPITCHROLL

#define INTERRUPT_PIN 2  // use pin 2 on Arduino Uno & most boards
#define LED_PIN 13 // (Arduino is 13, Teensy is 11, Teensy++ is 6)
bool blinkState = false;

bool dmpReady = false;  // set true if DMP init was successful
uint8_t mpuIntStatus;   // holds actual interrupt status byte from MPU
uint8_t devStatus;      // return status after each device operation (0 = success, !0 = error)
uint16_t packetSize;    // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount;     // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars
Quaternion q;           // [w, x, y, z]         quaternion container
```

```
VectorInt16 aa;         // [x, y, z]            accel sensor measurements
VectorInt16 aaReal;     // [x, y, z]            gravity-free accel sensor measurements
VectorInt16 aaWorld;    // [x, y, z]            world-frame accel sensor measurements
VectorFloat gravity;    // [x, y, z]            gravity vector
float euler[3];         // [psi, theta, phi]    Euler angle container
float ypr[3];           // [yaw, pitch, roll]   yaw/pitch/roll container and gravity vector

volatile bool mpuInterrupt = false;  // indicates whether MPU interrupt pin has gone high
float yaw;
float yawOffset = 0;
float pitch;
float pitchOffset = 0;
float roll;
float rollOffset = 0;
boolean repeat = false;


int encoder = 3;
int count = 0;
float distance;
boolean displayControl = true;
boolean countControl = true;

LiquidCrystal lcd(8, 9, 4, 5, 6, 7);

void dmpDataReady()
```

```
{
  mpuInterrupt = true;
}


void setup()
{
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
  Wire.begin();
  Wire.setClock(400000); // 400kHz I2C clock. Comment this line if having compilation difficulties
#elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
  Fastwire::setup(400, true);
#endif


  // initialize serial communication
  // (115200 chosen because it is required for Teapot Demo output, but it's
  // really up to you depending on your project)
  Serial.begin(115200);
  while (!Serial); // wait for Leonardo enumeration, others continue immediately


  // NOTE: 8MHz or slower host processors, like the Teensy 3.3V or Arduino
  // Pro Mini running at 3.3V, cannot handle this baud rate reliably due to
  // the baud timing being too misaligned with processor ticks. You must use
  // 38400 or slower in these cases, or use some kind of external separate
  // crystal solution for the UART timer.
```

```
  // initialize device
  //Serial.println(F("Initializing I2C devices..."));
  mpu.initialize();
  pinMode(INTERRUPT_PIN, INPUT);

  // verify connection
  //Serial.println(F("Testing device connections..."));
  //Serial.println(mpu.testConnection() ? F("MPU6050
connection successful") : F("MPU6050 connection failed"));

  // load and configure the DMP
  //Serial.println(F("Initializing DMP..."));
  devStatus = mpu.dmpInitialize();

  // supply your own gyro offsets here, scaled for min
sensitivity
  mpu.setXGyroOffset(220);
  mpu.setYGyroOffset(76);
  mpu.setZGyroOffset(-85);
  mpu.setZAccelOffset(1788); // 1688 factory default for my
test chip

  // make sure it worked (returns 0 if so)
  if (devStatus == 0) {
    // turn on the DMP, now that it's ready
    // Serial.println(F("Enabling DMP..."));
    mpu.setDMPEnabled(true);

    // enable Arduino interrupt detection
```

```
    //Serial.println(F("Enabling interrupt detection (Arduino external interrupt 0)..."));
    attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN), dmpDataReady, RISING);
    mpuIntStatus = mpu.getIntStatus();


    // set our DMP Ready flag so the main loop() function knows it's okay to use it
    //Serial.println(F("DMP ready! Waiting for first interrupt..."));
    dmpReady = true;


    // get expected DMP packet size for later comparison
    packetSize = mpu.dmpGetFIFOPacketSize();
  } else {
    // ERROR!
    // 1 = initial memory load failed
    // 2 = DMP configuration updates failed
    // (if it's going to break, usually the code will be 1)
    //Serial.print(F("DMP Initialization failed (code "));
    //Serial.print(devStatus);
    //Serial.println(F(")"));
  }


  // configure LED for output
  attachInterrupt (digitalPinToInterrupt(encoder), ISRencoder, RISING); // Interrupt to calculate number of counts detected by motor encoder
  pinMode(LED_PIN, OUTPUT);
  lcd.begin (16, 2);
```

```
  lcd.setCursor (0, 0);
  lcd.print ("Starting in...");
  while (millis () < 10000)
  {
    Serial.println ('d');
    lcd.setCursor(14, 1);
    lcd.print (int (20000 - millis()) / 1000);
  }
  lcd.clear ();
  lcd.setCursor (0, 0);
  lcd.print ("Starting in..."); // Pause for 20 seconds to allow MPU 6050 reading to stailize
  while (millis () < 20000)
  {
    Serial.println ('d');
    lcd.setCursor(15, 1);
    lcd.print (int (20000 - millis()) / 1000);
  }
  MPU (0); // after it has stabilized, set the first readings as offset to initialize the yaw, pitch and roll to 0
  yawOffset = yaw;
  pitchOffset = pitch;
  rollOffset = roll;
}


void loop()
{
  // put your main code here, to run repeatedly:
  /* while (pitch < 5)
  {
```

```
    Serial.println ('e');
  } */
  int startDistance = distance; // set the current distance as reference distance
  while (distance - startDistance < 20) // follow line with PID until the difference between the current distance and reference distance is more than 20cm
  {
    Serial.println ('e');
  }
  while (pitch < 20) // go straight until MPU 6050 senses the pitch is more than 20 degrees
  {
    MPU (0);
    forward (1);
  }
  forward(300); // move forward for 300 milliseconds
  displayControl = false; // stop the screen from displaying total distance and total time
  lcd.clear(); // clear LCD screen
  lcd.setCursor(0, 0);
  lcd.print("Ramp Angle:"); // print ramp angle
  MPU (2);
  freeze (500); // stop for 500 milliseconds
  forward (1800); // move forward for 1800 milliseconds
  freeze (3000); // stop for 3000 milliseconds
  countControl = false; // stop counting total distance starting from now
  turnLeft (2000); // turn left for 2000 milliseconds
  freeze (500); // stop for 500 milliseconds
  countControl = true; // start counting total distance whenever it moves
```

```
  lcd.clear();
  displayControl = true; // start displaying total distance and total time
  startDistance = distance;
  while (distance - startDistance < 30) // follow line for 30 cm with PID
  {
    Serial.println ('e');
  }
  startDistance = distance;
  displayControl = false; // stop displaying total distance and total time
  lcd.clear();
  lcd.setCursor (0, 0);
  lcd.print ("Count Distance:");
  while (distance - startDistance < 130) // follow line for 130 cm with PID
  {
    lcd.setCursor (13, 1);
    lcd.print (distance - startDistance);
    Serial.println ('e');
  }
  unsigned long int refertime = millis ();
  while (millis () - refertime <= 3000) // pause and displaying timer up to 3000 milliseconds
  {
  lcd.setCursor (0,1);
  lcd.print ("Time: ");
  lcd.print (millis() - refertime);
  freeze (1);
  }
```

```
  lcd.clear ();
  displayControl = true;
  while (distance < 1430) // follow line with PID until the total distance is no more than 1430 cm
  {
    Serial.println ('e');
  }
  while (true) // stop forever
  {
    freeze (1);
  }
}


void MPU (int printnumber) // This function resets the MPU 6050 when FIFO overflows
{
  do
  {
    if (!dmpReady) return;
    mpuInterrupt = false;
    mpuIntStatus = mpu.getIntStatus();


    // get current FIFO count
    fifoCount = mpu.getFIFOCount();


    // check for overflow (this should never happen unless our code is too inefficient)
    if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
      // reset so we can continue cleanly
      mpu.resetFIFO();
```

```
      //Serial.println(F("FIFO overflow!"));
      repeat =  true;

      // otherwise, check for DMP data ready interrupt (this should happen frequently)
    } else if (mpuIntStatus & 0x02) {
      // wait for correct available data length, should be a VERY short wait
      while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

      // read a packet from FIFO
      mpu.getFIFOBytes(fifoBuffer, packetSize);

      // track FIFO count here in case there is > 1 packet available
      // (this lets us immediately read more without waiting for an interrupt)
      fifoCount -= packetSize;
      repeat = false;
#ifdef OUTPUT_READABLE_YAWPITCHROLL
      mpu.dmpGetQuaternion(&q, fifoBuffer);
      mpu.dmpGetGravity(&gravity, &q);
      mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
      //Serial.print("ypr\t");
      //Serial.print(ypr[0] * 180/M_PI);
      //Serial.print("\t");
      //Serial.print(ypr[1] * 180/M_PI);
      //Serial.print("\t");
      //Serial.println(ypr[2] * 180/M_PI);
      if (ypr[0] < 0)
```

```
      {
        ypr[0] += 2 * M_PI;
      }
      yaw = ypr [0] * 180 / M_PI - yawOffset;
      pitch = ypr[1] * 180 / M_PI - pitchOffset;
      roll = ypr [2] * 180 / M_PI - rollOffset;
      //Serial.println (pitch);
      switch (printnumber)
      {
        case 1:
          lcd.print (yaw);
          break;

        case 2:
          lcd.print (pitch);
          break;

        case 3:
          lcd.print (roll);
          break;

        default:
          break;
      }
#endif
    }
  } while (repeat == true);
}
```

```
void forward (int duration) // this functions allows Hercules to move forward for whatever duration inputed in the argument
{
 unsigned long int before = millis ();
  while (millis() - before < duration)
  {
    Serial.println ('a');
  }
}

void turnLeft (int duration) // this functions allows Hercules to turn left for whatever duration inputed in the argument
{
  unsigned long int before = millis ();
  while (millis () - before < duration)
  {
    Serial.println ('c');
  }
}

void freeze (int duration) // this functions allows Hercules to stop for whatever duration inputed in the argument
{
  unsigned long int before = millis ();
  while (millis() - before < duration)
  {
    Serial.println ('d');
  }
}
```

```
void ISRencoder (void)
{
  if (countControl == true) // count total distance when countControl is set to TRUE
  {
    count++;
    distance = (count * 28.3) / 36;
  }
  if (displayControl == true) // display and total distance and total time since program starts when displayControl is set to TRUE
  {
   lcd.setCursor (0,0);
   lcd.print ("Distance: ");
   lcd.print (distance);
   lcd.setCursor (0,1);
   lcd.print ("Time: ");
   lcd.print (millis() / 1000);
  }
}
```

We faced a lot of challenges while making the codes.

There is an anomaly of the last sensor in the line follower sensor module. To overcome that, we used the constrain() function to limit the error calculated to a maximum and minimum value, so that it ensures the symmetry of the error calculated in the right and left sides of the sensor module.

The ‘int’ data type burst when we used in timer function. To prevent that, we used ‘unsigned long int’ to have a bigger data size.