Introduction to Robotics with the Raspberry Pi

INTRODUCTION TO PID CONTROL

Jeff Cicolani
Sunday, July 23rd, 2017

This workshop is assuming a you are using Windows. If you are using a Mac or Linux machine, good luck… I mean, you will need to look up the proper instructions for your OS. Sorry, I'm a Windows guy, now with a little Linux, but Windows is my got to OS for general use.

Introduction

So far, we've been working with large generic blocks of information; Python, sensors, motor control, etc. Last week we talked about using IR proximity sensors for detecting a line. This week we will delve into a very specific control algorithm. The PID controller is one of the most widely used control loops because of its versatility and simplicity. And we'll try to do it without getting math heavy.

Control Loops

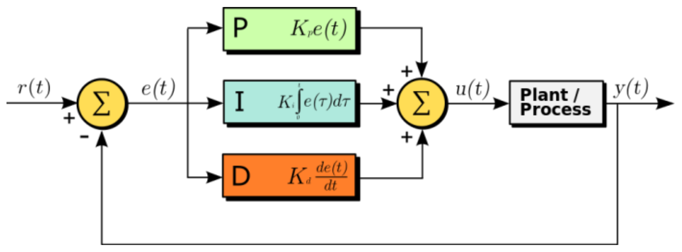
The PID Controller is a member of a group of algorithms called control loops. The purpose of a control loop is to use input from a measured process to make changes to a control, or controls, to compensate for discrepancies between the current state and a desired state. There are many different types of control loops, and is, in fact, a whole area of study called control theory.

For our purposes, this week, we really only care about one; proportional, integral, and derivative or PID.

Proportional, Integral, and Derivative Control

From Wikipedia:

> A PID controller continuously calculates an error value e(t) as the difference between a desired setpoint and a measured process variable and applies a correction based on proportional, integral, and derivative terms. PID is an initialism for Proportional-Integral-Derivative, referring to the three terms operating on the error signal to produce a control signal.

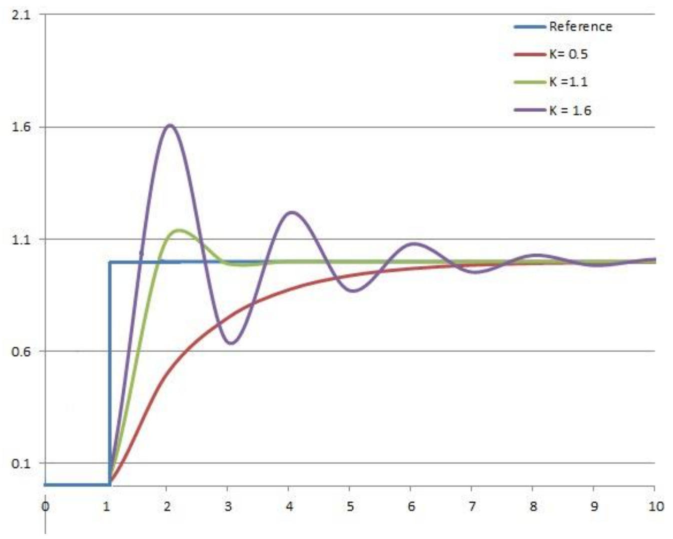


The purpose of the controller is to apply incremental adjustments to some output in order to achieve the desired result. In our application, we will be using the feedback from IR sensors to apply changes to our motors. The desired behavior is a robot that will keep a line centered while the robot moves forward. However, this process can be used with any sensors and outputs. For instance, PID is used in multirotor platforms to maintain level and stability.

As the name implies, the PID algorithm actually consists of three parts; proportional, integral, and derivative. Each part, in itself, is a type of control. However, if used independently, the resulting behavior would be erratic and difficult to predict.

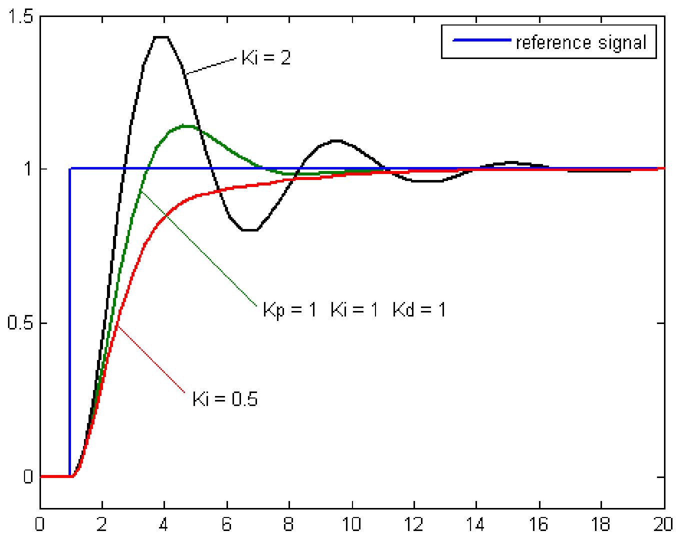
Proportional Control

In this control method, the amount of change is set based entirely on the size of the error. The larger the error, the more change is applied. By itself, a purely proportional control would reach a zero-error state, but will have difficulty dealing with drastic changes and would result in heavy oscillation.



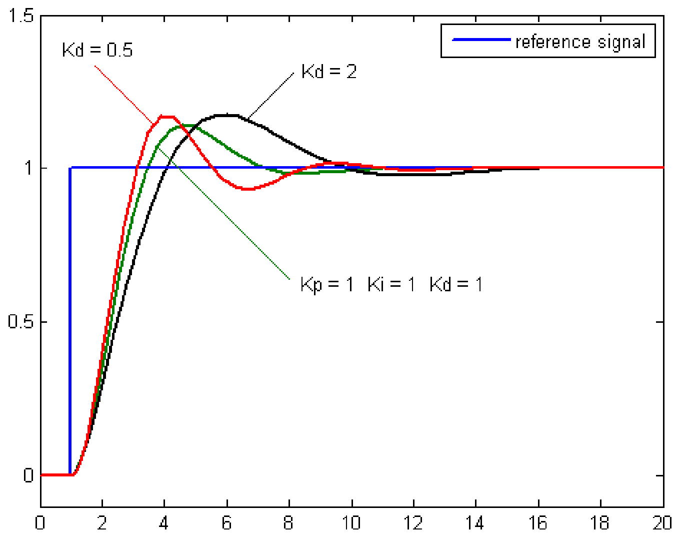
Integral Control

Integral control considers not only the error, but also the time it has persisted. The amount of change applied to compensate for the error will increase over time. A purely integral control could bring the device to a zero-error state, but it will react slowly and, again, tends toward over compensation and oscillation.



Derivative Control

Derivative control does not consider the error and, therefore, can never bring the device to a zero-error state. What it does, however, is try to reduce the change in error to zero. If too much compensation is applied, the algorithm will overshoot. It will then apply another correction, and this process will continue in this manner, producing a pattern of constant, increasing, or decreasing corrections. Whereas a state of decreasing oscillation is considered "stable", the algorithm will never reach a truly zero-error state.



Bringing Them Together

The PID controller is, simply, the sum of these three methods. By bringing the three of them together, the algorithm aims to produce a smooth correcting process which will bring the error to zero. Time for a little bit of math…

Let's start by defining some variables.

e(t) is the error in time where (t) is time or instantaneous time, a.k.a. the present.

K_p is a parameter representing the proportional gain. When we start coding, this will be our proportional variable.

K_i is the integral gain parameter. It will also become a variable.

K_d is the derivative gain parameter. And you guessed it, yet another future variable.

τ represents integrated values over time. We'll get to that.

The proportional term is basically the current error multiplied by the K_p value.

$$P_{out} = K_p e(t)$$

The integral portion is a bit more complicated because it takes into account all of the error that has happened. It is the sum of the error over time and the accumulated correction.

$$I_{out} = K_i \int_0^t e(\tau) d\tau$$

The derivative term is the difference between the original error and current error over time, then multiplied by our derivative parameter.

$$D_{out} = K_d \frac{de(t)}{dt}$$

So, to bring it all together, our PID equation looks like this:

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

Alright, that's it for the math. The next time you see these equations we'll be converting them over to code. And that's next.

Implementing the PID Controller

Why did we need to go through all of that math?

Because it is important to understand what is happening inside the equation. There are three parameters we will be adjusting to fine tune your PID controller. By understanding how these parameters are used, you will be able to determine which ones need adjusting when.

The Plan

In order to implement the controller, we need to know a couple of things. First, what is our desired outcome? Second, what is our input?

Our goal for this project is to create a simple robot that will follow a line. The robot consists of two drive motors that work in tandem to drive and steer the robot. It is also outfitted with four simple IR detectors that are tuned to indicate reflectance from the surface. When the input from the sensor is high, it is over a light area, when it is low, it is over a dark area. The four IR sensors are arrayed in a straight line across the front of the robot.

Given this description (which should sound very familiar), our outputs are the motors. More accurately, our output is the difference in speed between the two motors. Our inputs are the IR sensors which will be used to set an error value. When an outer sensor is over a dark area (the line), the error will be twice that of the inner sensors. In this way, we'll know the robot is a little off center, or a lot off center. Also, the left two sensors will have a negative value and the right side a positive value, so we will know which direction we are off.

The Code

In order to put this together in a logical format, we'll want to break down the code into functions. If you're feeling particularly ambitious, you may want to create a PID class. For the workshop, however, I'll just be putting together a simple program with a PID function.

Raspberry Pi Code

1. Open a terminal window. You can use the same one we were using for the installation, but as long as Idle is open, this terminal window will be locked.
2. Type

```
sudo idle
```

3. In the Idle IDE, create a new file and save it as "PID.py"
4. We'll start with importing the necessary libraries. Enter the following code:

```
from Adafruit_MotorHAT import Adafruit_MotorHAT as amhat
from Adafruit_MotorHAT import Adafruit_DCMotor as adcm
import time
import serial
```

5. Next, we'll create some variables and initialize the serial port for the Arduino and the motors:

```
long lastTime
int output, target
int error, sumError, lastError
int sensorErr
int kp, ki, kd
int speed, leftSpeed, rightSpeed, leftAdjust, rightAdjust

# create 2 motor objects
mh = amhat(addr=0x60)

motor1 = mh.getMotor(1)
motor2 = mh.getMotor(2)

ser = serial.Serial('/dev/ttyACM0',9600)
```

6. And now, the PID function

```
def PID(err):
        # get the current time
        now = int(round(time.time()*1000)
        duration = now-lastTime

        # calculate the error values
        error = target-err
        sumError += (error * duration)
        dErr = (error – lastError) / duration

        # calculate PID
        output = kp * error + ki * sumErr + kd *dErr

        # update variables
        lastError = error
        lastTime = now

        # return PID value
        return output
```

7. Create a drive function:

```
def drive (maxSpeed, pid)

        ratio = pid

        # if the ratio is negative, reduce left speed
        if ratio < 0: leftAdjust = (maxSpeed/2) + ratio
        # if the ratio is positive, reduce right speed
        elif ratio > 0: rightAdjust = (maxSpeed/2) – ratio
        # otherwise full speed ahead
        else:
                leftAdjust = 0
                rightAdjust = 0

        # set the speed of each motor
        leftSpeed = (maxSpeed/2) + leftAdjust
        rightSpeed = (maxSpeed/2) + rightAdjust

        motor1.setSpeed(leftSpeed)
        motor2.setSpeed(rightSpeed)

        # run the motors
        motor1.run(amhat.FORWARD)
        motor2.run(amhat. FORWARD)
```

8. Create the main program:

```
# set start speed
motor1.setSpeed(0)
motor2.setSpeed(0)

# these are starting values. You'll want to change these.
kp = 1
ki = 1
kd = 1

try:
        while True:
                serial_line = ser.readline()
                sensorErr = int(serial_line)

                pid = PID(sensorErr)
                drive(128, pid)
```

```
except KeyboardInterrupt:
        ser.close()
```

Arduino Code

Now that we have the drive program, we need to create the sensor program for the Arduino. We will assume sensor 1 is on the far left of the robot and sensor 4 is on the far right. This code should also look very familiar, since it is a slightly modified version of what we did last week.

```
int ir1 = 0;
int ir2 = 0;
int ir3 = 0;
int ir4 = 0;
int err = 0;

const int ir1pin = 3;
const int ir2pin = 4;
const int ir3pin = 5;
const int ir4pin = 6;

void setup() {
        pinMode(ir1pin, INPUT);
        pinMode(ir2pin, INPUT);
        pinMode(ir3pin, INPUT);
        pinMode(ir4pin, INPUT);

        Serial.begin(9600);
}

void loop() {
        ir1 = digitalRead(ir1pin);
        ir2 = digitalRead(ir2pin);
        ir3 = digitalRead(ir3pin);
        ir4 = digitalRead(ir4pin);

        if(ir1 == 1) {err = -2;}
        else if(ir2 == 1) {err = -1;}
        else if(ir3 == 1) {err = 1;}
        else if(ir4 == 1) {err = 2;}
        else {err = 0;}

        Serial.println(err)
}
```

Explore

A lot of work can go into fine tuning a PID controller. The results, however, are worth it. As you move forward in robotics you'll find yourself relying on this, or a variation of it, regularly.

The code we're working with here is very basic. There are a lot of things that need to be done to make this code more useful. For instance, it is using integers for its variables. You will be able to get better results by using floats. The motor control code assumes a constant forward movement. Play around with the code. See what you can do to make it more efficient.

Hint: This is very unlikely to work well, especially for the upcoming competition.

References and Resources:

https://en.wikipedia.org/wiki/PID_controller

https://en.wikipedia.org/wiki/Control_theory

http://www.ni.com/white-paper/3782/en/

http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/

And for those of you who want to shortcut this whole thing:

http://code.activestate.com/recipes/577231-discrete-pid-controller/

https://pypi.python.org/pypi/pypid/