Arduino PID Example Lab

Bret Comnes & A. La Rosa

1. Introduction to PID

PID (Proportional, Integral, Differential) is a control algorithm that tries to compensate for characteristics in your system. There are three primary components to think about in a PID control loop. Each component is prefixed with a gain constant, and when added together, give you the instantaneous control value that you use to drive your system. Typically, you are generating a voltage to control your system, so each component can be thought of as contributing a particular voltage to your final output.

You will have voltage corresponding to the current state of your system (position, temperature, etc) that is called your ``Process Variable" or ***PV***. The ***PV*** is the value you pass to your PID control loop to tell it about the state of the system. You also have a setpoint (***SP***) voltage, corresponding to the state you wish your ***PV*** to reach. Basically, you want your PID loop to drive your system so that ***SP*** and ***PV*** are equal. Third, you have a

control voltage, ***u***,

which corresponds to the instantaneous voltage value you are using to drive your system towards its ***SP*** voltage. Your control voltage ***u*** can be thought of as what is actually sent to the system to steer it where you want it to go. It's analogous to a gas pedal that your control loop is controlling.

The PID algorithm is show in Equation (1.1).

$$u(t) = MV(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t) \qquad (1.1)$$

There is a proportional, integral and differential part to Equation (1.1). The constants K_p, K_i, and K_d are used to set the sign and contribution gain of each part of this equation.

$e(t)$ is your proportional "error" corresponding to $SP - PV$.

The variable t corresponds to the current time in our system, and τ is simply a variable of integration.

The proportional portion of the equation takes into account how far away our PV is from our SP. The differential part takes into account how fast we are moving (if we move to fast near our SP, we will over shoot), and can be used to reduce the proportional portion if we are moving to fast, or speed us up if we are experiencing resistance despite our proportional contribution.

The integral part of the equation takes into account how long we have been off of the set point, contributing more to our output the longer we are missing the *SP*. This is important because our **P** and **D** contributions will typically lead our *PV* to sag slightly above or below our *SP* variable. (PID controller, 2013)

2. Controlling an LED with PID

2.1. Installing Arduino Libraries

Writing our own PID control loop isn't that hard, but there are a lot of details to take into account. (Beauregard, 2013) A good place to start is with an existing library, a set of tools someone has written for us to easily grab and use. There is no sense in re-inventing the wheel if we don't have to, and we can always go back and attempt to implement our own algorithm if we wish later. A decent place to start with an Arduino library is the libraries documentation. (Beauregard, PIDLibrary, 2013). This page will explain how to use the library, and is a good reference if you ever get stuck. We first have to install the library.

First download the libarary. You can get it from https://github.com/br3ttb/Arduino-PID-Library/archive/master.zip or find someone in class who already has a copy and grab if off of a thumb drive.

The next step varies from system to system so take your pick:

2.1.1. Windows

You can unzip the library to %userprofile%\documents\Arduino\libraries Make sure to grab the folder inside the folder that it is zipped in. You might have to change any "-" characters in the folder name to an underscore or else the Arduino IDE yells at you. The final path should look something like: %userprofile%\documents\Arduino\libraries\PID_v1

2.1.2. OS X

In OS X you can put the folder inside the zip into: ~/Documents/Arduino/libraries

2.2. Using the PID Library

To use a library in code, simply go to the **Toolbar -> Sketch -> Import Library -> PID_v1**. This should insert the following code at the top of your program:

```
#include <PID_v1.h>
```

This enables us to use the code discussed on the PID documentation website. (Beauregard, PIDLibrary, 2013)

2.3. The code

```
#include <PID_v1.h>
const int  photores = A0;        // Photo resistor input
const int  pot = A1;             // Potentiometer input
const int  led = 9;              // LED output
double  lightLevel;              // variable that stores the incoming light
                                    level

// Tuning parameters
float Kp=0;                      //Initial Proportional Gain
float Ki=10;                     //Initial Integral Gain
float Kd=0;                      //Initial Differential Gain

double Setpoint, Input, Output;  //These are just variables for storingvalues
PID myPID(&Input, &Output, &Setpoint, Kp, Ki, Kd, DIRECT);
                                 // This sets up our PDID Loop
                                 //Input is our PV
                                 //Output is our u(t)
                                 //Setpoint is our SP
const int sampleRate = 1; // Variable that determines how fast our PID loop
runs

// Communication setup
const long serialPing = 500;     //This determines how often we ping our loop
// Serial pingback interval in milliseconds
unsigned long now = 0;           //This variable is used to keep track of time
// placehodler for current timestamp
unsigned long lastMessage = 0;   //This keeps track of when our loop last
spoke to serial
// last message timestamp.

void setup(){
  lightLevel = analogRead(photores);          //Read in light level
  Input = map(lightLevel, 0, 1024, 0, 255);   //Change read scale to analog
                                                out scale
  Setpoint = map(analogRead(pot), 0, 1024, 0, 255);
                                              //get our setpoint from our pot
  Serial.begin(9600);                         //Start a serial session
  myPID.SetMode(AUTOMATIC);                   //Turn on the PID loop
  myPID.SetSampleTime(sampleRate);            //Sets the sample rate

  Serial.println("Begin");                    // Hello World!
  lastMessage = millis();                     // timestamp
}

void loop(){
  Setpoint = map(analogRead(pot), 0, 1024, 0, 255); //Read our setpoint
```

```
  lightLevel = analogRead(photores);          //Get the light level
  Input = map(lightLevel, 0, 900, 0, 255);    //Map it to the right scale
  myPID.Compute();                            //Run the PID loop
  analogWrite(led, Output);                   //Write out the output from the
                                                PID loop to our LED pin

  now = millis();                             //Keep track of time
  if(now - lastMessage > serialPing) {  //If it has been long enough give us
                                          some info on serial
    // this should execute less frequently
    // send a message back to the mother ship
    Serial.print("Setpoint = ");
    Serial.print(Setpoint);
    Serial.print(" Input = ");
    Serial.print(Input);
    Serial.print(" Output = ");
    Serial.print(Output);
    Serial.print("\n");
    if (Serial.available() > 0) { //If we sent the program a command deal
                                    with it
      for (int x = 0; x < 4; x++) {
        switch (x) {
          case 0:
            Kp = Serial.parseFloat();
            break;
          case 1:
            Ki = Serial.parseFloat();
            break;
          case 2:
            Kd = Serial.parseFloat();
            break;
          case 3:
            for (int y = Serial.available(); y == 0; y--) {
              Serial.read();  //Clear out any residual junk
            }
            break;
        }
      }
      Serial.print(" Kp,Ki,Kd = ");
      Serial.print(Kp);
      Serial.print(",");
      Serial.print(Ki);
      Serial.print(",");
      Serial.println(Kd);  //Let us know what we just received
      myPID.SetTunings(Kp, Ki, Kd); //Set the PID gain constants and start
running
    }

    lastMessage = now;
    //update the time stamp.
  }

}
```

2.4. Interpreting the code

You can get a copy of the code from:
https://github.com/bcomnes/315-lab-microcontroller/blob/master/code/pid_led_set_serial/pid_led_set_serial.ino

You should have a good idea of what is going on at this point, but to run through some new concepts, lets start with:

```
PID myPID(&Input, &Output, &Setpoint, Kp, Ki, Kd, DIRECT);
```

This line set up the PID library to use a PID process called myPID. We could have called it Arnold if we wanted. Its just a name. Any time we call

```
myPID.Compute();
```

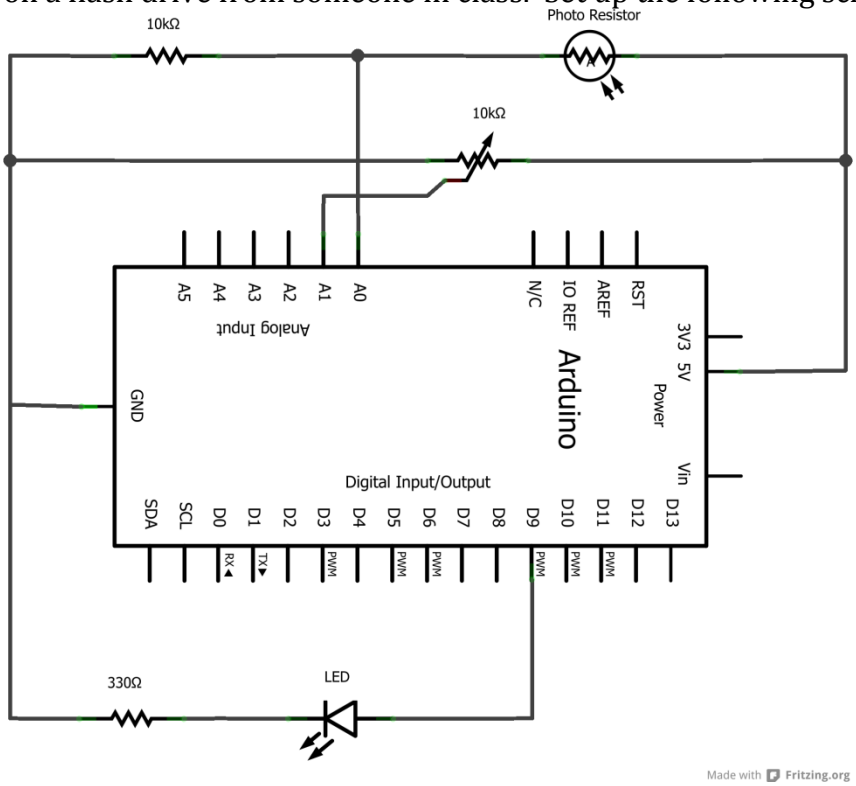
subsequently, myPID will take the variables `Input, Setpoint, Kp, Ki,` and `Kd,` and use them in its calculation, and then write whatever it determines to `Output.` `DIRECT` simply tells our PID which direction we are working in. If for example, the PID loop tries to make the LED dimmer when it should actually get brighter, we would change `DIRECT` to `REVERSE.`

The other weird thing that is going on is that we can't poll our serial line too often, or else the PID loop would not work very well. It would run to slowly. That is why we use time stamps to keep track of how long ago we provided some serial line feedback and decide whether to wait, or to write what our PID loop is doing.

The third weird thing that we are doing, is that we check to see if we have sent any commands to our Arduino every time we write to the serial line. I set up a very simple command parser that will take 3, comma separated float variables and set our `Kp, Ki, Kd` respectively with the new values. This lets us play with the tuning without having to re-flash our Arduino every time we want to change the tuning values.

2.5. Setting up the demo

Either type in the above code, or download it from the website or get a copy on a flash drive from someone in class. Set up the following schematic:



You should place your photo resistor so that it aims into the LED output, or attach your LED to a wire so you can vary how close it is to the photo resistor. Our initial values for our gains is 0 for Kp and Kd, and 10 for Ki. This make sense since our LED needs a constant voltage, and the Ki will turn itself up till it reaches 255 or the LED matches the desired set point.

With the LED close to your photo resistor, vary the potentiometer. This should increase and decrease the brightness of the LED. If you move the LED away, the PID loop should try to compensate and turn up the brightness to match the set point. If the set point is to low, the ambient light will be enough to exceed the set point. Since our system has no way to dim the room lights, it will simply settle for turning off altogether.

In the serial monitor, send in a command of "0,0.5,0" without the quotes. The Arduino should parse it, and set Ki to 0.5 instead of 10. **How does this change how fast the PID loop reacts to environmental changes or changes in set points?**

You are free to play with different values of Kp and Kd, but due to the quick response time and nature of an LED, these variable are best left set to 0.

3. PID Temperature Control (Extra!)

If you wish, try using a temperature sensor instead of your photo resistor. Instead of controlling an LED, use a fan. Be sure to use an NPN transistor to power the fan however, as the Arduino does not have enough current to drive it, and may shut down on you. You can drive the base of the transistor with the same PWM signal you used for the LED, but be sure to REVERSE the direction of the PID loop.

ABSOLUTELY DO NOT APPLY MORE THAN 5V TO THE ARDINO. Also, you shouldn't need more than about 0.2A to drive the fan.

Bibliography

PID controller. (2013, February 19). Retrieved February 2, 2013, from Wikipedia: http://en.wikipedia.org/wiki/PID_control

Beauregard, B. (2013). *Improving the Beginner's PID – Introduction*. Retrieved February 19, 2013, from brettbeauregard Project Blog: http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/

Beauregard, B. (2013). *PIDLibrary*. Retrieved from Arduino Playground: http://playground.arduino.cc/Code/PIDLibrary

Colville, M. (n.d.). Process Control with a Microcontroller: PWM output, PID control, and hardware implementation. *2012*. Portland, OR: Portland State University.