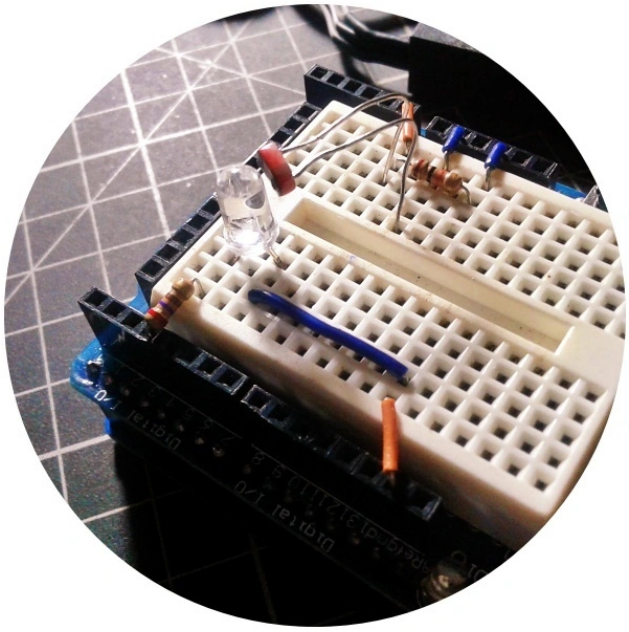
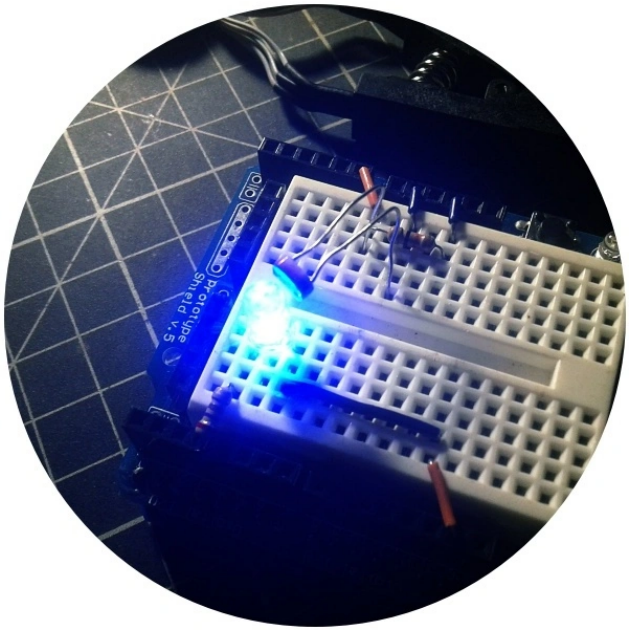
PID and Arduino Primer





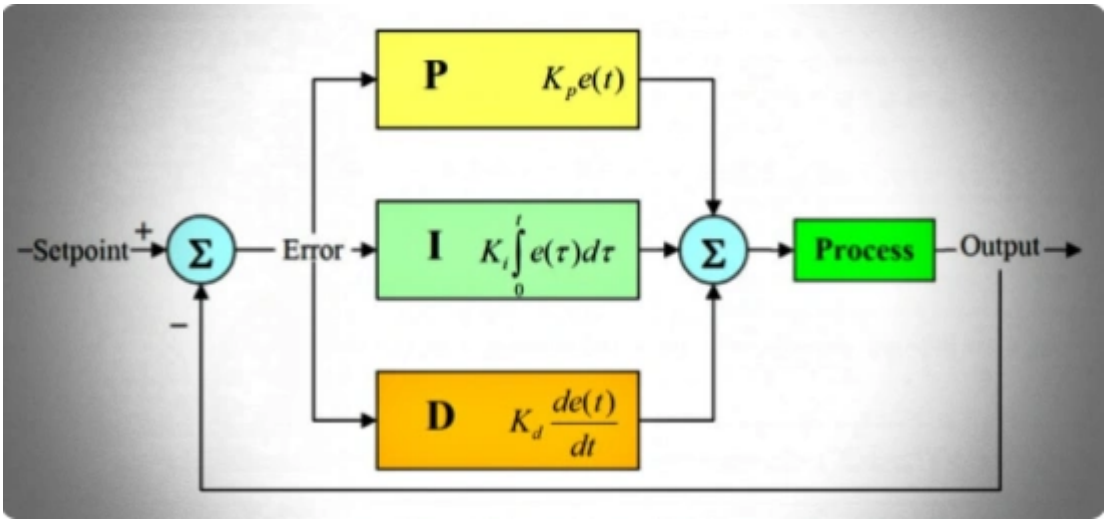
T.K. Hareendran

A few weeks ago, I installed an automatic water heater switch to my bathroom, but it was a crude device. What I really want to do is design a universal heater driver. That basic idea can be adapted to other applications. One of the applications is an intelligent swimming pool pump and heater switch, which needs maximum efficiency and safety. After a lot of searching and digging I got an idea. It was the creation of a cheap PID controller.

Within days I put together a PID controller for my swimming pool project, and it was a complex analog build. Unfortunately, things do not always go smoothly on the first try. After hours of operation, I noticed some defects in the pool heater driver circuit. It needs more testing. So, right now I am not going to post the 'swimming pool' project details, but I do want to share my random thoughts on PID controllers. I have learned a lot lately!

PID Controller – Basic Intro

A PID (Proportional-Integral-Derivative) controller is a control loop mechanism employing feedback that is widely used in industrial control systems and a variety of other applications requiring continuously modulated control. A PID controller continuously calculates an *error value* as the difference between a desired setpoint (SP) and a measured process variable (PV) and applies a correction based on proportional, integral, and derivative terms (denoted *P*,*I*, and *D* respectively), hence the name. In practice it automatically applies accurate and responsive correction to a control function. Wiki Read – https://en.wikipedia.org/wiki/PID_controller , https://en.wikipedia.org/wiki/PID_controller#Limitations_of_PID_control



The vast majority of industrial control loops utilize some combination of PID control. Remember, the human brain is the most prolific loop controller in existence. Every time we modify our behaviour based on a previous result; we create a control loop. We

continually frame new control loops and tweak old ones to govern our experience!

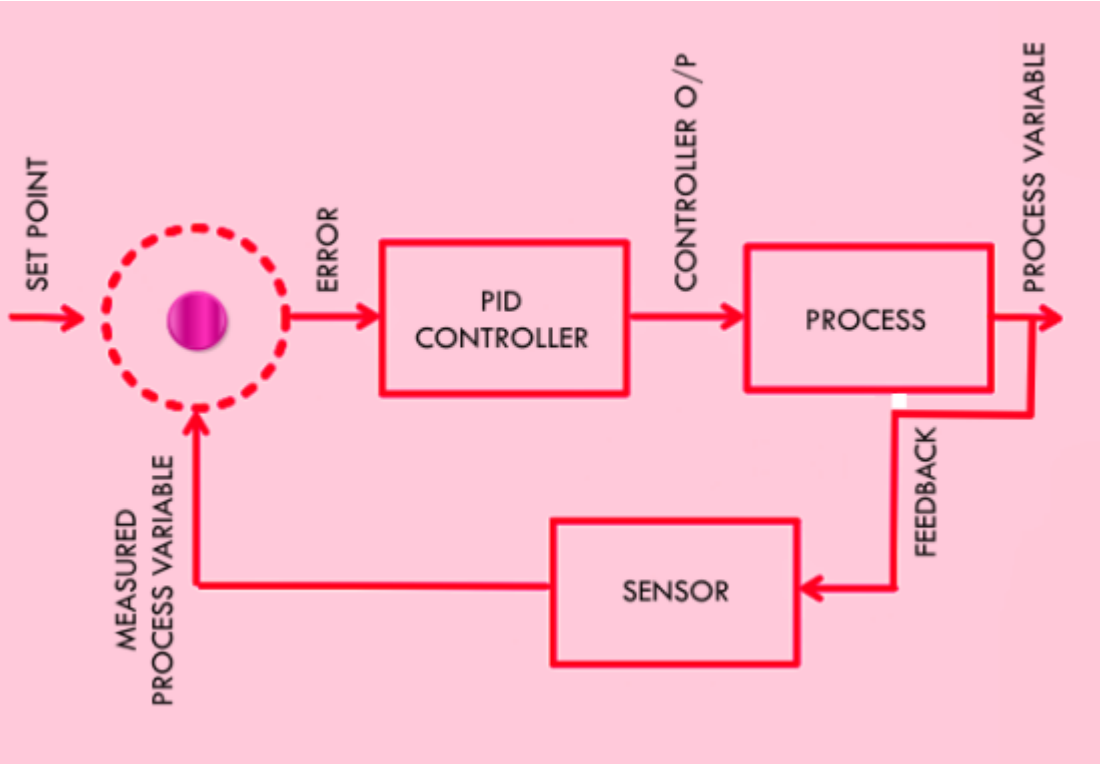
PID Process – Quick Breakdown

The proportional algorithm, the first ingredient in the control loop, is generally the most impactful and crucial of the three. The “P” component is responsible for adjusting the control variable proportionally to the amount of calculated error in the process. The integral is the aggregate of all recorded values, from the time counting begins until the time counting stops. The “I” component records all errors above or below zero error (compared to the setpoint) and continually strives to reduce it to zero or near zero. So, if the proportional is the present-error correction and the integral is the past-error correction, then the derivative is the future-error correction. The “D” component is the most complicated and is usually unnecessary in most applications. In fact, most proportional controllers employ a PI control loop only. Also note that these three basic coefficients are varied in each PID controller for specific application in order to get optimal response.

I had a go at explaining PID in detail at this point but it turned out to be far too long. Because I don’t want to bore you, I’ll see if I can add that up later in case it’s holding enough attention. Anyway, just note down that a PID controller is a control loop feedback mechanism commonly used in industrial control systems. A PID controller continuously

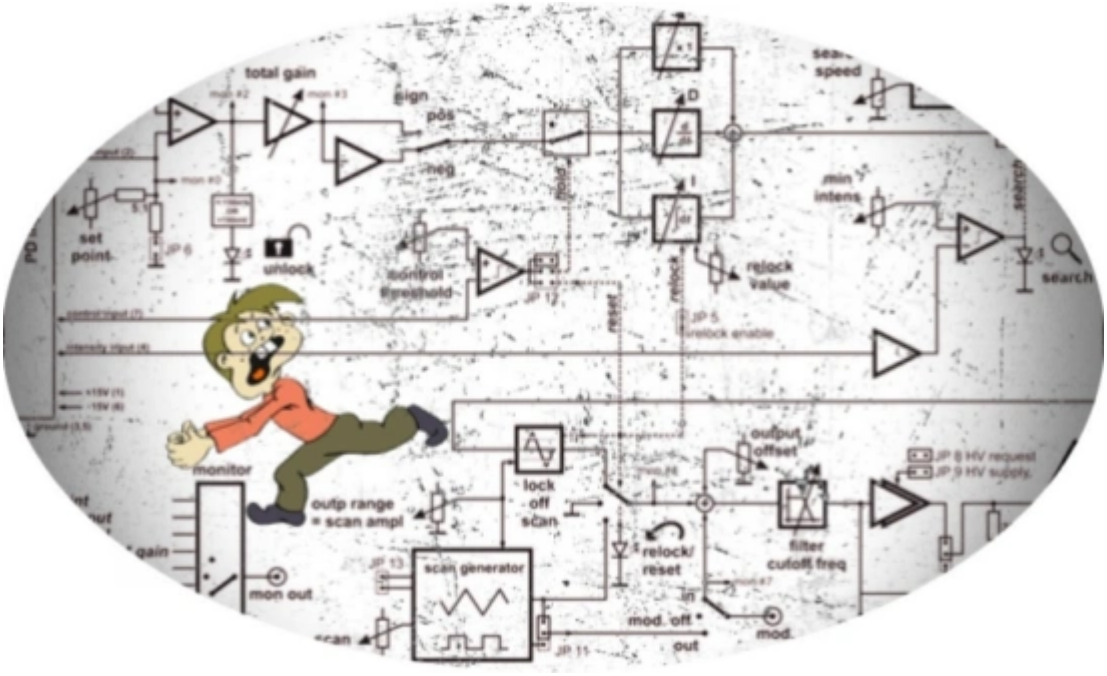
calculates an error value as the difference between a desired setpoint and a measured process variable. The controller attempts to minimize the error over time by adjustment of a control variable.

Consider the typical control system shown in below figure in which the process variable of a process has to be maintained at a particular level.



Assume that the process variable is temperature, and to measure the process variable a temperature sensor is used. The process has to be maintained at 60°C (set point), and the measured value from the temperature sensor is 40°C (process variable). This deviation of actual value from the set point (set point is the desired response of the process) in the PID control algorithm causes to produce the output to the actuator (let us say a heating coil) depending on the combination of proportional, integral and derivative responses. So, the PID controller continuously varies the output to the actuator till the process variable settle down to the set point.

In a nutshell, a PID controller is a combination of proportional (P), Integral (I), and Derivative (D) responses. In this “closed loop feedback system” the P-controller generates the control output proportional to the current error. The I-controller is primarily used to reduce the steady state error of the system. The D-controller sees how fast process variable changes per unit of time and produce the output proportional to the rate of change. In most controllers, this response depends only on process variable, rather than error. This avoids spikes in the output in case of rapid set point change by the operator.

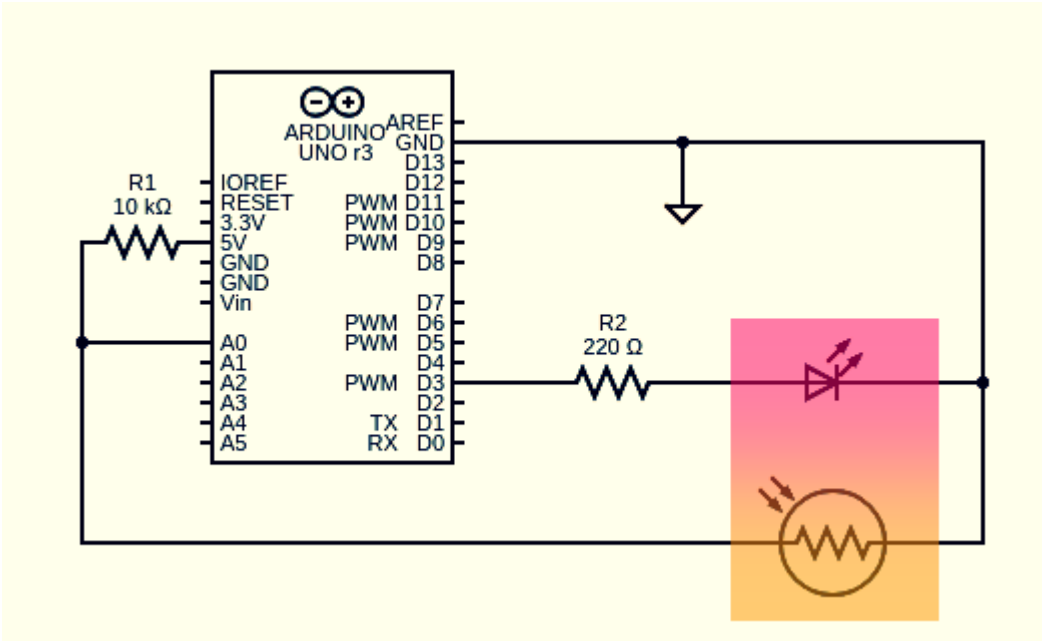


All of this may look somewhat daunting, perhaps even to some veteran engineers. Luckily, you do not have to spend much money to learn what is actually going on here. There's a lot of online resources to help you – Google is your best friend!

Practical PID – Arduino Experiment

To demo PID theory in action, I picked up an Arduino Uno from my messy drawer along with a GL5528 photoresistor (LDR) and a super bright 5mm blue LED. The experiment described here is based on components on-hand because I did not have any dedicated PID systems or components within easy reach yet. And, I'm lazy too!

This is the hardware setup diagram. Note, you should place your LDR so that it aims into the LED output.



Datasheet (GL5528 LDR):

https://cdn.sparkfun.com/datasheets/Sensors/LightImaging/SEN-09088.pdf

In this experiment I am trying a far-famed Arduino PID library (PID_v1) which is simple to setup and use. The tricky part is to set the relevant PID parameters (kp, ki and kd) according your actual needs and configuration. That means you should note that a PID controller is not usable out of the box. Tuning must be done to ensure that the desired performance is achieved. This can be done by determining the constants beforehand and changed them according to the actual response of the system until the optimum values are achieved. Frankly I have not done much experimentation with tuning but despite the lack of precise tuning, I have had pretty consistent results!

To implement a PID controller in an Arduino sketch, five parameters must be known: proportional, integral and derivative constants, input value and set point value. Well, let's start with a crude demo code. Here's the code to implement a PID control system using an Arduino Uno. See, simplicity of the code is because of the great PID_v1 Library (https://github.com/br3ttb/Arduino-PID-Library) which requires you to specify only a few values and you're good to go!

```
[code]

#include <PID_v1.h>

#define LDR_INPUT 0 //LDR to A0

#define LED_OUTPUT 3 //LED to D3

//Variables

double Setpoint, Input, Output;

//PID Parameters

double Kp = 2, Ki = 10, Kd = 1;

//Start PID Instance

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

void setup()

{

  //Start Serial

  Serial.begin(9600);

  //pinMode(LED_OUTPUT, OUTPUT);


  //Set point (brightness target)

  Input = analogRead(LDR_INPUT);

  Setpoint = 100;

  //Turn the PID on

  myPID.SetMode(AUTOMATIC);

  //Adjust PID values

  myPID.SetTunings(Kp, Ki, Kd);

}

void loop()

{

  //Read photoresistor value

  Input = map(analogRead(LDR_INPUT), 0, 1024, 0, 255);
```

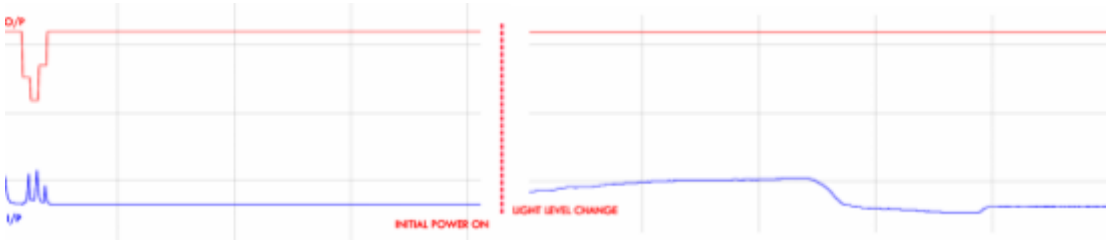
```
  //PID calculation
  myPID.Compute();
  //Write the output as calculated by the PID function
  analogWrite(LED_OUTPUT, Output);

  //Send data for serial monitoring
  Serial.print(Input);
  Serial.print(" ");
  Serial.println(Output);
}
[/code]
```

Related Library Links:

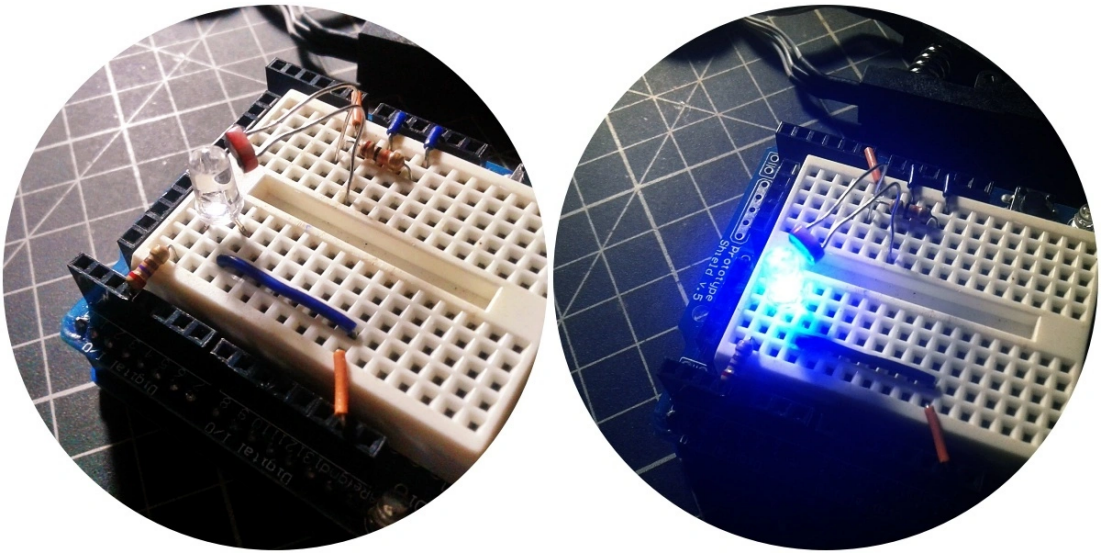
https://playground.arduino.cc/Code/PIDLibrary/

https://playground.arduino.cc/Code/PIDAutotuneLibrary/

To see how the well this code works, you can use the Serial Monitor in the normal way or the Serial Plotter comes with your Arduino IDE (1.6.6 and above). When I masked the photoresistor by putting a translucent film over its face, I casually noticed that the setup quickly reacts and tries to keep the LED brightness at the setpoint by managing the LED control value.



As you can see in the left portion of the above plot, there's a slightly annoying hiccup on initial start-up, which I think could be solved with better tuning (https://playground.arduino.cc/Code/PIDLibraryPonMExample/). However, once it's settled, there's hardly any discernible oscillation so far. I have not timed how long this narking phase lasts. I also forgot to measure the oscillation in the actual output point.



Closing Note

I have done a few analog PID controller projects, but this is my first time preparing a quick primer on PID controllers for the home builder. This was a learning attempt because I read much about PID controllers and conducted a few practical experiments. Someday, I'm going to refine the code demoed here, and make it appropriate for my swimming pool heater/water pump controller project. As might be expected, many great off-the-shelf PID controllers already exist, but I want to break the enigmas and do it cheaper and flexile!

…

Thanks…

https://www.arrow.com

https://www.electricaltechnology.org

https://www.nutsvolts.com

https://clippard.com

https://www.teachmemicro.com

www.dialog-semiconductor.com

& Google Images