Lab 5 Arduino PID library supplement

To get started, the Arduino PID library has some really good code documentation and examples. After you import the library, check out the examples and [documentation](http://playground.arduino.cc/Code/PIDLibrary). The documentation has the library’s built in functions further down the page, which is the most important section. It details what exactly the functions are doing, something vital to know if you are working with this library.

Unfortunately, the library can be a little confusing for non-CS majors to work with, so we are going to outline the important aspects and things that you need to do here. The reason we use the library is because it's a good teaching tool for arduino, and the code implementation works really well.

One other note, in this lab we will be going over *practical* PID. The terminology used and the expectations of the system might differ from what you went over in your controls class.

### Table of Contents

1. The & Sign
2. Setting up the PID
3. PID constants

# The & Sign

One confusing part is the usage of the ‘&’ sign in the code, shown in the setup function in the PID library examples. The only thing that you should understand about this is that certain variables (such as output) are changed in the extra PID library code without you seeing it.

That means that you only need to run PID.compute() and output will be different (pretty much magic for all intents and purposes of this class). There is no need to set output to anything. In fact if you do, it will only mess up things in the code. That being said, you can make setpoint equal to anything you want at any moment in the code and the next time that you run PID.compute() it will calculate your PID with the new setpoint. If you want to know a little more, keep reading this section. If you don’t, skip to the next section.

Here’s my attempt at a super simple explanation. When we set a variable, it’s like giving the arduino a box to store something in. Since the arduino has a lot of boxes, we will have to name the box, and we will have to label it with the things that we can store inside it. Therefore,

Int variable = 0;

Means that we are giving the arduino a box called ‘variable’ and we have put in a value of zero to start with. When we make the variable equal to something, we are putting that number inside the box and destroying the previous value.

With the ‘&’ sign, we already have created the box and put it down somewhere in the arduino’s memory. When we put the ‘&’ sign before a variable, what we are saying is that we are telling the arduino where the box is located. We do this because the arduino is not the owner of the box, the PID library is. So because the PID library is doing things with the variable a lot, and we want the arduino to be respectful of the PID library, we tell the arduino that ‘the variable is over there, I don’t know what the value is and it will change a lot, so just check that location when you need to use the box’s value’. If you want to look up more info on what’s going on here under the hood, lookup pointers.

# Setting up the PID

What exactly is PID? In its most basic definition PID is a bunch of fancy math that produces numbers (*the output*) depending on how much a certain number (*the current state*) is different from the number that we want (*the setpoint*). That’s literally it. PID is not smart, it's just math that computes numbers in a nice way for control. To do that, we need to tailor the math to get the PID to output the kind of behavior that we want. To help illustrate what I’m saying consider this: there are *no units* to the PID output. It’s just math.

But we are getting ahead of ourselves a little. Before we set the units, we need to look at the control system that we are trying to implement. What is the PID actually going to control? In general terms, the PID will control the *actuation method* that we will use to control the error. In our case, we will use the motor to get closer to a setpoint. Now let’s consider the PID output again. And what do we need in our output to control the motor? To control the motor correctly and for all cases in our system, we will need to consider motor *speed* and *direction*. To do this, we need to modify the PID output to encode the motor speed and direction inside the value of output. We will do that by modifying the range that the PID can output.

Now that we have established what we will be using the PID for, let’s look at how we can tune the PID output to contain the motor speed and direction. In the provided PID library, there’s a function for this called [SetOutputLimits()](http://playground.arduino.cc/Code/PIDLibrarySetOutputLimits). The documentation page tells us that the **default limits are 0 to 255**. This nicely encodes our motor value into the output, but what about direction? I’ll leave you to figure out what the new range should be for the output (look up the analogWrite command), just remember - since this is all fancy math that we can set to whatever we want.

Once the output is changed, you will realize that we can’t just use the output in analogWrite. Instead, we will have to use an if statement to extract both the speed and direction from the output value.This is also something that I will leave you to figure out.

***Note:***Something else we need to set is what exactly the input of the PID should be. The easiest way to set that, especially for this lab, is to look at the units of the Setpoint. In our case, the setpoint is 1500 *encoder ticks*. Therefore, by just looking at the units of the setpoint should make it obvious (although I won’t say *exactly* what it is so you can learn it). Just as a general rule, choose something that we can keep track of that has the same units and is acted upon by the *actuation method* described above for the PID output/control signal. Also make sure that you regularly reset the input value.

***NOTE2****:* Be careful when you write your logic, because it’s very easy to make a mistake. Remember to double check everything!

# PID Constants

Regarding the constants for PID, these are going to be very low… under 1. Remember that only specific variable types can handle decimals, so code accordingly.

Let’s think about why the constants will need to be less than 1. The PID equation relies on the error between your current state and your desired state. We do different operations on that error to control our motor to reach our desired point. Sometimes the way we calculate our error can lead to astronomical values if the PID constants are over 1. Again, PID is just math, so it relies on the mathematical relationship between the input and the output. In our case, the encoder ticks (and the setpoint error) can get to very high numbers, especially compared to our 0-255 range. Therefore, we should consider PID constants under 1.

As a proof for my statement above, let’s consider a hypothetical example for the lab: if you set the proportional constant to 1, and there are 1000 encoder ticks between your current and desired states, then the PID function will give as big of a number as it can: 255. We won’t get any less than that until we get to an error of 255, which is relatively close to the desired point. We don’t want to be telling the motor to be going full speed close to the desired point or it will overshoot, so it will have to be less than 1.

Your PID will have a steady state error. That’s ok, as long as it’s acceptably close (for the lab, about 0-2% from the setpoint). You will have to tweak the values of your constants. A little tweak can do a lot. We want you to get a feel for the tweaking, so I will leave that to you. The [Ziegler-Nichols](https://en.wikipedia.org/wiki/Ziegler%E2%80%93Nichols_method) method is a help resource, however don’t expect it to work perfectly. You can’t escape the need to finetune things.

To reward you for reading the whole document, I will tell you a little more about the PID table that you should construct. In terms of K\_p, you will want to use values over 0, but keep things within the range specified above (within 1… *maybe* go a tiny bit higher to prove a point). Your K\_d should start at 0, so you can purely see the effect that K\_p has on the system, and then keep K\_d within the range specified above. NEVER bring it over 1, otherwise K\_d will dominate everything and you won’t see any change in response. Typically, shoot to have K\_p greater than K\_d.

One last helpful hint of advice, when trying to figure out the PID constants, I would suggest at the very least, outputting these variables to the serial monitor. They will ensure that you have all the information to tune well, although you might need more depending on your implementation:

‘Encoder Ticks’ , ‘Error’ , ‘PID Output value’ , ‘millis() value’

*Note*: the millis() value is for plotting the values on the graph nicely.