Arduino PID Autotune Library « Project Blog

At long last, I've released an Autotune Library to compliment the Arduino PID Library. When I released the current version of the PID Library, I did an insanely extensive series of posts to get people comfortable with what was going on inside.

While not nearly as in-depth, that's the goal of this post. I'll explain what the Autotune Library is trying to accomplish, and how it goes about its business.

Attribution

For A couple years I've wanted to have an Autotune Library, but due to an agreement with my employer, I wasn't able to write one. BUT! when I found the AutotunerPID Toolkit by William Spinelli I was good to go; My company had no problem with me porting and augmenting and existing open source project.

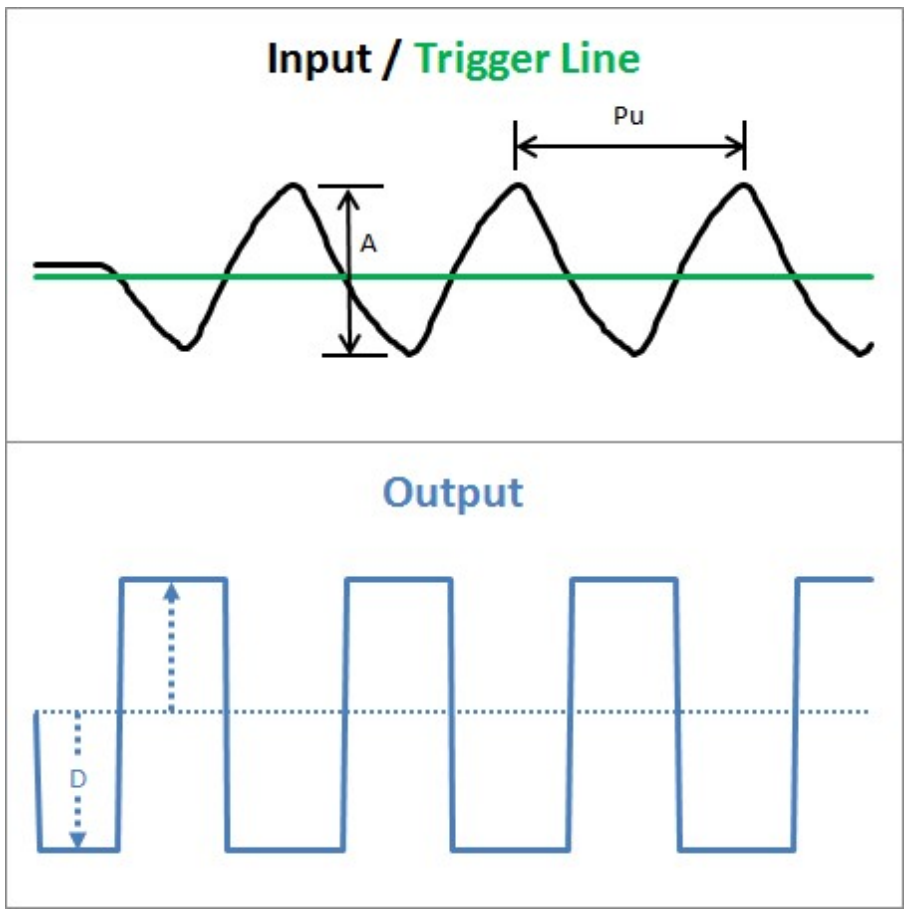
I converted the code from matlab, made some tweaks to the peak identification code, and switched it from the Standard form (Kc, Ti, Td) to the Ideal form (Kp, Ki, Kd.) Other than that, all credit goes to Mr. Spinelli.

The Theory

The best tuning parameters (Kp, Ki, Kd,) for a PID controller are going to depend on what that controller is driving. The best tunings for a toaster oven are going to be different than the best tunings for a sous-vide cooker.

Autotuners attempt to figure out the nature of what the controller is driving, then back-calculate tuning parameters from that. There are various methods of doing this, but most involve changing the PID Output in some way then observing how the Input responds.

The method used in the library is known as the relay method. here's how it works:



Starting at steady state (both Input and Output are steady,) the Output is stepped in one direction by some distance D. When the Input crosses a trigger line, the output changes to the other direction by distance D.

By analyzing how far apart the peaks are, and how big they are in relation to the output changes, the Autotuner can tell the difference between one type of process and another. As a result, different systems will get custom tuning parameters:

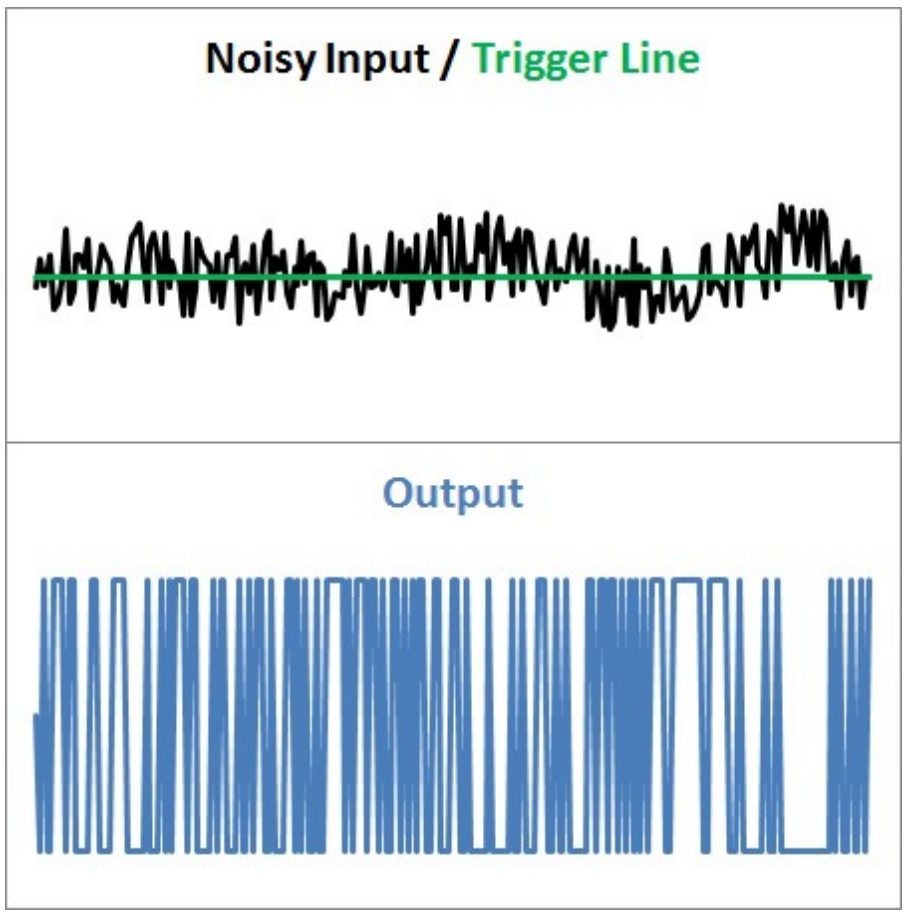
Control Type	Kp	Ki	Kd
PI	0.4 * Ku	0.48 * Ku / Pu	0
PID	0.6 * Ku	1.2 * Ku / Pu	0.075 * Ku * Pu
Where Ku = 4 * D / (A * π)			

The Implementation

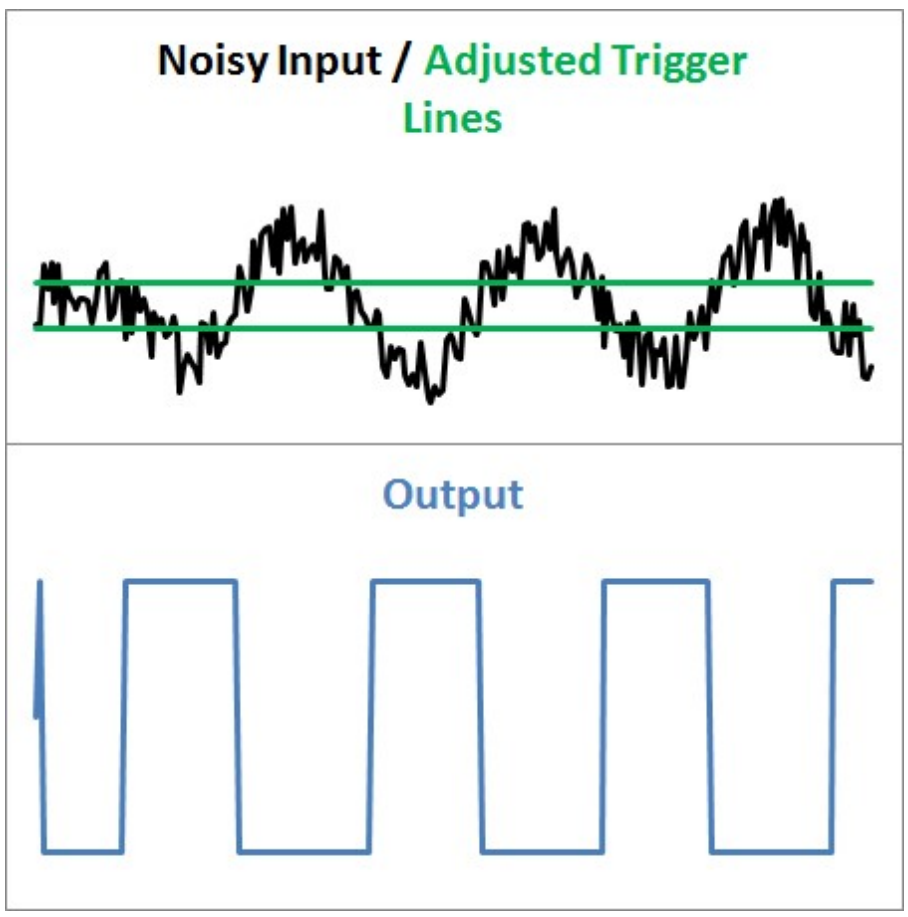
This works well in theory, but real-world data isn't very cooperative. The input signal is usually noisy, which causes two main problems.

Problem #1: When to step?

Since a noisy signal is choppy, it's likely that the trigger line will be crossed several times as the Input moves past it. This can cause mild chatter in the output, or if severe, can completely destroy things:

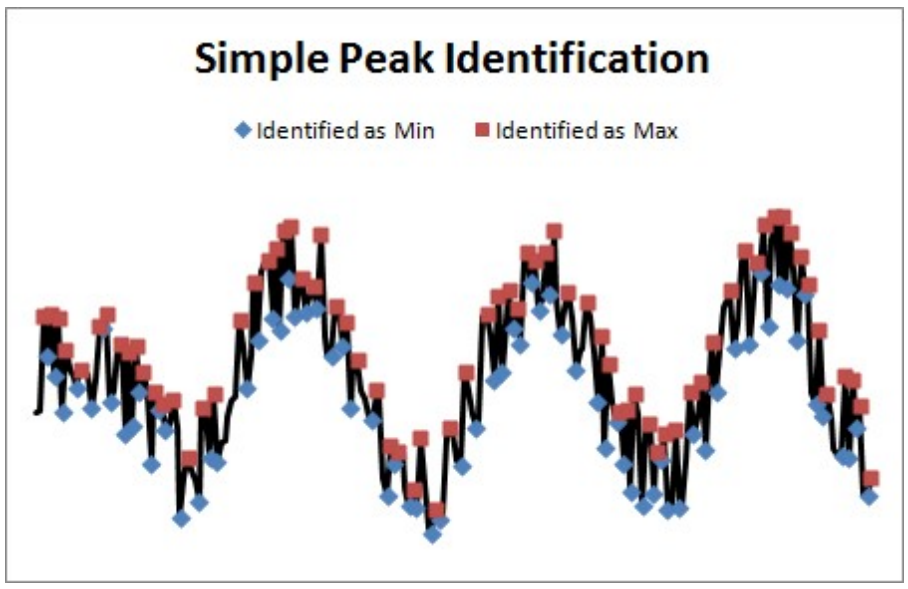


The way I chose to side-step this issue was to have the user specify a noise band. In effect, this creates two trigger lines. Since the distance between them is equal to the noise (if properly set) it's less likely that multiple crossings will occur due to signal chatter.



Problem #2: Peak Identification

In a simulated world, identifying the peaks is easy: when the Input signal changes direction, that's a minimum or a maximum (depending an which change occured.) In a noisy world however, this method fails:

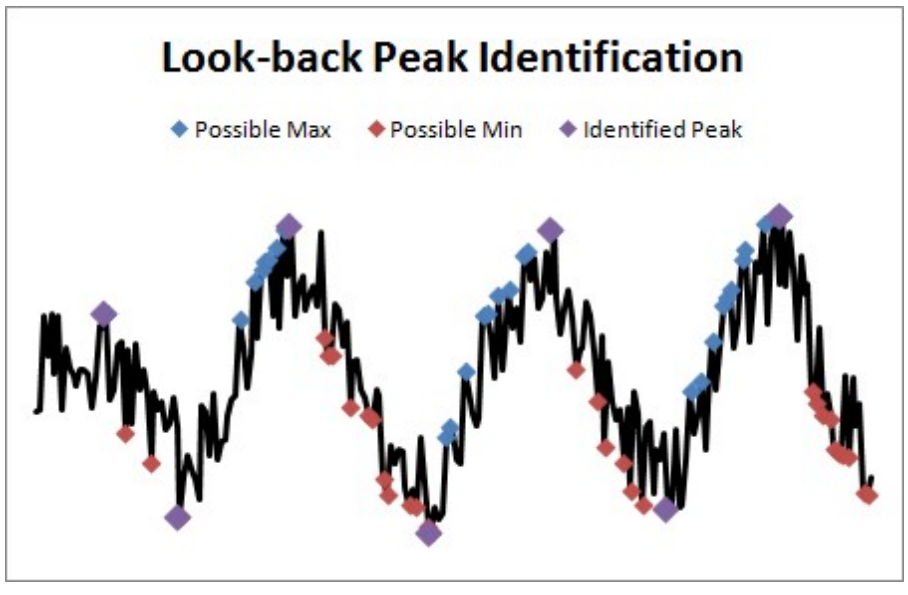


Every noise blip is a direction change. To deal with this issue I added a "look-back time" parameter. It's an awful name. If you can

think of something better let me know.

At any rate, the user defines some window, say 10 seconds. The Library then compares the current point to the last ten seconds of data. If it is a min or a max, it gets flagged as a possible peak.

When the flagged point switches from being a max to a min, or vice versa, the previously flagged point is confirmed as a peak.



Another way of explaining the look-back time is that a point will be identified as a peak if it is the largest (or smallest) value within one look-back into the future or past. Like I said: awful name.

You should also know…

- The number of cycles performed will vary between 3 and 10. The algorithm waits until the last 3 maxima have been within 5% of each other. This is trying to ensure that we've reached a stable oscillation and there's no external strangeness happening. This leads me to…
- I'm not the biggest fan of Autotune. I've often said, and still believe, that a moderately trained person will beat an Autotuner every day of the week. There's just so much that can go wrong without the algorithm knowing about it. That being said, Autotune is a valuable tool to help the novice get into the ballpark.

Tags: Autotune, osPID, PID

This entry was posted on Saturday, January 28th, 2012 at 2:41 pm and is filed under Coding, PID. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response, or trackback from your own site.