PID Tuning Overview
with Special Consideration for the New TBS

Ryan French
Technical Development, Tannas Co.

October 12, 2017

Abstract

To control temperature in the New TBS, we use a PID (Proportional, Integral, Derivative) algorithm from the Arduino library *PID*. Three methods of determining PID parameters have been used during initial testing with the New TBS prototypes: manual tuning, Ziegler-Nichols, and Cohen-Coon. Each method has had varying success, with manual tuning being the preferred method at the time of this publication. Further work with the Ziegler-Nichols and Cohen-Coon methods is needed to make them work for our purposes.

1 PID Theory

For a process variable (e.g. temperature) to reach and hold at a given value, it has to know how much controller output (e.g. pulse-width modulation of current) is needed at a given time. PID controller theory is one method of achieving reliable communication between the process variable [p(t)] and the controller output [u(t)].

Ideal PID theory involves a differential equation based on error, e(t), which controls output. Here, error is defined as the difference between the setpoint, s(t), and the process variable, p(t). For example, if we want a temperature of 150°C but our system is currently at 100°, then our error at the given time, t_0, is: $s(t_0) - p(t_0) = 150 - 100 = 50 = e(t_0)$. The full ideal equation for controller output is:

$$u(t) = K_p \left[e(t) + \frac{1}{T_i} \int_0^t e(\tau) d\tau + T_d \frac{de(t)}{dt} \right] \tag{1}$$

where K_p is the proportional constant, T_i is the integral time constant, and T_d is the derivative time constant. Note that the integral and derivative parameters are: $K_i = K_p / T_i$ and $K_d = K_p \cdot T_d$, respectively.

The second term is an integral from the initial time (usually zero) until the current time. In that way, the integral term is the correction due to an accumulation of error. The third term is a time derivative of error. This derivative term, in practice, is more often a time derivative of p(t). This avoids large increases in u(t) when s(t) is changed; dp(t)/dt instead is only concerned with changes in the process variable itself. Using this and adapting equation 1 to a discrete system, we arrive at the equation that is most often used, including in our PID library:

$$u(t) = K_p \left[e(t) + \frac{1}{T_i} \sum_{n=0}^{t} e(n) \Delta n + T_d \frac{\Delta p(t)}{\Delta t} \right] \tag{2}$$

We can see that in equation 2, the integral becomes a sum of all errors times the change in time from zero to the current time, and the derivative is now a change in p(t) over a change in time (Δt is often the sample period).

2 Introduction to Ziegler-Nichols Method

A thorough, concise overview of this method is found in [1].

2.1 Initialization

Initializing this method requires all three PID parameters: K_p, K_i, and K_d, to be set to zero. K_p is turned up in small increments until a sustained, periodic oscillation occurs (e.g., a consistent period). Noting the value of the proportional constant which led to consistent oscillation as K_u and the period of oscillation as T_u, we can calculate the PID parameters.

2.2 Calculation of Parameters

The parameters are found as follows:

$$\begin{aligned} K_p &= \frac{K_u}{1.7} \quad , \\ T_i &= \frac{T_u}{2} \quad , \\ T_d &= \frac{T_u}{8} \end{aligned} \tag{3}$$

or [keeping in mind the relationship between PID parameters and their time constants (section 1)]:

$$\begin{aligned} K_p &= \frac{K_u}{1.7} \quad , \\ K_i &= \frac{2K_u}{1.7T_u} \quad , \\ K_d &= \frac{K_u T_u}{8 \cdot 1.7} \end{aligned} \tag{4}$$

An alternate method of calculating the parameters is called the Tyreus-Luyben method. It tends to reduce oscillatory effects and improves robustness [1]. Using the same values calculated in initalization, we get:

$$\begin{aligned} K_p &= \frac{K_u}{2.2} \quad , \\ K_i &= \frac{K_u}{(2.2)^2 T_u} \quad , \\ K_d &= \frac{K_u T_u}{2.2 \cdot 6.3} \end{aligned} \tag{5}$$

3 Introduction to Cohen-Coon Method

A general introduction to the Cohen-Coon method, especially in using an approximated slope, can be found in [2].

3.1 Initialization

Initializing this method requires turning off all PID functionality in the temperature controller. For this method, u(t) is the only thing that will be controlled (in our case, u(t) is the heater output, a value between 0 and 255 which controls pulse-width modulation frequency). To begin, set u(t) = 0. Once p(t) (temperature) is stable at this u(t), step u(t) up to another value. With temperature logging software running, wait again for the temperature to increase and level out at a maximum, stable temperature. For this method, the time plots of u(t) and p(t) are required.

3.2 Calculation of Necessary Values

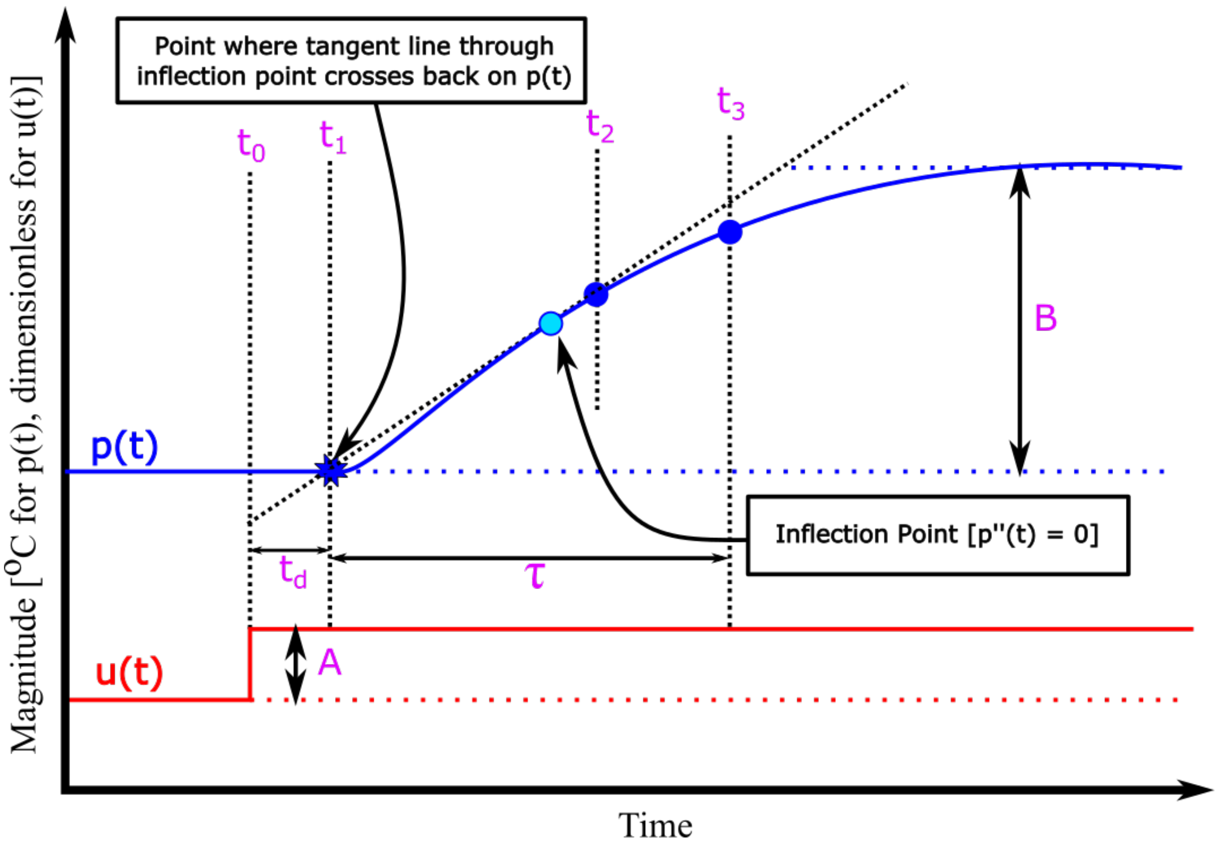
Point where tangent line through inflection point crosses back on p(t)
t_0
t_1
t_2
t_3
B
p(t)
Inflection Point [p''(t) = 0]
t_d
τ
u(t)
A
Magnitude [^{O}C for p(t), dimensionless for u(t)]
Time

Figure 1: This is what the plots of u(t) and p(t) should look like. All constants derived in this subsection are labeled in the plot.

The time constants are defined as such: t_0 is the time at which u(t) shifts from one input to another (e.g., 0 to 100), t_1 is the time at which the tangent line of greatest slope (occurring at $\frac{d^2p(t)}{dt^2} = 0$) crosses p(t), t_2 is the time for p(t) to reach half of its maximum value (keep in mind that this is halfway between the starting p(t) and final p(t)), and t_3 is the time for p(t) to reach 63.2% of its maximum value. A is the difference between initial and final u(t) values, and B is the difference between initial and final p(t) values.

Using these values, we can calculate the other two time constants, plus two more constants which simplify the final math:

$$\begin{aligned}
&t_d = t_1 - t_0 \quad \text{(''dead time''),} \\
&\tau = t_3 - t_1 \quad , \\
&r = \frac{t_d}{\tau} \quad , \\
&K = B/A
\end{aligned}$$

3.3 Calculation of Parameters: Slope Method

By plotting u(t) and p(t), we can find the greatest slope of p(t) [that is, $(\frac{dp(t)}{dt})_{max}$], and therefore, the point where this tangent line crosses p(t). This is t_1, and it needs to be calculated in this way before calculating the rest of the constants mentioned in the previous subsection. This method is more intensive than the next one, but should yield more accurate results. This method also calculates a proportional gain that is one-half of what is calculated in the next method. This reduces oscillation time [3].

The equations to calculate parameters is as follows:

$$\begin{aligned}
&K_p = \frac{0.67}{K}\left(\frac{1}{r} + 0.185\right) \\
&T_i = 2.5t_d\left(\frac{\tau + 0.185t_d}{\tau + 0.611t_d}\right), \quad K_i = K_p/T_i \\
&T_d = 0.37t_d\left(\frac{\tau}{\tau + 0.185t_d}\right), \quad K_d = K_pT_d
\end{aligned} \tag{6}$$

3.4 Calculation of Parameters: Approximation Method

For most applications using a PID controller, the transfer function takes on a predictable form, with a First Order with Dead Time (FODT) model transfer function [G(s), the function that converts u(s) into p(s) in LaPlace space]. We can, therefore, calculate the time constant t_1 using the other time constants, without going through the hassle of trying to do it on a plot (note: the author has not personally confirmed the accuracy of this method's t_1 constant vs. the slope method discussed previously).

Calculating t_1 now involves only algebra:

$$t_1 = \frac{t_2 - \ln(2)t_3}{1 - \ln(2)}$$

Now that we have the constants that we need, we can calculate the PID parameters as follows:

$$\begin{aligned} K_p &= \frac{1}{K \cdot r}\left(\frac{4}{3} + \frac{r}{4}\right) \\ T_i &= t_d\left(\frac{32+6r}{13+8r}\right), \quad K_i = K_p/T_i \\ T_d &= t_d\left(\frac{4}{11+2r}\right), \quad K_d = K_p T_d \end{aligned} \tag{7}$$

4 Introduction to Manual Method

While there is no "best" way to manually tune a PID controller, this section will introduce the method that the author has used.

4.1 Method

First, make sure that the controller is in PID mode (unlike the Cohen-Coon method). Set all gains to zero. Increase K_p until the response to a disturbance of p(t) is a steady oscillation. Once this is done, increase K_d until these oscillations go away (e.g., it is critically damped). Repeat the steps in increasing K_p and K_d until increasing K_d does not stop the oscillations.

Make sure that K_p and K_d are at their last stable values. Increase K_i until it brings you to your setpoint (in our case, the target temperature) in the number of oscillations that is needed. This is a matter of preference, as overshoot can lead to a quicker settling time, but the oscillations about the setpoint may be unwanted.

Once this is done, test it by changing p(t): change setpoints and watch for roughly acceptable settling, and manually change p(t) (e.g., cool down the system manually) and watch for roughly acceptable settling.

If thus far the results are acceptable for early tuning, it is time to start fine tuning. For our purposes, we are more concerned with p(t) remaining very accurate at a setpoint, so set p(t) to an acceptable value (I set temperature to 150°C for high-temp PID) and watch the movement of p(t) about the setpoint. Keeping in mind that K_p and K_i respond to the error, e(t), and K_d responds only to a change in p(t), we can analyze the waveform of p(t) and change parameters accordingly.

I will list a couple of issues that I had experienced in fine tuning as examples:

- A small change in p(t) (e.g., a small spike in temperature) causes a large response in the direction opposite of the change in p(t).
  - K_d is likely too large. This makes a small change in p(t) have a large effect on u(t). Also, pay attention to the location at which this spike happened. If it happened near the setpoint, K_p may also be too large. It is important to consider both of these parameters when this happens.
- p(t) is running slightly above or slightly below the setpoint without approaching it.
  - K_p and/or K_i likely aren't large enough. Start by increasing K_i to see if the accumulation of error will bring p(t) to the setpoint without adding noise/oscillations. If it does cause oscillations, try manipulating both K_p and K_i slightly until the wanted effect is achieved.
- Settling time is too long.
  - Decrease K_i. Sometimes, slightly increasing K_p can have the same effect of decreasing K_i. Increasing K_d can also decrease settling time.

Most of the time, manual tuning takes time to get used to. Experimenting with independent increases and decreases in parameters can help you get a feel for what each of them does. For reference, here is a table listing general attributes of the parameters [4]:

Effect of Increasing Each PID Parameter				
Parameter	**Rise Time**	**Overshoot**	**Settling Time**	**Steady-State Error**
K_p	Decrease	Increase	Minor Change	Decrease
K_i	Decrease	Increase	Increase	Eliminate
K_d	Minor Change	Decrease	Decrease	Minor Change

Table 1: The effect of increasing each parameter on p(t).

References

[1] Co, Tomas B. Michigan Technological University, Department of Chemical Engineering. http://pages.mtu.edu/ tbco/cm416/zn.html

[2] Co, Tomas B. Michigan Technological University, Department of Chemical Engineering. http://pages.mtu.edu/ tbco/cm416/cctune.html

[3] Dataforth Co. *Tuning Control Loops for Fast Response - Dataforth*. https://www.dataforth.com/tuning-control-loops-for-fast-response.aspx

[4] Zhong, Jinghua. PowerPoint Presentation: *PID Controller Tuning: A Short Tutorial*. Purdue University. http://saba.kntu.ac.ir/eecd/pcl/download/PIDtutorial.pdf