PID CONTROLLER

PID CONTROLLERS

A PID Controller is a proportional, integral, derivative controller.

These controllers are really nice because they learn as they are running.

The goal is to have a constant steady-state output that matches a desired set point and in a timely manner. For example, when we use cruise control in our cars, we want our car to try to stay at a constant speed and not oscillate between speeds. If our car slows down due to a big hill, then once we get over the hill we want the car to catch back up to the correct speed as fast as possible without too much overshoot.

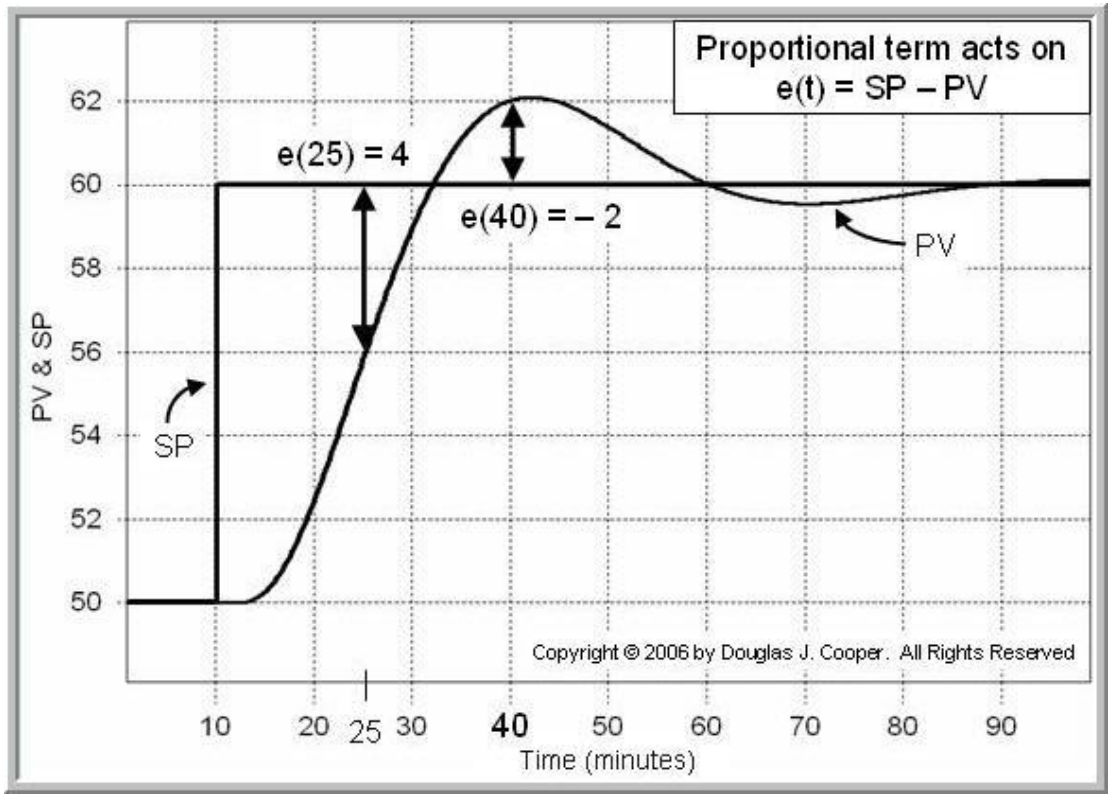
PID CONTROLLERS

The P, I, and D terms are all added together to contribute to the output.

The P, I, and D terms all have coefficients associated with them that determine how much each term contributes to the output. Let's call these coefficients K_P, K_I, and K_D.

So if K_P =10, K_I =0.5, and K_D=0, then we have a PI controller since the derivative term equals 0.

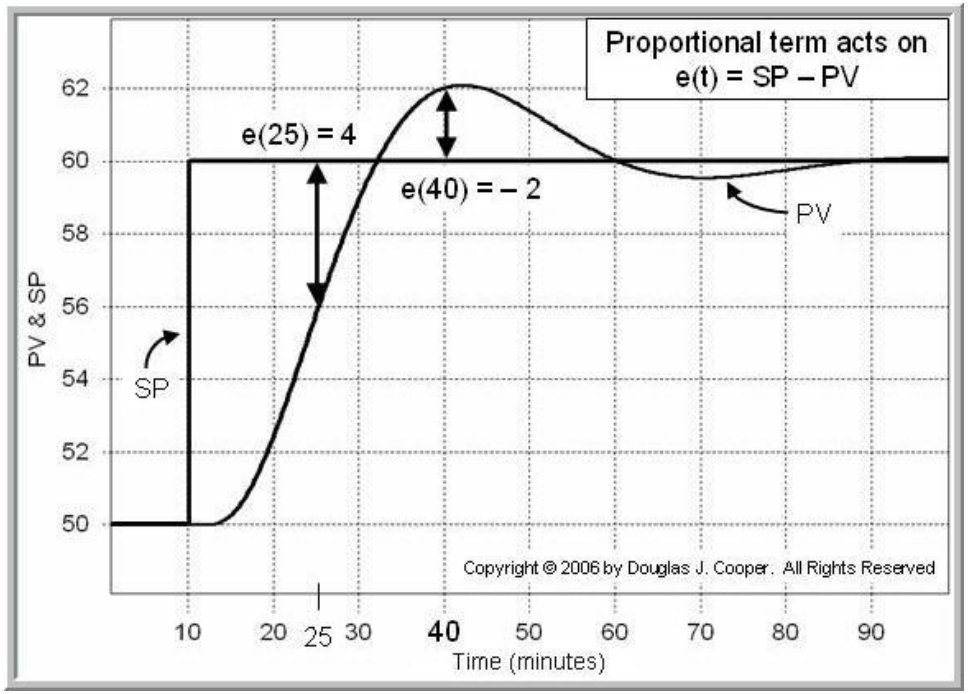
ERROR



The goal with controlling is to respond to some change, so SP (set point) in this graph is the desired output. This kind of desired output is known as a "step function" and the response is known as a "step response." Error [e(t)] is the difference between the desired output and the actual output at any moment in time. The picture shows an example of different errors at time equal to 25 and 40.

PROPORTIONAL TERM

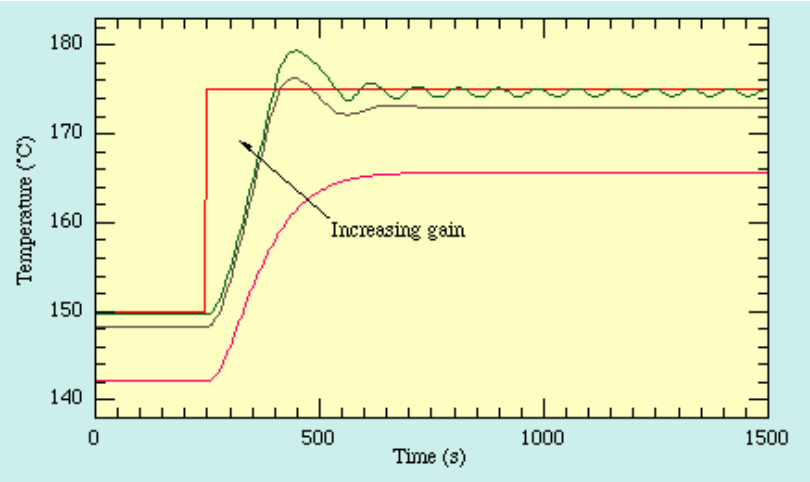
The proportional term contributes to the output by multiplying the error by the coefficient K_P. So when the error is large, the controller P term contributes a large value to the input and when the error is 0, the P term contributes 0. In the diagram, the P term would contribute more at t = 25 than it subtracts at t = 40.



SMALL PROPORTIONAL COEFFICIENT

If we make coefficient K_P really small, then we never quite reach the desired output. The pink line is an example of a small K_P.

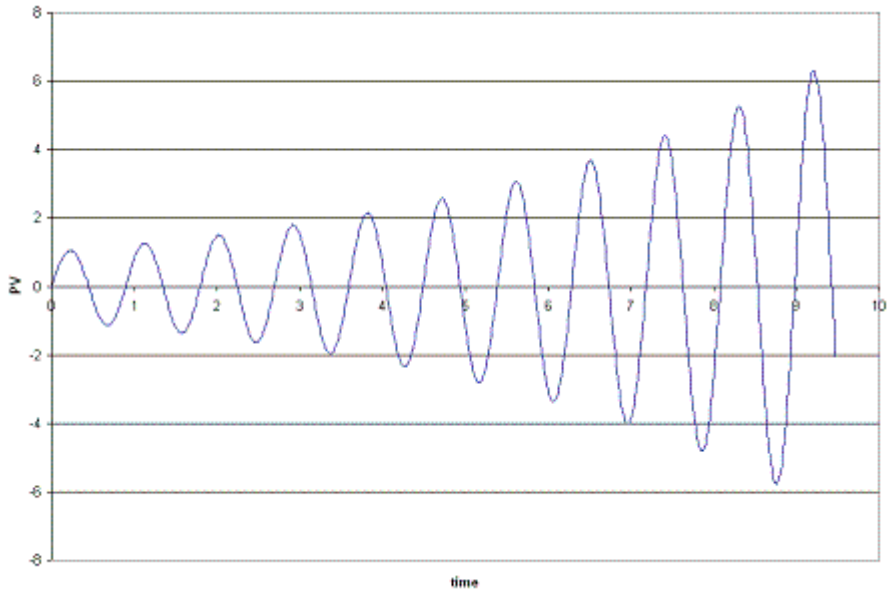
In the graph, the desired output is 175 degrees Celsius. Notice that the value of the pink line converges to ≈165 degrees after a long period of time. This gives us what we call a steady-state error (e_{ss}) of 10 degrees.



As we increase K_P, we get a smaller steady-state error. If we increase even further we get oscillations. If we increase even further, then the system can become unstable *(see next slide)*.

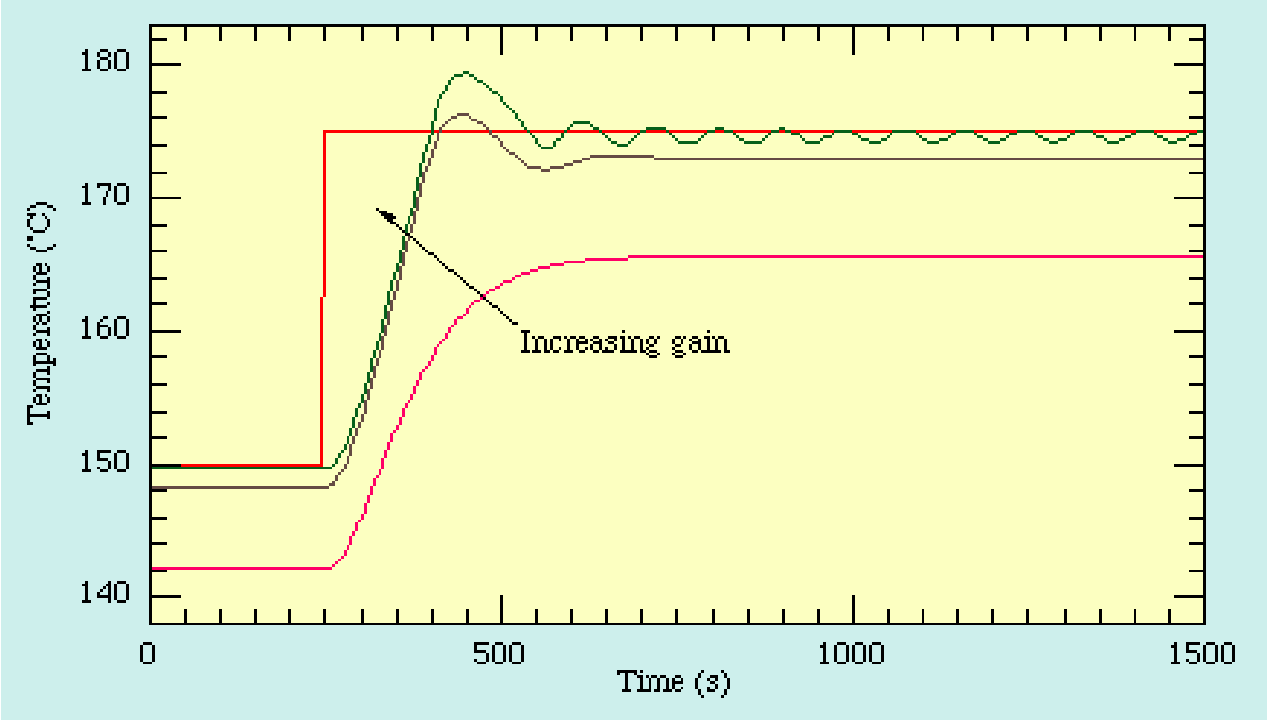
LARGE PROPORTIONAL COEFFICIENT (UNSTABLE)

If we make K_P too big, then we eventually get oscillations that become larger and larger and we have an unstable system.



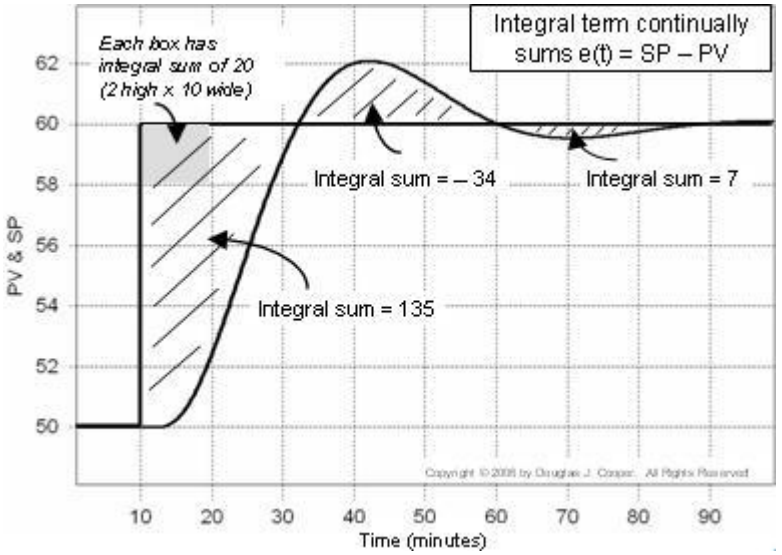
GETTING A STEADY-STATE ERROR = 0

The green graph performs OK, but it still has a small steady-state error. Only by adding an integral term will it ensure that we get a steady-state error equal to 0 (makes a PI controller). This is important because a P controller alone may not respond correctly with different loads (maybe we're controlling a motor). The I term is nice because it allows the controller to learn with time and ensures e_{ss}=0.



INTEGRAL TERM

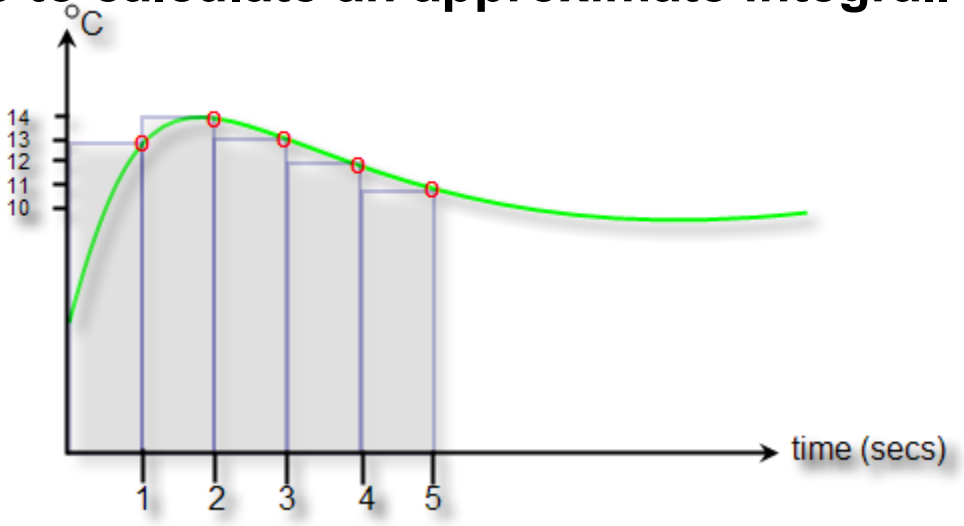
The integral term keeps a total sum of the error over time. With a PI controller, the steady-state error will be 0 and the total error will only depend on initial areas under the curve when the system was oscillatory. The total sum of the error in the diagram is 135-34+7=108. Assuming everything after 100 minutes has an error of 0, this 108 will remain approximately constant for the lifetime that we are controlling. So the I term will continually contribute to the output 108 * some coefficient value (K_I).



INTEGRAL TERM

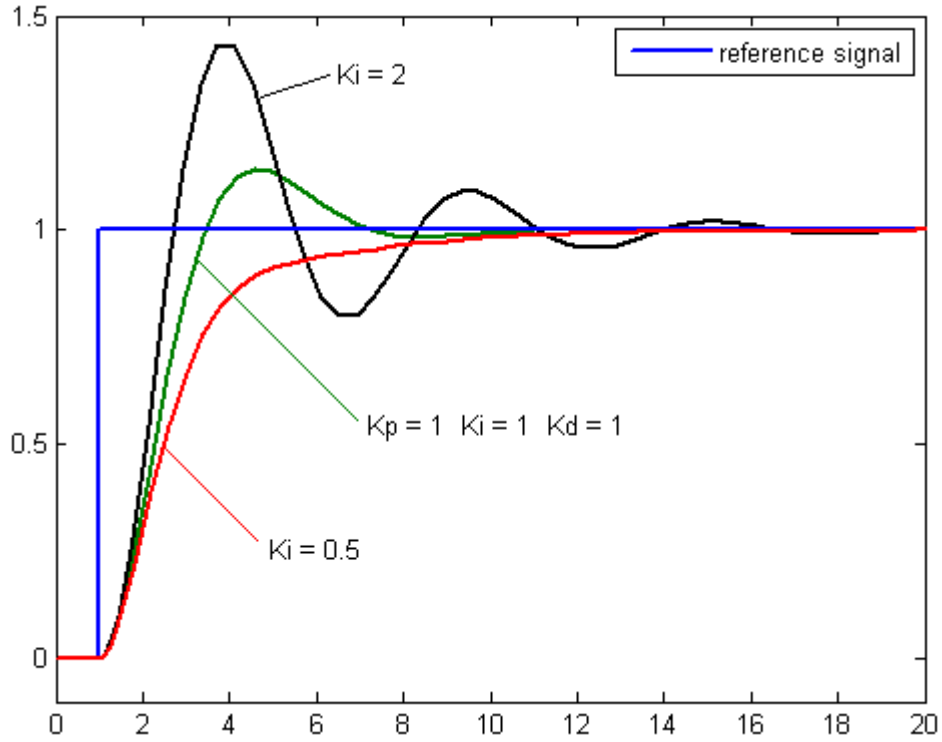
The real world can't sample the sensor's value continuously, so it will sample discrete steps at some sample rate. Your program will calculate the integral using these discrete steps where the width depends on the sample rate

(*t = 1/sample rate*). The sample rate of this diagram is 1 Hz *(which is fast enough for some applications and too slow for other applications)*. Your program will need to add all the areas of the rectangles to calculate an approximate integral.



INTEGRAL TERM

We need to choose the correct K_I value. If K_I is too big, then you will get large oscillations on the output. *Note: These oscillations may continue forever.* If K_I is too small, then it may take too long to reach the desired output. Compare the black line (large K_I) with the red line (small K_I).



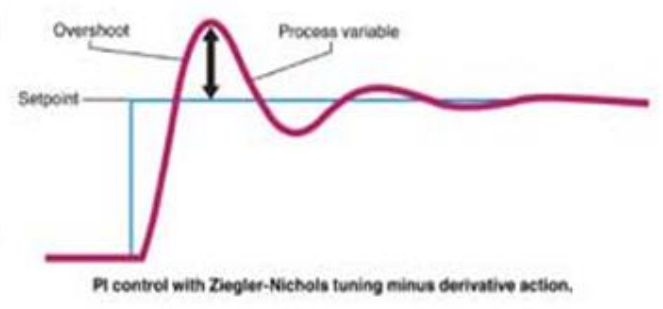
DERIVATIVE TERM

The derivative term is used to smooth out the response.

For example, let's say we are controlling a motor that is rated for 10,000 RPM when we provide 12V and 100% duty cycle. Let's say our goal is to spin at 2,000 RPM *(also assume that the ideal duty cycle for 2000 RPM is 35%, but we don't know this)*.

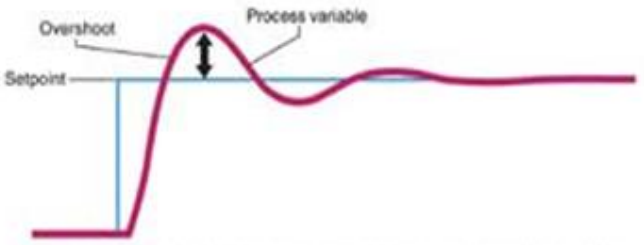
When we first turn the motor on, the error is going to be massive and our P term is going to likely contribute too much to the input (it will probably make the duty cycle go from 0% to 100% in one time step). This causes the output to overshoot tremendously. ☹

The D term makes the slope of the duty cycle decrease a bit so that the motor ramps up smoother (so now maybe the duty cycle goes from 0% to 25% to 50%) and helps to prevent the large overshoot.



PI control with Ziegler-Nichols tuning minus derivative action.

As you can see in the graphs, the D term doesn't drastically change the response. The D term has the least amount of impact on the controller so many times you may see PI controllers instead of PID.



PID control with Ziegler-Nichols tuning including moderate derivative action.

PID FORMULA

The formula for the PID controller is

$$Input = K_P E + K_I \int (E) dt + K_D \left(\frac{dE}{dt} \right)$$

where *E* is the error. Since we will be sampling the output at discrete times (t = 1/Sample Rate)**, we will use an approximation for the integral and derivative. Now our formula becomes**

$$Input = K_P E_n + K_I \Delta t \sum E_n + K_D \left(\frac{E_n - E_{n-1}}{\Delta t} \right)$$

where E_n is the current error and E_{n-1} is the previous error.

You can make a PID controller with an analog circuit (using amplifiers and low & high pass filters), but it's easier to create by programming software.

PID TUNING

Let's say you've written your program correctly for a PID controller and now you have 3 coefficients K_P, K_I, and K_D and you have to decide on values for each. What values do you choose?

Remember, increasing K_P decreases steady-state error and speeds up the response, but increases the overshoot and degrades the stability.

Like K_P, increasing K_I also speeds up the response, increases overshoot, and degrades stability. It also increases settling time ☹. But having K_I eliminates steady-state error, so it's typically necessary.

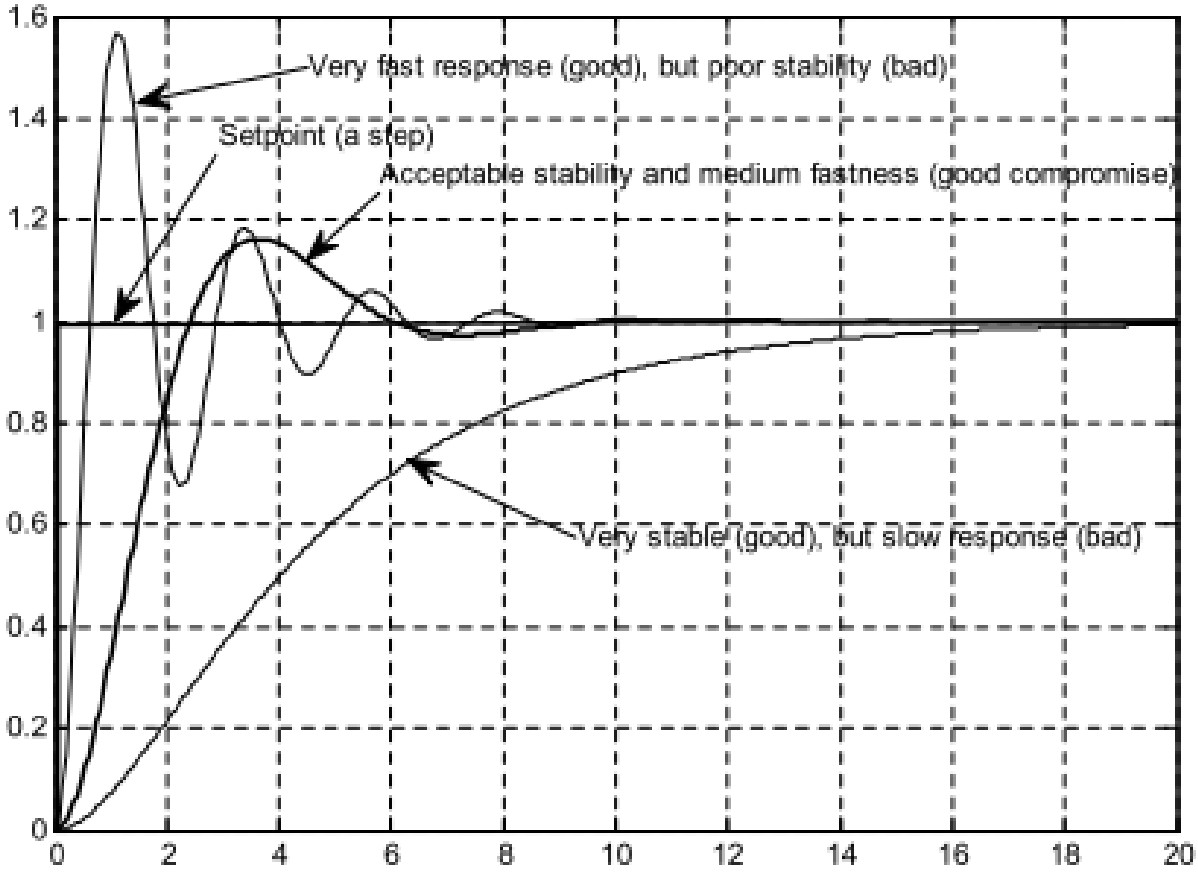
K_D slightly speeds up response, slightly decreases overshoot, and slightly decreases settling time. It's good to include if this value is small!

It sounds like K_P is most important, K_I is somewhat important, and K_D is least important. So an approach to tuning might be to increase K_P until you get a response that is close the desired output where the output is oscillating. Then decrease K_P and add in a K_I that is smaller than K_P. Then add in a K_D that is smaller than both K_P and K_I.

Some people have jobs that are entirely focused on tuning PID controllers. This should tell you that tuning can be complicated.

PID TUNING

The following graph shows the typical sacrifice one would make to determine the best parameters for a PID controller.



PID TUNING METHODS

There are many algorithms for tuning. Two popular methods are the Ziegler-Nichols method and Cohen-Coon method.

Ziegler-Nichols method requires a PID controller to already be in place, whereas the Cohen-Coon method just requires a P controller at first.

The Cohen-Coon method is supposedly better if there's a deadtime between the input and output. Deadtime is the time it takes to observe a change in the output once the change in the output signal is applied. An example of deadtime . . . if we make a thermostat for a ceramic heater, it takes a lot of time for the heat to propagate through the ceramic. So by the time the temperature sensor registers that the heater is up to the setpoint temperature, it's typically too late and the heater's temperature overshoots the setpoint temperature. The deadtime decreases the stability of the system. ☹

ZIEGLER-NICHOLS TUNING METHOD

PID Formula: $V_{IN} = K_P E + K_I \int (E) dt + K_D \frac{dE}{dt}$

The Ziegler-Nichols method is a closed loop method that starts with a P controller. We start with a small gain (K_P) then increase this gain until we get oscillations at a constant amplitude. This value is known as the ultimate gain (K_U). The period of these oscillations is P_U. We then use these values to determine the value for K_P, K_I, and K_D.

Note: These two charts are the same thing. The first chart is in relation to K_P & P_U and the second chart is in relation to K_U & P_U.

Ziegler–Nichols method

Control Type	K_p	K_i	K_d
P	$0.50K_u$	-	-
PI	$0.45K_u$	$1.2K_p/P_u$	-
PID	$0.60K_u$	$2K_p/P_u$	$K_pP_u/8$

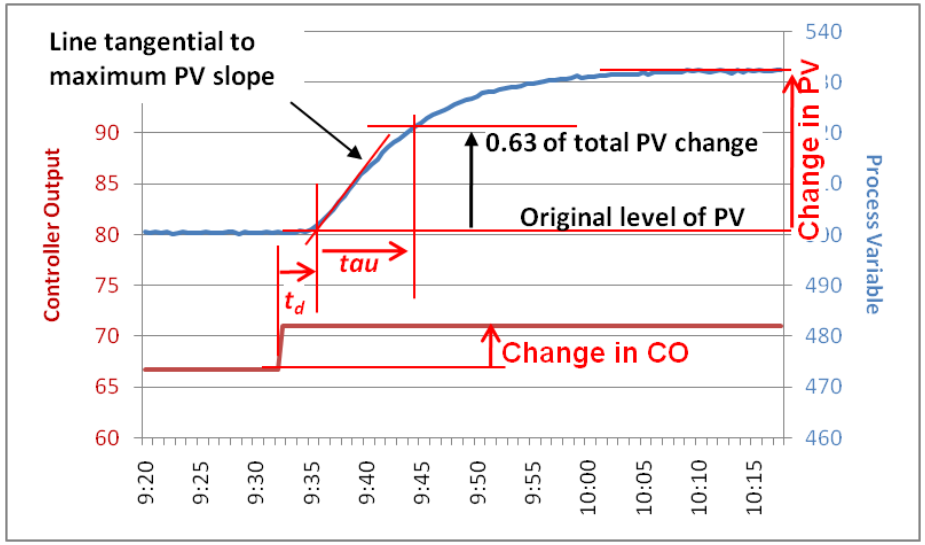
Control Type	K_P	K_I	K_D
P	$0.50K_U$	-	-
PI	$0.45K_U$	$0.54K_U/P_U$	-
PID	$0.60K_U$	$1.2K_U/P_U$	$0.075K_U*P_U$

These values are good starting points to tuning the PID controller. You typically find better results with some additional manual tuning.

COHEN-COON TUNING METHOD

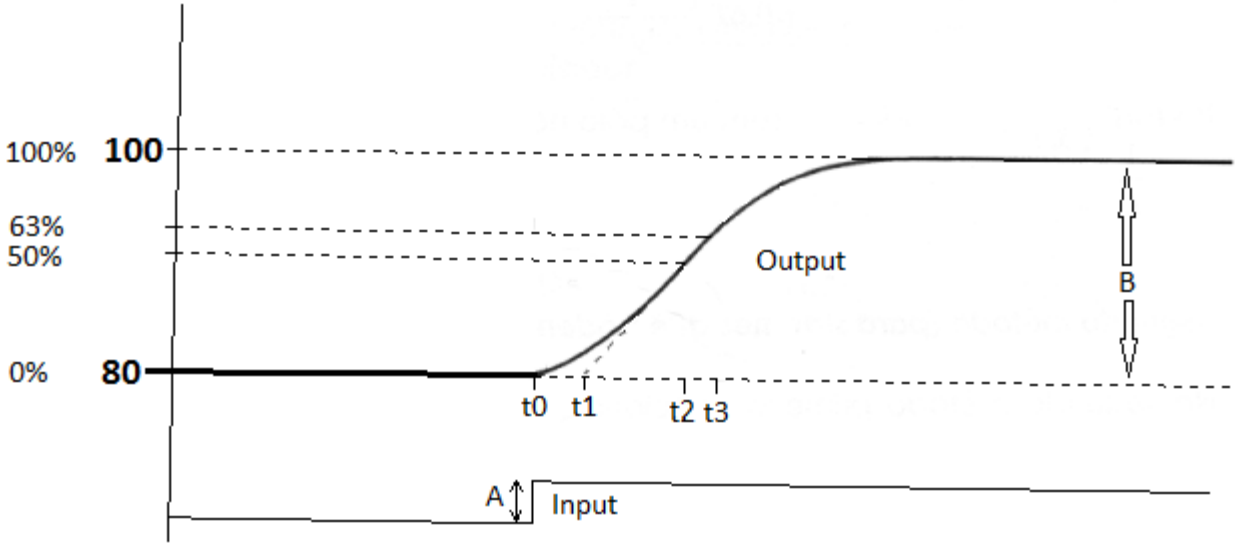
The Cohen-Coon method is an open loop method (no feedback) so not much setup is required. It involves turning on an output and waiting for it to stabilize, then applying a small increase in the output (step response) that should be at least 5x greater than the noise and graphing the response.

For example, let's say we provide a duty cycle of 80% to a motor and wait until it is stable. Then we apply a duty cycle of 97% to the motor and graph how the motor's response changes.



Looking at the plot, the output changed from 500 to 532. The change in the output is B (equals 32). The change in the input is A (which is 17%). Now we need to find Tau.

COHEN-COON TUNING METHOD



This graph from t_1 onwards looks kind of like the charging of a capacitor graph.

Remember, the formula for charging a capacitor is . . .

$$V(t) = V_{max}\left(1 - e^{-t/\tau}\right)$$

If we raise this graph up so it doesn't start at *V=0*, we get . . .

$$V(t) = (V_{max} - V_{min}) * \left(1 - e^{-t/\tau}\right) + V_{min}$$

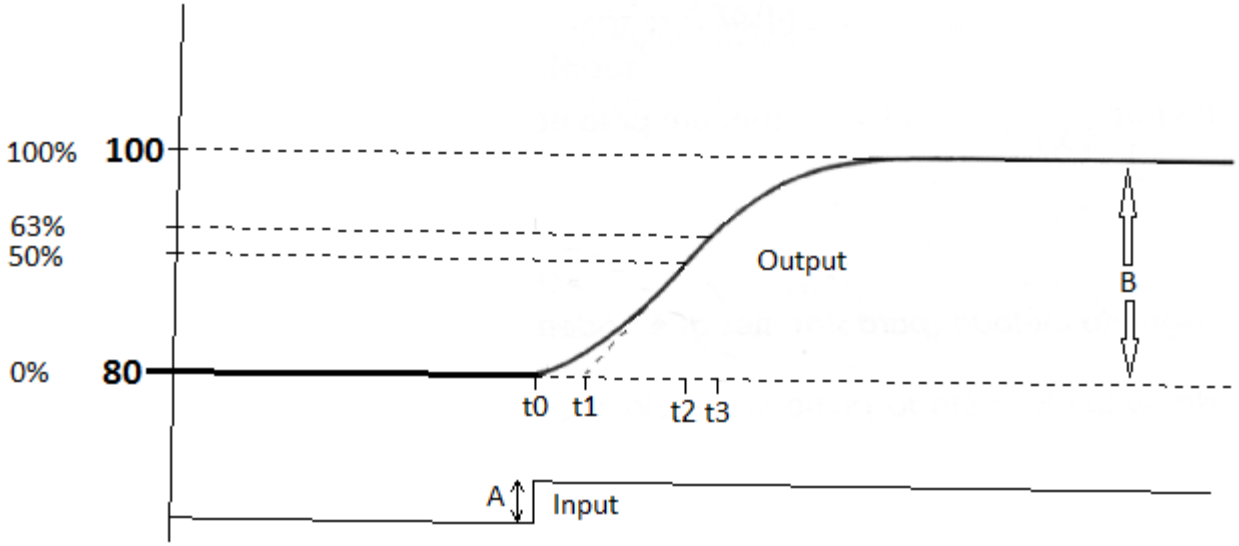
If we shift this graph to the right so that *t* starts at t_1 and not *t=0*, we get . . .

$$V(t) = (V_{max} - V_{min}) * \left(1 - e^{-(t-t_1)/\tau}\right) + V_{min}$$

Remember Tau is difference in time from 0 to 63%. If we plug in $\tau = t_3 - t_1, V = f$, we get the generic formula of . . .

$$f(t) = (f_{max} - f_{min}) * \left(1 - e^{-(t-t_1)/(t_3-t_1)}\right) + f_{min}$$

COHEN-COON TUNING METHOD



$$f(t) = (f_{max} - f_{min}) * \left(1 - e^{-(t-t_1)/(t_3-t_1)}\right) + f_{min}$$

We are going to have this graph as our data so we'll know f_{max}, f_{min}, t_0, t_2, and t_3. We won't know t_1, but if we plug in t_2 into this equation we can find t_1.

$$f(t_2) = (f_{max} - f_{min}) * \left(1 - e^{-(t_2-t_1)/(t_3-t_1)}\right) + f_{min}$$

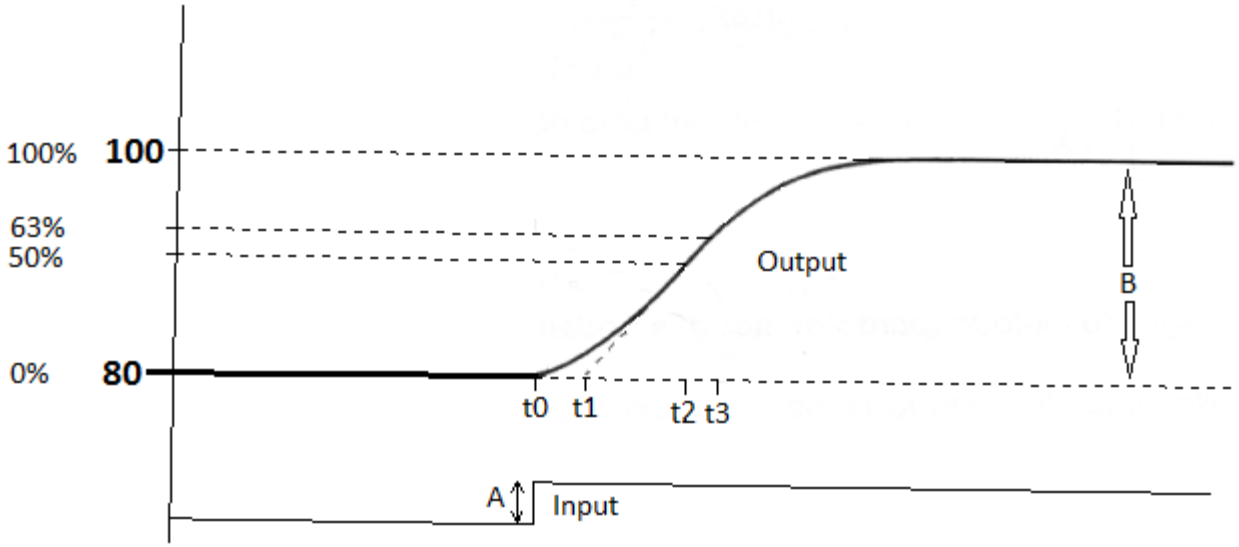
$$0.5 * (f_{max} - f_{min}) + f_{min} = (f_{max} - f_{min}) * \left(1 - e^{-(t_2-t_1)/(t_3-t_1)}\right) + f_{min}$$

$$0.5 * (f_{max} - f_{min}) = (f_{max} - f_{min}) * \left(1 - e^{-(t_2-t_1)/(t_3-t_1)}\right)$$

$$0.5 = \left(1 - e^{-(t_2-t_1)/(t_3-t_1)}\right)$$

(continues, stay tuned! hee hee)

COHEN-COON TUNING METHOD



$$0.5 = \left(1 - e^{-(t_2 - t_1)/(t_3 - t_1)}\right)$$

$$-0.5 = -e^{-(t_2 - t_1)/(t_3 - t_1)}$$

$$0.5 = e^{-(t_2 - t_1)/(t_3 - t_1)}$$

$$ln(0.5) = -(t_2 - t_1)/(t_3 - t_1)$$

$$ln(0.5) * (t_3 - t_1) = -(t_2 - t_1)$$

$$t_3 * ln(0.5) - t_1 * ln(0.5) = -t_2 + t_1$$

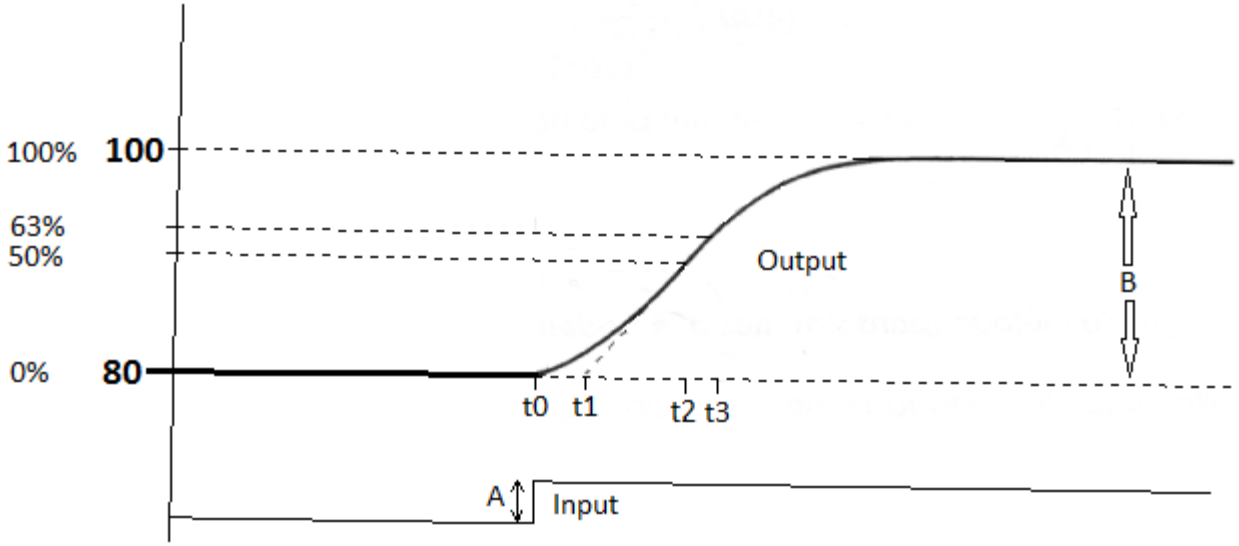
$$-t_1 - t_1 * ln(0.5) = -t_2 - t_3 * ln(0.5)$$

$$t_1 + t_1 * ln(0.5) = t_2 + t_3 * ln(0.5)$$

$$t_1\left(1 + ln(0.5)\right) = t_2 + t_3 * ln(0.5)$$

$$\boldsymbol{t_1 = \frac{t_2 + t_3 * ln(0.5)}{1 + ln(0.5)}}$$

COHEN-COON TUNING METHOD



$t_1 = \frac{t_2 + t_3 * ln(0.5)}{1 + ln(0.5)}$, $\tau = t_3 - t_1$, ***A = delta input, B = delta output (in this example B = 20)***, **let** $g_p = B/A$ **and** $t_d = t_1 - t_0$.

www.opticontrols.com	Controller Gain	Integral Time	Derivative Time
P Controller:	$K_c = \frac{1.03}{g_p}\left(\frac{\tau}{t_d} + 0.34\right)$		
PI Controller:	$K_c = \frac{0.9}{g_p}\left(\frac{\tau}{t_d} + 0.092\right)$	$T_I = 3.33 t_d \frac{\tau + 0.092 t_d}{\tau + 2.22 t_d}$	
PD Controller:	$K_c = \frac{1.24}{g_p}\left(\frac{\tau}{t_d} + 0.129\right)$		$T_D = 0.27 t_d \frac{\tau - 0.324 t_d}{\tau + 0.129 t_d}$
PID Controller: (Noninteracting)	$K_c = \frac{1.35}{g_p}\left(\frac{\tau}{t_d} + 0.185\right)$	$T_I = 2.5 t_d \frac{\tau + 0.185 t_d}{\tau + 0.611 t_d}$	$T_D = 0.37 t_d \frac{\tau}{\tau + 0.185 t_d}$

$$K_P = K_C$$
$$K_I = \frac{K_C}{T_I}$$
$$K_D = K_C * T_D$$