PID Tuning

Q: How should we select the gains k_p, k_i, k_d?

There are many ways:

- optimization
- heuristics-based methods
- "Brute Force" searches
- Guessing ← Don't do this.

Ziegler-Nichols Method

redefine gain parameters → $u(t) = k_p e + k_d \dot{e} + k_i \int e\,dt$ ← factor out k_p

$$= k_p\left(e + \frac{k_d}{k_p}\dot{e} + \frac{k_i}{k_p}\int e\,dt\right)$$

define $T_d = k_d/k_p$ and $T_i = \frac{k_p}{k_i}$ } notice inversion

1st method: For non-oscillatory systems

Issue a step input and observe the response

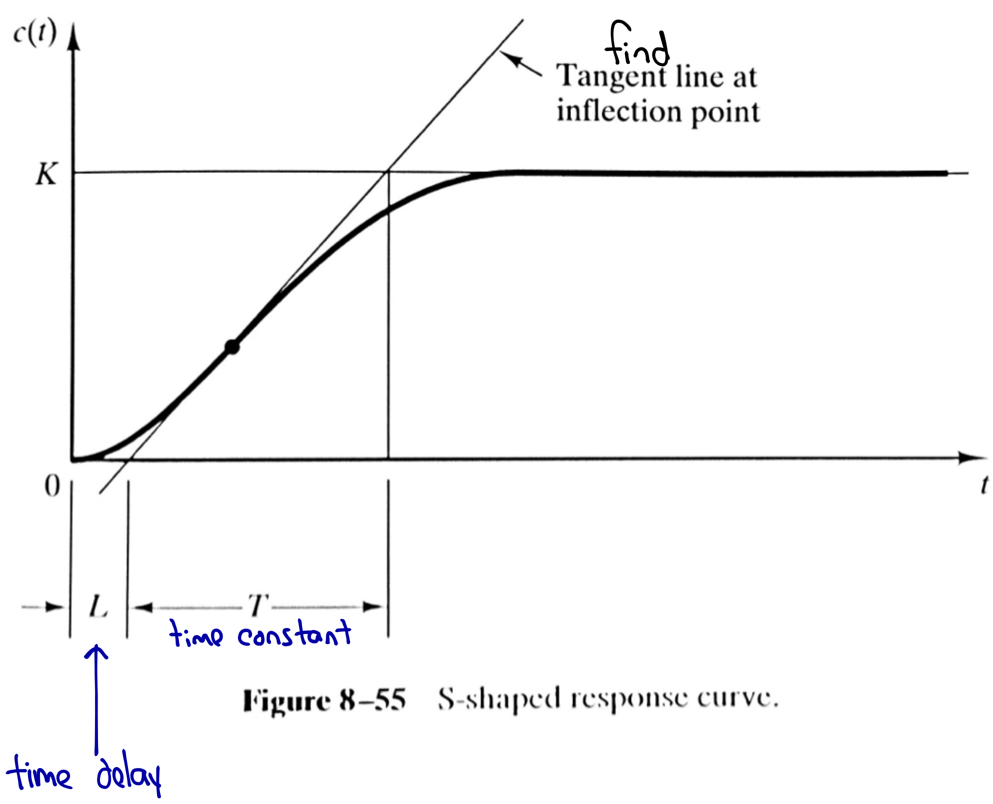

Figure 8–55 S-shaped response curve.

Note: If the response doesn't look like this, then this method does not apply.

Once you find T and L, use the table below to select the gains.

Controller Type	K_p	T_i	T_d
Proportional (P)	$\frac{T}{L}$		
Proportional-Integral (PI)	$0.9\frac{T}{L}$	$\frac{L}{0.3}$	
Proportional-Integral-Derivative (PID)	$1.2\frac{T}{L}$	$2L$	$0.5L$

Ziegler-Nichols Method (cont)

2nd Method: (for oscillatory systems)

1) Start with proportional control.
2) Slowly increase k_p until sustained oscillations occur (like below)

Note: if sustained oscillation cannot be induced, this method does not apply.

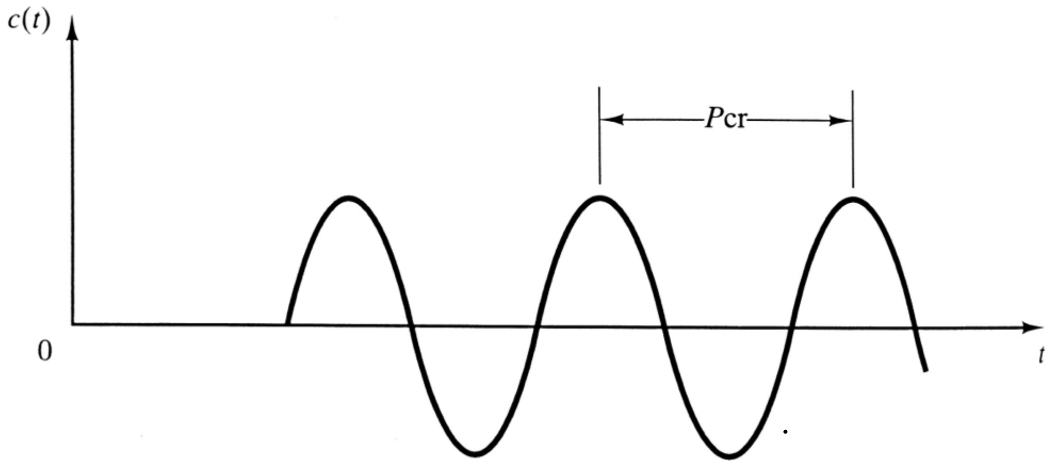

Figure 8–57 Sustained oscillation with period P_{cr}.

3) Define: K_{cr} – gain at which sustained oscillation occurs

P_{cr} – period of oscillation at this gain.

4) Pick gains from the table below.

Controller Type	K_p	T_i	T_d
Proportional (P)	$0.50K_{cr}$		
Proportional-Integral (PI)	$0.45K_{cr}$	$\frac{1}{1.2}P_{cr}$	
Proportional-Integral-Derivative (PID)	$0.60K_{cr}$	$0.5P_{cr}$	$0.125P_{cr}$

PID Implementation

$$u(t) = k_p e + k_d \dot{e} + k_i \int e \, dt$$

Some implementation detail:

- Most loops are not strictly timed (each loop may take a slightly different time)

Pseudo-Code

```
output_type PID(kp,ki,kd){
   get current time
   deltaT = last time - current time   } How long has it been since last updated?

   compute:                           Xd      -   X
      current error = desired state - measured states
      error sum = last error sum + (current error * deltaT) <-- numerical integration
      error derivative = (current error - last error) / deltaT <- Numerical deriv. via Finite difference

   output = (kp * current error) + (kd * error derivative)
         + (ki * error sum)

   last time = current time              } save current data into global
   last error sum = error sum            } variables for use in the next loop.

   return output
}
```

PID Implementation Concerns

From: http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction

Sample Time - The PID algorithm functions best if it is evaluated at a regular interval. If the algorithm is aware of this interval, we can also simplify some of the internal math.

Derivative Kick - Not the biggest deal, but easy to get rid of, so we're going to do just that.

On-The-Fly Tuning Changes - A good PID algorithm is one where tuning parameters can be changed without jolting the internal workings.
Reset Windup Mitigation -We'll go into what Reset Windup is, and implement a solution with side benefits

On/Off (Auto/Manual) - In most applications, there is a desire to sometimes turn off the PID controller and adjust the output by hand, without the controller interfering

Initialization - When the controller first turns on, we want a "bumpless transfer." That is, we don't want the output to suddenly jerk to some new value

Controller Direction - This last one isn't a change in the name of robustness per se. it's designed to ensure that the user enters tuning parameters with the correct sign.

PID Implementation Concerns - Sample Time

The algorithm (is designed and) works best at a fixed sampling rate.

Most micro-controllers do not operate at fixed cycles.

A few options:

- do nothing (is performance okay?... then okay)
- programmatically enforce fixed update rate
- interrupts (code that says "stop everything else and do this" at an event such as button push or timer)

A change to our pseudo-code

```
define desired sample time

output_type PID(kp,ki,kd){
   get current time
   deltaT = last time - current time

   if (deltaT >= desired sample time){

       compute:
      current error = desired state - measured states
      error sum = last error sum + (current error * sample time)
      error derivative = (current error - last error) / sample time

       output = (kp * current error) + (kd * error derivative)
         + (ki * error sum)

   last time = current time
   last error sum = error sum

   return output
   }
   else {
       don't do anything
   }

}
```

only update if the time since last update is ≥ desired sample time

Notice that these are now sample_time, not ΔT...

Because of this we could simplify the math further.

if the desired sample time has not elapsed, don't do anything