intro.pid Introducing PID control

1 One of the most ubiquitous types is the **proportional-integral-derivative** (PID) *controller*. It has a transfer function with real constants K_P, K_I, and K_D:

PID controller

$$C(s) = \underbrace{K_P}_{\text{proportional}} + \underbrace{K_I/s}_{\text{integral}} + \underbrace{K_D s}_{\text{derivative}}. \quad (1)$$

Remember: the controller operates on the error $E(s)$, so the PID controller effectively sums terms proportional to the error, its integral, and its derivative. Inspecting this in the time domain with error $e(t)$ by taking the inverse Laplace transform of the output $U(s) = C(s)E(s)$,

$$u(t) = \underbrace{K_P e(t)}_{\text{proportional}} + \underbrace{K_I \int_0^t e(\theta)\, d\theta}_{\text{integral}} + \underbrace{K_D \dot{e}(t)}_{\text{derivative}}. \quad (2)$$

2 So the control effort u is responsive to:

- **P** the amount and direction of error (reactive, spring-like),
- **I** the accumulation of error over time (memoried, mass-like), and
- **D** the time rate of change of the error (anticipatory, damper-like).

Although the mechanical spring-mass-damper analog above has its limitations, it is helpful for our intuition. More generally, we can consider the three constants K_P, K_I, and K_D to be "knobs" with which we can include more or less of each term.

3 Just how a controller will affect the closed-loop response is significantly dependent on the *plant* dynamics. Therefore, there is no way to make fully general statements about the impact of each of the PID terms. This is why we need the detailed analytic design tools of Chapter rldesign and the intervening chapters hence. However, for some simple systems, we can make the assertions of Table pid.1.

Table pid.1: occasionally true generalities about PID controller terms.

Proportional	Integral	Derivative
• is the workhorse • speeds up responses • can lead to instability when too large	• improves or eliminates steady-state error • slows down the response • becomes a liability when it can't forget (integral windup)	• speeds up the response • can yield jitter when measurement noise is large • can lead to instability when measurement noise is large

4 There are many methods of **tuning** a PID controller: selecting K_P, K_I, and K_D to meet certain performance criteria. The **root locus**

tuning

root locus

design method of Chapter rldesign and the **frequency response** design method of Chapter freqd allow us to precisely design for specific performance criteria. However, there are times when specific performance criteria and involved analysis are not available or convenient. In these cases, hand-tuning is possible via several algorithms. One such algorithm is presented in the following section.

frequency response

Ziegler–Nichols tuning method

5 The Ziegler–Nichols method of tuning a PID controller is presented in the following algorithm.

1. Set $K_P, K_I, K_D = 0$.
2. Increase K_P until a marginally stable response[1] is observed.
3. Record this **ultimate gain** K_u and the **oscillation period** T_u.
4. Set the controller gains:

$$K_P = 0.6K_u \quad K_I = 1.2K_u/T_u \quad K_D = 3K_uT_u/40. \tag{3}$$

1. This can be the impulse, step, or free response. Furthermore, it can be oscillatory.

ultimate gain K_u

oscillation period T_u

Example intro.pid-1

re: hand-tuning a PID controller

For the block diagram of Fig. pid.1, with the plant

$$G(s) = \frac{15000}{s^4 + 50s^3 + 875s^2 + 6250s + 15000}$$

use the Ziegler–Nichols method to design a PID controller $C(s)$.

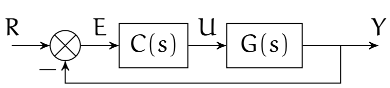

Figure pid.1: block diagram for Example intro.pid-1.

We proceed with Matlab, symbolically at first. Let's define the transfer functions.

```
syms S kp ki kd % S is the laplace transform s
G_sym = 15000/(S^4+50*S^3+875*S^2+6250*S+15000); % plant
C_sym = kp + ki/S + kd*S; % PID controller transfer fun
```

From the preceding lecture's **??**, the closed-loop transfer function is as follows.

```
CL_sym = simplify( ...
```

```
    C_sym*G_sym/(1+C_sym*G_sym) ...
)
```

```
CL_sym =

(15000*kd*S^2 + 15000*kp*S + 15000*ki)/(15000*S +
↪  15000*ki + 15000*S*kp + 15000*S^2*kd + 6250*S^2
↪  + 875*S^3 + 50*S^4 + S^5)
```

I have created a function `sym_to_tf` that creates a `tf` object, which we'll need for simulation.[a]

```
type sym_to_tf.m
```

```
function tf_obj = sym_to_tf(sym_tf,s_var)
  % TODO test to make sure s_var is in
  ↪  symvar(sym_tf) ...
  syms(symvar(sym_tf))
  syms s
  sym_tf = subs(sym_tf,s_var,s);
  tf_str = char(sym_tf);
  s = tf([1,0],[1]);
  eval(['tf_obj = ',tf_str,';']);
```

Let's wrap it in a function of our own `K_sub`, which will create a closed-loop `tf` object from our `CL_sym` with the PID gains included.

```
K_sub = @(Kp,Ki,Kd) sym_to_tf( ...
    subs( ...
        CL_sym, ...
        {kp,ki,kd}, ...
        {Kp,Ki,Kd} ...
    ), ...
    S ...
);
K_sub(1,0,0) % e.g.
```

```
ans =

                  15000 s
  -------------------------------------------
  s^5 + 50 s^4 + 875 s^3 + 6250 s^2 + 30000 s

Continuous-time transfer function.
```

Now let's use `impulse` to simulate the response starting with a small proportional gain.

```
[y,t] = impulse(K_sub(1,0,0));
```

Now, we should plot the result – see Fig. pid.2.

```
figure
plot(t,y)
grid on
xlabel('time (s)')
ylabel('impulse response')
```

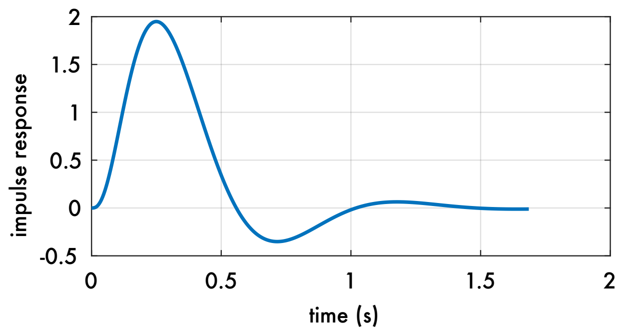

Figure pid.2: impulse response with (small) $K_P = 1$.

If we iteratively increase $K_P = 1 \to 3 \to 5.25$ (the response for each of these values is plotted in Fig. pid.3), we find that around the last value, the system becomes marginally stable and therefore

$$K_u = 5.25. \tag{4}$$

The oscillation period appears to be around $T_u = 0.56$ seconds. Defining these quantities, we can now compute K_I and K_D from Eq. 3.

```
Ku = 5.25;
Tu = 0.56;
KP = 0.6*Ku;
KI = 1.2*Ku/Tu;
KD = 3*Ku*Tu/40;
disp(sprintf( ...
    'KP = %0.2f, KI = %0.2f, KD = %0.2f', ...
    KP,KI,KD ...
))
```

```
KP = 3.15, KI = 11.25, KD = 0.22
```

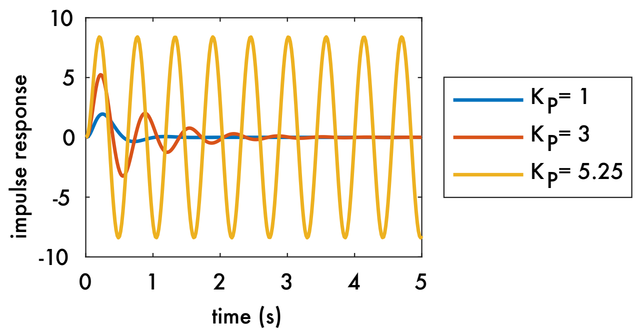

Figure pid.3: impulse responses with $K_I = K_D = 0$ and K_P as shown.

Let's try out this controller for step response and see how it looks.

```
[y,t] = step(K_sub(KP,KI,KD));
figure
plot(t,y)
xlabel('time (s)')
ylabel('step response')
```

The resulting step response is plotted in Fig. pid.4. We didn't have specific expectations for performance, here, but this result is a nice, average-looking step response with some overshoot and a decent settling time.

a. The function is available in the repo: `github.com/ricopicone/matlab-rico`.