intro.pidi · An interactive PID controller design

1 In this lecture, we will build an interactive PID control design tool in Python. However, you need not install Python[2] to try the design tool: it is available at the following web page.

2. For more on Python, see `python.org`.

click to launch interactive page in browser

It may take a few minutes to load the Jupyter notebook.[3] Once it does, click Cell ⟩ Run All . This will run the Python code that comprises the remainder of this lecture. Scroll to the bottom of the webpage to interact with the PID gains that update the closed-loop step response plot!

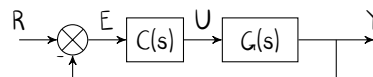3. For more on Jupyter, see `jupyter.org`.

Figure pidi.1: a unity feedback control loop.

2 For the unity feedback block diagram of Fig. pidi.1, we will design a PID controller C(s). Design requirements are (a) less than 20 percent overshoot, (b) an initial peak in less than 0.2 seconds, and (c) zero steady-state error for a step response.

3. Python code in this section was generated from a Jupyter notebook named `pid_interactive_design_python.ipynb` with a python3 kernel.

First, load some general-purpose Python packages.

```python
import numpy as np # for numerics
import sympy as sp # for symbolics
import control as c # the Control Systems module
import matplotlib as mpl # for plots
import matplotlib.pyplot as plt # also for plots
from IPython.display import display, Markdown, Latex
```

The following Python packages are specific for
the interactive widget.

```python
from ipywidgets import *
%matplotlib widget
```

Symbolic transfer functions

Let's investigate the transfer functions
symbolically. We begin by defining the Laplace s
and gain symbolic variables.

```python
s,K_p,K_i,K_d = sp.symbols('s K_p K_i K_d')
```

We will design a PID controller for a plant
with the following transfer function.

```python
G_sym = 15000/(s**4+50*s**3+875*s**2+6250*s+15000)
display(G_sym)
```

$$\frac{15000}{s^4 + 50s^3 + 875s^2 + 6250s + 15000}$$

The controller has the following symbolic
transfer function.

```
C_sym = K_p + K_i/s + K_d*s
display(C_sym)
```

$$K_d s + \frac{K_i}{s} + K_p$$

The closed-loop transfer function for the unity feedback system is as follows.

```
T_sym = sp.simplify(
    C_sym*G_sym/(1+C_sym*G_sym)
)
T_num, T_den = list( # for simplifying
    map(
        lambda x: sp.collect(x,s),
        sp.fraction(T_sym)
    )
)
T_sym = T_num/T_den
display(T_sym)
```

$$\frac{15000K_i + s\left(15000K_d s + 15000K_p\right)}{15000K_i + s\left(15000K_d s + 15000K_p + s^4 + 50s^3 + 875s^2 + 6250s + 15000\right)}$$

Symbolic to control transfer functions

The control package has objects of type TransferFunction that will be useful for simulation in the next section. We begin by defining a function to convert a symbolic transfer function to a control TransferFunction object.

```python
def sym_to_tf(tf_sym,s_var):
    global s # changes s globally!
    S = s_var
    s = sp.symbols('s')
    tf_sym = tf_sym.subs(S,s)
    tf_str = str(tf_sym)
    s = c.TransferFunction.s
    ldict = {}
    exec('tf_out = '+tf_str,globals(),ldict)
    tf_out = ldict['tf_out']
    return tf_out
```

This isn't smooth, but it works. Note that
tf_sym must have no symbolic variables besides
s_var, the Laplace s. We can apply this to G_sym,
then, but not yet C_sym.

```python
type(sym_to_tf(G_sym,s))
```

```
control.xferfcn.TransferFunction
```

Defining the closed-loop function

We need to create a function that specifies the
gains, substitutes them into the symbolic
closed-loop transfer function, then converts it
to a control package TransferFunction
object via sym_to_tf.

```python
def pid_CL_tf(CL_sym,Kp=0,Ki=0,Kd=0):
    sp.symbols('K_p K_i K_d')
    s = c.TransferFunction.s
    CL_subs = CL_sym.subs({K_p: Kp, K_i: Ki, K_d: Kd})
    return sym_to_tf(CL_subs,s)
```

For instance, we can let $K_p = 1$ and $K_i = K_d = 0$.

```
display(
    pid_CL_tf(T_sym,Kp=1)
)
```

$$\frac{1.5 \times 10^4}{s^4 + 50s^3 + 875s^2 + 6250s + 3 \times 10^4}$$

Step response

It is straightforward to use the control package's **step_response** function to get a step response for a single set of gains.

```
gains = {'Kp':2, 'Ki':1, 'Kd':0.1}
sys_CL = pid_CL_tf(T_sym,**gains)
t_step = np.linspace(0,3,200)
t_step,y_step = c.step_response(sys_CL, t_step)
```

Now let's plot it. The result is shown in Fig. pidi.2.

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
line, = ax.plot(t_step, y_step)
plt.xlabel('time (s)')
plt.ylabel('step response')
plt.show()
```
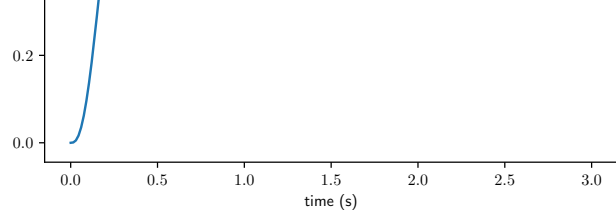
Figure pidi.2: step response with $K_p, K_i, K_d = 2, 1, 0.1$.

Interactive step response

The following essentially repeats the same process of

1. setting the PID gains with `pid_CL_tf`,
2. simulating with `step_response`, and
3. plotting the response.

The caveat is that this happens with a GUI interaction callback function update that sets new gains (based on the GUI sliders), simulates, and replaces the old line on the plot. The final plot is shown in **??**. It appears to meet our performance requirements.

```
%matplotlib widget
# simulate
t_step = np.linspace(0,3,200)
sys_CL = pid_CL_tf(T_sym,Kp=1)
t_step,y_step = c.step_response(sys_CL, t_step)

# initial plot
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
line, = ax.plot(t_step, y_step)
plt.xlabel('time (s)')
plt.ylabel('step response')
plt.show()

# GUI callback function
def update(Kp = 1.0, Ki = 0.0, Kd = 0.0):
    global t_step, kp, ki, kd
    kp,ki,kd = Kp,Ki,Kd
    sys_CL = pid_CL_tf(T_sym,Kp=Kp,Ki=Ki,Kd=Kd)
    t_step,y_step = c.step_response(sys_CL, t_step)
    line.set_ydata(y_step)
    ax.relim()
```

```
    ax.autoscale_view()
    fig.canvas.draw_idle()
    plt.show()

# interaction definition
interact(
  update,
  Kp=(0.0,10.0),
  Ki=(0.0,20.0),
  Kd=(0.0,1.0)
);
```
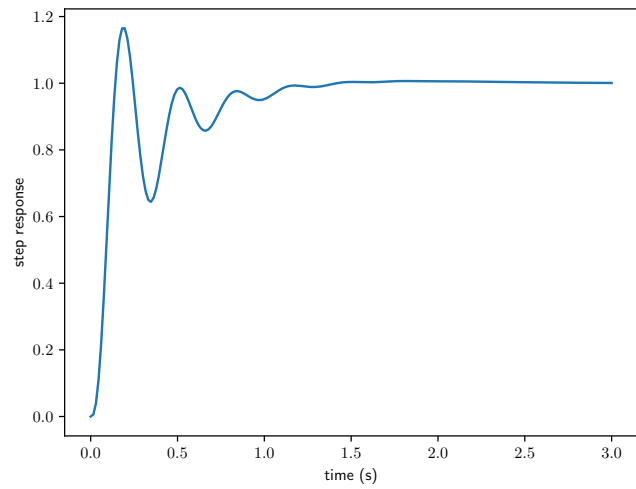

Figure pidi.3: step response from interaction with $K_p, K_i, K_d = 3.1, 6.2, 0.8$.
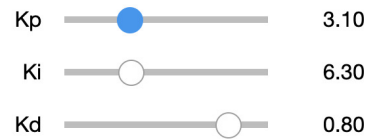
The sliders appear as shown in Fig. pidi.4.

Figure pidi.4: this is how the sliders should look.