

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 Python² 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 [\[a\]](#) [\[b\]](#). 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.

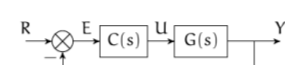


Figure pidi.1: a unity feedback control loop.

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

```
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.

```
from ipynbwidgets import *
import matplotlib widget
```

Symbolic transfer functions

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

```
s, Kp, Ki, Kd = sp.symbols('s Kp Ki Kd')
```

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

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

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

The controller has the following symbolic transfer function.

```
C_symb = Kp + Ki/s + Kd*s
display(C_symb)
```

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

```
T_symb = sp.simplify(
    C_symb*G_symb/(1+C_symb*G_symb)
)
T_num, T_den = list(c.for_simplifying(
    sp)
    lambda x: sp.collect(x,s),
    sp.fraction(T_symb)
)
T_symb = T_num/T_den
display(T_symb)
```

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

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.

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

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

```
type(sym_to_tf(G_symb,a))
```

`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`.

```
def pid_cl_tf(CL_symb,Kp=0,Ki=0,Kd=0):
    sp.symbols('Kp Ki Kd')
    a = c.TransferFunction.a
    CL_symb = CL_symb.subs({Kp: Kp, Ki: Ki, Kd: Kd})
    return sym_to_tf(CL_symb,a)
```

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

```
display(
    pid_cl_tf(T_symb,Kp=1)
)
```

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

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_symb,*gains)
t_step = np.linspace(0,3,200)
y_step, y_step_c = c.step_response(sys_cl, t_step)
```

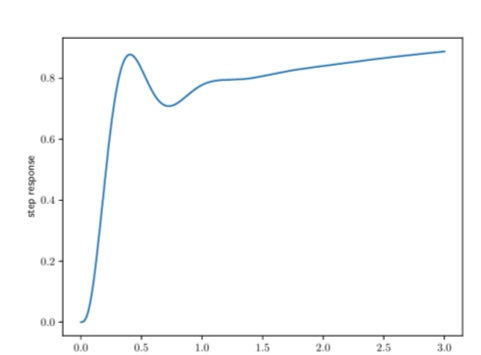


Figure pidi.2: step response with $K_p, K_i, K_d = 2, 1, 0.1$.

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.

```
import matplotlib widget
# simulate
t_step = np.linspace(0,3,200)
sys_cl = pid_cl_tf(T_symb,Kp=1)
y_step,y_step_c = 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_symb,Kp=Kp,Ki=Ki,Kd=Kd)
    t_step,y_step_c = c.step_response(sys_cl, t_step)
    line.set_ydata(y_step_c)
    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)
);
```

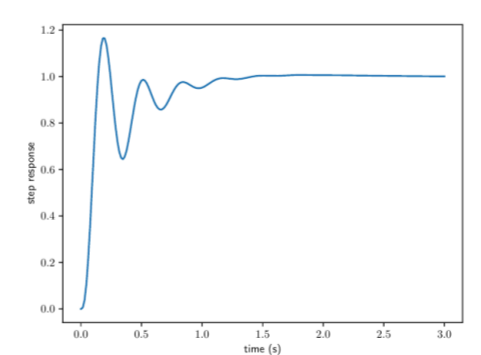


Figure pidi.3: step response from interaction with $K_p, K_i, K_d = 3.1, 6.2, 0.8$.

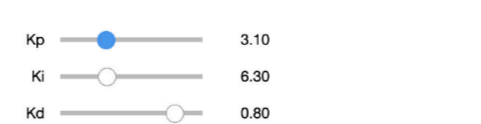


Figure pidi.4: this is how the sliders should look.