Simulating Saturation

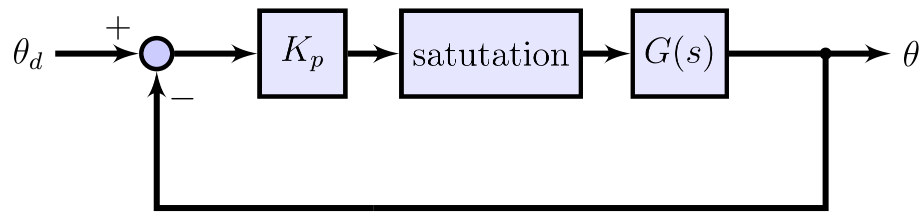


Figure 1: Block diagram representing saturation in a feedback control system

This notebook gives two options for simulating s feedback control system that includes saturation, like the one shown above. The first option is to use the `python-control` module and specifically `control.forced_response` inside of a `for` loop. The main advantage of this method is that it still allows the user to think in terms of transfer functions. Using `control.forced_response` can be a little slow for plants with pure integrators. The second option is to fall back to numeric integration (Runge-Kutta) using `scipy.intergrate.odeint`. This approach can be very fast and can handle arbitrary nonlinearities, but it requires programming differential equations rather than using transfer functions.

The first step under the first option is simulating the affects of saturation on a closed-loop system is simulating the system one time step at a time so that the code can limit the input to the plant transfer function. The key challenge in simulating the response one step at a time is that the initial conditions to `control.forced_response` need to be the ending conditions from the previous time step.

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import control
```

Consider a plant under proportional control:

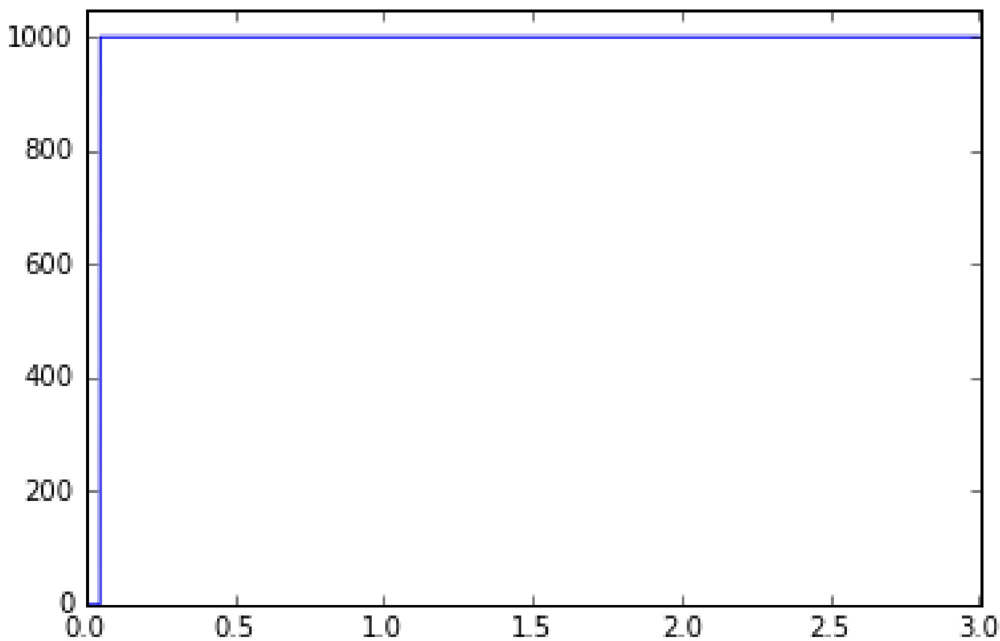
```
p = 6
g = 5
G = control.TransferFunction(g*p,[1,p,0])
```

Simple Simulation Ignoring Saturation

If we just wanted to simulate the closed-loop response without concern for saturation, we could do the following:

```
kp = 5
cltf = control.feedback(kp*G)
```

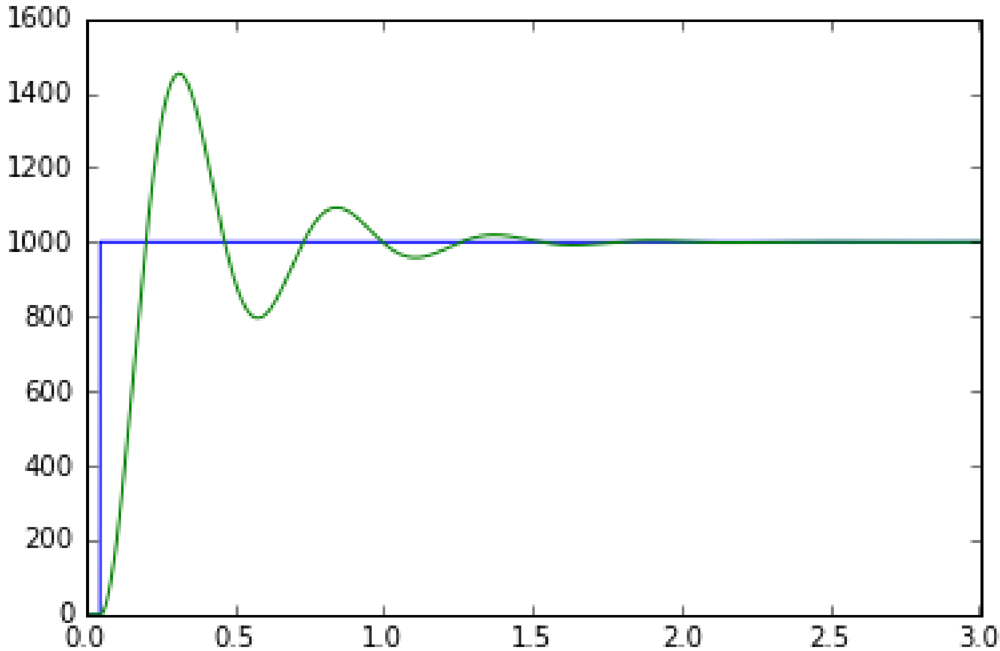
```
cltf

      150
---------------
s^2 + 6 s + 150

t = np.arange(0,3,0.001)
amp = 1000
u = np.zeros(len(t))
u[50:] = amp
plt.figure()
plt.plot(t,u)
plt.ylim([0,1050])

(0, 1050)
```




```
t, y_fb, x = control.forced_response(cltf, t, u)

y_fb.shape

(3000,)

x.shape

(2, 3000)
```

```
plt.figure()
plt.plot(t,u,t,y_fb)
```

```
[<matplotlib.lines.Line2D at 0x11597de48>,
 <matplotlib.lines.Line2D at 0x11597dfd0>]
```



One Step at a Time Simulation

```
n = len(G.pole())
x_prev = np.zeros(n)
y_one_step = np.zeros(len(t))
dt = t[1]-t[0]
pwm_vect = np.zeros(len(t))

import pdb

for i, t_i in enumerate(t):
    e = u[i] - y_one_step[i-1]
    pwm = kp*e
    pwm_vect[i] = pwm
    t_temp, y_temp, x_temp = control.forced_response(G,[t_i-dt,t_i],[pwm
    y_one_step[i] = np.squeeze(y_temp[-1])
    x_prev = np.squeeze(x_temp[:,-1])
```

```
x_temp
```

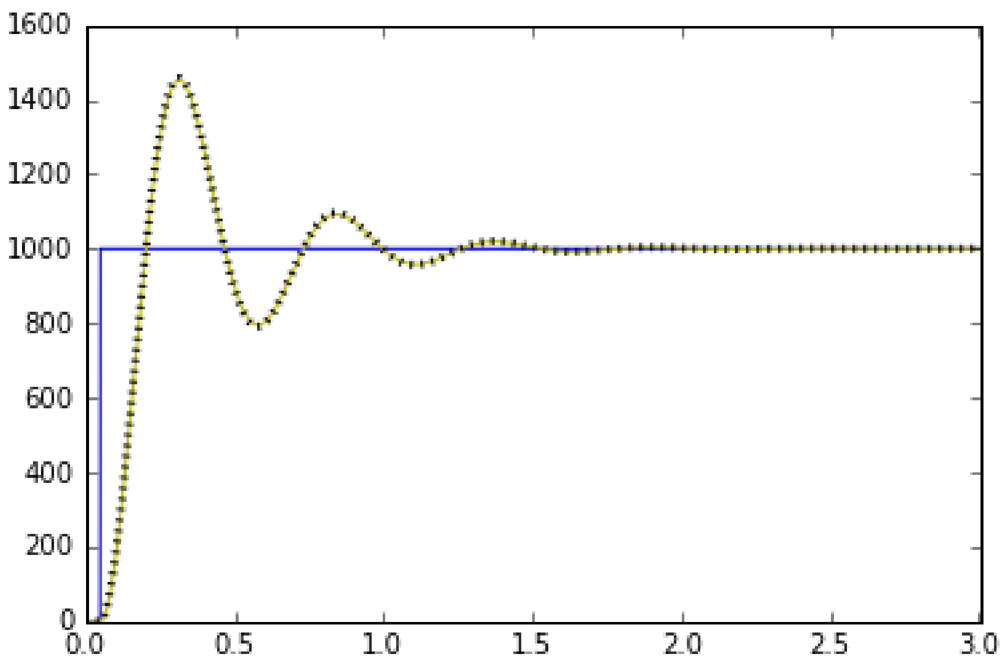
```
array([[ -3.16497465e-02,  -3.22616017e-02],
       [  3.33386906e+01,   3.33386587e+01]])
```

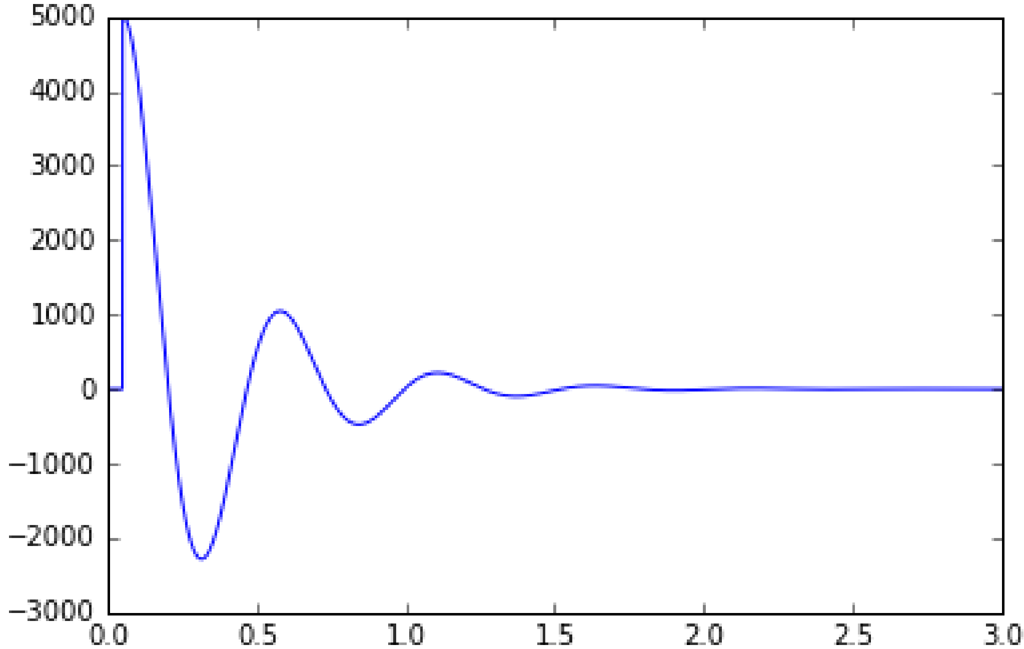
```
y_one_step
```

```
array([    0.        ,     0.        ,     0.        , ...,  1000.16165
        1000.16071821,  1000.15975953])
```

```
plt.figure()
plt.plot(t,u,t,y_fb,'y')
plt.plot(t, y_one_step,'k:',linewidth=3)

plt.figure()
plt.plot(t,pwm_vect)
```

```
[<matplotlib.lines.Line2D at 0x115b3bb38>]
```






```
G.pole()
```

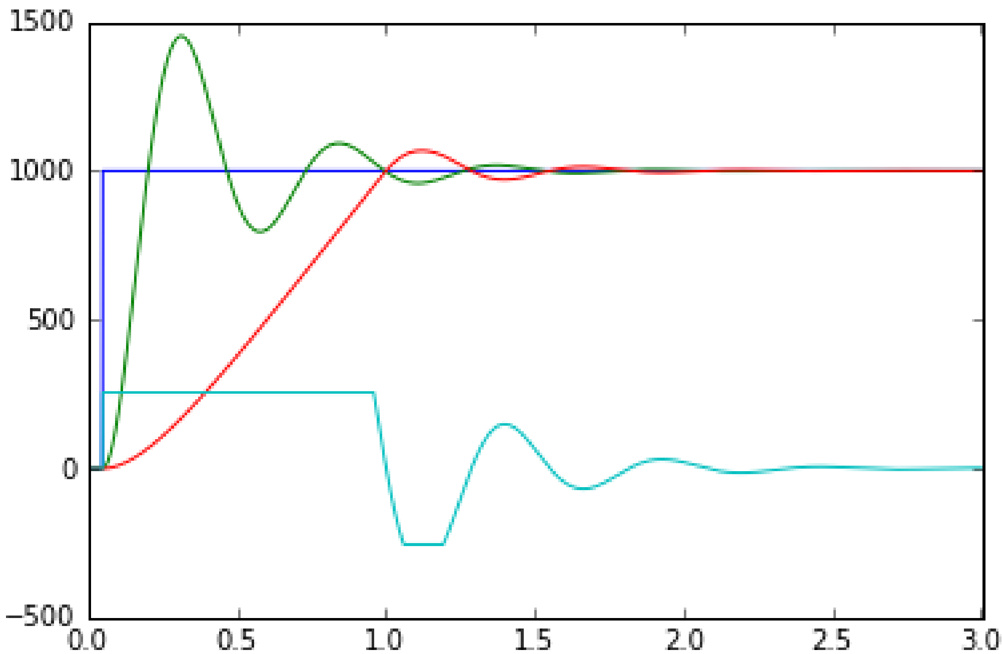
```
array([-6.,  0.])
```

Saturation Control Function

```
def P_control_sat_sim(G, Kp, t, u, pos_sat=255, neg_sat=-255):
    def mysat(pwmin):
        if pwmin > pos_sat:
            pwmout = pos_sat
        elif pwmin < neg_sat:
            pwmout = neg_sat
        else:
            pwmout = pwmin
        return pwmout

    n = len(G.pole())
    x_prev = np.zeros(n)
    y_one_step = np.zeros(len(t))
    dt = t[1]-t[0]
    pwm_vect = np.zeros(len(t))

    for i, t_i in enumerate(t):
```

```
        e = u[i] - y_one_step[i-1]
        pwm = kp*e
        pwm_star = mysat(pwm)
        pwm_vect[i] = pwm_star
        t_temp, y_temp, x_temp = control.forced_response(G,[t_i-dt,t_i]
                                                          [pwm_star,pwm_
        y_one_step[i] = np.squeeze(y_temp[-1])
        x_prev = np.squeeze(x_temp[:,-1])

    return y_one_step, pwm_vect

y_sat, pwm_vect = P_control_sat_sim(G,5,t,u)

plt.figure()
plt.plot(t,u,t,y_fb,t,y_sat, t, pwm_vect)
```

```
[<matplotlib.lines.Line2D at 0x115d26a58>,
 <matplotlib.lines.Line2D at 0x115d26c18>,
 <matplotlib.lines.Line2D at 0x115d2d588>,
 <matplotlib.lines.Line2D at 0x115d2dda0>]
```




```
mylist = ['a','b','c']

for i, item in enumerate(mylist):
    print('%i: %s' % (i,item))
```

```
0: a
1: b
2: c
```

Differential Equation Approach

An alternative to using `control.forced_response` inside a `for` loop is to use `integrate.odeint`. `odeint` avoids transfer functions entirely and works directly on the differential equation. An advantage of `odeint` is that it can handle fully nonlinear systems.

If the transfer function is

$$G(s) = \frac{\Theta(s)}{V(s)} = \frac{gp}{s^2 + ps} \tag{1}$$

then the corresponding differential equation is

$$\ddot{\theta} + p\dot{\theta} = gpv \tag{2}$$

In order to use `odeint`, the differential equation model needs to be rearranged into a series of first order differential equations by defining states:

$$x_1 = \theta \tag{3}$$

$$x_2 = \dot{\theta} \tag{4}$$

Solving for the derivatives of the states gives

$$\dot{x}_1 = x_2 \tag{5}$$

$$\dot{x}_2 = gpv - px_2 \tag{6}$$

```python
def mysat(vin):
    if vin > 255:
        vout = 255
    elif vin < -255:
        vout = -255
    else:
        vout = vin
    return vout

def dxdt(x,t,kp,theta_d,use_sat=True):
    theta = x[0]
    theta_dot = x[1]
    e = theta_d - theta
```

```
    v = kp*e
    if use_sat:
        v = mysat(v)
    out = [theta_dot, g*p*v-p*theta_dot]
    return out

from scipy import integrate

x_mat1 = integrate.odeint(dxdt,[0,0],t,args=(5,1000,False))

x_mat2 = integrate.odeint(dxdt,[0,0],t,args=(5,1000,True))

plt.figure()
plt.plot(t,x_mat1[:,0])
plt.plot(t,x_mat2[:,0])
```

```
[<matplotlib.lines.Line2D at 0x11586c278>]
```