Final Exam – Simulation Results

ECEn 483/ ME 431

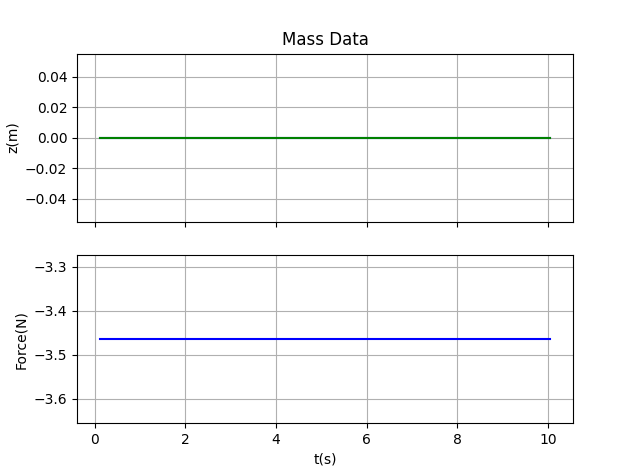
Winter 2021

Name: Mitchell Gehlbach

At the end of the exam, make sure to include this file with your submission.

# Part 2. Design models

2.2 Insert plot of the output of the simulation model with initial condition  and input directly below this line.

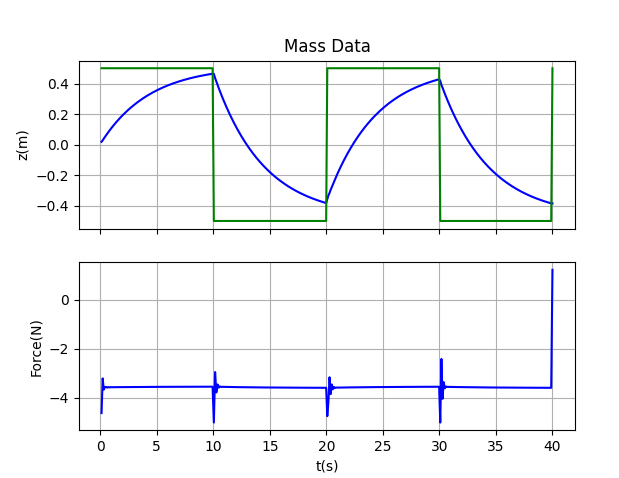


2.6 Insert the code you used for simulating the dynamics (this should nominally be massDynamics.py, unless you chose to use the complied version and lose 10 points):

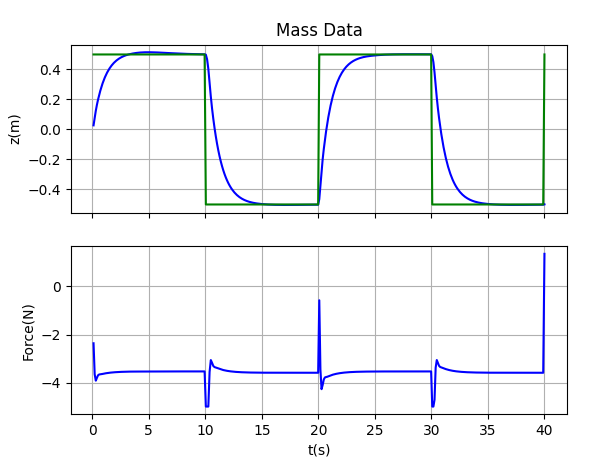
import numpy as np  
import massParam as P  
  
class massDynamics:  
 def \_\_init\_\_(self, alpha=0.0):  
 self.state = np.array([[0.0],  
 [0.0]])  
 self.g = P.g  
 self.theta = 45 \* np.pi / 180 \* (1.+alpha\*(2.\*np.random.rand()-1.))  
 self.m = P.m \* (1.+alpha\*(2.\*np.random.rand()-1.))  
 self.Fmax = P.F\_max  
 self.k1 = P.k1\* (1.+alpha\*(2.\*np.random.rand()-1.))  
 self.k2 = P.k2 \* (1.+alpha\*(2.\*np.random.rand()-1.))  
 self.b = P.b \* (1.+alpha\*(2.\*np.random.rand()-1.))  
 self.Ts = P.Ts  
 self.force\_limit = P.F\_max  
  
 def update(self, u):  
 u = self.saturate(u, self.force\_limit)  
 self.rk4\_step(u)  
 y = self.h()  
 return y  
  
 def f(self, state, F):  
 # Return xdot = f(x,u), the system state update equations  
 # re-label states for readability  
 z = state.item(0)  
 zdot = state.item(1)  
 xdot = np.array([[zdot],  
 [(1/self.m)\*(F - self.b\*zdot - self.k1\*z - self.k2\*z\*\*3 + (1/np.sqrt(2))\*self.m\*self.g)]])  
  
 return xdot  
  
 def h(self):  
 # return the output equations  
 # could also use input u if needed  
 z = self.state.item(0)  
 y = np.array([  
 [z],  
 ])  
 return y  
  
 def rk4\_step(self, u):  
 # Integrate ODE using Runge-Kutta RK4 algorithm  
 F1 = self.f(self.state, u)  
 F2 = self.f(self.state + self.Ts / 2 \* F1, u)  
 F3 = self.f(self.state + self.Ts / 2 \* F2, u)  
 F4 = self.f(self.state + self.Ts \* F3, u)  
 self.state += self.Ts / 6 \* (F1 + 2 \* F2 + 2 \* F3 + F4)  
  
 def saturate(self, u, limit):  
 if abs(u) > limit:  
 u = limit \* np.sign(u)  
 return u

# Part 3. PID Control

3.6 Insert a plot that shows both and when is a square wave with magnitude meters and frequency 0.05 Hz, and when using a PD controller.



3.7 Insert a plot that shows both and when is a square wave with maginitude meters and frequency 0.05 Hz, and when using a PID controller.



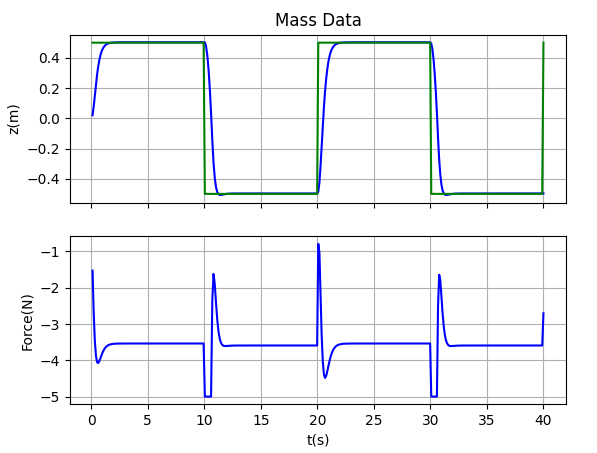
**NOTE: I found that using the calculated kd value seemed to overdamp the system. When implementing the PID controller, I changed kd to be 0.5 instead of 2.147. This gave a much better rise time. I used a ki of 2 to eliminate steady state error.**

3.8 Insert the controller code that implements PID control directly below this line.

import numpy as np  
import massParam as P  
  
class controllerPID:  
  
 def \_\_init\_\_(self):  
 self.kp = P.kp  
 self.ki = P.ki  
 self.kd = P.kd  
 self.limit = P.F\_max  
 self.beta = (2\*P.sigma-P.Ts)/(2\*P.sigma+P.Ts)  
 self.Ts = P.Ts  
 self.z\_d1 = 0.0  
 self.z\_dot = 0.0  
 self.error\_d1 = 0.0  
 self.error\_dot = 0.0  
 self.integrator = 0.0  
 self.F\_e = P.F\_e  
  
 def update(self, z\_r, y):  
 z = y.item(0)  
 error = z\_r - z  
  
 # derivative control  
 self.error\_dot = self.beta \* self.error\_dot + (1 + self.beta) \* ((error - self.error\_d1) / self.Ts)  
 self.error\_d1 = error  
 self.z\_dot = self.beta \* self.z\_dot + (1 + self.beta) \* ((z - self.z\_d1) / self.Ts)  
 self.z\_d1 = z  
  
 # integral control  
 threshold = 0.1  
 if np.abs(self.error\_dot) <= threshold:  
 self.integrator = self.integrator + (self.Ts / 2.0) \* (error + self.error\_d1)  
 self.error\_d1 = error  
  
 # calculate the control force  
 F\_unsat = self.kp \* error - self.kd \* self.z\_dot + self.ki \* self.integrator  
  
 F\_e = P.k1 \* z + P.k2 \* z \*\* 3 - (1 / np.sqrt(2)) \* P.m \* P.g  
  
 F = self.saturate(F\_unsat + F\_e)  
  
 return F  
  
 def saturate(self, u):  
 if abs(u) > self.limit:  
 u = self.limit\*np.sign(u)  
 return u  
  
 def integratorAntiWindup(self, F, F\_unsat):  
 if self.ki != 0.0:  
 self.integrator = self.integrator + self.Ts/self.ki \* (F - F\_unsat)

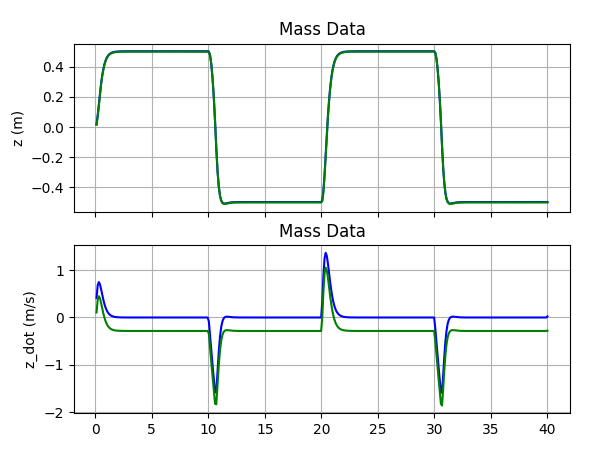
# Part 4. Observer based control

4.5. Insert a plot of the step response of the system for the complete observer based control.



**NOTE: In order to get this performance, I changed the desired poles from -2+/- 2j to -5+/-0.1j. I also changed the eigenvalues of A-LC to be 10 times that of A-BK, instead of 5 times as great. I kept the integrator pole the same at -5. See the param file for the implementation.**

4.6. Insert a plot of the state estimation error.

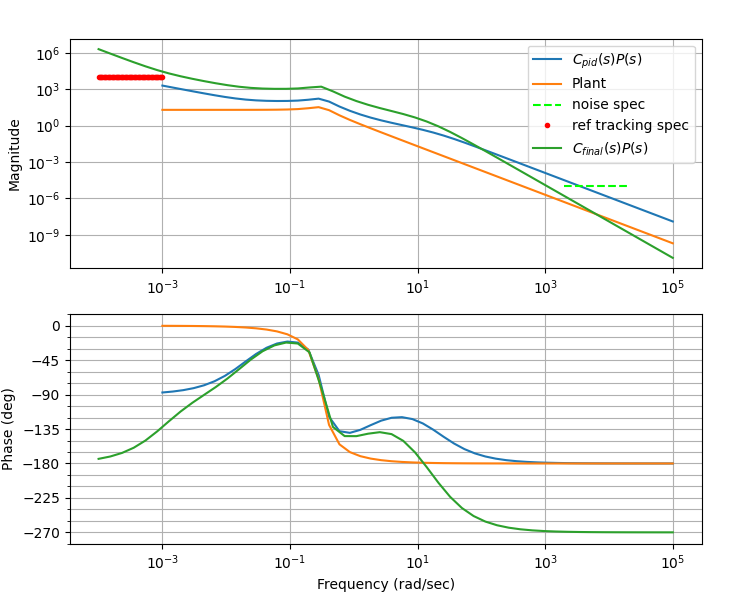


4.7. Insert a copy of your controller code which includes both the state feedback controller AND the observer.

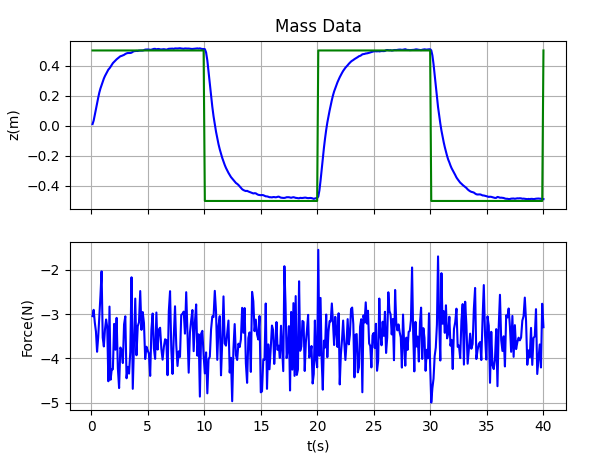
import numpy as np  
import massParam as P  
  
class controllerObsv:  
 def \_\_init\_\_(self):  
 self.observer\_state = np.array([])  
 self.x\_hat = np.array([[0.0],  
 [0.0]])  
 self.F\_d1 = 0.0  
 self.integrator = 0.0  
 self.error\_d1 = 0.0  
 self.K = P.K  
 self.ki = P.ki2 # grab the second one, not the one found from PID control  
 self.L = P.L  
 self.A = P.A  
 self.B = P.B  
 self.C = P.C  
 self.limit = P.F\_max  
 self.F\_e = P.F\_e  
 self.Ts = P.Ts  
  
 def update(self, z\_r, y):  
 x\_hat = self.update\_observer(y)  
 z\_hat = self.x\_hat.item(0)  
  
 # integrate error  
 error = z\_r - z\_hat  
 self.integrateError(error)  
  
 # compute the state feedback controller  
 F\_tilde = -self.K @ x\_hat - self.ki\*self.integrator  
  
 F\_e = P.k1 \* z\_hat + P.k2 \* z\_hat \*\* 3 - (1 / np.sqrt(2)) \* P.m \* P.g  
 F = self.saturate(F\_tilde.item(0) + F\_e)  
  
 self.F\_d1 = F  
 # self.integratorAntiWindup(F, F\_tilde+F\_e)  
  
 return F, x\_hat  
  
 def update\_observer(self, y):  
 # update the observer using RK4 integration  
 F1 = self.observer\_f(self.x\_hat, y)  
 F2 = self.observer\_f(self.x\_hat + self.Ts / 2 \* F1, y)  
 F3 = self.observer\_f(self.x\_hat + self.Ts / 2 \* F2, y)  
 F4 = self.observer\_f(self.x\_hat + self.Ts \* F3, y)  
 self.x\_hat += self.Ts / 6 \* (F1 + 2 \* F2 + 2 \* F3 + F4)  
 x\_hat = np.array([[self.x\_hat.item(0)],  
 [self.x\_hat.item(1)]])  
 return x\_hat  
  
 def observer\_f(self, x\_hat, y\_m):  
 xhat\_dot = self.A @ x\_hat + self.B \* self.F\_d1 + self.L @ (y\_m - self.C @ x\_hat)  
 return xhat\_dot  
  
 def integrateError(self, error):  
 self.integrator = self.integrator + (self.Ts / 2.0) \* (error + self.error\_d1)  
 self.error\_d1 = error  
  
 def integratorAntiWindup(self, F, F\_unsat):  
 if self.ki != 0.0:  
 self.integrator = self.integrator + self.Ts/self.ki \* (F-F\_unsat)  
  
 def saturate(self,u):  
 if abs(u) > self.limit:  
 u = self.limit\*np.sign(u)  
 return u

# Part 5. Loopshaping

5.6 Insert the Bode plots for the original plant, the PID controlled plant, and the loopshaped controlled plant below this line.



5.7 Insert simulation results for the loopshaping controller below this line.



5.8 Insert a copy of your loopshaping AND controller code below this line.

import massParam as P  
import matplotlib.pyplot as plt  
from control import tf, tf2ss  
import control as cnt  
import numpy as np  
  
# in this file, you should will need to:  
#### 1)define xfer functions for the plant, and the PID controller  
#### 2)define specifications (for C\_pid \* P, and for the final controller C)  
#### 3)make bode plots to help you in meeting specifications and responding to questions  
  
# Compute plant transfer functions  
# using new gains  
kp = 5  
kd = 2.9  
ki = 0.1  
  
Plant = tf([1.0/P.m], [1, P.b/P.m, P.k1/P.m])  
C\_pid = tf([(kd+kp\*P.sigma), (kp+ki\*P.sigma), ki],  
 [P.sigma, 1, 0])  
  
# Question 5.1  
omegas = [0.2, 20.0] # omega\_d, omega\_no  
mag, phase, omegas = cnt.bode(Plant\*C\_pid, plot=False, dB=False, omega = omegas)  
print('Mag at 0.2 rad/s: ', mag[0])  
print('Mag at 20 rad/s: ', mag[1])  
  
# display bode plots of transfer functions  
# fig1 = plt.figure()  
# cnt.bode([Plant, Plant\*C\_pid, Plant\*C\_pid/(1+Plant\*C\_pid)], dB=False)  
# plt.legend(('No control - $P(s)$', '$C\_{pid}(s)P(s)$', 'Closed-loop PID'))  
# fig1.axes[0].set\_title('Mass Spring Damper')  
#  
# fig2 = plt.figure()  
# cnt.bode([Plant, Plant\*C\_pid], dB=False, margins=True)  
# fig2.axes[0].set\_title('Mass Spring Damper - Stability Margins')  
# plt.show()  
  
# Calculate the phase and gain margin  
gm, pm, Wcg, Wcp = cnt.margin(Plant \* C\_pid)  
  
dB\_flag = False  
if dB\_flag:  
 print("gm: ", gm, " pm: ", pm, " Wcg: ", Wcg, " Wcp: ", Wcp)  
else:  
 print("gm: ", cnt.mag2db(gm), " pm: ", pm, " Wcg: ", Wcg, " Wcp: ", Wcp)  
  
# compute final compensator C, and prefilter F  
def add\_control\_lag(C, z, M):  
 Lag = tf([1, z], [1, z/M])  
 return C \* Lag  
  
def add\_control\_lpf(C, p):  
 LPF = tf(p, [1, p])  
 return C \* LPF  
  
def add\_control\_proportional(C, kp):  
 proportional = tf([kp], [1])  
 return C \* proportional  
  
def add\_spec\_noise(gamma\_n, omega\_n, dB\_flag = False):  
 w = np.logspace(np.log10(omega\_n), 1 + np.log10(omega\_n))  
 fig = plt.gcf()  
 if dB\_flag == False:  
 fig.axes[0].loglog(w, gamma\_n \* np.ones(len(w)),  
 '--', color=[0, 1, 0], label = 'noise spec')  
 else:  
 fig.axes[0].semilogx(w, 20.0\* np.log10(gamma\_n) \* np.ones(len(w)),  
 '--', color=[0, 1, 0], label = 'noise spec')  
  
def add\_spec\_ref\_tracking(gamma\_r, omega\_r, dB\_flag = False):  
 w = np.logspace(np.log10(omega\_r)-1, np.log10(omega\_r))  
 fig = plt.gcf()  
 if dB\_flag == False:  
 fig.axes[0].loglog(w, 1./gamma\_r\*np.ones(len(w)),  
 '.', color=[1, 0, 0],label='ref tracking spec')  
 else:  
 fig.axes[0].semilogx(w, 20.\*np.log10(1/gamma\_r)\*np.ones(len(w)),  
 '.', color=[1, 0, 0],label='ref tracking spec')  
  
C = C\_pid  
C = add\_control\_lag(C, z=.001, M=(1/1e-6))  
C = add\_control\_lpf(C, p=10000)  
C = add\_control\_proportional(C, kp=1)  
F = tf(1, 1)  
F = add\_control\_lpf(F, p=0.8)  
  
# convert them to state space models if you are going to use that method  
# otherwise you can just use C and F directly in the controllerLoop.py file.  
Css=cnt.tf2ss(C)  
Fss=cnt.tf2ss(F)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # calculate bode plot and gain and phase margin for original PID \* plant dynamics  
 mag, phase, omega = cnt.bode(Plant \* C\_pid, dB=dB\_flag,  
 omega=np.logspace(-3, 5),  
 plot=True, label="$C\_{pid}(s)P(s)$")  
  
 gm, pm, Wcg, Wcp = cnt.margin(Plant \* C\_pid)  
 print("for original C\_pid system:")  
 print(" pm: ", pm, " Wcp: ", Wcp, "gm: ", gm, " Wcg: ", Wcg)  
 mag, phase, omega = cnt.bode(Plant, dB=dB\_flag,  
 omega=np.logspace(-3, 5),  
 plot=True, label="Plant")  
  
 #########################################  
 # Define Design Specifications  
 #########################################  
  
 # ----------- noise specification --------  
 omega\_n = 2000 # attenuate noise above this frequency  
 gamma\_n = 1e-5 # attenuate noise by this amount  
 add\_spec\_noise(gamma\_n, omega\_n)  
  
 # ----------- general tracking specification --------  
 omega\_r = 0.001 # track signals below this frequency  
 gamma\_r = 1e-4 # tracking error below this value  
 add\_spec\_ref\_tracking(gamma\_r, omega\_r)  
  
 ## plot the effect of adding the new compensator terms  
 mag, phase, omega = cnt.bode(Plant \* C, dB=dB\_flag,  
 omega=np.logspace(-4, 5),  
 plot=True, label="$C\_{final}(s)P(s)$")  
  
 gm, pm, Wcg, Wcp = cnt.margin(Plant \* C)  
 print("for final C\*P:")  
 print(" pm: ", pm, " Wcp: ", Wcp, "gm: ", gm, " Wcg: ", Wcg)  
  
 fig = plt.gcf()  
 fig.axes[0].legend()  
 plt.show()  
  
 ############################################  
 # now check the closed-loop response with prefilter  
 ############################################  
 # Closed loop transfer function from R to Y - no prefilter  
 CLOSED\_R\_to\_Y = (Plant \* C / (1.0 + Plant \* C))  
 # Closed loop transfer function from R to Y - with prefilter  
 CLOSED\_R\_to\_Y\_with\_F = (F \* Plant \* C / (1.0 + Plant \* C))  
 # Closed loop transfer function from R to U - no prefilter  
 CLOSED\_R\_to\_U = (C / (1.0 + Plant \* C))  
 # Closed loop transfer function from R to U - with prefilter  
 CLOSED\_R\_to\_U\_with\_F = (F \* C / (1.0 + Plant \* C))  
  
 fig = plt.figure()  
 plt.grid(True)  
 mag, phase, omega = cnt.bode(CLOSED\_R\_to\_Y, dB=dB\_flag, plot=True,  
 color=[0, 0, 1], label='closed-loop $\\frac{Y}{R}$ - no pre-filter')  
 mag, phase, omega = cnt.bode(CLOSED\_R\_to\_Y\_with\_F, dB=dB\_flag, plot=True,  
 color=[0, 1, 0], label='closed-loop $\\frac{Y}{R}$ - with pre-filter')  
 fig.axes[0].set\_title('Closed-Loop Bode Plot')  
 fig.axes[0].legend()  
  
 plt.figure()  
 plt.subplot(211), plt.grid(True)  
 T = np.linspace(0, 2, 100)  
 \_, yout\_no\_F = cnt.step\_response(CLOSED\_R\_to\_Y, T)  
 \_, yout\_F = cnt.step\_response(CLOSED\_R\_to\_Y\_with\_F, T)  
 plt.plot(T, yout\_no\_F, 'b', label='response without prefilter')  
 plt.plot(T, yout\_F, 'g', label='response with prefilter')  
 plt.legend()  
 plt.ylabel('Step Response')  
  
 plt.subplot(212), plt.grid(True)  
 \_, Uout\_no\_F = cnt.step\_response(CLOSED\_R\_to\_U, T)  
 \_, Uout\_F = cnt.step\_response(CLOSED\_R\_to\_U\_with\_F, T)  
 plt.plot(T, Uout\_no\_F, color='b', label='control effort without prefilter')  
 plt.plot(T, Uout\_F, color='g', label='control effort with prefilter')  
 plt.ylabel('Control Effort')  
 plt.legend()  
  
 plt.show()

import massParam as P  
import loopshape\_mass as L  
import numpy as np  
from discreteFilter import discreteFilter  
  
class controllerLoop:  
 def \_\_init\_\_(self):  
 self.A\_C = L.Css.A  
 self.B\_C = L.Css.B  
 self.C\_C = L.Css.C  
 self.D\_C = L.Css.D  
 self.A\_F = L.Fss.A  
 self.B\_F = L.Fss.B  
 self.C\_F = L.Fss.C  
 self.D\_F = L.Fss.D  
 n = self.A\_C.shape[0]  
 self.x\_C = np.zeros((n, 1))  
 n = self.A\_F.shape[0]  
 self.x\_F = np.zeros((n, 1))  
 self.limit = P.F\_max  
 self.F\_e = P.F\_e  
 self.Ts = P.Ts  
  
 self.prefilter = discreteFilter(L.F.num, L.F.den, P.Ts)  
 self.control = discreteFilter(L.C.num, L.C.den, P.Ts)  
  
 def update(self, z\_r, y):  
 z = y.item(0)  
  
 # prefilter  
 z\_r\_filtered = self.prefilter.update(z\_r)  
  
 # error signal  
 error = z\_r\_filtered - z  
  
 # find the control  
 F\_tilde = self.control.update(error)  
  
 # compute total force  
 F = self.saturate(F\_tilde + self.F\_e)  
  
 return F.item(0)  
  
 def saturate(self, u):  
 if abs(u) > self.limit:  
 u = self.limit\*np.sign(u)  
 return u

# 6. Insert your parameter (or other code) file that computes all control gains here.

import numpy as np  
  
# one of these two forms for importing control library functions may be helpful.  
import control as cnt  
#from control import tf, bode, etc.  
  
# physical parameters  
g = 9.8  
theta = 45.0\*np.pi/180.0 #this variable just defines the slope of the block, and may not be used explicitly  
m = 0.5  
k1 = 0.05  
k2 = 0.02  
F\_max = 5.0  
b = 0.1  
  
# INITIAL CONDITIONS  
z0 = 0.0  
zdot0 = 0.0  
z\_e = 0.0  
  
# Equilibrium Force  
F\_e = k1\*z\_e + k2\*z\_e\*\*3 - (1/np.sqrt(2))\*m\*g  
  
## PID CONTROL GAIN CALCULATIONS ##  
sigma = 0.005  
zeta = 0.707  
kp = F\_max / 1.0  
wn = np.sqrt(k1/m + kp/m)  
kd = 0.5 # m\*(2\*zeta\*wn - (b/m))  
ki = 2.0  
  
print('kp: ', kp)  
print('kd: ', kd)  
print('ki: ', ki)  
  
# ----------------------------------------------  
# FULL STATE FEEDBACK AND OBSERVER DESIGN  
  
# Statespace equations  
A = np.array([[0.0, 1.0],  
 [(-k1/m), (-b/m)]])  
B = np.array([[0.0],  
 [(1/(m))]])  
C = np.array([[1.0, 0.0]])  
  
# Augmented Statespace system for use with integrator full state feedback  
A1 = np.array([[0.0, 1.0, 0.0],  
 [(-k1/m), (-b/m), 0.0],  
 [-1.0, 0.0, 0.0]])  
B1 = np.array([[0.0],  
 [(1/m)],  
 [0.0]])  
  
# gain calculation  
des\_poles = np.array([-5+0.1j, -5-0.1j])  
polesI = np.append(des\_poles, -5)  
if np.linalg.matrix\_rank(cnt.ctrb(A1, B1)) != 3:  
 print('The system is not controllable.')  
else:  
 K1 = cnt.place(A1, B1, polesI)  
 K = np.array([K1.item(0), K1.item(1)])  
 ki2 = K1.item(2)  
  
# observer calculation  
obsv\_poles = 10 \* des\_poles  
  
# compute gains if the system is observable  
if np.linalg.matrix\_rank(cnt.ctrb(A.T, C.T)) != 2:  
 print('The system is not observable.')  
else:  
 L = cnt.acker(A.T, C.T, obsv\_poles).T  
  
print('K: ', K)  
print('ki2: ', ki2)  
print('L^T: ', L.T)  
  
# ----------------------------------------------  
  
  
# simulation parameters  
t\_start = 0.0  
t\_end = 40.0  
Ts = 0.01  
t\_plot = 0.1  
sigma = 0.05