Practice Final Exam – Simulation Results

ECEn 483/ ME 431

Winter 2023

Name:_____Jacob Child______________________

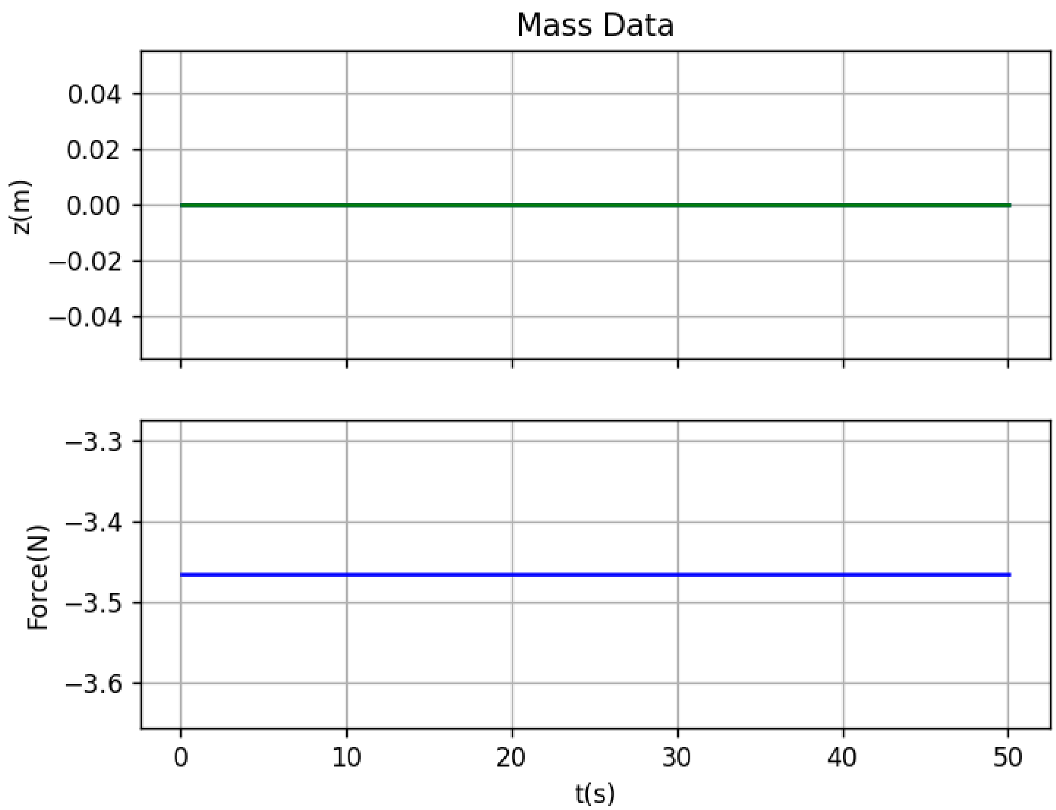
At the end of the exam, print this file and staple it to the handout portion of the exam.

To print to the digital lab Caedm printer, go to https://webprint.et.byu.edu/ and then select EB423.

Part I (25 pts)	
Part II (25 pts)	
Part III (25 pts)	
Part IV (25 pts)	
Total: (100 pts)	

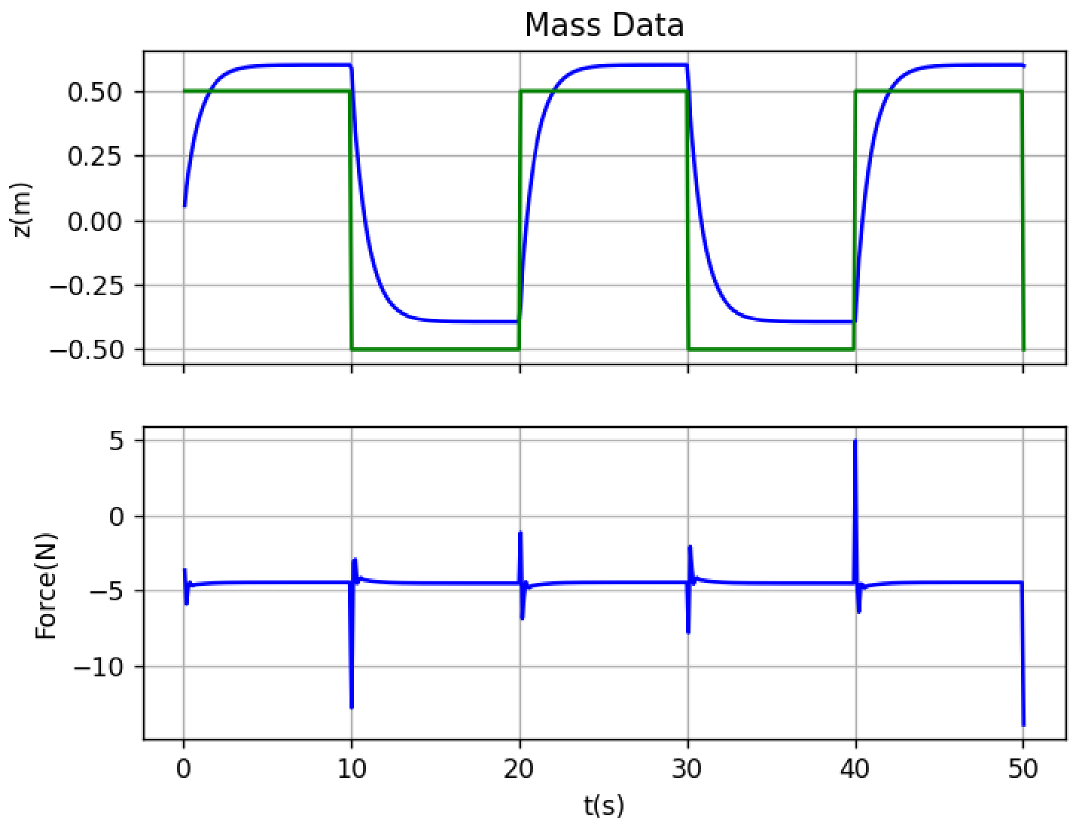
Part 1. Design models

1.2 Insert plot of the output of the simulation model with initial condition $z(0) = z_e$ and input F_e directly below this line.

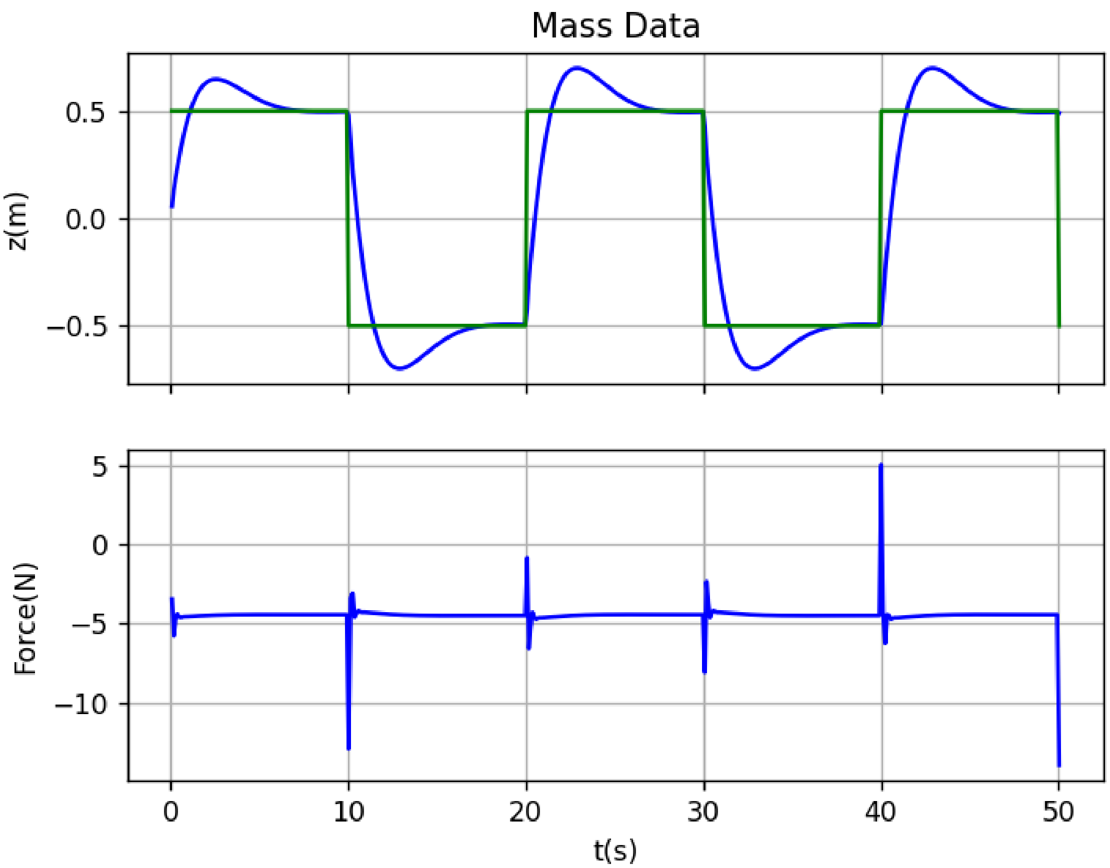


Part 2. PID Control

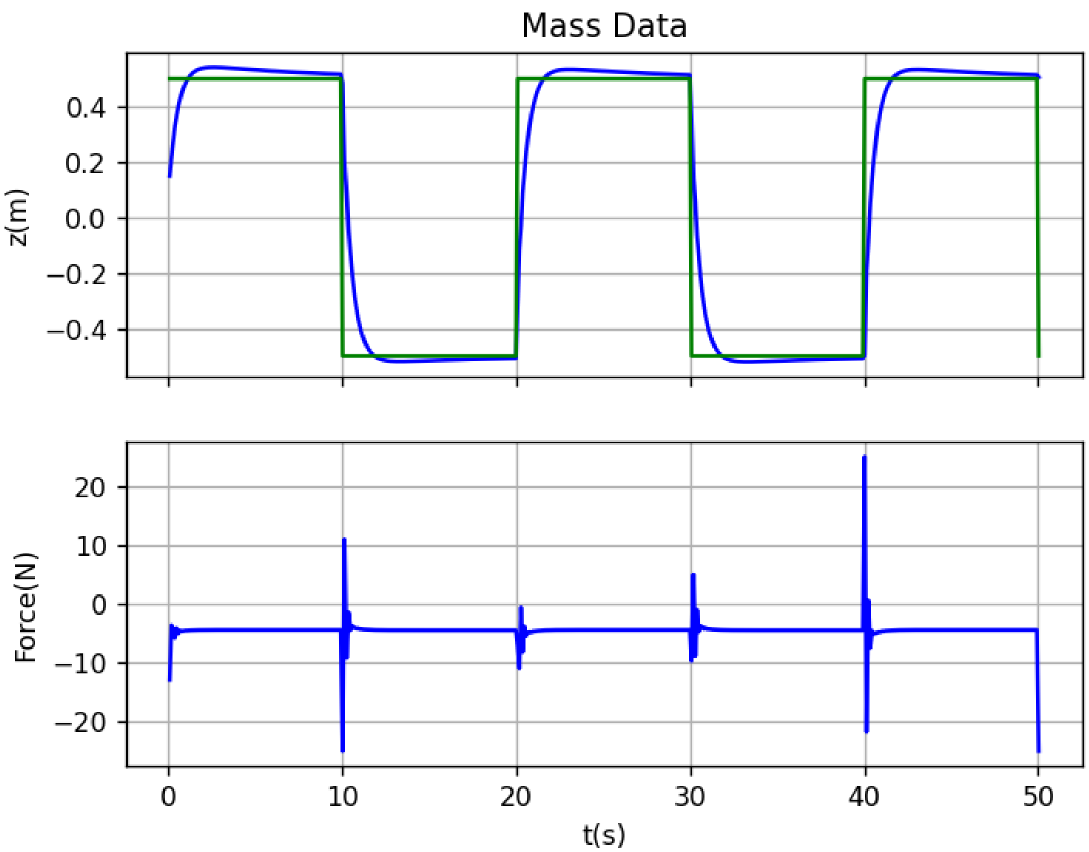
2.4 Insert a plot that shows both z and z^r when z^r is a square wave with magnitude ± 0.5 meters and frequency 0.05 Hz, and when using a PD controller.



2.5 Insert a plot that shows both z and z^r when z^r is a square wave with maginitude ± 0.5 meters and frequency 0.05 Hz, and when using a PID controller.



****If you set tr = 0.25 (and leave everything else the same) you get a bit better performance although some more tuning would need to be done to get rid of the wee bit of steady state error lingering*



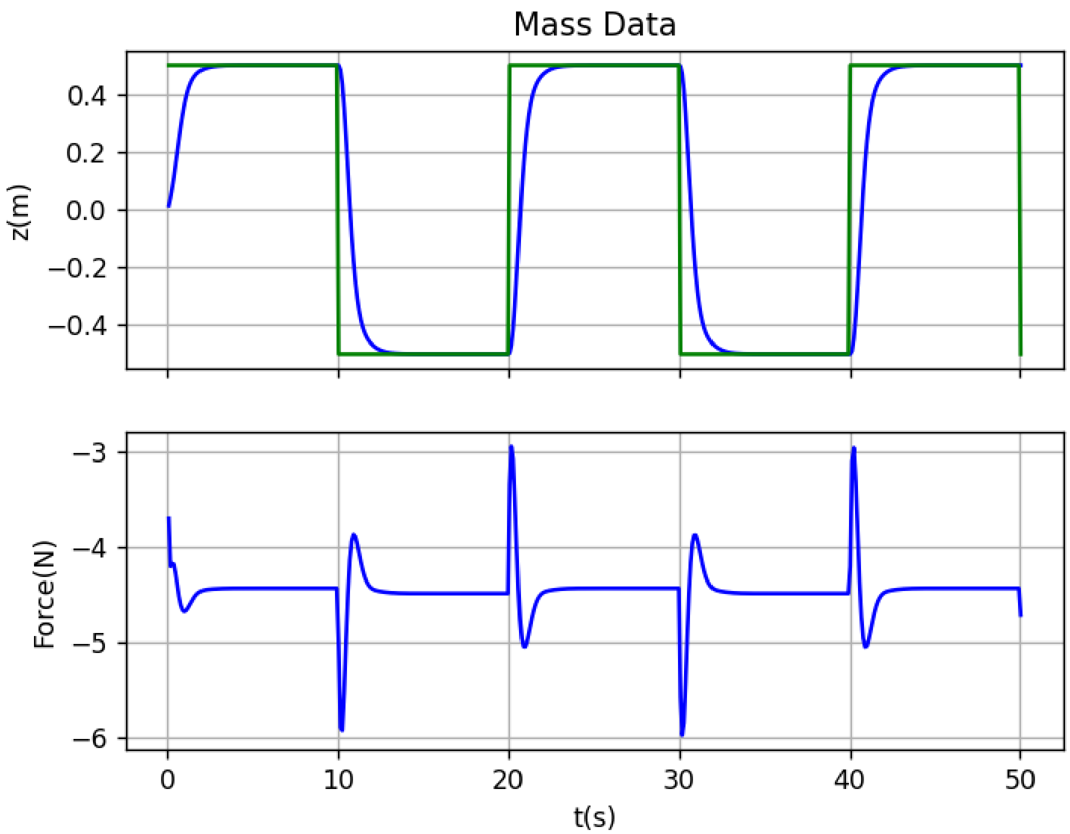
2.6 Insert the Python code for `ctrlPID.py` that implements PID control directly below this line.

```
import numpy as np
import slopedMassParam as P

class ctrlPID:
    def __init__(self):
        tr = 0.5 #sec
        wn = 2.2/tr
        zeta = 2.0
        a1 = P.b/ (P.m)
        b0 = 1.0 / (P.m)
        a0 = P.k1 / (P.m)
        self.kd = (2.0*zeta*wn - a1) / b0 #these are general equations and should work
for all PD systems
        self.kp = (wn**2 - a0) / b0
        self.ki = 5.0 #Integrator gain that I tune
        print("kd: ", self.kd, " kp: ", self.kp, " ki: ", self.ki)
        #other needed parameters
        self.sigma = P.sigma #0.05 I believe
```

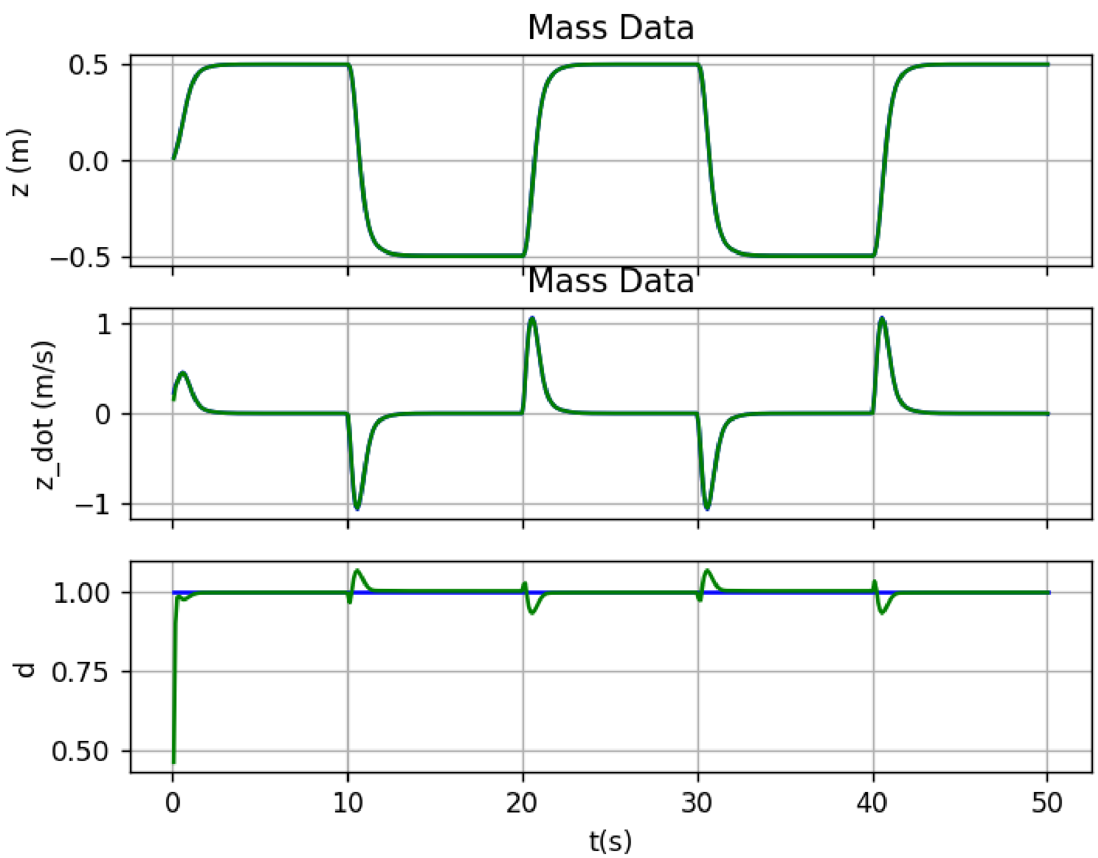
```
        self.Ts = P.Ts
        self.beta = (2.0 * self.sigma - P.Ts) / (2.0 * self.sigma + P.Ts) #dirty derivative
gain
        self.limit = P.F_max #his built in saturation function uses self.limit
        #variables and delayed variables for calculation
        self.zdot = 0.0
        self.integrator = 0.0
        self.error_d1 = 0.0
        self.z_d1 = 0.0 #delayed z

    def update(self, z_r, z):
        z = z #[0][0]
        error = z_r - z
        #integrate on error
        self.integrator = self.integrator + (P.Ts/2.0)*(error + self.error_d1)
        #compute derivative
        self.zdot = self.beta*self.zdot + (1.0-self.beta) * ((z - self.z_d1) / P.Ts)
        F_tilde = self.kp * error - self.kd * self.zdot + self.ki * self.integrator

        z_eq = 0.0
        F_eq = P.k1 * z_eq + P.k2 * z_eq**3 - P.m * P.g * np.sin(np.pi/4)
        F = self.saturate(F_tilde+F_eq)
        #integrator anti windup
        if self.ki != 0.0:
            self.integrator =  self.integrator + P.Ts/self.ki*(F - (F_tilde+F_eq)) #?ie if it is
saturating decrease the integrator

        #update delayed variables
        self.error_d1 = error
        self.z_d1 = z
        return F


    def saturate(self, u):
        if abs(u) > self.limit:
            u = self.limit*np.sign(u)
        return u


    #this is the saturate function he gave us, I will use the one from the practice final
    # def saturate(u, limit):
    #     if abs(u) > limit:
    #         u = limit*np.sign(u)
    #     return u
```

Part 3. Observer based control

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



3.6 Insert a plot of the state estimation error.



3.7 Insert a copy of `ctrlObsv.py` that implements the observer based controller directly below this line.

```
import numpy as np
import slopedMassParam as P
import control as cnt

class ctrlObsv:
    def __init__(self):
        tr = 0.5
        tr_obs = tr/5.0 #this satisfies the 5x faster requirment
        zeta = 0.707
        wn = 2.2/tr
        wn_obs = 2.2/tr_obs
        integrator_pole = -2.0 #make sure when I make the poly this is a positive value
so it comes out negative in the left hand plane
        zeta_obs = 0.707
        self.limit = P.F_max

        #State Space Matrices
        self.A = np.array([[0.0, 1.0],
```

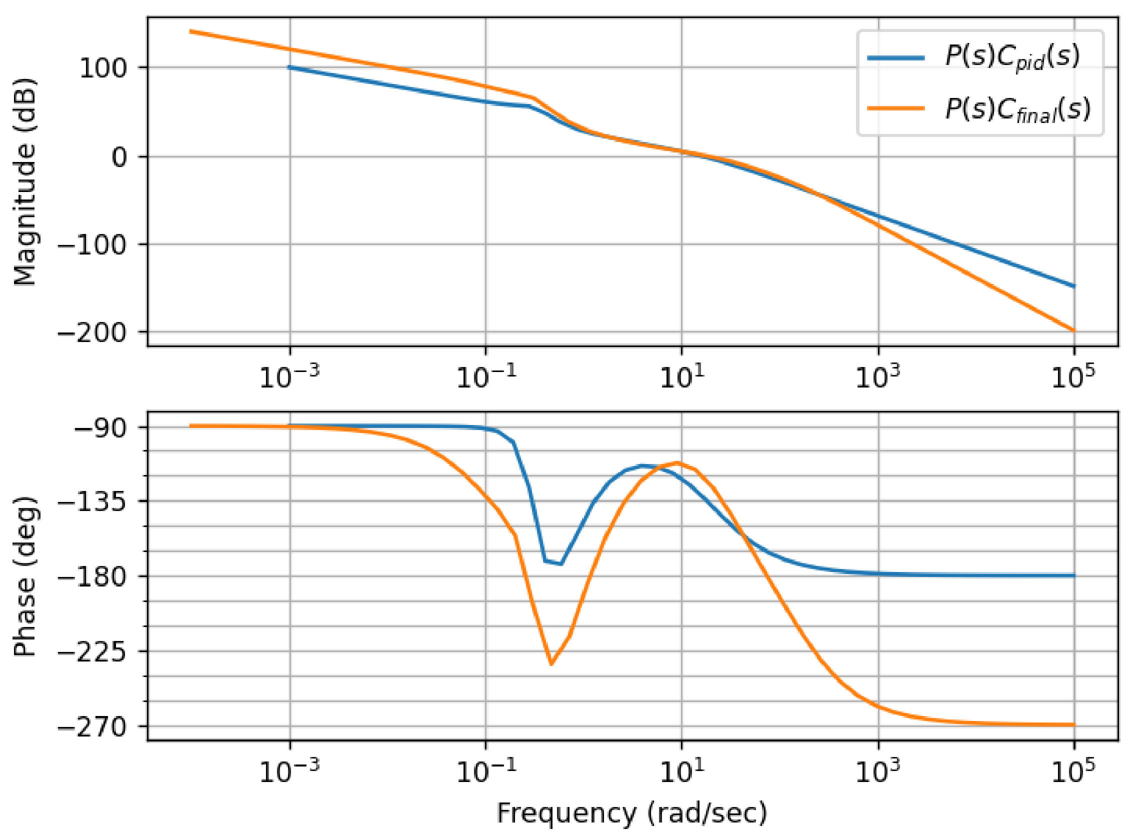
```
            [-P.k1/(P.m), -P.b/(P.m)]])
        self.B = np.array([[0.0],
            [1/(P.m)]])
        self.C = np.array([[1.0, 0.0]])
        self.D = np.array([[0.0]])

        #form augmented system
        A1 = np.vstack((np.hstack((self.A, np.zeros((np.size(self.A, 1),1)))),
             np.hstack((-self.C, np.array([[0.0]]))) ))
        self.B1 = np.vstack((self.B, 0.0))
        #gain calculation

        des_char_poly = np.convolve([1, 2 * zeta*wn, wn**2],
                            [1, -integrator_pole]) #!when is the integrator pole negative vs
positive?
        des_poles = np.roots(des_char_poly)
        # Compute the gains if the system is controllable
        if np.linalg.matrix_rank(cnt.ctrb(A1, self.B1)) != 3:
          print("The system is not controllable")
        else:
          self.K1 = (cnt.place(A1, self.B1, des_poles))
          self.K = self.K1[0][0:2]
          self.Ki = self.K1[0][2]
        print('K: ', self.K)
        print('ki: ', self.Ki)
        #print(des_poles)

        #?3.3 for disturbance observer
        #do this
        #augmented matrices for observer design
        self.A2 = np.concatenate((
                np.concatenate((self.A, self.B), axis=1),
                np.zeros((1, 3))),
                axis=0)
        self.B2 = np.concatenate((self.B, np.zeros((1, 1))), axis=0)
        self.C2 = np.concatenate((self.C, np.zeros((1, 1))), axis=1)

        #disturbance observer design
        dist_obs_pole = -20.0 #same as above, both negative or both positive
        wn_obs = 2.2/tr_obs
        des_obs_char_poly = np.convolve([1, 2*zeta_obs*wn_obs, wn_obs**2],
                            [1.0, -dist_obs_pole]) #! should this pole input be negative or
positive?
        des_obs_poles = np.roots(des_obs_char_poly)
        #compute the gains if the system is observable
        if np.linalg.matrix_rank(cnt.ctrb(self.A2.T, self.C2.T)) != 3:
```

```
            print("The system is not observable")
        else:
            self.L2 = cnt.acker(self.A2.T, self.C2.T, des_obs_poles).T
        print('L2: ', self.L2)
        print("\n")
        print('A2: ', self.A2)
        print("\n")
        print('B1: ', self.B1)
        print("\n")
        print("C2: ", self.C2)

        #variables to stay behind
        self.zdot = 0.0 #estimated derivative of z
        self.z_d1 = 0.0 #z delayed by one sample
        self.integrator = 0.0
        self.error_d1 = 0.0
        self.x_hat = np.array([[0.0], #z_hat_0
                               [0.0]]) #zdot_hat_0
        self.F_d1 = 0.0
        self.obs_state = np.array([
            [0.0], #z_hat
            [0.0], #zdot_hat
            [0.0], # estimate of the disturbance
        ])

    def update(self, z_r, y):
        x_hat, d_hat = self.update_observer(y)
        z_hat = x_hat[0][0]
        error = z_r -z_hat
        #integrate the error
        self.integrator = self.integrator + (P.Ts/2.0)*(error + self.error_d1)
        self.error_d1 = error #update the error
        #copmute the state feedback controller
        z_eq = 0.0 #do I use 0.0 or z_hat?
        F_eq = P.k1 * z_eq + P.k2 * z_eq**3 - P.m * P.g * np.sin(np.pi/4)
        F_tilde = -self.K @ x_hat - self.Ki * self.integrator - d_hat
        F = self.saturate(F_tilde.item(0)+F_eq)
        # self.F_d1 = F
        self.F_d1 = F_tilde #make sure down below in the observer that F_d1 *does not*
include F_eq
        return F, x_hat, d_hat

    def update_observer(self, y):
        # update the observer using RK4 integration
        F1 = self.observer_f(self.obs_state, y)
        F2 = self.observer_f(self.obs_state + P.Ts / 2 * F1, y)
```

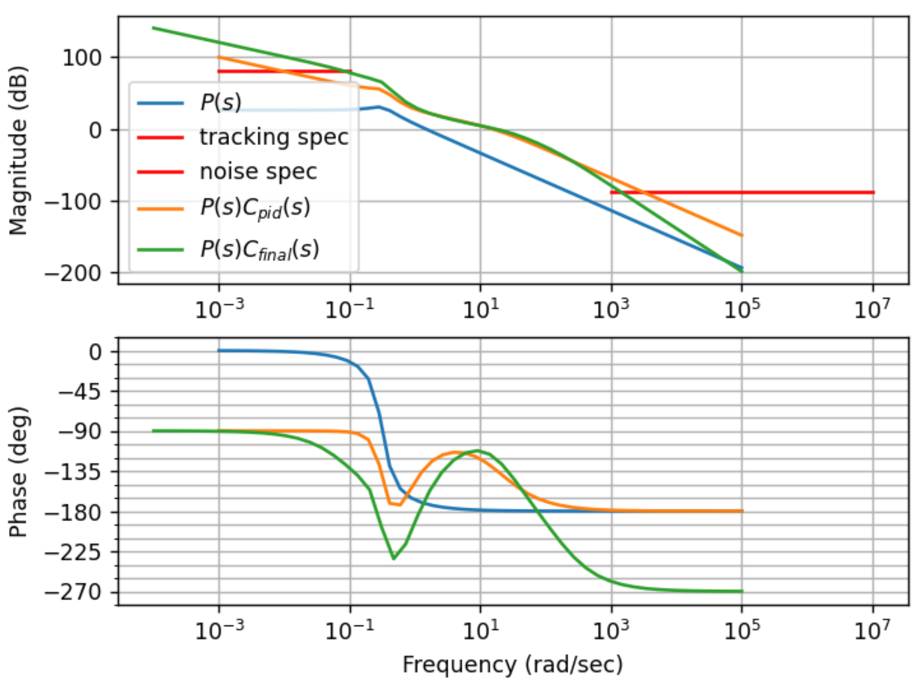
```
        F3 = self.observer_f(self.obs_state + P.Ts / 2 * F2, y)
        F4 = self.observer_f(self.obs_state + P.Ts * F3, y)
        self.obs_state += P.Ts / 6 * (F1 + 2 * F2 + 2 * F3 + F4)
        x_hat = self.obs_state[0:2]
        d_hat = self.obs_state[2][0]
        return x_hat, d_hat

    def observer_f(self, x_hat, y):
        #this is called in the update observer function for RK4
        # xhat = [z_hat, zdot_hat]

        # xhatdot = A*(xhat-xe) + B*(u-ue) + L(y-C*xhat)
        #!is it always going to be B1 and A2 and C2 etc????
        xhat_dot = self.A2 @ x_hat\
                + self.B1 * (self.F_d1)\
                + self.L2 * (y - self.C2 @ x_hat)
        return xhat_dot

    def saturate(self,u):
        if abs(u) > self.limit:
            u = self.limit*np.sign(u)
        return u

#this is the saturate function he gave us, I am going to use the one from the practice final
# def saturate(u, limit):
#     if abs(u) > limit:
#         u = limit * np.sign(u)
#     return u
```

Part 4. Loopshaping

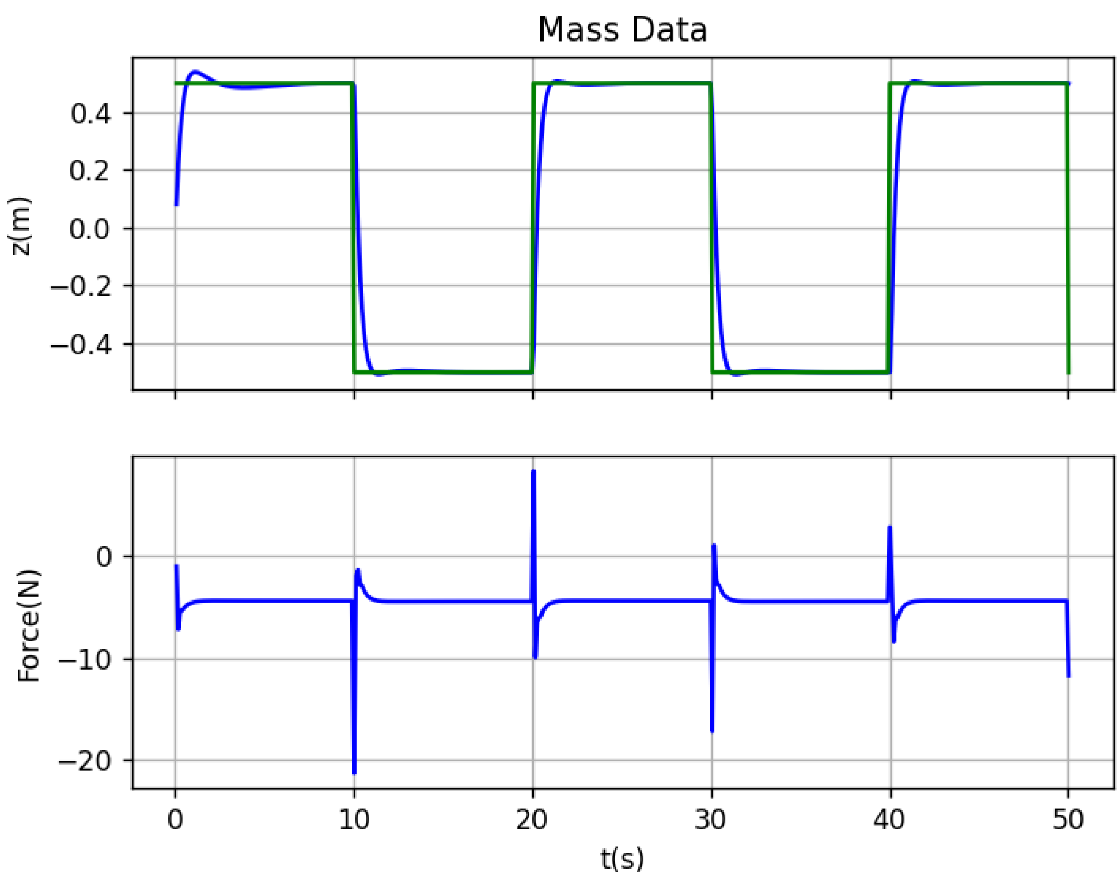
4.6 Insert the Bode plots for the PID controlled plant, and the loopshaped controlled plant below this line.



This plot below I used to help tune etc…



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



4.8 Insert the file `loopshapeRodMass.py` for the controller below this line.

```
import slopedMassParam as P
import matplotlib.pyplot as plt
from control import TransferFunction as tf
from control import tf, bode, margin, step_response, mag2db
import numpy as np
import loopshape_tools as ls
from ctrlPID import ctrlPID
PID = ctrlPID()

# Compute plant transfer functions
Plant = tf([1.0/(P.m )], #numerator
        [1.0, P.b/(P.m), P.k1 / (P.m)]) #this comes from the plant, make sure each
term has something, even if a 0.0
C_pid = tf([[(PID.kd+PID.kp*PID.sigma),
         (PID.kp+PID.ki*PID.sigma),
         PID.ki],
```

```
        [PID.sigma, 1, 0]) # this should be the same for every PID controller I believe

PLOT = True
dB_flag = True

############################################################
#############
#   Control Design
############################################################
#############
C = C_pid * ls.lead(w=14.7769,M=125.0) * ls.lag(z=12.0, M =120.0) * ls.lpf(p=27.0)
#lead is for phase margin, lag is for disturbances/tracking, lpf is for noise
print('C(s)= ', C)

############################################################
#
# add a prefilter to eliminate the overshoot
############################################################
#
F = tf(1, 1) * ls.lpf(p=3.0) #originally this was p=3.0 from the example I think?
print('F(s)= ', F)

###############################################
#  Convert Controller to State Space Equations if following method in 18.1.7
###############################################
C_num = np.asarray(C.num[0])
C_den = np.asarray(C.den[0])
F_num = np.asarray(F.num[0])
F_den = np.asarray(F.den[0])

if __name__ == "__main__":
   # calculate bode plot and gain and phase margin for just the plant dynamics
      #for the above see the quick code just below

   #### Code added to find gammaN and gammaR and to plot the noise and tracking
specifications
   #for the controller
   # #Also quick code to plot just the plant
   mag, phase, omega = bode(Plant, dB=True,
                    omega=np.logspace(-3, 5),
                    Plot=True, label="$P(s)$")

   gm, pm, Wcg, Wcp = margin(Plant * C_pid)

   magCP, phaseCP, omegaCP = bode(Plant*C_pid, plot=False,
```

```
                    omega = [0.01, 1000.0], dB=dB_flag) #TODO fill out these omega's for
gammaN and gammaR
   mag4Plt, phase4Plt, omega4Plt = bode(Plant*C_pid, plot=False,
                    omega = [0.1, 1000.0], dB=dB_flag) #TODO fill out these omegas for
the tracking and noise specifications

   #Tracking and noise specifications
   ls.spec_tracking(gamma=0.1*1.0/mag4Plt[0], omega=0.1, flag=dB_flag) #tracking
specification, the 0.1 is a "factor of 10", omega is at where
   ls.spec_noise(gamma=0.1*mag4Plt[1], omega=1000.0, flag=dB_flag)


   print("MagCP: ", magCP)
   print("for original C_pid system:")
   #It will spit out absolute magnitude, so I will not need to convert
   print(" pm: ", pm, " Wcp: ", Wcp, "gm: ", gm, " Wcg: ", Wcg)
   print("gammaR = ", 1.0/magCP[0])
   print("gammaN = ", magCP[1])
   #A few additional print statements
   print("\n")
   print("C(s): ", C)
   print("\n")
   print("F(s): ", F)

   ##### End of my input stuff



   # calculate bode plot and gain and phase margin for original PID * plant dynamics
   mag, phase, omega = bode(Plant * C_pid, dB=True,
                    omega=np.logspace(-3, 5),
                    Plot=True, label="$P(s)C_{pid}(s)$")

   gm, pm, Wcg, Wcp = margin(Plant * C_pid)
   print("for original C_pid system:")
   if dB_flag is True:
      print(" pm: ", pm, " Wcp: ", Wcp, "gm: ", mag2db(gm), " Wcg: ", Wcg)
   elif dB_flag is False:
      print(" pm: ", pm, " Wcp: ", Wcp, "gm: ", gm, " Wcg: ", Wcg)

   ###########################################
   #   Define Design Specifications
   ###########################################
   # specs go here
   # ls.spec_...
```

```
# plot the effect of adding the new compensator terms
mag, phase, omega = bode(Plant * C, dB=dB_flag,
                    omega=np.logspace(-4, 5),
                    Plot=True, label="$P(s)C_{final}(s)$")

gm, pm, Wcg, Wcp = margin(Plant * C)
print("for final P*C:")
if dB_flag is True:
   print(" pm: ", pm, " Wcp: ", Wcp, "gm: ", mag2db(gm), " Wcg: ", Wcg)
elif dB_flag is False:
   print(" pm: ", pm, " Wcp: ", Wcp, "gm: ", gm, " Wcg: ", Wcg)

plt.figure(1)
fig = plt.gcf()
fig.axes[0].legend()

##############################################
# 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(2)
plt.clf()
plt.grid(True)
mag, phase, omega = 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 = 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(4)
plt.clf()
plt.subplot(211), plt.grid(True)
T = np.linspace(0, 2, 100)
_, yout_no_F = step_response(CLOSED_R_to_Y, T)
_, yout_F = 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 = step_response(CLOSED_R_to_U, T)
_, Uout_F = 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()
```