Final Exam – Simulation Results

ECEn 483/ ME 431

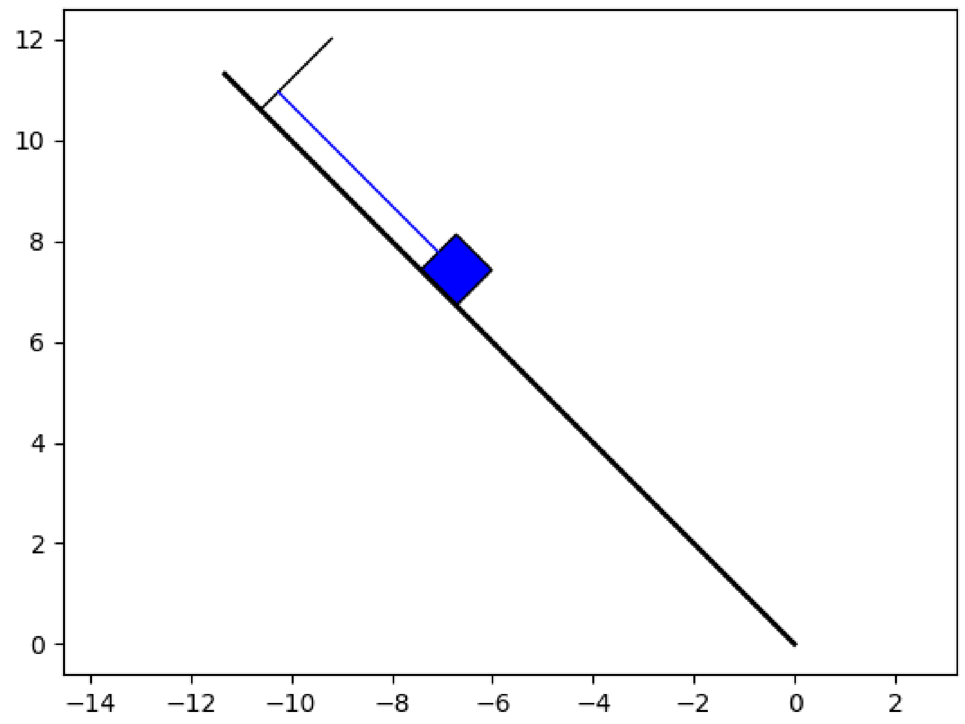
Fall 2024

Name: Connor Mabey

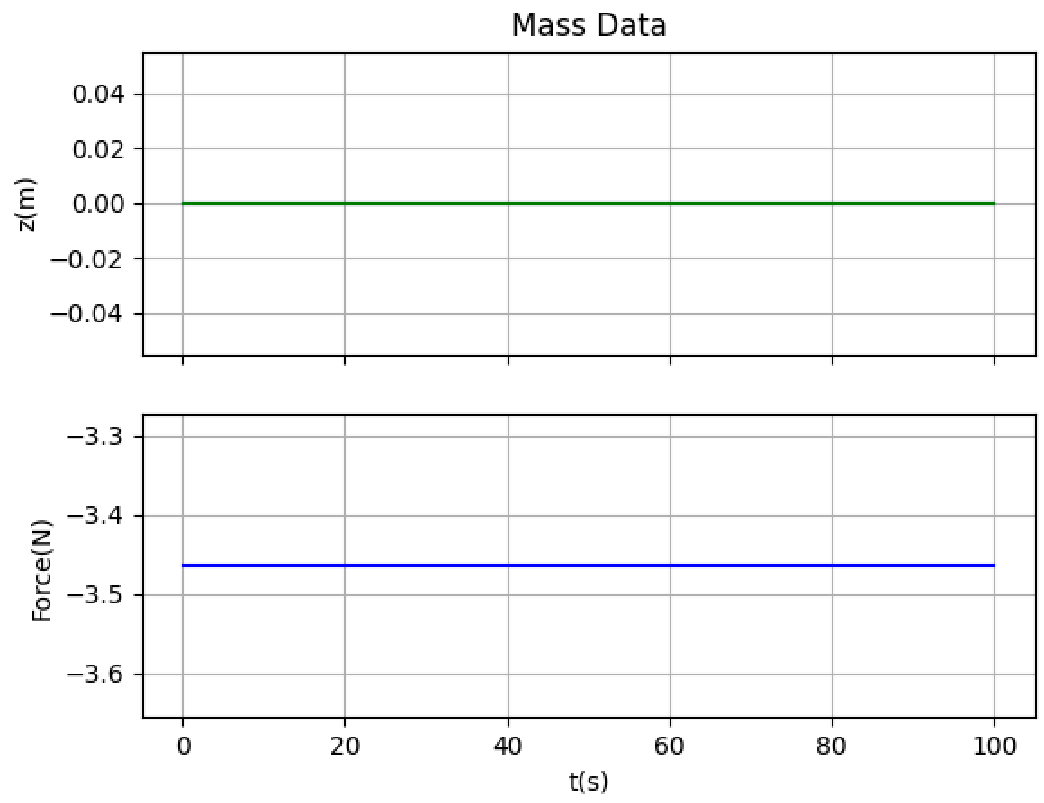
At the end of the exam, make sure to include this file with your submission.

Part 2. Design models

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



12
10
8
6
4
2
0
−14
−12
−10
−8
−6
−4
−2
0
2

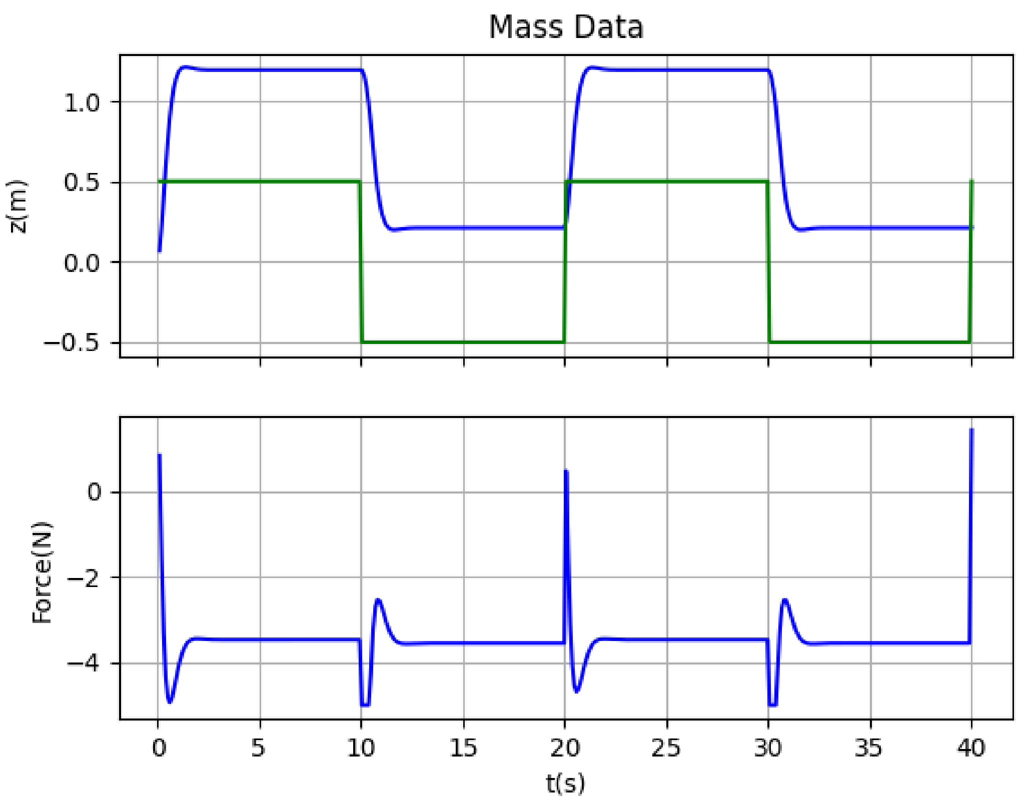


2.6 Insert the code you used for simulating the dynamics (this should nominally be massDynamics.py, unless you chose to use the complied version and lose 10 points):

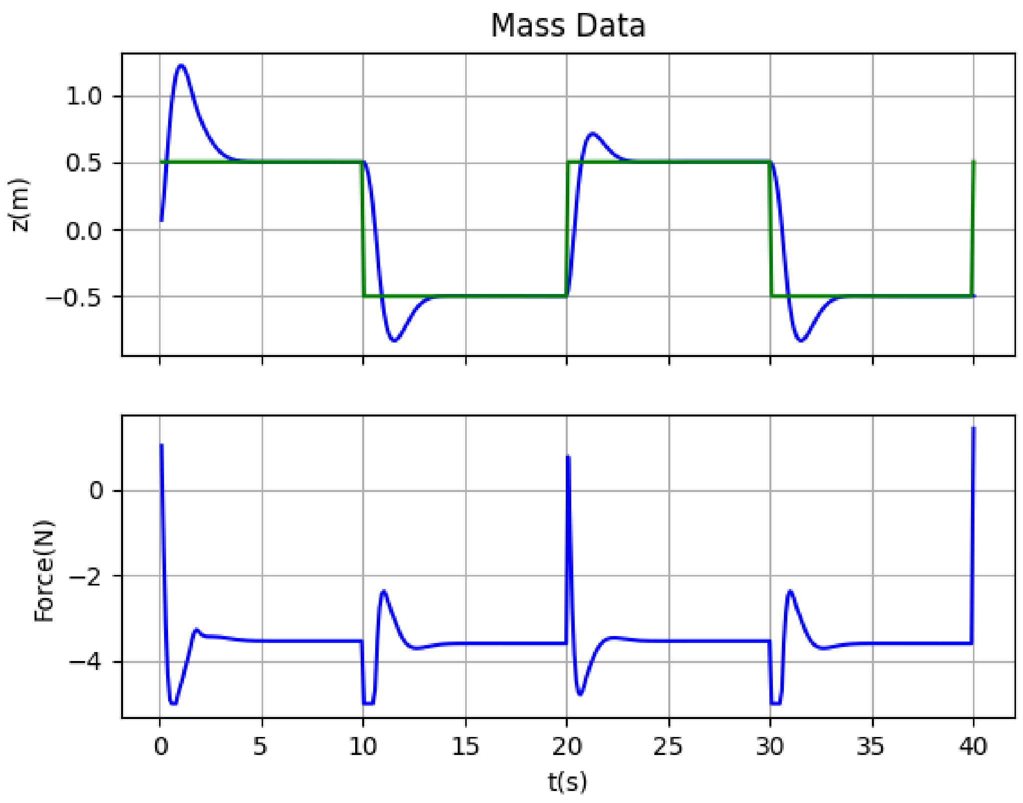
```
import numpy as np
import massParam as P

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)
        z_dot = state.item(1)
        xdot = np.array([
            [z_dot],
            [(F - self.b*z_dot + self.g*self.m*np.sqrt(2)/2 -
self.k1*z - self.k2*z**3)/self.m]
        ])
        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 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.



3.7 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.



i have 70 minutes left and a lot to do. i can't tune this further, i'm sorry

k_i = 2.25

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):
        zeta = 0.707
        tr = 0.6998
        wn = 2.2/tr
        self.kp = 5
        self.ki = 2.25   # tune this
        self.kd = zeta*wn - 0.1
        self.limit = P.F_max
        self.beta = (2.0 * P.sigma - P.Ts) / (2.0 * P.sigma + P.Ts)
        self.Ts = P.Ts
        self.z_d1 = 0.0
        self.z_dot = 0.0
        self.error_d1 = 0.0
        self.integrator = 0.0
        self.Fe = P.Fe

    def update(self, z_r, y):
        z = y[0][0]

        error = z_r - z

        # integrate error
        self.integrator = self.integrator + (P.Ts / 2) * (error +
self.error_d1)

        # dirty deriv
        self.z_dot = self.beta * self.z_dot + (1 - self.beta) * ((z
- self.z_d1) / P.Ts)


        # compute the linearized torque using PD control
        F_unsat = self.kp * error + self.ki * self.integrator -
self.kd * self.z_dot
        F = saturate(P.F_max, F_unsat)

        # integrator anti - windup
        if np.abs(self.z_dot) < 0.5:
            self.integrator = self.integrator + (P.Ts / 2) * (error
+ self.error_d1)

        self.error_d1 = error
        self.z_d1 = z
        return F

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

Part 4. Observer based control

yeah sorry this is too much for me. i've been at this for 5 hours and 45 minutes and i'm still so far from being done.

if we had literally ANY amount of experience with the control library this would be so. much. easier.

i get that we don't use cnt so that we actually learned fundamentals but this 6-8 hour exam would probably take less than an hour if i knew the different functions and finer details of cnt.

even from the small amount that i know i can that it's so, so, so, so useful but we never talked about it. seems like a really big misstep, especially for those who are interested and want to learn more about control theory. i'm not really one of those people but on just the final itself it would have been a genuine life saver. the tediousness and general unpleasantness of tuning and matrix match made up about 90% of the frustration with this class, but cnt would solve so many of those issues.

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

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.

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.

6. Insert your code that computes all control gains for each of the other parts here. And make sure to upload all of your code as part of your submission. Include all relevant files in the “python” folder.