Final Exam – Simulation Results

ECEn 483/ ME 431

Fall 2024

Name: Brendan Bakker

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.

A screenshot of a computer

Description automatically generated

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):

        # Initial state conditions

        self.state = np.array([

            [P.z0],      # initial position

            [P.zdot0]    # initial velocity

        ])

        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[0][0]

        zdot = state[1][0]

        zddot = (F - self.b\*zdot + np.sqrt(2)/2\*self.g\*self.m - self.k1\*z - self.k2\*z\*\*3)/self.m

        xdot = np.array([[zdot], [zddot]])

        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

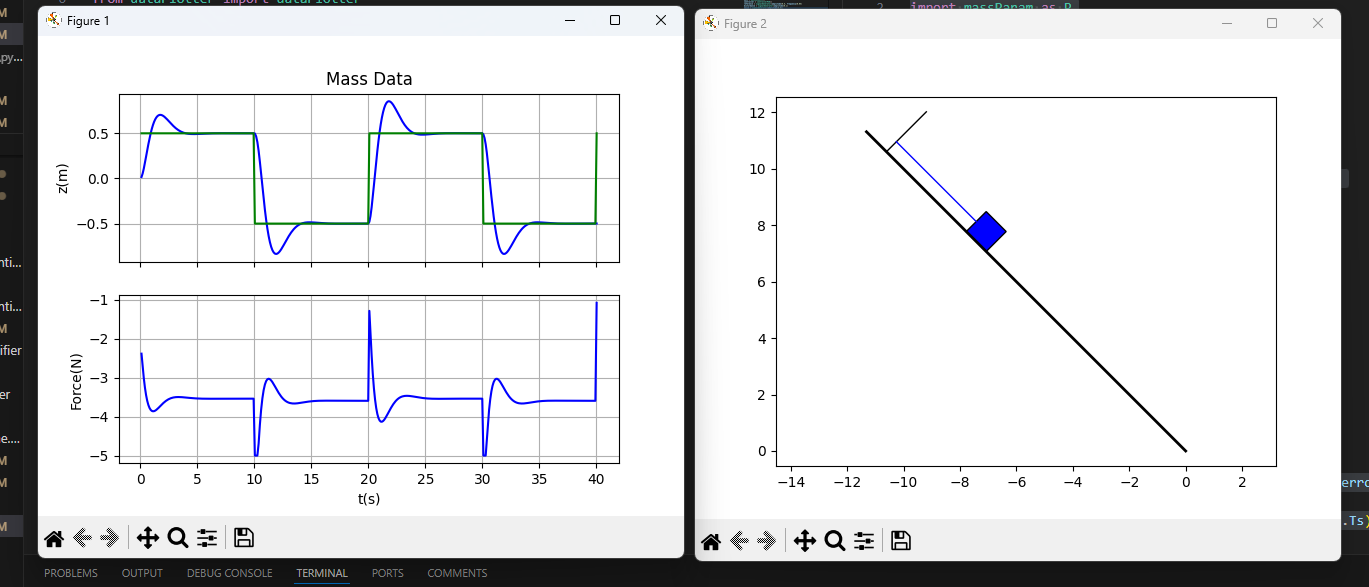
# Part 3. PID Control

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.

A screenshot of a computer

Description automatically generated

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.



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 = 2.5

        self.ki = 1.75

        self.kd = 1.497

        self.limit = P.F\_max

        self.sigma = 0.05 # from test document and email

        self.beta = (2.0 \* self.sigma - P.Ts) \

                / (2.0 \* self.sigma + P.Ts)

        self.Ts = P.Ts

        self.z\_d1 = 0.

        self.z\_dot = 0.

        self.error\_d1 = 0.

        self.integrator = 0.

        self.F\_e = P.F\_e

    def update(self, z\_r, y):

        z = y[0][0]

        # define integrator for PID control:

        error = z\_r - z

        self.integrator = self.integrator + (P.Ts/2) \* (error + self.error\_d1)

        self.z\_dot = self.beta \* self.z\_dot \

                + (1 - self.beta) \* ((z - self.z\_d1) / P.Ts)

        # feedback linearized force

        F\_fl = P.k1 \* z

        # # compute the linearized force using PD control

        # F\_tilde = self.kp \* (z\_r - z) - self.kd \* self.zdot

        # PID control

        F\_tilde = self.kp \* error \

            + self.ki \* self.integrator \

            - self.kd \* self.z\_dot

        # compute the final force and saturate

        F\_unsat = F\_fl + F\_tilde + P.F\_e

        F = self.saturate(F\_unsat)

        # integrator anti-windup

        if self.ki != 0.0:

            self.integrator = self.integrator + P.Ts / self.ki \* (F - F\_unsat)

        # 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

# Part 4. Observer based control

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.