Practice Final Exam – Simulation Results

ECEn 483/ ME 431

Winter 2023

Name:\_\_\_\_Jacob Child\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

|  |  |
| --- | --- |
|  |  |
| 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  and input directly below this line.

A picture containing chart

Description automatically generated

# Part 2. PID Control

2.4 Insert a plot that shows both and when is a square wave with magnitude degrees and frequency 0.1 Hz, and when using a PD controller.

\*\*\*would the first plot hurt me???\*\*\*

Chart, line chart

Description automatically generated

Chart

Description automatically generated with medium confidence

2.5 Insert a plot that shows both and when is a square wave with maginitude degrees and frequency 0.1 Hz, and when using a PID controller.

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

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

import numpy as np

import rodMassParam as P

class ctrlPID:

    def \_\_init\_\_(self):

        tr = 0.1 #sec

        wn = 2.2/tr

        zeta = 0.707

        a1 = P.b/ (P.m \* P.ell\*\*2)

        b0 = 1.0 / (P.m \* P.ell\*\*2)

        a0 = P.k1 / (P.m \* P.ell\*\*2)

        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 = 1.0 #Integrator gain that I tune

        print("kd: ", self.kd, " kp: ", self.kp)

        #other needed parameters

        self.sigma = 0.005

        self.Ts = P.Ts

        self.beta = (2.0 \* self.sigma - P.Ts) / (2.0 \* self.sigma + P.Ts) #dirty derivative gain

        self.limit = P.tau\_max #his built in saturation function uses self.limit

        #variables and delayed variables for calculation

        self.thetadot = 0.0

        self.integrator = 0.0

        self.error\_d1 = 0.0

        self.theta\_d1 = 0.0 #delayed theta

    def update(self, theta\_r, y):

        theta = y#[0][0]

        error = theta\_r - theta

        #integrate on error

        #!do I need an anti-windup scheme?

        self.integrator = self.integrator + (P.Ts/2.0)\*(error + self.error\_d1)

        #compute derivative

        self.thetadot = self.beta\*self.thetadot + (1.0-self.beta) \* ((theta - self.theta\_d1) / P.Ts)

        tau\_tilde = self.kp \* error - self.kd \* self.thetadot + self.ki \* self.integrator

        #?no feedback linearized force as I did the Jacobian linearization earlier

        tau = self.saturate(tau\_tilde)

        #integrator anti windup just in case

        if self.ki != 0.0:

            self.integrator =  self.integrator + P.Ts/self.ki\*(tau - tau\_tilde) #?ie if it is saturating decrease the integrator

        #update delayed variables

        self.error\_d1 = error

        self.theta\_d1 = theta

        return tau

    def saturate(self, u):

        if abs(u) > self.limit:

            u = self.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.

Chart, line chart

Description automatically generated

3.6 Insert a plot of the state estimation error.

Chart

Description automatically generated

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

import numpy as np

import rodMassParam as P

import control as cnt

class ctrlObsv:

    def \_\_init\_\_(self):

        #tuning parameters

        tr = 0.1

        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 = -10.0 #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.tau\_max

        #State Space Matrices

        self.A = np.array([[0.0, 1.0],

                      [-P.k1/(P.m\*P.ell\*\*2), -P.b/(P.m\*P.ell\*\*2)]])

        self.B = np.array([[0.0],

                      [1/(P.m\*P.ell\*\*2)]])

        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)

        #? I did the below because of 3.2, but then 3.3 immediately wants a disturbance observer?

        #observer design

        # des\_obs\_char\_poly = [1,2\*zeta\_obs\*wn\_obs, wn\_obs\*\*2]

        # 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.A.T, self.C.T)) != 2:

        #     print("The system is not observable")

        # else:

        #     self.L = cnt.place(self.A.T, self.C.T, des\_obs\_poles).T

        # print('L.T: ', self.L.T)

        #?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 = -1.0 #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.thetadot = 0.0 #estimated derivative of z

        self.theta\_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.Tau\_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, theta\_r, y):

        x\_hat, d\_hat = self.update\_observer(y)

        theta\_hat = x\_hat[0][0]

        error = theta\_r -theta\_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

        th\_eq = theta\_hat

        tau\_eq =  P.m\*P.g\*P.ell \* np.cos(th\_eq) + P.k1 \* th\_eq + P.k2 \* th\_eq\*\*3

        Tau\_tilde = -self.K @ x\_hat - self.Ki \* self.integrator - d\_hat

        tau = self.saturate(Tau\_tilde.item(0)+tau\_eq)

        # self.Tau\_d1 = tau

        self.Tau\_d1 = Tau\_tilde

        return tau, 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.Tau\_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

# Part 4. Loopshaping

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

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.