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.

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.

# 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.