A Design Study Approach to Classical Control

Randal W. Beard Timothy W. McLain
Brigham Young University

Updated: December 28, 2020

Homework D.14

(a) Modify your solution from HW D.13 so that the uncertainty parameter is $\alpha = 0.2$, representing 20% inaccuracy in the knowledge of the system parameters, and so that the input disturbance is 0.25. Also, add noise to the output channels z_m and θ_m with standard deviation of 0.001.

(b) Add a disturbance observer to the controller, and verify that the steady state error in the estimator has been removed. Tune the system to get good response.

Solution

Python code used to design the observer based controller is shown below:

```
# Single link mass Parameter File
import numpy as np
import control as cnt
import sys
sys.path.append('..')  # add parent directory
import massParam as P

Ts = P.Ts  # sample rate of the controller
beta = P.beta  # dirty derivative gain
F_max = P.F_max  # limit on control signal

#  tuning parameters
```

```
tr = 2.5
zeta = 0.707
integrator_pole = np.array([-10.0])
tr_obs = tr/10.0  # rise time for observer
zeta_obs = 0.707  # damping ratio for observer
dist_obsv_pole = np.array([-1.0])  # pole for disturbance observer

# State Space Equations
# xdot = A*x + B*u
# y = C*x
A = np.array([[0.0, 1.0],
               [-P.k/P.m, -P.b/P.m]])

B = np.array([[0.0],
               [1.0/P.m]])

C = np.array([[1.0, 0.0]])

# form augmented system
A1 = np.array([[0.0, 1.0, 0.0],
               [-P.k/P.m, -P.b/P.m, 0.0],
               [-1.0, 0.0, 0.0]])

B1 = np.array([[0.0],
               [1.0/P.m],
               [0.0]])

# gain calculation
wn = 2.2/tr  # natural frequency
des_char_poly = np.convolve(
    [1, 2*zeta*wn, wn**2],
    np.poly(integrator_pole))
des_poles = np.roots(des_char_poly)

# Compute the gains if the system is controllable
if np.linalg.matrix_rank(cnt.ctrb(A1, B1)) != 3:
    print("The system is not controllable")
else:
    K1 = cnt.acker(A1, B1, des_poles)
    K = np.array([K1.item(0), K1.item(1)])
    ki = K1.item(2)

# observer design
# Augmented Matrices
A2 = np.concatenate((
```

```
        np.concatenate((A, B), axis=1),
        np.zeros((1, 3))),
        axis=0)
C2 = np.concatenate((C, np.zeros((1, 1))), axis=1)

wn_obs = 2.2/tr_obs
des_obsv_char_poly = np.convolve(
    [1, 2*zeta*wn_obs, wn_obs**2],
    np.poly(dist_obsv_pole))
des_obsv_poles = np.roots(des_obsv_char_poly)

# Compute the gains if the system is controllable
if np.linalg.matrix_rank(cnt.ctrb(A2.T, C2.T)) != 3:
    print("The system is not observerable")
else:
    L2 = cnt.acker(A2.T, C2.T, des_obsv_poles).T
    L = L2[0:2,0]
    Ld = L2[2,0]

print('K: ', K)
print('ki: ', ki)
print('L^T: ', L.T)
print('Ld: ', Ld)
```

Python code for the observer based control is shown below:

```
import numpy as np
import massParamHW14 as P

class massController:
    def __init__(self):
        self.observer_state = np.array([
            [0.0],
            [0.0],
            [0.0],  # initial estimate for disturbance
        ])
        self.force_d1 = 0.0          # control, delayed by one sample
        self.integrator = 0.0        # integrator
        self.error_d1 = 0.0          # error signal delayed by 1 sample
        self.K = P.K                 # state feedback gain
        self.ki = P.ki               # Input gain
        self.L = P.L2                 # observer gain
        self.A = P.A2                 # system model
        self.B = P.B1
```

```
        self.C = P.C2
        self.limit = P.F_max          # Maxiumum force
        self.Ts = P.Ts                # sample rate of controller

    def update(self, z_r, y_m):
        # update the observer and extract z_hat
        x_hat, d_hat = self.update_observer(y_m)
        z_hat = x_hat.item(0)

        # integrate error
        error = z_r - z_hat
        self.integrateError(error)

        # Compute the state feedback controller
        force_tilde = -self.K @ x_hat \
                      - self.ki * self.integrator \
                      - d_hat

        # compute total torque
        force = self.saturate(force_tilde.item(0))
        self.force_d1 = force
        return force, x_hat

    def update_observer(self, y_m):
        # update the observer using RK4 integration
        F1 = self.observer_f(self.observer_state, y_m)
        F2 = self.observer_f(self.observer_state + self.Ts / 2 * F1, y_m)
        F3 = self.observer_f(self.observer_state + self.Ts / 2 * F2, y_m)
        F4 = self.observer_f(self.observer_state + self.Ts * F3, y_m)
        self.observer_state += self.Ts / 6 * (F1 + 2 * F2 + 2 * F3 + F4)
        x_hat = np.array([[self.observer_state.item(0)],
                          [self.observer_state.item(1)],
                          ])
        d_hat = self.observer_state.item(2)
        return x_hat, d_hat

    def observer_f(self, x_hat, y_m):
        # xhatdot = A*xhat + B*(u-ue) + L(y-C*xhat)
        xhat_dot = self.A @ x_hat \
                     + self.B * self.force_d1 \
                     + self.L @ (y_m - self.C @ x_hat)
        return xhat_dot

    def integrateError(self, error):
        self.integrator = self.integrator + (self.Ts/2.0)*(error + self.error_d1)
```

```
        self.error_d1 = error

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

See the wiki for the complete solution.