A Design Study Approach to Classical Control

Randal W. Beard Timothy W. McLain
Brigham Young University

Updated: December 28, 2020

Homework E.14

(a) Modify your solution from HW E.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 0.5. 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:

```
# ballbeam Parameter File
import numpy as np
import control as cnt
import sys
sys.path.append('..')  # add parent directory
import ballbeamParam as P
import numpy as np
from scipy import signal
import control as cnt

##################################################
#                    State Space
```

```
#####################################################
# tuning parameters
tr_z = 1.2        # rise time for position
tr_theta = 0.5    # rise time for angle
zeta_z = 0.707  # damping ratio position
zeta_th = 0.707  # damping ratio angle
integrator_pole = np.array([-4.0])
# pick observer poles
wn_th_obs = 8.0*2.2/tr_theta
wn_z_obs = 5.0*2.2/tr_z
dist_obsv_pole = np.array([-15.0])

# State Space Equations
# xdot = A*x + B*u
# y = C*x
A = np.array([[0.0, 0.0, 1.0, 0.0],
               [0.0, 0.0, 0.0, 1.0],
               [0.0, -P.g, 0.0, 0.0],
               [-P.m1*P.g/((P.m2*P.length**2)/3.0+P.m1*(P.length/2.0)**2), 0.0, 0

B = np.array([[0.0],
              [0.0],
              [0.0],
              [P.length / (P.m2 * P.length ** 2 / 3.0 + P.m1 * P.length ** 2 / 4.

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

# form augmented system
A1 = np.array([[0.0, 0.0, 1.0, 0.0, 0.0],
               [0.0, 0.0, 0.0, 1.0, 0.0],
               [0.0, -P.g, 0.0, 0.0, 0.0],
               [-P.m1*P.g/((P.m2*P.length**2)/3.0+P.m1*(P.length/2.0)**2), 0.0, 0
               [-1.0, 0.0, 0.0, 0.0, 0.0]])

B1 = np.array([[0.0],
               [0.0],
               [0.0],
               [P.length/(P.m2*P.length**2/3.0+P.m1*P.length**2/4.0)],
               [0.0]])

# gain calculation
wn_th = 2.2/tr_theta  # natural frequency for angle
wn_z = 2.2/tr_z  # natural frequency for position
des_char_poly = np.convolve(
```

```
    np.convolve([1, 2*zeta_z*wn_z, wn_z**2],
                [1, 2*zeta_th*wn_th, wn_th**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)) != 5:
    print("The system is not controllable")
else:
    K1 = cnt.acker(A1, B1, des_poles)
    K = np.array([K1.item(0), K1.item(1), K1.item(2), K1.item(3)])
    ki = K1.item(4)

# compute observer gains
# Augmented Matrices
A2 = np.concatenate((
        np.concatenate((A, B), axis=1),
        np.zeros((1, 5))),
        axis=0)
C2 = np.concatenate((C, np.zeros((2, 1))), axis=1)

des_obs_char_poly = np.convolve(
    np.convolve([1, 2*zeta_z*wn_z_obs, wn_z_obs**2],
                [1, 2*zeta_th*wn_th_obs, wn_th_obs**2]),
    np.poly(dist_obsv_pole))
des_obs_poles = np.roots(des_obs_char_poly)

# Compute the gains if the system is observable
if np.linalg.matrix_rank(cnt.ctrb(A2.T, C2.T)) != 5:
    print("The system is not observable")
else:
    # place_poles returns an object with various properties.  The gains are acces
    # .T transposes the matrix
    L2 = signal.place_poles(A2.T, C2.T, des_obs_poles).gain_matrix.T
    L = L2[0:4, 0:2]
    Ld = L2[4:5, 0:2]


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 ballbeamParam as P
import ballbeamParamHW14 as P14

class ballbeamController:
    def __init__(self):
        self.observer_state = np.array([
            [0.0],  # initial estimate for z_hat
            [0.0],  # initial estimate for theta_hat
            [0.0],  # initial estimate for z_hat_dot
            [0.0],  # initial estimate for theta_hat_dot
            [0.0],  # estimate of the disturbance
        ])
        self.F_d1 = 0.0  # Computed Force, delayed by one sample
        self.integrator = 0.0        # integrator
        self.error_d1 = 0.0          # error signal delayed by 1 sample
        self.K = P14.K               # state feedback gain
        self.ki = P14.ki             # Integral gain
        self.L = P14.L2               # observer gain
        self.A = P14.A2               # system model
        self.B = P14.B1
        self.C = P14.C2
        self.limit = P.Fmax          # Maximum force
        self.Ts = P.Ts               # sample rate of controller

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

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

        # Construct the state
        xe = np.array([[P.ze], [0.0], [0.0], [0.0]])
        x_tilde = x_hat - xe

        # equilibrium force
        F_e = P.m1*P.g*P.ze/P.length + P.m2*P.g/2.0
        # Compute the state feedback controller
        F_tilde = -self.K @ x_tilde \
                    - self.ki * self.integrator
        F_unsat = F_e + F_tilde.item(0) - d_hat
        F = self.saturate(F_unsat)
```

```
        self.integratorAntiWindup(F, F_unsat)
        self.F_d1 = F
        return F, x_hat, d_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)],
                          [self.observer_state.item(2)],
                          [self.observer_state.item(3)]])
        d_hat = self.observer_state.item(4)
        return x_hat, d_hat

    def observer_f(self, x_hat, y_m):
        xe = np.array([[P.ze], [0.0], [0.0], [0.0], [0.0]])
        # equilibrium force
        F_e = P.m1*P.g*P.ze/P.length + P.m2*P.g/2.0
        # xhatdot = A*(xhat-xe) + B*(u-ue) + L(y-C*xhat)
        xhat_dot = self.A @ (x_hat - xe) \
                   + self.B * (self.F_d1 - F_e) \
                   + 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 integratorAntiWindup(self, F, F_unsat):
        # integrator anti - windup
        if self.ki != 0.0:
            self.integrator = self.integrator + P.Ts/self.ki*(F-F_unsat)

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

See the wiki for the complete solution.