A Design Study Approach to Classical Control

Randal W. Beard   Timothy W. McLain
Brigham Young University

Updated: December 28, 2020

Homework E.13

The objective of this problem is to design an observer that estimates the state of the system and to use the estimated state in the controller designed in Homework E.12.

(a) For the sake of understanding the function of the observer, for this problem we will use exact parameters, without an input disturbance. Modify the ball beam dynamics so that the parameters known to the controller are the actual plant parameters (uncertainty parameter $\alpha = 0$).

(b) Verify that the state space system is observable by checking that $\text{rank}(\mathcal{O}_{A,C}) = n$.

(c) In the control block, add an observer to estimate the state $\hat{x}$, and use the estimate of the state in your feedback controller. Tune the poles of the controller and observer to obtain good performance.

(d) Modify the simulation files so that the controller outputs both u and $\hat{x}$. Add a plotting routine to plot both the state and the estimated state of the system on the same graph.

(e) As motivation for the next chapter, add an input disturbance to the system of 0.5 and observe that there is steady state error in the response even though there is an integrator. This is caused by a steady state error in the observation error. In the next chapter we will show how to remove the steady state error in the observation error.

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 = 0.25     # rise time for angle
zeta_z   = 0.707  # damping ratio position
zeta_th  = 0.707  # damping ratio angle
integrator_pole = -5.0
# pick observer poles
wn_th_obs = 5.0*2.2/tr_theta
wn_z_obs = 5.0*2.2/tr_z

# 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.matrix([[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.matrix([[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
des_obs_char_poly = np.convolve([1, 2*zeta_z*wn_z_obs, wn_z_obs**2],
                                [1, 2*zeta_th*wn_th_obs, wn_th_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(A.T, C.T)) != 4:
    print("The system is not observable")
else:
    # place_poles returns an object with various properties.  The gains are acces
    # .T transposes the matrix
    L = signal.place_poles(A.T, C.T, des_obs_poles).gain_matrix.T

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

Python code for the observer based control is shown below:

```python
import numpy as np
import ballbeamParam as P
import ballbeamParamHW13 as P13

class ballbeamController:
    def __init__(self):
        self.x_hat = 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
        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 = P13.K               # state feedback gain
        self.ki = P13.ki             # Integral gain
        self.L = P13.L                 # observer gain
        self.A = P13.A                 # system model
        self.B = P13.B
        self.C = P13.C
        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 = 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)
```

```
        F = self.saturate(F_unsat)
        self.integratorAntiWindup(F, F_unsat)
        self.F_d1 = F
        return F, x_hat

    def update_observer(self, y):
        # update the observer using RK4 integration
        F1 = self.observer_f(self.x_hat, y)
        F2 = self.observer_f(self.x_hat + self.Ts / 2 * F1, y)
        F3 = self.observer_f(self.x_hat + self.Ts / 2 * F2, y)
        F4 = self.observer_f(self.x_hat + self.Ts * F3, y)
        self.x_hat += self.Ts / 6 * (F1 + 2 * F2 + 2 * F3 + F4)
        return self.x_hat

    def observer_f(self, x_hat, y):
        # xhatdot = A*(xhat-xe) + B*u + L(y-C*xhat)
        xe = np.array([[P.ze], [0.0], [0.0], [0.0]])

        # equilibrium force
        F_e = P.m1*P.g*P.ze/P.length + P.m2*P.g/2.0
        xhat_dot = self.A @ (x_hat - xe) \
                   + self.B * (self.F_d1 - F_e) \
                   + self.L @ (y - 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.