A Design Study Approach to Classical Control

Randal W. Beard Timothy W. McLain
Brigham Young University

Updated: December 28, 2020

Homework D.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 D.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 mass 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 0.25 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:

```
# 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 = -10.0
tr_obs = tr/10.0  # rise time for observer
zeta_obs = 0.707  # damping ratio for 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
wn_obs = 2.2/tr_obs
des_obsv_char_poly = [1, 2*zeta_obs*wn_obs, wn_obs**2]
des_obsv_poles = np.roots(des_obsv_char_poly)

# Compute the gains if the system is controllable
if np.linalg.matrix_rank(cnt.ctrb(A.T, C.T)) != 2:
    print("The system is not observerable")
else:
    L = cnt.acker(A.T, C.T, des_obsv_poles).T

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

Python code for the observer based control is shown below:

```
import numpy as np
import massParamHW13 as P

class massController:
    def __init__(self):
        self.x_hat = np.matrix([
            [0.0],
            [0.0],
        ])
        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.L                    # observer gain
        self.A = P.A                    # system model
        self.B = P.B
        self.C = P.C
        self.limit = P.F_max            # Maxiumum 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 = self.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
        # 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.x_hat, y_m)
        F2 = self.observer_f(self.x_hat + self.Ts / 2 * F1, y_m)
        F3 = self.observer_f(self.x_hat + self.Ts / 2 * F2, y_m)
        F4 = self.observer_f(self.x_hat + self.Ts * F3, y_m)
        self.x_hat += self.Ts / 6 * (F1 + 2 * F2 + 2 * F3 + F4)
        x_hat = np.array([[self.x_hat.item(0)],
                          [self.x_hat.item(1)],
                          ])
        return x_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.