A Design Study Approach to Classical Control

Randal W. Beard   Timothy W. McLain
Brigham Young University

Updated: December 28, 2020

Homework E.12

(a) Modify the state feedback solution developed in Homework E.11 to add an integrator with anti-windup to the feedback loop for z.

(b) Add a constant input disturbance of 1 Newtons to the input of the plant and allow the plant parameters to vary up to 20%.

(c) Tune the integrator pole (and other gains if necessary) to get good tracking performance.

Solution

Step 1. The original state space equations are

$$\dot{x} = \begin{pmatrix} 0 & 0 & 1.0000 & 0 \\ 0 & 0 & 0 & 1.0000 \\ 0 & -18.1923 & 0 & 0 \\ -9.8000 & 0 & 0 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \\ 2.6519 \\ 0 \end{pmatrix} u$$
$$y_r = \begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix} x,$$

therefore the augmented system is

$$A_1 = \begin{pmatrix} A & \mathbf{0} \\ -C & \mathbf{0} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1.0000 & 0 & 0 \\ 0 & 0 & 0 & 1.0000 & 0 \\ 0 & -18.1923 & 0 & 0 & 0 \\ -9.8000 & 0 & 0 & 0 & 0 \\ 0 & -1.0000 & 0 & 0 & 0 \end{pmatrix}$$

$$B_1 = \begin{pmatrix} B \\ \mathbf{0} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 2.6519 \\ 0 \\ 0 \end{pmatrix}$$

Step 2. Following the design in HW E.11, we use the tuning parameters $t_{r_z} = 1.2$, $t_{r_\theta} = 0.5$, $\zeta_z = 0.707$, $\zeta_\theta = 0.707$. In addition, we will add an integrator pole at $p_I = -5$. The new controllability matrix

$$\mathcal{C}_{A_1,B_1} = [B_1, A_1B_1, A_1^2B_1] = \begin{pmatrix} 0 & 2.6519 & 0 & 0 & 0 \\ 0 & 0 & 0 & -25.9890 & 0 \\ 2.6519 & 0 & 0 & 0 & 472.7979 \\ 0 & 0 & -25.9890 & 0 & 0 \\ 0 & 0 & 0 & 0 & 25.9890 \end{pmatrix}.$$

The determinant is $det(\mathcal{C}_{A_1,B_1}) = -1.2345e + 05 \neq 0$, therefore the system is controllable.

The open loop characteristic polynomial

$$\begin{aligned} \Delta_{ol}(s) = \det(sI - A_1) &= \det \begin{pmatrix} s & 0 & -1.0000 & 0 & 0 \\ 0 & s & 0 & -1.0000 & 0 \\ 0 & 18.1923 & s & 0 & 0 \\ 9.8000 & 0 & 0 & s & 0 \\ 0 & 1.0000 & 0 & 0 & s \end{pmatrix} \\ &= s^5 - 178.2842s, \end{aligned}$$

which implies that

$$\mathbf{a}_{A_1} = \begin{pmatrix} 0 & 0 & 0 & -178.2842 & 0 \end{pmatrix}$$

$$\mathcal{A}_{A_1} = \begin{pmatrix} 1 & 0 & 0 & 0 & -178.2842 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

The desired closed loop polynomial

$$\begin{aligned}\Delta_{cl}^d(s) &= (s^2 + 2\zeta_z\omega_{n_z}s + \omega_{n_z}^2)(s^2 + 2\zeta_\theta\omega_{n_\theta}s + \omega_{\theta_z}^2)(s+5) \\ &= s^5 + 13.8139s^4 + 82.9192s^3 + 265.3469s^2 + 420.5664s + 325.3556,\end{aligned}$$

which implies that

$$\boldsymbol{\alpha} = \begin{pmatrix} 13.8139, & 82.9192, & 265.3469, & 420.5664, & 325.3556 \end{pmatrix}.$$

The augmented gains are therefore given as

$$\begin{aligned}K_1 &= (\boldsymbol{\alpha} - \mathbf{a}_{A_1})\mathcal{A}_{A_1}^{-1}\mathcal{C}_{A_1,B_1}^{-1} \\ &= \begin{pmatrix} 31.2675 & -23.0425 & 5.2090 & -10.2100 & 12.5190 \end{pmatrix}\end{aligned}$$

Step 3. The feedback gains are therefore given by

$$\begin{aligned}K &= K_1(1:2) = \begin{pmatrix} 31.2675 & -23.0425 & 5.2090 & -10.2100 \end{pmatrix} \\ k_I &= K_1(3) = 12.5190.\end{aligned}$$

Alternatively, we could have used the following Python script

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


####################################################
```

```
#                   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 = -5.0


# 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.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.matrix([K1.item(0), K1.item(1), K1.item(2), K1.item(3)])
    ki = K1.item(4)

print('K: ', K)
print('ki: ', ki)
```

Python code that implements the associated controller listed below.

```
import numpy as np
import ballbeamParam as P
import ballbeamParamHW12 as P12

class ballbeamController:
    def __init__(self):
        self.integrator = 0.0  # integrator
        self.error_d1 = 0.0  # error signal delayed by 1 sample
        self.K = P12.K  # state feedback gain
        self.ki = P12.ki  # Integral gain
        self.limit = P.Fmax  # Maximum force
        self.Ts = P.Ts  # sample rate of controller

    def update(self, z_r, x):
        z = x.item(0)
        # integrate error
        error = z_r - z
        self.integrateError(error)
        # Construct the linearized state
        x_tilde = x - np.array([[P.ze], [0], [0], [0]])
        zr_tilde = z_r - P.ze

        # 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
        F = self.saturate(F_unsat)
```

```
        self.integratorAntiWindup(F, F_unsat)
        return F

    def differentiateZ(self, z):
        self.z_dot = self.beta*self.z_dot + (1-self.beta)*((z - self.z_d1) / self
        self.z_d1 = z

    def differentiateTheta(self, theta):
        self.theta_dot = self.beta*self.theta_dot + (1-self.beta)*((theta - self.
        self.theta_d1 = theta

    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):
        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
```

The complete simulation files are contained on the wiki associated with this book.