A Design Study Approach to Classical Control

Randal W. Beard   Timothy W. McLain
Brigham Young University

Updated: December 28, 2020

Homework F.14

(a) Modify your solution from HW F.13 so that the uncertainty parameter is $\alpha = 0.2$, representing 20% inaccuracy in the knowledge of the system parameters. Modify `VTOL_dynamics` to add an altitude disturbance of 1.0 and a wind disturbance of 1.0 m/s. A Python code snippet that implements these disturbances is given by

```
from math import sin
# disturbances
wind = 1.0
altitude_dist = 1.0

# add wind disturbance
zdot = zdot + wind

zddot = -(fr+fl)*sin(theta)/(mc+2*mr)
hddot = (-(mc+2*mr)*g  \
          + (fr+fl)*cos(theta))/(mc+2*mr)  \
          + altitude_dist
thetaddot = d*(fr-fl)/(Jc+2*mr*d**2)
```

Before adding the disturbance observer, run the simulation and note that the controller is not robust to the input disturbance.

(b) Add a disturbance observer to both controllers, and verify that the steady state error in the estimator has been removed and that the disturbances have been adequately compensated. Tune the system to get good response.

Solution

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

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


####################################################
#                  State Space
####################################################
# tuning parameters
wn_h     = 1.0
zeta_h   = 0.707
wn_z     = 0.9905
zeta_z   = 0.707
wn_th    = 13.3803
zeta_th = 0.707
integrator_h = np.array([-3.0])
integrator_z = np.array([-2.0])

# observer gains
wn_h_obs     = 10.0*wn_h
wn_z_obs     = 10.0*wn_z
wn_th_obs    = 5.0*wn_th
dist_obsv_pole_lon = np.array([-10.0])
dist_obsv_pole_lat = np.array([-10.0])


# State Space Equations
A_lon = np.array([[0.0, 1.0],
                   [0.0, 0.0]])
B_lon = np.array([[0.0],
                   [1.0/(P.mc+2.0*P.mr)]])
C_lon = np.array([[1.0, 0.0]])
```

```
A_lat = np.array([[0.0, 0.0, 1.0, 0.0],
                  [0.0, 0.0, 0.0, 1.0],
                  [0.0, -P.Fe/(P.mc+2.0*P.mr), -(P.mu/(P.mc+2.0*P.mr)), 0.0],
                  [0.0, 0.0, 0.0, 0.0]])
B_lat = np.array([[0.0],
                  [0.0],
                  [0.0],
                  [1.0/(P.Jc+2*P.mr*P.d**2)]])
C_lat = np.array([[1.0, 0.0, 0.0, 0.0],
                  [0.0, 1.0, 0.0, 0.0]])

# form augmented system
A1_lon = np.array([[0.0, 1.0, 0.0],
                   [0.0, 0.0, 0.0],
                   [-1.0, 0.0, 0.0]])
B1_lon = np.array([[0.0],
                   [1.0/(P.mc+2.0*P.mr)],
                   [0.0]])
A1_lat = 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.Fe/(P.mc+2.0*P.mr), -(P.mu/(P.mc+2.0*P.mr)), 0.0, 0.
                   [0.0, 0.0, 0.0, 0.0, 0.0],
                   [-1.0, 0.0, 0.0, 0.0, 0.0]])
B1_lat = np.array([[0.0],
                   [0.0],
                   [0.0],
                   [1.0/(P.Jc+2*P.mr*P.d**2)],
                   [0.0]])

# gain calculation
des_char_poly_lon = np.convolve([1.0, 2.0*zeta_h*wn_h, wn_h**2],
                                np.poly(integrator_h))
des_poles_lon = np.roots(des_char_poly_lon)

des_char_poly_lat = np.convolve(
                        np.convolve([1.0, 2.0*zeta_z*wn_z, wn_z**2],
                                    [1.0, 2.0*zeta_th*wn_th, wn_th**2]),
                        np.poly(integrator_z))
des_poles_lat = np.roots(des_char_poly_lat)


# Compute the gains if the system is controllable
if np.linalg.matrix_rank(cnt.ctrb(A1_lon, B1_lon)) != 3:
    print("The longitudinal system is not controllable")
else:
```

```
    K1_lon = cnt.acker(A1_lon, B1_lon, des_poles_lon)
    K_lon = np.array([K1_lon.item(0), K1_lon.item(1)])
    ki_lon = K1_lon.item(2)

if np.linalg.matrix_rank(cnt.ctrb(A1_lat, B1_lat)) != 5:
    print("The lateral system is not controllable")
else:
    K1_lat = cnt.acker(A1_lat, B1_lat, des_poles_lat)
    K_lat = np.array([K1_lat.item(0), K1_lat.item(1), K1_lat.item(2), K1_lat.item
    ki_lat = K1_lat.item(4)

# form augmented system for observer design
# form augmented system
A2_lon = np.array([[0.0, 1.0, 0.0],
                    [0.0, 0.0, 1.0/(P.mc+2.0*P.mr)],
                    [0.0, 0.0, 0.0]])
C2_lon = np.array([[1.0, 0.0, 0.0]])
A2_lat = 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.Fe/(P.mc+2.0*P.mr), -(P.mu/(P.mc+2.0*P.mr)), 0.0, 0.
                    [0.0, 0.0, 0.0, 0.0, 1.0/(P.Jc+2*P.mr*P.d**2)],
                    [0.0, 0.0, 0.0, 0.0, 0.0]])
C2_lat = np.array([[1.0, 0.0, 0.0, 0.0, 0.0],
                    [0.0, 1.0, 0.0, 0.0, 0.0]])

# compute observer poles
des_obs_char_poly_lon = np.convolve([1.0, 2.0*zeta_h*wn_h_obs, wn_h_obs**2],
                                    np.poly(dist_obsv_pole_lon))
des_obs_poles_lon = np.roots(des_obs_char_poly_lon)
des_obs_char_poly_lat = np.convolve(
                            np.convolve([1.0, 2.0*zeta_z*wn_z_obs, wn_z_obs**2],
                                        [1.0, 2.0*zeta_th*wn_th_obs, wn_th_obs**2
                            np.poly(dist_obsv_pole_lat))
des_obs_poles_lat = np.roots(des_obs_char_poly_lat)

if np.linalg.matrix_rank(cnt.ctrb(A2_lon.T, C2_lon.T)) != 3:
    print("The longitudinal system is not observable")
else:
    # place_poles returns an object with various properties.  The gains are acces
    # .T transposes the matrix
    L2_lon = signal.place_poles(A2_lon.T, C2_lon.T, des_obs_poles_lon).gain_matri
    L_lon = L2_lon[0:2, 0:1]
    Ld_lon = L2_lon[2:3, 0:1]

if np.linalg.matrix_rank(cnt.ctrb(A2_lat.T, C2_lat.T)) != 5:
```

```
    print("The lateral system is not observable")
else:
    # place_poles returns an object with various properties.  The gains are acces
    # .T transposes the matrix
    L2_lat = signal.place_poles(A2_lat.T, C2_lat.T, des_obs_poles_lat).gain_matri
    L_lat = L2_lat[0:4, 0:2]
    Ld_lat = L2_lat[4:5, 0:2]

print('K_lon: ', K_lon)
print('ki_lon: ', ki_lon)
print('L_lon^T: ', L_lon.T)
print('Ld_lon: ', Ld_lon)
print('K_lat: ', K_lat)
print('ki_lat: ', ki_lat)
print('L_lat^T: ', L_lat.T)
print('Ld_lat: ', Ld_lat)
```

Python code for the observer based control is shown below:

```
import numpy as np
import VTOLParam as P
import VTOLParamHW14 as P14

class VTOLController:
    def __init__(self):
        self.xhat_lon = np.array([[0.0], [0.0], [0.0]])
        self.xhat_lat = np.array([[0.0], [0.0], [0.0], [0.0], [0.0]])
        self.integrator_z = 0.0      # integrator on position z
        self.error_z_d1 = 0.0        # error signal delayed by 1 sample
        self.integrator_h = 0.0      # integrator on altitude h
        self.error_h_d1 = 0.0        # error signal delayed by 1 sample
        self.F_d1 = 0.0  # Force signal delayed by 1 sample
        self.tau_d1 = 0.0  # torque signal delayed by 1 sample
        self.limit = P.fmax
        self.Ts = P.Ts

    def update(self, r, y):
        z_r = r.item(0)
        h_r = r.item(1)
        y_lat = np.array([[y.item(0)],[y.item(2)]])
        y_lon = y.item(1)
        # update the observers
        xhat_lat, dhat_lat = self.update_lat_observer(y_lat)
        xhat_lon, dhat_lon = self.update_lon_observer(y_lon)
```

```
        z_hat = xhat_lat.item(0)
        h_hat = xhat_lon.item(0)
        theta_hat = xhat_lat.item(1)

        # integrate error
        error_z = z_r - z_hat
        self.integrateErrorZ(error_z)
        error_h = h_r - h_hat
        self.integrateErrorH(error_h)

        # Construct the states
        # Compute the state feedback controllers
        F_tilde = -P14.K_lon @ xhat_lon \
                  - P14.ki_lon * self.integrator_h
        F = P.Fe/np.cos(theta_hat) \
            + F_tilde.item(0) \
            - dhat_lon
        tau = -P14.K_lat @ xhat_lat \
              - P14.ki_lat*self.integrator_z \
              - dhat_lat
        u = np.array([[F], [tau.item(0)]])
        self.F_d1 = F
        self.tau_d1 = tau.item(0)
        return u, xhat_lat, xhat_lon

    def update_lat_observer(self, y_m):
        # update the observer using RK4 integration
        F1 = self.observer_f_lat(self.xhat_lat, y_m)
        F2 = self.observer_f_lat(self.xhat_lat + self.Ts / 2 * F1, y_m)
        F3 = self.observer_f_lat(self.xhat_lat + self.Ts / 2 * F2, y_m)
        F4 = self.observer_f_lat(self.xhat_lat + self.Ts * F3, y_m)
        self.xhat_lat += self.Ts / 6 * (F1 + 2 * F2 + 2 * F3 + F4)
        xhat = np.array([[self.xhat_lat.item(0)],
                         [self.xhat_lat.item(1)],
                         [self.xhat_lat.item(2)],
                         [self.xhat_lat.item(3)],
                        ])
        dhat = self.xhat_lat.item(4)
        return xhat, dhat

    def observer_f_lat(self, x_hat, y_m):
        # xhatdot = A*xhat + B*u + L(y-C*xhat)
        xhat_dot = P14.A2_lat @ x_hat \
                   + P14.B1_lat * self.tau_d1 \
                   + P14.L2_lat @ (y_m - P14.C2_lat @ x_hat)
```

```
        return xhat_dot

    def update_lon_observer(self, y_m):
        # update the observer using RK4 integration
        F1 = self.observer_f_lon(self.xhat_lon, y_m)
        F2 = self.observer_f_lon(self.xhat_lon + self.Ts / 2 * F1, y_m)
        F3 = self.observer_f_lon(self.xhat_lon + self.Ts / 2 * F2, y_m)
        F4 = self.observer_f_lon(self.xhat_lon + self.Ts * F3, y_m)
        self.xhat_lon += self.Ts / 6 * (F1 + 2 * F2 + 2 * F3 + F4)
        xhat = np.array([[self.xhat_lon.item(0)],
                         [self.xhat_lon.item(1)],
                        ])
        dhat = self.xhat_lon.item(2)
        return xhat, dhat

    def observer_f_lon(self, x_hat, y_m):
        # xhatdot = A*xhat + B*u + L(y-C*xhat)
        xhat_dot = P14.A2_lon @ x_hat \
                   + P14.B1_lon * (self.F_d1 - P.Fe) \
                   + P14.L2_lon @ (y_m - P14.C2_lon @ x_hat)
        return xhat_dot

    def integrateErrorZ(self, error_z):
        self.integrator_z = self.integrator_z \
                            + (P.Ts/2.0)*(error_z + self.error_z_d1)
        self.error_z_d1 = error_z

    def integrateErrorH(self, error_h):
        self.integrator_h = self.integrator_h \
                            + (P.Ts/2.0)*(error_h + self.error_h_d1)
        self.error_h_d1 = error_h

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

See the wiki for the complete solution.