A Design Study Approach to Classical Control

Randal W. Beard   Timothy W. McLain
Brigham Young University

Updated: December 28, 2020

Homework F.12

(a) Modify the state feedback solution developed in Homework F.11 to add an integrator with anti-windup to the altitude feedback loop and to the position feedback loop.

(b) Allow the plant parameters to vary up to 20% and add a constant input disturbance of 0.1 Newtons to the input of position dynamics simulating wind. *Hint: The best place to add the wind force is in the class that implements the dynamics. For example, one possibility is to modify the z dynamics as*

```
zddot = (-(fr+fl)*sin(theta)+F_wind)/(P.mc+2*P.mr).
```

(c) Tune the integrator poles on both loops (and other gains if necessary) to get good tracking performance.

Solution

The following Python script solves for the controller gains:

```
# VTOL Parameter File
import numpy as np
import control as cnt
```

```python
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 = -1.0
integrator_z = -1.0

# 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.matrix([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.matrix([K1_lat.item(0), K1_lat.item(1), K1_lat.item(2), K1_lat.ite
    ki_lat = K1_lat.item(4)

print('K_lon: ', K_lon)
print('ki_lon: ', ki_lon)
print('K_lat: ', K_lat)
print('ki_lat: ', ki_lat)
```

Python code that implements the associated controller is listed below.

```
import numpy as np
import VTOLParam as P
import VTOLParamHW12 as P12
```

```
class VTOLController:
    def __init__(self):
        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.limit = P.fmax

    def update(self, r, x):
        z_r = r.item(0)
        h_r = r.item(1)
        z = x.item(0)
        h = x.item(1)
        theta = x.item(2)
        # integrate error
        error_z = z_r - z
        self.integrateErrorZ(error_z)
        error_h = h_r - h
        self.integrateErrorH(error_h)

        # Construct the states
        x_lon = np.array([[x.item(1)], [x.item(4)]])
        x_lat = np.array([[x.item(0)], [x.item(2)], [x.item(3)], [x.item(5)]])
        # Compute the state feedback controllers
        F_tilde = -P12.K_lon @ x_lon - P12.ki_lon * self.integrator_h
        F = P.Fe/np.cos(theta) + F_tilde.item(0)
        tau = -P12.K_lat @ x_lat - P12.ki_lat*self.integrator_z
        return np.array([[F], [tau.item(0)]])

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

    def differentiateH(self, h):
        self.h_dot = P.beta*self.h_dot + (1-P.beta)*((h - self.h_d1) / P.Ts)
        self.h_d1 = h

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

    def integrateErrorZ(self, error_z):
        self.integrator_z = self.integrator_z + (P.Ts/2.0)*(error_z + self.error_
        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_
        self.error_h_d1 = error_h

    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.