A Design Study Approach to Classical Control

Randal W. Beard Timothy W. McLain
Brigham Young University

Updated: December 28, 2020

Homework F.10

The objective of this problem is to implement the PID controller using only measured outputs of the system.

(a) Modify the system dynamics file so that the parameters m_c, J_c, d, and μ vary by up to 20% of their nominal value each time they are run (uncertainty parameter = 0.2).

(b) Change the simulation files so that the input to the controller is the output and not the state. Implement the nested PID loops in Problems F.8. Use the dirty derivative gain of $\tau = 0.05$. Tune the integrator so that there is no steady state error. The controller should only assume knowledge of the position z, the angle θ, the altitude h, the reference position z_r, and the reference altitude h_r.

Solution

The solution is on the wiki page associated with the book.

The Python code for the controller is shown below.

```
import numpy as np
import sys
sys.path.append('..')  # add parent directory
import VTOLParam as P
import VTOLParamHW10 as P10
```

```python
from PIDControl import PIDControl

class VTOLController:
    def __init__(self):
        self.zCtrl = PIDControl(P10.kp_z, P10.ki_z, P10.kd_z, \
            P.fmax, P.beta, P.Ts)
        self.hCtrl = PIDControl(P10.kp_h, P10.ki_h, P10.kd_h, \
            P.fmax, P.beta, P.Ts)
        self.thetaCtrl = PIDControl(P10.kp_th, 0.0, P10.kd_th, \
            P.fmax, P.beta, P.Ts)

    def update(self, r, y):
        z_r = r.item(0)
        h_r = r.item(1)
        z = y.item(0)
        h = y.item(1)
        theta = y.item(2)
        F_tilde = self.hCtrl.PID(h_r, h, error_limit=1.0, \
                flag=False)
        F = F_tilde + P.Fe
        theta_ref = self.zCtrl.PID(z_r, z, flag=False)
        tau = self.thetaCtrl.PID(theta_ref, theta,flag=False)
        return np.array([[F], [tau]])
```

```python
import numpy as np

class PIDControl:
    def __init__(self, kp, ki, kd, limit, beta, Ts):
        self.kp = kp                 # Proportional control gain
        self.ki = ki                 # Integral control gain
        self.kd = kd                 # Derivative control gain
        self.limit = limit           # The output will saturate at this limit
        self.beta = beta             # gain for dirty derivative
        self.Ts = Ts                 # sample rate

        self.y_dot = 0.0              # estimated derivative of y
        self.y_d1 = 0.0              # Signal y delayed by one sample
        self.error_dot = 0.0          # estimated derivative of error
        self.error_d1 = 0.0          # Error delayed by one sample
        self.integrator = 0.0        # integrator

    def PID(self, y_r, y, error_limit=1000.0, flag=True):
        '''
            PID control,
```

```

            if flag==True, then returns
                u = kp*error + ki*integral(error) + kd*error_dot.
            else returns
                u = kp*error + ki*integral(error) - kd*y_dot.

            error_dot and y_dot are computed numerically using a dirty derivative
            integral(error) is computed numerically using trapezoidal approximati
        '''

        # Compute the current error
        error = y_r - y
        # integral needs to go before derivative to update error_d1 correctly
        if np.abs(error)<=error_limit:  # only integrate when close
            self.integrateError(error)
        # differentiate error and y
        self.differentiateError(error)
        self.differentiateY(y)

        # PID Control
        if flag is True:
            u_unsat = self.kp*error + self.ki*self.integrator + self.kd*self.erro
        else:
            u_unsat = self.kp*error + self.ki*self.integrator - self.kd*self.y_do
        # return saturated control signal
        u_sat = self.saturate(u_unsat)
        self.integratorAntiWindup(u_sat, u_unsat)
        return u_sat

    def PD(self, y_r, y, flag=True):
        '''
            PD control,

            if flag==True, then returns
                u = kp*error + kd*error_dot.
            else returns
                u = kp*error - kd*y_dot.

            error_dot and y_dot are computed numerically using a dirty derivative
        '''

        # Compute the current error
        error = y_r - y
        # differentiate error and y
        self.differentiateError(error)
```

```
        self.differentiateY(y)

        # PD Control
        if flag is True:
            u_unsat = self.kp*error + self.kd*self.error_dot
        else:
            u_unsat = self.kp*error - self.kd*self.y_dot
        # return saturated control signal
        u_sat = self.saturate(u_unsat)
        return u_sat

    def differentiateError(self, error):
        '''
            differentiate the error signal
        '''
        self.error_dot = self.beta*self.error_dot + (1-self.beta)*((error - self.
        self.error_d1 = error

    def differentiateY(self, y):
        '''
            differentiate y
        '''
        self.y_dot = self.beta*self.y_dot + (1-self.beta)*((y - self.y_d1) / self
        self.y_d1 = y

    def integrateError(self, error):
        '''
            integrate error
        '''
        self.integrator = self.integrator + (self.Ts/2)*(error+self.error_d1)

    def integratorAntiWindup(self, u_sat, u_unsat):
         # integrator anti - windup
         if self.ki != 0.0:
            self.integrator = self.integrator + self.Ts/self.ki*(u_sat-u_unsat);

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