Aircraft Pitch

Authors:

Grace Hutchinson

Erik Perales

Brandon Serna

Lab #3

12/20/2020

Abstract

The purpose of this lab was to model aircraft pitch by implementing several controller design methods, simulate and analyze using MATLAB/Simulink/Simscape, implement a GUI, and design an aircraft animation to show transient response. Using MATLAB, we were able to design several compensators that utilized PID, Root Locus, Frequency Domain, and State-Space controller design methods. We found the implementing State-Space control design methods proved to far more powerful and optimal, while Frequency Domain control design methods revealed how well our aircraft pitch system operates at different frequencies. In the end, we were able to not only meet the designated design requirements, but also improve upon the requirements described by the CTMS website.

Table of Contents

Project Introduction

Background

There are three main axes that control the movement of an aircraft:

- Yaw axis (normal axis) that governs the direction of the aircraft
- Roll axis (longitudinal axis) that governs the rotation of the aircraft
- Pitch axis (transverse axis) that governs the vertical movement of the aircraft

This project was focused on modeling the pitch of the aircraft controlled by the elevators at the end, where a positive pitch had the aircraft nose pointed upward and a negative pitch had the aircraft pitch pointed downward.

The system modeling would normally include the following:

- Lift, that allows the aircraft to be suspended in air and directly opposes weight
- Weight, caused by gravitational pull and directly opposes lift
- Thrust, generated in the engines to move the aircraft and much overcome drag
- Drag, generated on multiple parts of the aircraft, and opposes thrust

Due to the complexity of the system with lift, weight, thrust, and drag, we opted to design using the most ideal conditions, where lift and weight balance out, thrust and weight balance out, the aircraft is at constant altitude, constant velocity, and any change in pitch does not affect the speed of the aircraft.

Objectives

In order to optimize and test several different system controllers we set up the following requirements to meet:

- Percent overshoot $< 5\%$
- Rise Time < 1 second
- Settling Time < 5 seconds
- Steady State Error $< 2\%$

With MATLAB, we modeled the pitch system using transfer functions and state-space forms to be utilized in various controller design to meet requirement for PID (P, PI), root locus, frequency domain, and state-space controller. This was tested using both open and closed loop with to view the step responses to gauge stability. A lead compensator was used by plotting and analyzing the root locus and frequency response to determine the bode plot, phase, and gain margin of the system. Using Simulink, we modeled the state-space model to generate the different responses to open loop and closed loop simulations. From these responses we designed controllers that to improve upon the system and meet our design requirements. Finally, using MATLAB, we used state space and LQR to design and animate the transient system behavior of the aircraft depending on the user desired pitch and weight factor. This allowed for the user to see how the physical system moved and looked depending on the pitch and weight factor and compare to the step input.

Aircraft Pitch: System Modeling

Introduction

The purpose of this experiment is to model our aircraft pitch system using transfer functions and state-space forms that will later be utilized to design various controllers to meet certain requirements using PID, root locus, frequency domain, and state-space controller design methods.

Objectives

By assuming the aircraft is in steady cruise at constant altitude, velocity, and pitch angle, we can represent the aircraft pitch system as a set of linearized longitudinal and lateral equations. We will then be able to design various controllers to meet certain design requirements using PID, root locus, frequency domain, and state-space controller design methods.

Problems

The two-dimensional problem we want to model our aircraft pitch system is based on the following figure and quantities:

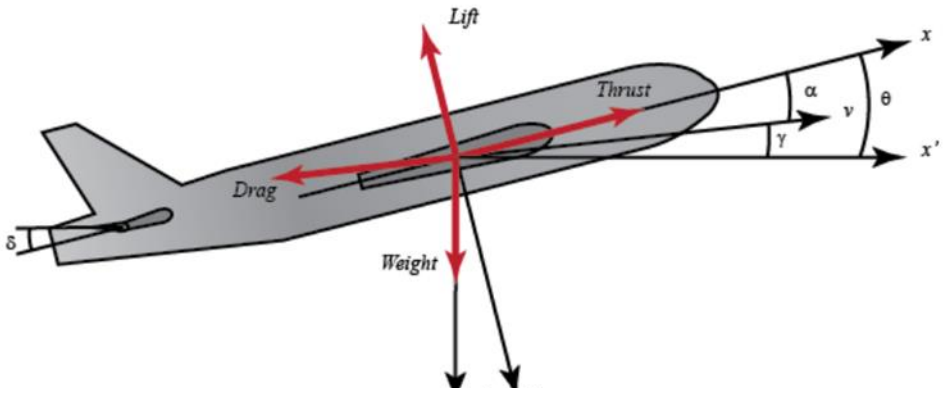


Figure 1) Free Body Diagram of the aircraft showing forces of lift, weight, thrust, and drag

In Figure 1, we will input an elevator deflection angle δ that will output the pitch angle θ while assuming the change in pitch angle will not change the speed of the aircraft, further simplifying our problem. The values and coefficients are taken from the data of Boeing's commercial aircraft. For the PID, root locus, frequency response, and state-space controller design methods the requirements for these methods are as follows:

- Settling time less than 5 seconds
- Rise time less than 1 seconds
- Overshoot less than 5%
- Steady-State error less than 2%

Equations

To obtain both the transfer functions and state-space equations for the aircraft pitch system, we first assume that the aircraft is in steady-cruise at constant altitude and velocity, as well as assume the pitch angle will not change the speed of the aircraft. Thus, the longitudinal equations of motion for the aircraft are the following:

$\alpha = Angle\ of\ attack$

$q = Pitch\ rate$

$\theta = Pitch\ angle$

$\delta = Elevator\ deflection\ angle$

$\mu = \frac{\rho S_{\bar{c}}}{4m}$

$\rho = Density\ of\ air$

$S = Platform\ area\ of\ the\ wing$

$\bar{c} = Average\ chord\ length$

$m = Mass\ of\ the\ aircraft$

$\Omega = \frac{2U}{\bar{c}}$

$U = Equilibrium\ flight\ speed$

$C_T = Coefficient\ of\ thrust$

$C_D = Coefficient\ of\ drag$

$C_L = Coefficient\ of\ lift$

$C_W = Coefficient\ of\ weight$

$C_M = Coefficient\ of\ pitch\ movment$

$\gamma = Flight\ path\ angle$

$\sigma = \frac{1}{1 + \mu C_L} = Constant$

$i_{yy} = Normalized\ movement\ of\ inertia$

$\eta = \mu\sigma C_M = Constant$

$$\dot{\alpha} = \mu\Omega\sigma[-(C_{L+}C_D)\alpha + \frac{1}{(\mu - C_L)}q - (C_W sin\gamma)\theta + C_L]$$

$$\dot{q} = \frac{\mu\Omega}{2i_{yy}}[[C_M - \eta(C_{L+}C_D)]\alpha + [C_M + \alpha C_M(1 - \mu C_L)]q + (\eta C_W sin\gamma)\delta]$$

$$\dot{\theta} = \Omega q$$

After plugging in values and coefficients, we get the following simplified equations:

$$\dot{\alpha} = -0.313\alpha + 56.7q + 0.232\delta$$

$$\dot{q} = -0.0139\alpha - 0.426q + 0.0203\delta$$

$$\dot{\theta} = 56.7q$$

Taking the Laplace transform of the modeling equations, we can find the transfer function of our system with δ as our input and θ as our output after some algebra:

$$sA(s) = -0.313A(s) + 56.7Q(s) + 0.232\Delta(s)$$

$$sQ(s) = -0.0139A(s) - 0.426Q(s) + 0.0203\Delta(s)$$

$$\dot{\theta} = 56.7Q(s)$$

$$\boldsymbol{P_{aircraft}(s) = \frac{\theta(s)}{\Delta(s)} = \frac{1.151s + 0.1774}{s^3 + 0.739s^2 + 0.921s} \left[\frac{rad}{sec}\right]}$$

We can also let the linearized equations of motion be represented in state-space form as a series of standard form matrices. We will use MATLAB to simply this process in the next section.

MATLAB Implementation

Once we have our linearized system of equations, we can represent our aircraft pitch system as a transfer function and a series of state space equations using the following MATLAB code:

```
%% Aircraft Pitch: System Modeling
clear clc
close all
clearvars

% Transfer Function of the System
s = tf('s');
P_aircraft = (1.151*s + 0.1774)/(s^3 + .739*s^2 +0.921*s);
% Transfer Function of the aircraft pitch system

inputs = {'S(delta)'};
outputs = {'theta'};

set(P_aircraft,'InputName',inputs)
set(P_aircraft,'OutputName',outputs)

P_aircraft
```

Figure 2) MATLAB implementation of the Aircraft Pitch system transfer functions

Using the MATLAB tf() function, we can represent the position of the cart and pendulum as a linear SISO transfer function:

```
P_aircraft =

  From input "S(delta)" to output "theta":
      1.151 s + 0.1774
  ------------------------
  s^3 + 0.739 s^2 + 0.921 s

Continuous-time transfer function.
```

Figure 3) Transfer functions of our aircraft pitch system

Figure 3 shows the transfer function for our aircraft pitch system. We can already determine that the system is stable based off the roots of the denominators lying somewhere on the left-hand complex plane, but we will study it further in the next section. Lastly, we can rearrange the following linearized equations derived previously:

$$\dot{\alpha} = -0.313\alpha + 56.7q + 0.232\delta$$

$$\dot{q} = -0.0139\alpha - 0.426q + 0.0203\delta$$

$$\dot{\theta} = 56.7q$$

To represent the aircraft pitch system using state space equations through MATLAB:

```
%%
% State-Space Representation of the System

A = [ -0.313   56.7 0;
      -0.0139 -0.426 0;
            0   56.7 0];
% System Matrix

B = [0.232; 0.0203; 0];
% Input Matrix

C = [0 0 1];
% Output Matrix

D = 0;
% Feedforward Matrix

states = {'alpha' 'q' 'theta'};
inputs = {'S(delta)'};
outputs = {'theta'};
sys_ss = ss(A,B,C,D,'statename',states,'inputname',inputs,'outputname',outputs)
aircraft_pitch = tf(sys_ss)
```

Figure 4) MATLAB code for state-space equations of aircraft pitch system

```
sys_ss =

  A =
           alpha        q    theta
   alpha   -0.313     56.7        0
   q       -0.0139   -0.426       0
   theta         0     56.7       0

  B =
          S(delta)
   alpha     0.232
   q        0.0203
   theta         0

  C =
          alpha      q  theta
   theta      0      0      1

  D =
          S(delta)
   theta         0

Continuous-time state-space model.
```

```
aircraft_pitch =

  From input "S(delta)" to output "theta":
       1.151 s + 0.1774
  -------------------------
  s^3 + 0.739 s^2 + 0.9215 s

Continuous-time transfer function.
```

Figure 5) State-Space and Transfer Function representation of aircraft pitch system

With both the transfer function and state-space representation of the inverted pendulum system we can begin to analyze the stability of the system and design controllers using various methods to meet the design requirements.

Conclusion

In this section we were able to model our aircraft system as a two-dimensional problem and use the forces acting upon the aircraft to determine the linear equations of motion. We were then able to represent our linear equations of motion as a transfer function (Figure 3) and state-space equations (Figure 5) to be analyzed later to design controllers to meet the desired design requirements.

Aircraft: System Analysis

Introduction

The purpose of this section is to analyze the open-loop and closed-loop step response of the aircraft pitch system using MATLAB. In doing so, we will gauge the stability system and design controllers using PID, root, locus, frequency response, and state space design methods in later sections.

Objectives

By utilizing MATLAB and the open-loop transfer functions derived in the previous section (Figure 3), we will plot and analyze open-loop and closed-loop step responses to determine the stability of the aircraft pitch system without a controller.

Problems

For an input δ of 0.2 radians, we want to use PID, root locus, frequency response, and state-space controller design methods to meet the following design requirements:

- Settling time less than 5 seconds
- Rise time less than 1 seconds
- Overshoot less than 5%
- Steady-State error less than 2%

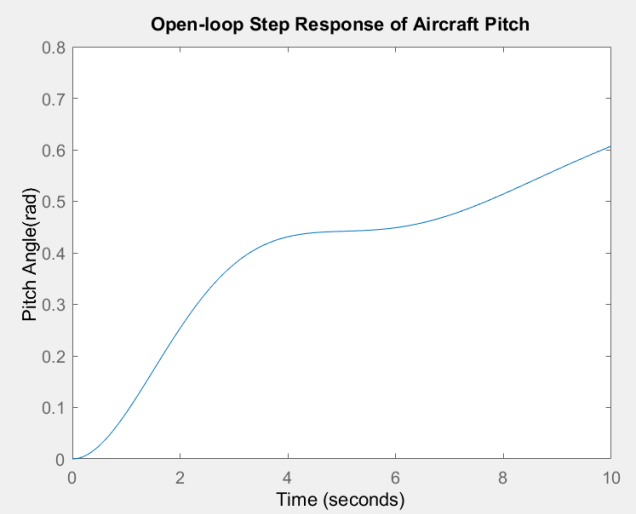
MATLAB Implementation

Using the same MATLAB code in Figure 3 to implement the plant of our aircraft pitch system, we examined the uncompensated open-loop system performance due to an elevator angle step input (δ) of 0.2 radians:

```
%%
% Open-Loop Response

t = [0:0.01:10];
figure()
step(0.2*P_aircraft,t);
axis([0 10 0 0.8]);
ylabel('Pitch Angle(rad)');
title('Open-loop Step Response of Aircraft Pitch');
% Plots the open-loop step response of the system
Poles = pole(P_aircraft)
% Poles of the open-loop aircraft pitch system
Zeros = zero(P_aircraft)
% Zeros of the open-loop aircraft pitch system
```

Figure 6) MATLAB code to plot the open-loop response of our aircraft pitch system




```
Poles =

   0.0000 + 0.0000i
  -0.3695 + 0.8857i
  -0.3695 - 0.8857i


Zeros =

   -0.1541
```

Figure 7) Open-Loop Step Response and the zero and poles of our system

Based on the plot in Figure 7, the plant of our aircraft pitch system does not meet our design requirements and is also unstable. This is due to a pole on the imaginary axis which acts as an integrator, making the output grow to infinity when step input is applied. To stabilize this system, we implemented a feedback controller using the following control architecture:

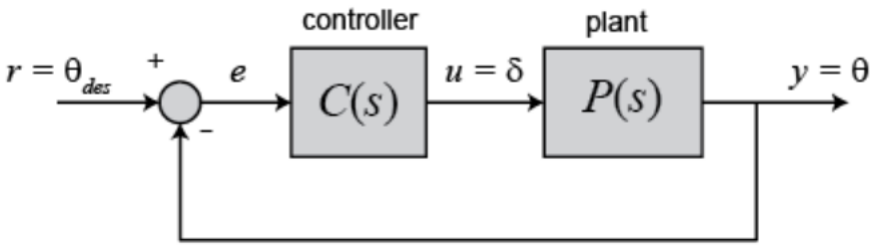
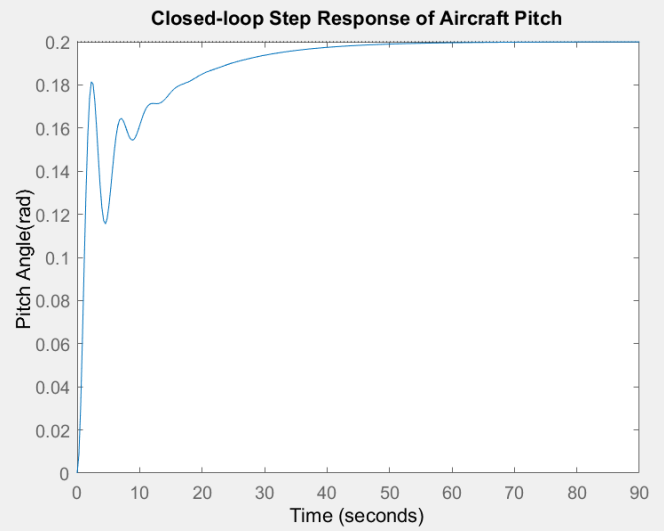


Figure 8) Control architecture of our aircraft pitch system

Using the MATLAB command feedback(), we can easily generate the closed-loop step response of our system:

```
%%
% Closed-Loop Response
cl_sys = feedback(P_aircraft,1)
% Closed-Loop TF of the aircraft pitch system
figure()
step(0.2*cl_sys);
ylabel('Pitch Angle(rad)');
title('Closed-loop Step Response of Aircraft Pitch');
Poles = pole(cl_sys)
Zeros = zero(cl_sys)
% Poles and zeros of the closed-loop aircraft pitch system
stepinfo(0.2*cl_sys)
```

Figure 9) MATLAB code to generate the closed-loop step response, poles, and zeros of the aircraft pitch system




```
cl_sys =

          1.151 s + 0.1774
  ----------------------------------
  s^3 + 0.739 s^2 + 2.072 s + 0.1774

Continuous-time transfer function.
```

```
ans =

  struct with fields:

       RiseTime: 1.7882
   SettlingTime: 35.0896
    SettlingMin: 0.1155
    SettlingMax: 0.2000
      Overshoot: 0
     Undershoot: 0
           Peak: 0.2000
       PeakTime: 98.7611
```

```
Poles =

  -0.3255 + 1.3816i
  -0.3255 - 1.3816i
  -0.0881 + 0.0000i


Zeros =

  -0.1541
```

Figure 10) Closed-loop step response, poles, zeros, and performance of the aircraft pitch system

Based off of the closed-loop step response in Figure 10, we can determine that our aircraft pitch system is stable but does not meet the design the requirements because the settling time is 35 seconds and the rise time is 1.7 seconds, despite having no overshoot. We also know our system is stable because the poles of the system lay on the left-hand complex plane. To get a better understanding of how the poles and zeros affect the closed-loop system response, we transformed the output back to the time domain and generate a system response using a time function. Using MATLAB commands zpk(), residue(), and ilaplace(), we can determine the time domain representation of our aircraft pitch system and plot the closed-loop step response:

```
%%
% Closed-Loop Response in Time Domain
s = tf('s');
R = 0.2/s;
% Step w/ a magnitude of 0.2
Ys = zpk(cl_sys*R);
% System Output due to a step
[r,p,k] = residue(0.2*[1.151 0.1774],[1 0.739 2.072 0.1774 0])
[num,den] = residue(r(1:2),p(1:2),k);
tf(num,den);
syms s
Ys = (0.2/s) - (0.0881/(s+0.08805)) - ((0.1121*s + 0.08071)/(s^2 + 0.6509*s +2.015));
% Partial Fraction Expansion of our system
yt = ilaplace(Ys);
% Calculating the time domain represenation of our system
t = [0:0.1:70];
yt = 0.2 - 0.0881*exp(-0.08805*t) - exp(-0.3255*t).*(0.1121*cos(1.3816*t)+0.0320*sin(1.3816*t));
figure()
plot(t,yt)
xlabel('Time(sec)');
ylabel('Pitch Angle(rad)');
title('Closed-loop Step Response of Aircraft Pitch');
```

Figure 11) MATLAB code to generate the closed-loop step response of the aircraft pitch system in time domain

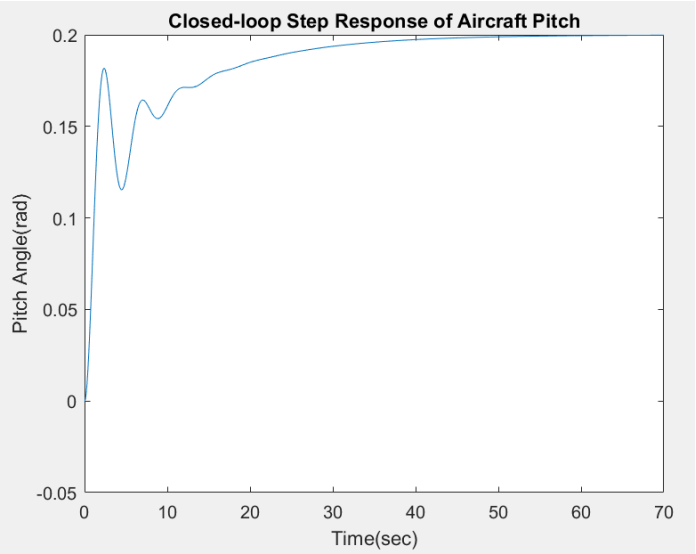


Figure 12) Closed-loop step response of the aircraft pitch system in time domain

After taking the partial fraction expansion and inverse Laplace transform of our aircraft pitch system, we get the following time domain expression for our system:

$$y(t) = 0.2 - 0.0881e^{-0.08805t} - e^{-0.3255t}(0.1121\cos(1.13816t) + 0.0320sin(1.3816t))$$

Each term represents a pole of Y(s) where the real part describes the exponential behavior, and the imaginary part describes the frequency of oscillation of the mode. The zeros alter the relative contribution of each mode by altering the coefficients of each mode. The plot in Figure 12 confirms our results in Figure 10, concluding that while our closed-loop system is stable, it fails to meet the design requirements without proper controller compensation.

Conclusion

In this section we were able to analyze the step response of the open-loop and closed-loop transfer functions of the aircraft pitch system using MATLAB. In doing so, we determined that the open-loop system is unstable due to a pole lying on the imaginary axis. However, the closed-loop response of the system ended up being stable but failed to meet the design criteria of our experiment. In next few sections, we will use PID, root locus, frequency domain, and state-space controller design methods to design a series of controllers that will fulfill the design requirements of this lab.

Aircraft Pitch: PID Controller Design

Introduction

The purpose of this section is to design a PID controller using MATLAB's control system designer for the aircraft pitch system. More specifically, we will analyze the effects of P, PI, and PID control under unity-feedback architecture to meet the design requirements of this lab.

Objectives

By utilizing MATLAB, we will design and analyze the effects of a P, PI, and PID controller to meet the aircraft's desired pitch angle requirements when an elevator pitch angle (δ) of 0.2 radians is applied.

Problems

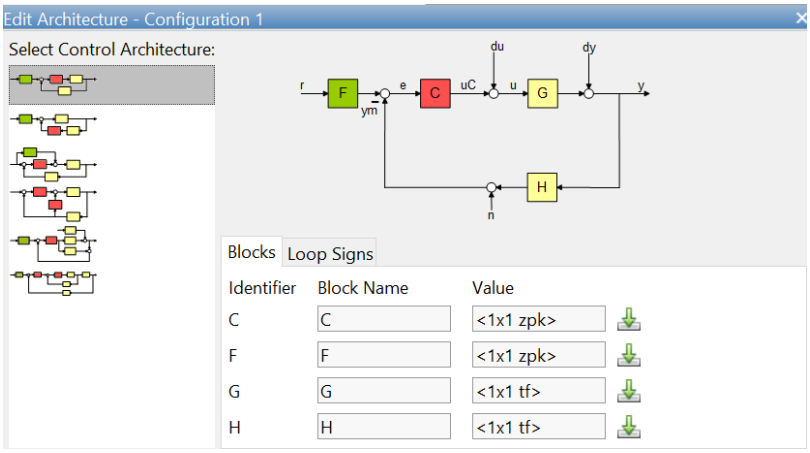
For an input δ of 0.2 radians, we want to design a PID controller to meet the following design requirements:

- Settling time less than 5 seconds
- Rise time less than 1 second
- Overshoot less than 5%
- Steady-State error less than 2%

In addition, we will also analyze the effects a P and PI controller will have on the response of the aircraft pitch system under unity-feedback.

MATLAB Implementation

Using MATLAB's controlSystemDesigner() function we can begin to design a P, PI, and PID controller to better understand how different combinations of the three controllers affect the overall response of the system. The following MATLAB code and Control System Designer Architecture are as follows:




```
%% Aircraft Pitch: PID Controller Design
clear clc
close all
clearvars

s = tf('s');
P_aircraft = (1.151*s + 0.1774)/(s^3 + .739*s^2 +0.921*s);
% Plant of our aircraft pitch system
controlSystemDesigner(P_aircraft);
```

Figure 13) Closed-loop step response of the aircraft pitch system in time domain

In Figure 13, we used the same transfer function found in Figure 3 for our aircraft pitch system and incorperated it into our Control Architectur where G is our plant, C is our controller, and F is our input. After changing our F value to 0.2 to reflect our refrence of a step function of 0.2 radians we can design a simple P controller by setting our C block (Kp in this case) to 2. Then we used PID Tuning to change the compensator block C in order to meet certain response time and transient behavior requirements. Since we desire a rise time of 1 second, we set our response time to 0.9904 and observed the following closed-loop step response:

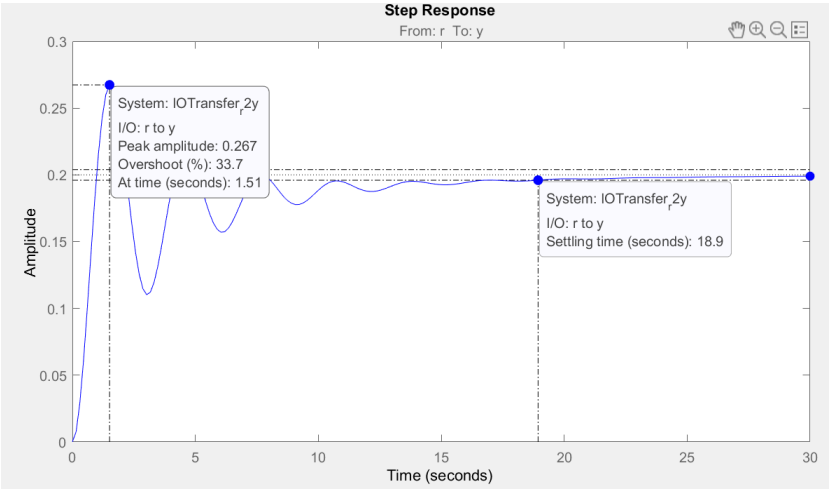


Figure 14) Closed-loop response of the aircraft pitch system under Proportional control

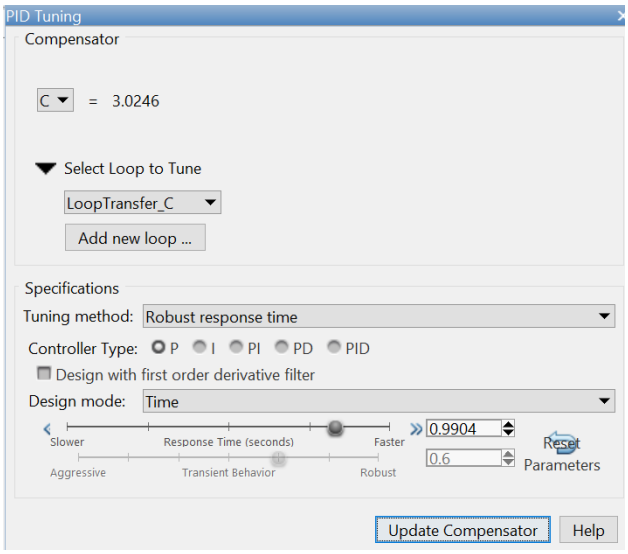


Figure 15) PID Tuning Settings for P compensator

With a Kp of 3.0246 chosen by the algorithm in Figure 15, our closed-loop step response shown in Figure 14 indicated that while our P compensated system meets the desired rise time and steady state error requirement, the overshoot and settling time are too high. In short, a proportional controller will not be enough to meet the given requirements. Changing our controller type to a PI controller using the PID Tuning option, we can add an integrator to our compensator and observe the closed-loop step response under PI control:

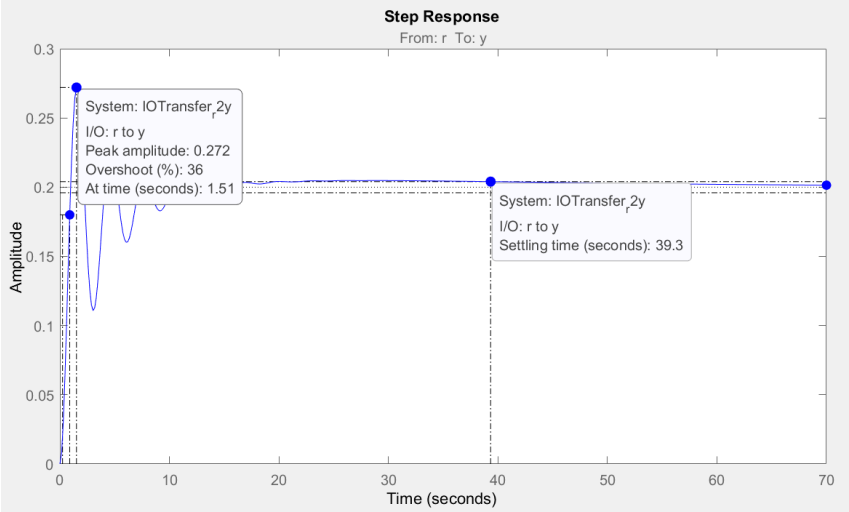


Figure 16) Closed-loop response of the aircraft pitch system under PI control

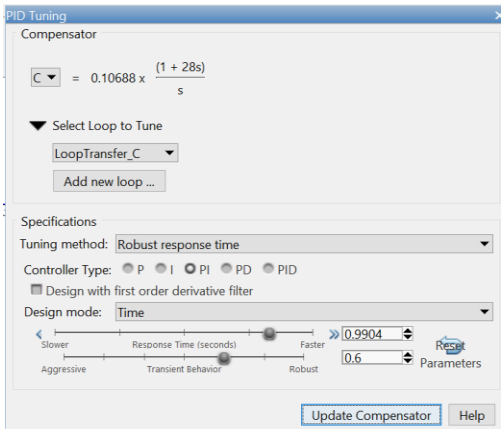


Figure 17) PID Tuning Settings for PI compensator

With a Kp of 2.99 and a Ki of $\frac{0.010688}{s}$, we observe that our closed-loop step response in Figure 16 indicates that the introduction of an integral controller increases overshoot and settling time immensely while our rise time and steady-state error remain unchanged, just as expected. To combat this this, introducing a derivative controller would decreases the overshoot and settling time, thus removing the oscillatory behavior and increases the performance of our system. After changing our controller type to PID and through much trial and error, we were able to achieve the following closed-loop step response and PID controller gains:

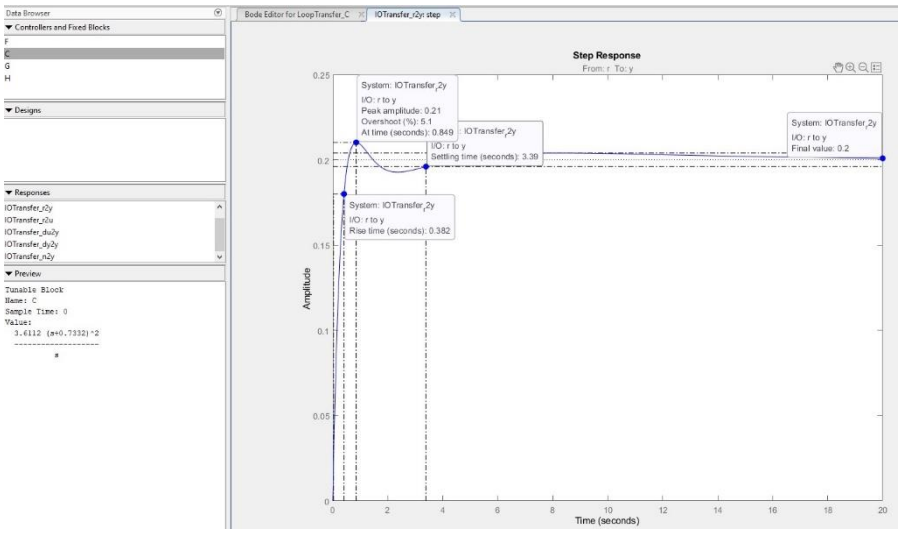


Figure 18) Closed-loop response of the aircraft pitch system under PID control

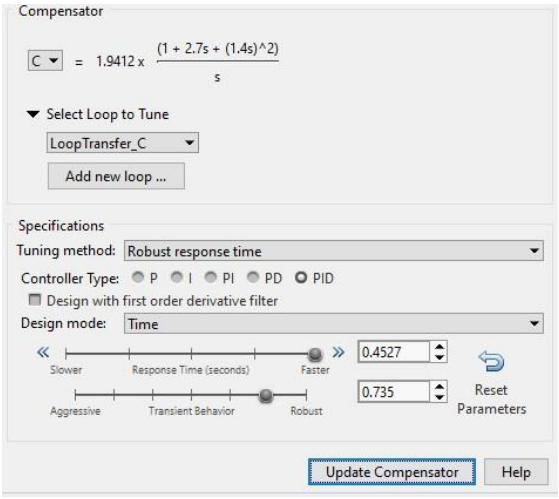


Figure 19) PID Tuning Settings for PID compensator

With a Kp of 5.295, Ki of $\frac{1.941}{s}$, Kd of $3.6112s$, we were able to successfully decreases the overshoot and settling time to 5.1% and 3.39 seconds respectively, as shown in Figure 18. With the introduction of derivative controller, we were able to counter act the overshoot and settling time impact that a proportional and integral controller have on our aircraft pitch system.

Conclusion

In this section, we were able to design and analyze the behavior of a P, PI, and PID controller using MATLAB for our aircraft pitch system. With a P controller, we were able to meet the rise time and steady-state error requirements but made our system oscillatory. A PI controller implementation further increased the overshoot and settling time of our system but reduced the steady-state error percentage to 0. Adding a derivative controller reduces the impact a PI controller had on the overshoot and settling time, allowing us to change the controller gains until the design requirements are met.

Aircraft Pitch: Root Locus Design

Introduction

The purpose of this section is to design a lead compensator for our aircraft pitch system by plotting and analyzing the root locus of the open-loop system with and without a compensator using MATLAB.

Objectives

By utilizing MATLAB, we will design a lead compensator through analyzing the root locus of the open-loop aircraft pitch system to meet the desired angle pitch requirements. We will then plot and analyze the closed-loop step response to determine the stability of our system with a lead controller designed based off the root locus of the open loop system.

Problems

For an input δ of 0.2 radians, we want to design a PID controller to meet the following design requirements:

- Settling time less than 5 seconds
- Rise time less than 1 second
- Overshoot less than 5%
- Steady-State error less than 2%

In addition, we will also analyze the effects a P and PI controller will have on the response of the aircraft pitch system under unity-feedback.

MATLAB Implementation

To plot and analyze the root locus of the open-loop aircraft pitch system, we once again used MATLAB's controlSystemDesigner() to plot the root locus and closed-loop step response of our system:

```
%% Aircraft Pitch: Root Locus Design
clear clc
close all
clearvars

s = tf('s');
P_aircraft = (1.151*s + 0.1774)/(s^3 + .739*s^2 +0.921*s);
controlSystemDesigner('rlocus',P_aircraft);
```

Figure 20) MATLAB code to plot the root locus and closed-loop step response of our aircraft pitch system

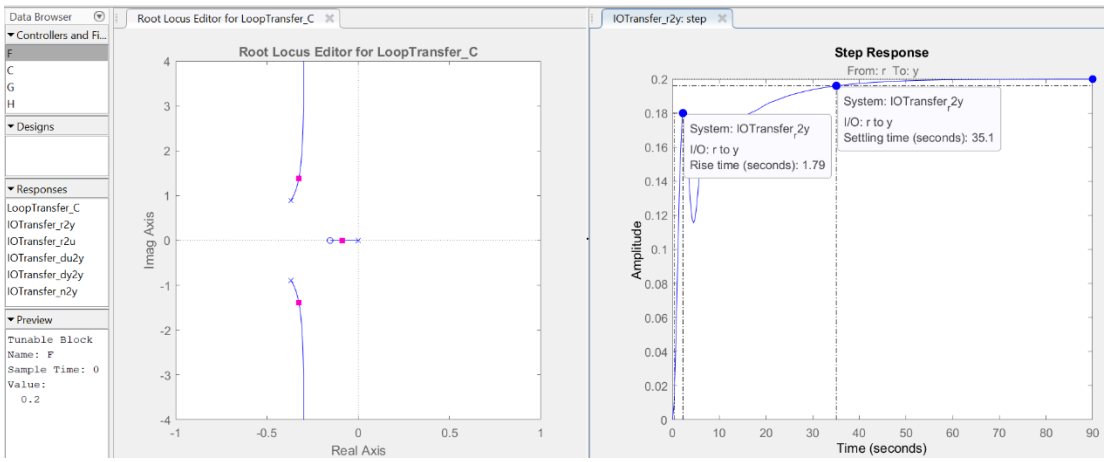


Figure 21) MATLAB code to plot the root locus and closed-loop step response of our aircraft pitch system

Based on Figure 21, we determine that the current placement of our closed-loop poles on the root locus plot will not fulfill our design requirements. To get a better understanding of where our closed-loop poles should lie on the root locus, we can add our design requirements to the root-locus plot. Specifically, we will add our overshoot, settling time, and rise time design

requirements. Despite rise time being an option, we can approximate the desired natural frequency using the following relation:

$$w_n = \frac{1.8}{T_r}$$

Because we want a rise time of 1 second, our desired natural frequency is 1.8 rad/sec. Adding these design requirements to our root locus, we get the following plot:

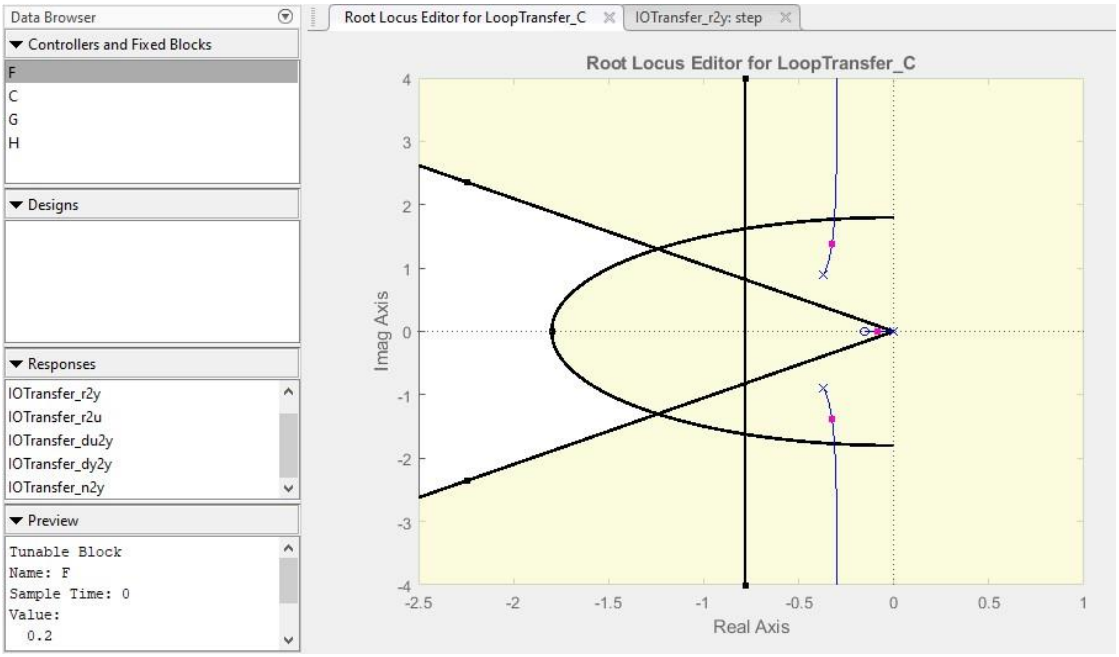


Figure 22) Root Locus plot of our aircraft system w/ design requirements

Based off of the root locus plot in Figure 22, we determined the unshaded regions where we should place our closed-loop poles. The vertical line around s = -0.8 represents the settling time requirement, the curved line represents the rise time requirement, and the two rays centered around the origin represent the overshoot requirement. Therefore, since none of the 3 root locus branches enters the unshaded region, we cannot place the closed loop poles anywhere that would meet the design requirements. To shift the root locus branches more to the left of the complex

plane, a lead compensator is needed where the magnitude of the pole is greater than the zero. We will place the zero of our lead compensator around z = -1.8 to ensure that as we increase our gain K, the branch of the root locus that approaches the open-loop zero will not exit the unshaded region. For the pole of our compensator, we want to place it further away left from the zero to meet the desired transient and steady state response. Lately, we will choose our gain K so that the two slowest closed-loop poles will approach the zeros, therefore reducing the rise time and settling time of the system. Choosing our p = -65 and K = 901.9, we were able to achieve the following step response:

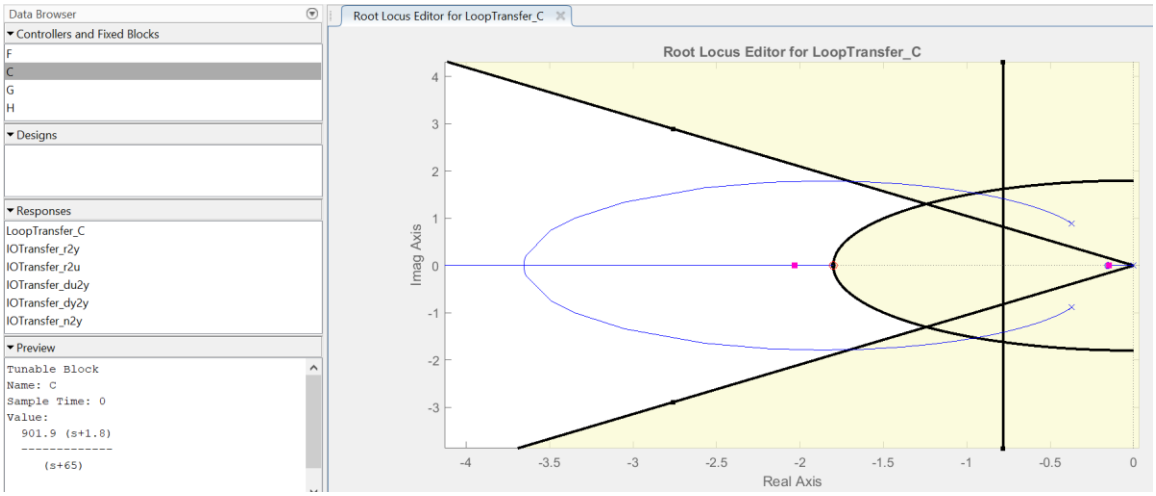


Figure 23) Root Locus plot of our aircraft system w/ lead compensator

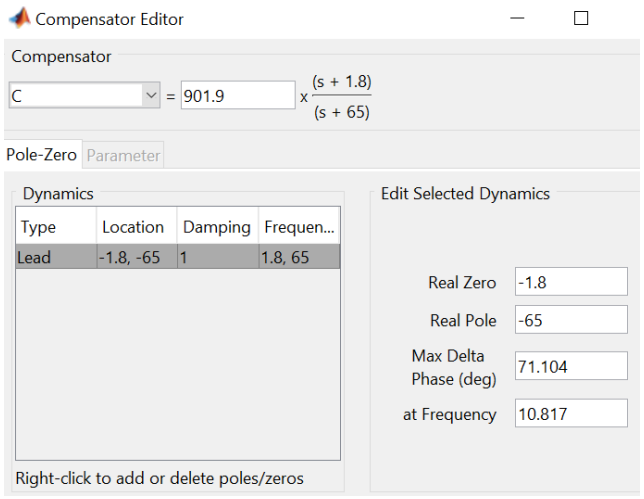


Figure 24) Lead Compensator values

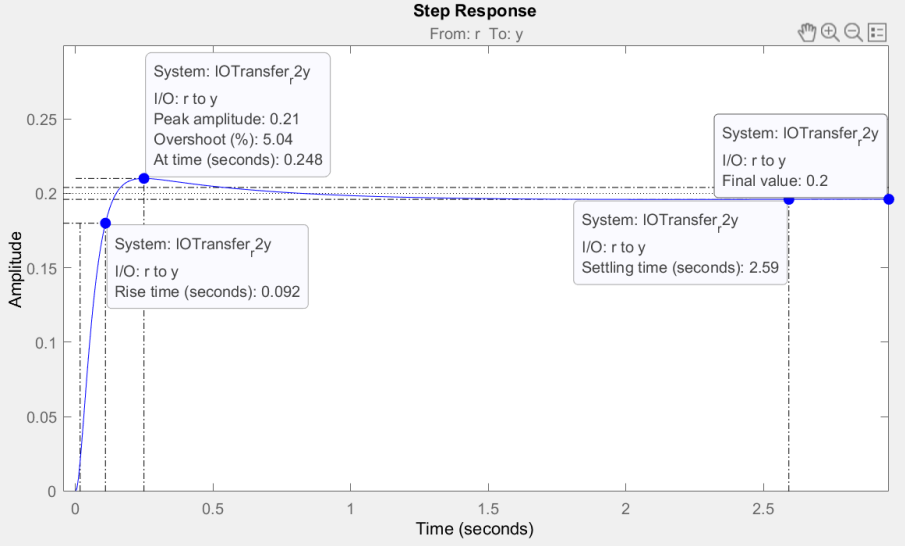


Figure 25) Closed-loop step response of our system w/ lead compensator

Figure 23 shows the effect our lead compensator has on the root locus of our system, which moves certain parts of the root locus branches to exist in the shaded region, allowing us to place our closed-loop poles in those shaded regions in order to meet the design requirements. Figure 25 shows the closed-loop step response of our system under lead compensator control, which confirms that our compensated system meets the desired design requirements.

Conclusion

In this section we analyzed the root locus of our open-loop system to determine how to meet the design requirements. We found that a lead compensator was needed in order to move the branches closer to the left and side of the complex plane after choosing zero, pole, and gain to be z = - 1.8, p = -65, and K = 901.9 respectively, we were able to meet the desired closed-loop step response requirements.

Aircraft Pitch: Frequency Domain Methods for Controller Design

Introduction

The purpose of this section is to design a compensator that will meet the system design requirements by analyzing the frequency response of the closed-loop aircraft pitch system using the margin() command in MATLAB, which will determine the bode plot, phase, and gain margin of the system.

Objectives

By utilizing MATLAB, we will design a compensator by analyzing the frequency response, phase margin, and gain margin of the closed-loop aircraft pitch system to meet the desired angle requirements. We will then plot and analyze the bode plot and step response of the system to determine the stability and performance of the aircraft pitch system with compensation.

Problems

For an input δ of 0.2 radians, we want to design a controller to meet the following design requirements:

- Settling time less than 5 seconds
- Rise time less than 1 second
- Overshoot less than 5%

MATLAB Implementation

To design our controller using the frequency domain, first we confirmed its stability by examining the frequency response of our closed-loop aircraft pitch system, specifically the phase and gain margin. Using the MATLAB commands feedback() and margin(), we can generate a closed-loop transfer function and bode plot of our aircraft pitch system that will determine its stability and performance based off of the closed-loop step response:

```
%% Aircraft Pitch: Frequency Domain Design
clear clc
close all
clearvars
s = tf('s');
P_aircraft = (1.151*s + 0.1774)/(s^3 + .739*s^2 +0.921*s);
% Plant of our aircraft pitch system
figure()
margin(P_aircraft)
% Frequency Response of the Open-Loop system
grid
Ts = feedback(P_aircraft,1);
% Closed-Loop Transfer Function of our system
figure()
step(0.2*Ts);
% Closed-Loop Step Response of our system
```

Figure 26) MATLAB code to generate the bode plot and step-response system w/out compensation

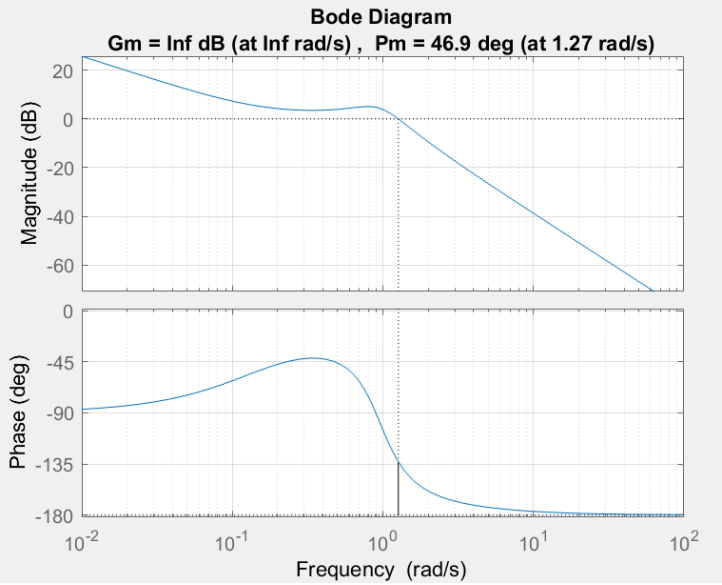


Figure 27) Frequency response of our open-loop system

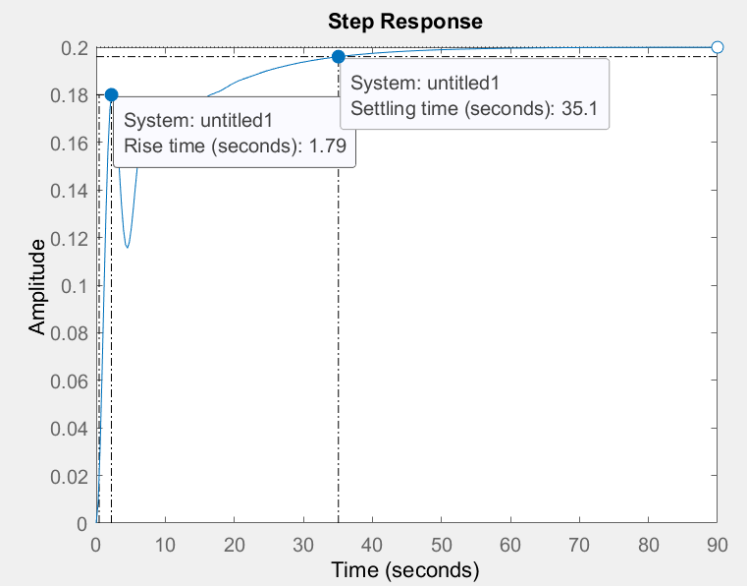


Figure 28) Closed-Loop Step Response of our system

Based on the positive gain and phase margins in Figure 27, our closed-loop system is indeed stable, which is apparent in the closed-loop step response in Figure 28. However, the settling time and rise time fail to meet the design requirements, indicating we need to increase the response time by increasing the crossover frequency, which increases the overshoot in return. To combat this, we also know that the phase margin is inversely proportional to the closed-loop system's overshoot. As a result, a lead compensator will be implemented to both increase the gain crossover frequency and the phase margin. The general form for how lead compensator is as follows:

$$C(s) = K\frac{Ts + 1}{\alpha Ts + 1}$$

Because our system is type 1, the steady-state error for a step input will be zero for any value of K as long as it >1. Thus, we will arbitrarily choose K = 10, which will in turn reduce the system's phase margin and increase the closed-loop system's overshoot as shown blow:

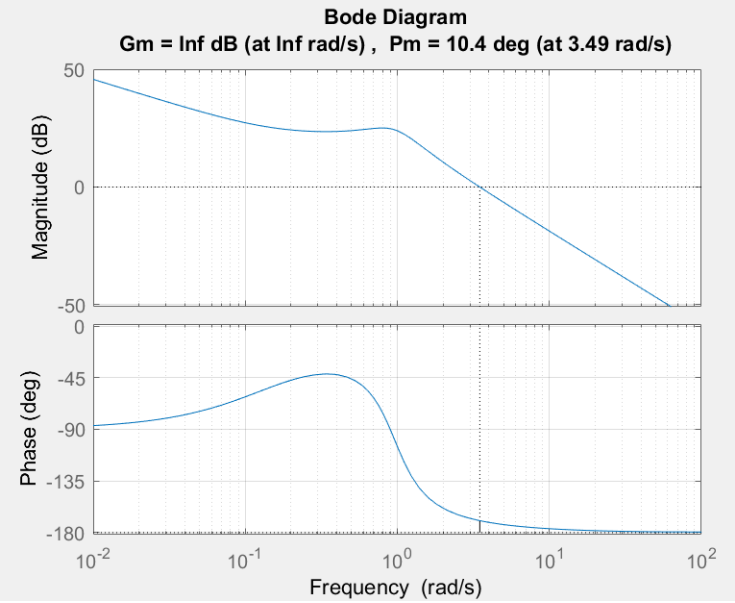


Figure 29) Frequency response of our open-loop system with K = 10

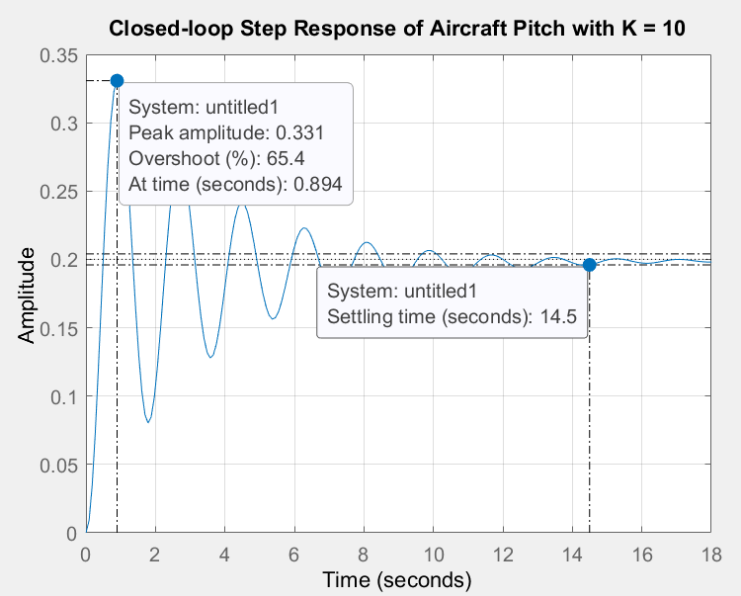


Figure 30) Closed-Loop Step Response of our system

In Figure 29, the introduction of a gain K >1 shifted our magnitude plot up while also decreasing the phase margin and increasing the gain crossover frequency. As a result, the closed-loop response experienced a drastic increase in the overshoot while decreasing the rise time and settling time. The lead compensator will help by adding damping to the system, which will

reduce overshoot and settling time. We can choose our parameter α, which is the ratio between the zero and pole, using the following equation:

$$\alpha = \frac{1 - \sin(PM)}{1 + \sin(PM)}$$

The larger the separation between the zero and pole, the greater the phase margin allowed for our system. PM represents the amount of phase lead we want to add to our system, which has a current phase margin of 10.4 degrees when K = 10. To determine how much phase we should add, we can use the following equation to approximate PM:

$$\zeta = \frac{PM}{100}$$

Because we desire an overshot less than 5%, $\zeta = 0.7$, thus the phase margin of our system must be at least 70 degrees. Therefore, letting PM = 65 will give us enough phase margin to meet the requirement, and the addition of 5 degrees to account for the increase in the gain crossover frequency. Our α in this case is:

$$\alpha = \frac{1 - \sin(PM)}{1 + \sin(PM)} = \frac{1 - \sin(65)}{1 + \sin(65)} = 0.0491$$

From α, we can determine the amount of magnitude that will be added at the new crossover frequency w_{max}:

$$Mag = 20\log\left(\frac{1}{\sqrt{\alpha}}\right) = 20\log\left(\frac{1}{\sqrt{0.0491}}\right) = 13dB$$

Examining the bode plot in Figure 28, we see that at -13dB, our gain cross over frequency will move from w = 3.49 rad/sec to 7.41 rad/sec. With w_{max}, we can calculate T using the following equation:

$$T = \frac{1}{w_{max}\sqrt{\alpha}} = \frac{1}{7.41\sqrt{0.0491}} = 0.6087$$

With K = 10, α = 0.0491, and T = 0.6087, we can use MATLAB to generate our lead compensator and analyze the bode plot and closed-loop step response of our compensated system:

```
OS = 5;
K = 10;
zeta = (-log(OS/100))/(sqrt(pi^2 + (log(OS/100))^2))
phase_marg = zeta*100
% Desired PM
alpha = (1-sind(65))/(1+sind(65))
mag_wc = 20*log10(1/sqrt(alpha))
% @ -13dB, wm = 7.41
wm = 7.41
% New crossover frequency
T = 1/(wm*sqrt(alpha))
Cs= K*(T*s + 1)/(alpha*T*s + 1)
% Lead Compensator
figure()
margin(Cs*P_aircraft)
grid
sys_cl = feedback(Cs*P_aircraft,1);
figure()
step(0.2*sys_cl)
xlim([0 10])
grid
title('Closed-loop Step Response of Aircraft Pitch with Lead Compensator')
stepinfo(0.2*sys_cl)
```

Figure 31) MATLAB code to generate the bode plot and step-response system w/ compensation

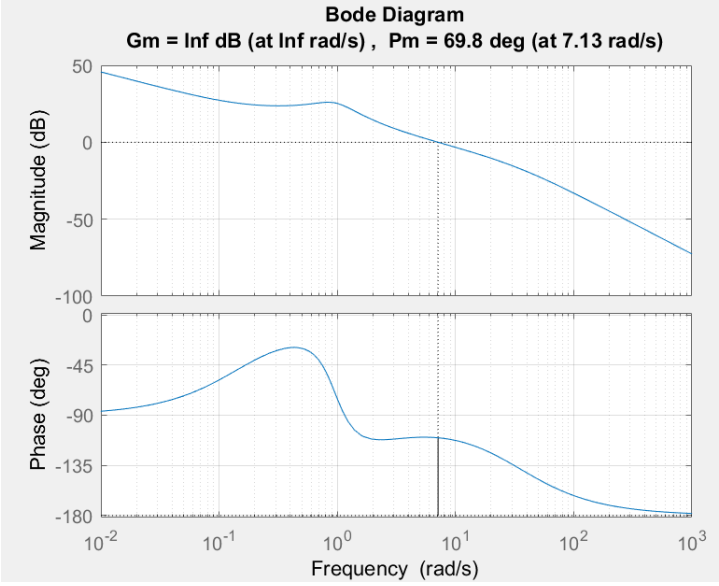


Figure 32) Frequency response of our open-loop system w/ compensation

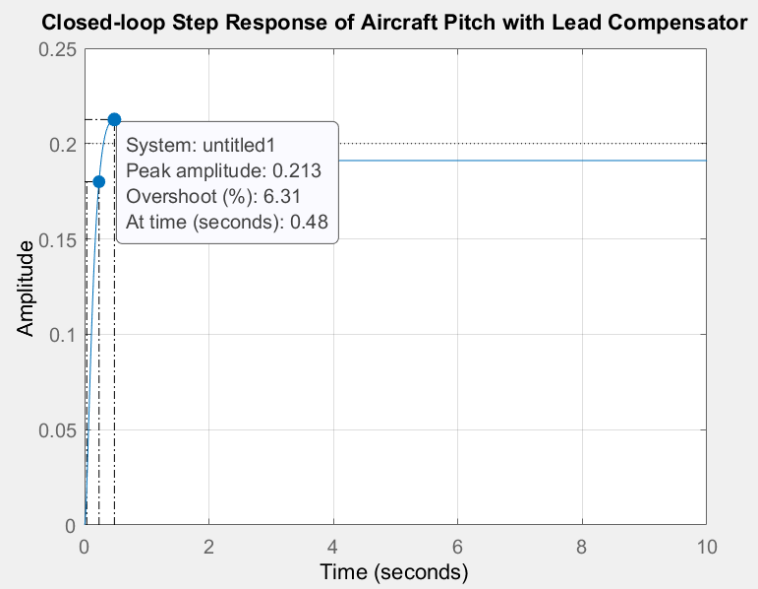


Figure 33) Closed-loop step response of our open-loop system w/ compensation

Based on the frequency response of our system shown in Figure 32, we were unable to reach the desired phase margin of 70 degrees necessary to meet the overshoot requirement, which is confirmed in the closed-loop step response of our aircraft pitch system shown in Figure 33 that fails to meet the overshoot requirement. We can iterate the process again by adding 5 degrees to our desired PM, totaling out to 70 degrees:

$$\alpha = \frac{1 - \sin(PM)}{1 + \sin(PM)} = \frac{1 - \sin(70)}{1 + \sin(70)} = 0.0311$$

From α, w_{max} is approximately the following:

$$Mag = 20\log\left(\frac{1}{\sqrt{\alpha}}\right) = 20\log\left(\frac{1}{\sqrt{0.0311}}\right) = 15dB$$

Based on the open loop bode plot, w_{max} is approximately 8.2 rad/sec, making our T to be the following:

$$T = \frac{1}{w_{max}\sqrt{\alpha}} = \frac{1}{8.2\sqrt{0.0311}} = 0.6916$$

Applying MATLAB once again to model our new compensator:

```
alpha = (1-sind(70))/(1+sind(70))
mag_wc = 20*log10(1/sqrt(alpha))
% @-15db, wmax is approximatley 8.2
wm = 8.2
T = 1/(wm*sqrt(alpha))
Cs= K*(T*s + 1)/(alpha*T*s + 1)
% New lead compensator
figure()
margin(Cs*P_aircraft)
grid
sys_cl = feedback(Cs*P_aircraft,1);
figure()
step(0.2*sys_cl)
xlim([0 5])
grid
title('Closed-loop Step Response of Aircraft Pitch with Lead Compensator')
stepinfo(0.2*sys_cl)
```

Figure 34) MATLAB code to generate the bode plot and step-response system w/ new compensator

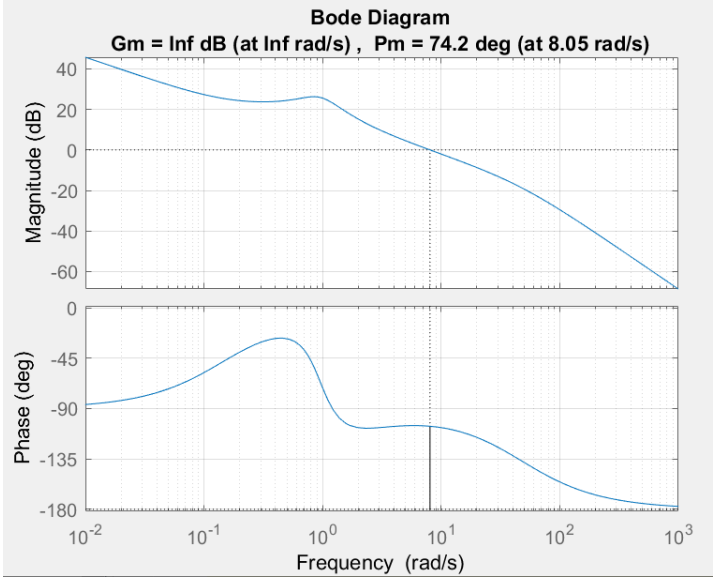


Figure 35) Frequency response of our open-loop system w/ new compensator

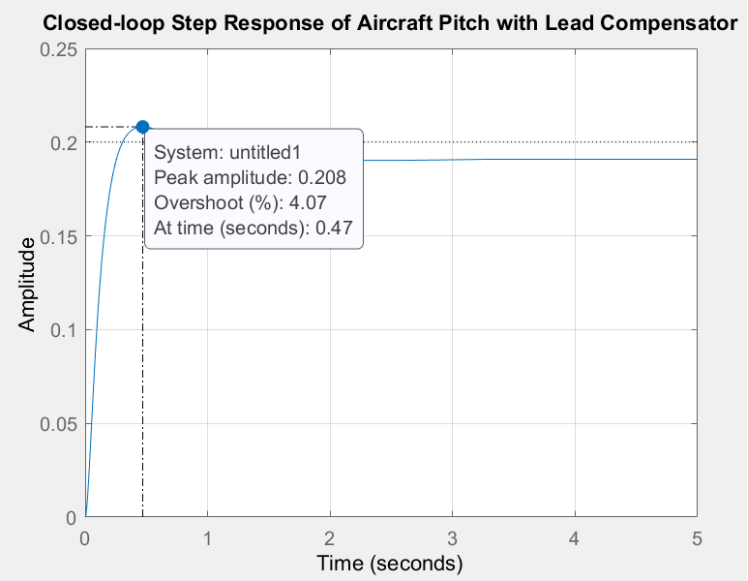


Figure 36) Closed-loop step response of our open-loop system w/ new compensator

Based on Figure 35, by adding an additional 5 degrees of phase, we were able reach a phase margin of 74.2 degrees at a crossover frequency of 8.05 rad/s, which more than meets the PM needed to achieve an overshoot less than 5%. Our closed-loop step response in Figure 36 reflects these results, indicating our compensated aircraft craft pitch system meets the designated design requirements.

Conclusion

In this section we were able to design a lead compensator by analyzing the frequency response of the open-loop aircraft pitch system to meet the desired design requirements. By analyzing the bode plot and phase margin, we were able to determine how much phase to add to our system and predict what crossover frequency was needed to maximize the system's resulting phase margin. After a few iterations, we were able to design a lead compensator that would meet the system's design requirements.

Aircraft Pitch: State-Space Methods for Controller Design

Introduction

The purpose of this section is to design a compensator through MATLAB using the state-space design methods of pole-placement and LQR that will meet our system design requirements.

Objectives

By utilizing MATLAB, we will design a compensator using pole-placement and Linear Quadratic Regulation (LQR). We will then examine the step response of the system and adjust performance accordingly using pole-placement and LQR.

Problems

For an input δ of 0.2 radians, we want to design a controller based on state-space methods to meet the following design requirements:

- Settling time less than 5 seconds
- Rise time less than 1 second
- Overshoot less than 5%
- Steady-State error less than 2%

MATLAB Implementation

Pole-Placement Control Design

Before we can use state-space controller design techniques to design a controller for our aircraft pitch system, we first need to determine if our system is completely state controllable. We use the MATLAB commands ctrb() and rank() to find the controllability matrix and the rank of the controllability matric respectively:

```
% State-Space Representation of the System
A = [ -0.313   56.7 0;
      -0.0139 -0.426 0;
           0   56.7 0];
% System Matrix
B = [0.232; 0.0203; 0];
% Input Matrix
C = [0 0 1];
% Output Matrix
D = 0;
% Feedforward Matrix
air_pitch = ss(A,B,C,D)
% State-Space Model of our Aircraft Pitch system
Co = ctrb(air_pitch)
% Controllability Matrix of our system
rank(Co)
% Rank of our controllability matrix
```

Figure 37) MATLAB code to find the controllability matrix and its rank

```
Co =

    0.2320    1.0784   -1.0107
    0.0203   -0.0119   -0.0099
         0    1.1510   -0.6732


ans =

     3
```

Figure 38) Controllability matrix of our system and its respective rank

Since our controllability matrix is 3x3 and is of rank 3 as shown in Figure 38, our system is completely state controllable. We can then use the following schematic to implement a full-state feedback control system:

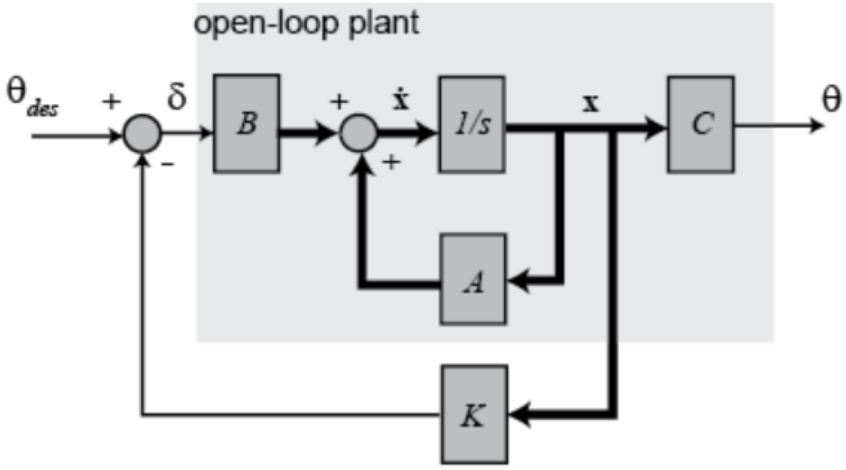


Figure 39) Full-State feedback control system block diagram

Where K is the control gain matrix, x is the state vector [α, q, θ]', θ_{des} is our reference input r, δ is our control input ($\theta_{des} = Kx$), and θ is our output. Because θ is our only output and all of the state variables in the vector are measured, a state observer won't be needed to estimate the other state variables. Our linear state space equations then become the following:

$$\dot{x} = (A - BK)x + B\theta_{des}$$

$$\theta = Cx$$

Since the determinate of $[sI(A - BK)]$ is a third-order polynomial, as presented in Figure 5 where the transfer function of our aircraft pitch system plant is of third order, there are three closed-loop poles that we can place anywhere because our system is completely state controllable. Examining our transfer function:

$$\boldsymbol{P_{aircraft}(s) = \frac{\theta(s)}{\Delta(s)} = \frac{1.151s + 0.1774}{s^3 + 0.739s^2 + 0.921s} \left[\frac{rad}{sec}\right]}$$

We note that there is a zero located at s = -0.154, so choosing one of our poles to be at that location will cancel out the affect that our plant's zero has on our system. For the two other complex poles, we can use the following second order characteristic equation to approximate where to place those poles:

$$q(s) = s^2 + 2w_n\zeta s + {w_n}^2$$

Because we desire a 5% overshoot and a settling time less than 1, ζ can be approximated using the following equation:

$$\boldsymbol{\zeta = \frac{-\ln\left(\frac{OS}{100}\right)}{\sqrt{\pi^2 + \left(\ln\frac{OS}{100}\right)^2}} = \frac{-\ln\left(\frac{5}{100}\right)}{\sqrt{\pi^2 + \left(\ln\frac{5}{100}\right)^2}} = 0.6901}$$

With ζ, we can find w_n using the following equation:

$$\boldsymbol{w_n = \frac{4}{\zeta * Ts} = \frac{4}{0.6901 * (1)} = 5.7962}$$

Plugging w_n and ζ back into our characteristic equation:

$$q(s) = s^2 + 8s + 33.6$$

The roots/desired closed-loop poles of our system are therefore s = -4 ± 4.2i, but because we are dealing with a higher order system will place our closed-loop poles 5x farther to the left to the complex plane(s = -20 ± 20i), which will have little contribution to the transient response. Before we use the MATLAB command place() or acker() to place our poles and determine the gain matrix using Ackerman's formula, we need to add prefilter to scale the reference input to ensure the output equals the reference in steady state. Otherwise, we would not expect the output to equal the commanded reference due to the full-state feedback not comparing the output to the reference input. The system block diagram for our full-state feedback system with prefilter is shown below:

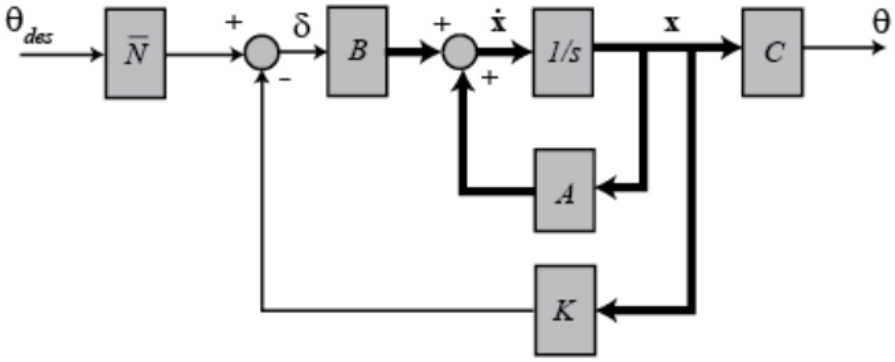


Figure 40) Full-State feedback control system w/ prefilter block diagram

Our prefiler $\bar{N}$ can be calculated by the following equation:

$$\bar{N} = \frac{1}{C(-A + Bk^T)^{-1}B}$$

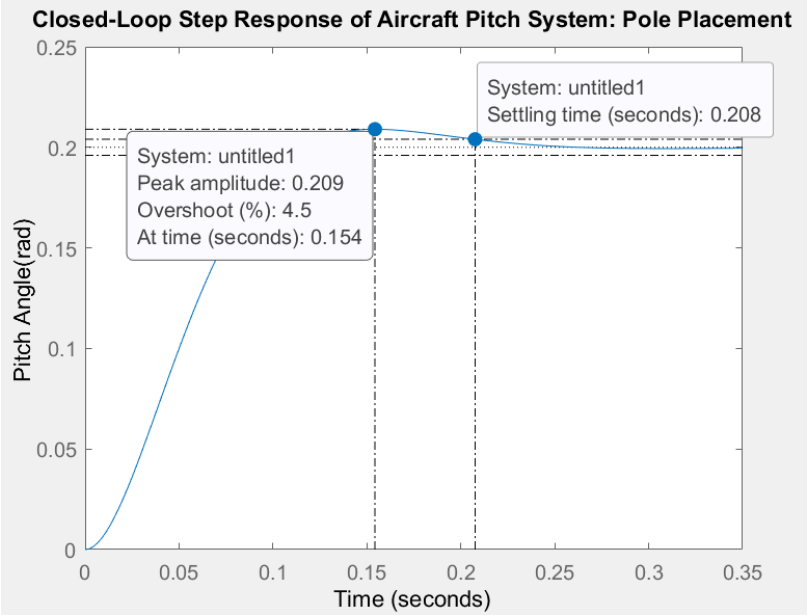
where k^T is our gain matrix determined from the MATLAB commands place() or acker(). Using MATLAB to calculate $\bar{N}$ and place() to determine our gain matrix through Ackerman's formula, we get the following closed-loop step response:

```
%%
% Pole Placement Implementation

Ts = 1;   % Settling Time
OS = 5;  % Percent Overshoot
zeta = (-log(OS/100))/(sqrt(pi^2 + (log(OS/100))^2));
wn = 4/(zeta*Ts);

P = [-20+20.4i -20-20.4i -0.154];
% Desired closed-loop pole placements
K = place(A,B,P)
% Gain matrix
Ac = [(A-B*K)];
N = 1/(C*((-Ac)^-1)*B)
% Precompensation Gain
cl_sys = ss(Ac,N*B,C,D);
% State-Space Model of Closed-Loop System
step(.2*cl_sys);
ylabel('Pitch Angle(rad)');
title('Closed-Loop Step Response of Aircraft Pitch System: Pole Placement');
stepinfo(cl_sys)
```

Figure 41) MATLAB code to implement full-state feedback and prefilter through pole placement




```
    RiseTime: 0.0746
SettlingTime: 0.2076
 SettlingMin: 0.9127
 SettlingMax: 1.0450
   Overshoot: 4.5003
  Undershoot: 0
        Peak: 1.0450
    PeakTime: 0.1543
```

Figure 42) Closed-Loop Step Response of our Aircraft Pitch system due to pole-placement and prefilter design

Based on closed-loop step response in Figure 42, our full-state feedback controller based on pole-placement successfully meets our design requirements. The addition of a prefilter ensures that that output equals the reference in steady state without affecting the transient response of the system. However, the drawbacks to pole placement control is that for higher order systems such as ours, it is harder to determine where the closed-loop poles should be and there is no sense of the amount of energy needed to implement pole-placement control. In the next section, we will implement LQR control design and examine the benefits of determining our gain matrix based on how well our system preforms and how much effort it takes to get that performance.

LQR Control Design

Using the MATLAB command lqr(), we can generate the optimal gain matrix K without choosing the location of our closed-loop poles. This is achieved by balancing the system error and control effort based on the state-cost matrix Q and control weight matrix R, where Q weighs the importance of the steady-state error of the system and R weighs the effort of our controller. Setting our Q = 30, R = 1, and implementing a prefilter to ensure our output matches the reference input, we used MATLAB to simulate the closed-loop step response under LQR control:

```
%%
% LQR Implementation
p = 30;
% State-Cost Weighting Factor
Q = p*C'*C;
R = 1;
% Control Weighting Factor
K = lqr(A,B,Q,R)
% Gain matrix
Ac = [(A-B*K)];
kr = 1/(C*((-Ac)^-1)*B)
% Prefilter
cl_sys = ss(Ac,kr*B,C,D);
step(0.2*cl_sys)
ylabel('Pitch Angle(rad)');
title('Closed-Loop Step Response of Aircraft Pitch System: LQR');
stepinfo(0.2*cl_sys)
```

Figure 43) MATLAB code to implement LQR and prefilter control design

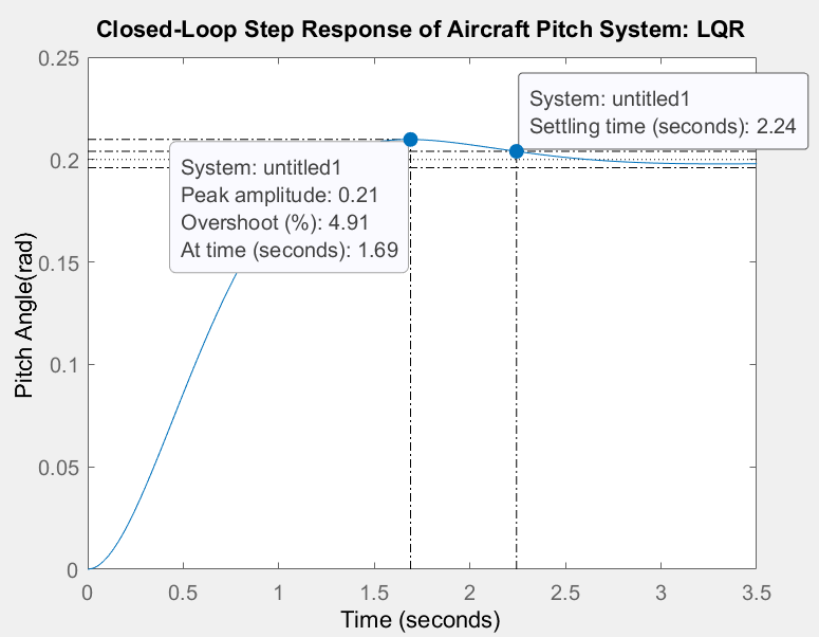


Figure 44) Closed-Loop Step response of our aircraft pitch system w/ LQR when Q = 30

```
K =

   -0.6320  144.8542    5.4772


kr =

    5.4772


ans =

  struct with fields:

       RiseTime: 0.8217
   SettlingTime: 2.2435
    SettlingMin: 0.1815
    SettlingMax: 0.2098
      Overshoot: 4.9076
     Undershoot: 0
           Peak: 0.2098
       PeakTime: 1.6892
```

Figure 45) Resulting K gain matrix, prefilter gain Kr, and step info of our system

Based off of Figure 44 and 45, our current LQR controlled system meets the overshoot and steady-state error design requirement, the settling time is not as quick as we would desire. To increase the performance of our system, we will weigh the importance of the error Q more than that of the importance of the control effort R. After trial and error, we found that a Q of 55 was sufficient enough to achieve a settling time of about 2 seconds:

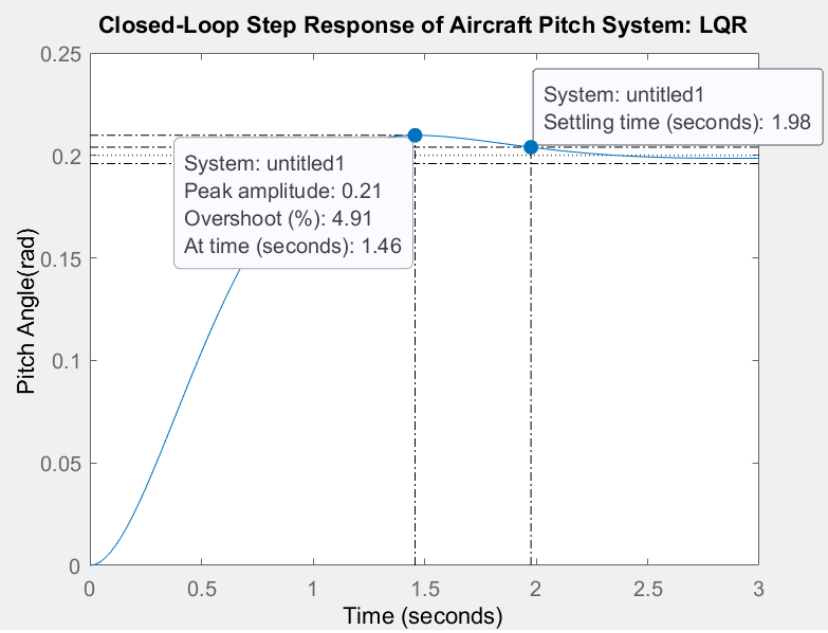


Figure 46) Closed-Loop Step response of our aircraft pitch system w/ LQR when Q = 55

```
K =

   -0.6453  174.6666    7.4162


kr =

    7.4162


ans =

  struct with fields:

       RiseTime: 0.7121
   SettlingTime: 1.9768
    SettlingMin: 0.1823
    SettlingMax: 0.2098
      Overshoot: 4.9055
     Undershoot: 0
           Peak: 0.2098
       PeakTime: 1.4573
```

Figure 47) Resulting K gain matrix, prefilter gain Kr, and step info of our system

Based on the closed-loop step response of our system in Figure 46 we notice that placing more weight on Q increases prefilter and gain matrix values shown in Figure 47, but also reduces our settling time and rise time while maintaining the same overshoot. Therefore, through LQR control design and prefilter implementation, we were able to mee the desired design requirements.

Conclusion

In this section we were able to design a compensator using the state-space methods of pole-placement and Linear Quadratic Regulation (LQR) while also implementing a prefilter to meet the achieve zero steady state error. We noted that although pole-placement is very powerful, we do not have a sense of the amount of energy needed to achieve pole-placement control. However, when using LQR control design, we were able to optimally choose our gain matrix values and successfully design a controller that met the design requirements of our aircraft pitch system.

Aircraft Pitch: Simulink Modeling

Introduction

The purpose of this section is to model the state-space model that we obtained in the first part of the lab onto MATLAB's Simulink environment. Once we obtain the desired model, we will generate the different responses to open loop and closed loop simulations. Using these responses, we will design controllers that will improve the response in order to meet our design requirements.

Objectives

Using the six nonlinear differential equations that have been linearized to longitudinal and lateral we will be able to create the mathematical representation on Simulink. Here we know that the aircraft pitch control is governed by the longitudinal dynamics and we will design the autopilot for this aircrafts pitch. For this particular model we will find the desired K values that we found in the previous section using the Linear Quadratic Regulator Method in order to obtain the controller that we want for the feedback gain.

Physical Setup and System Equations

In this section we will refer back to the state space model that was designed in the first part of the lab as shown below. This model follows the general state-space form.

$$\begin{bmatrix} \dot{\alpha} \\ \dot{q} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} -0.313 & 56.7 & 0 \\ -0.0139 & -0.426 & 0 \\ 0 & 56.7 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ q \\ \theta \end{bmatrix} + \begin{bmatrix} 0.232 \\ 0.0203 \\ 0 \end{bmatrix} [\delta]$$

$$y = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ q \\ \theta \end{bmatrix}$$

Figure 48) State-Space model used for the simulation

Implementation

We will build the model of the equations shown in the physical setup section which will follow the following model shown:

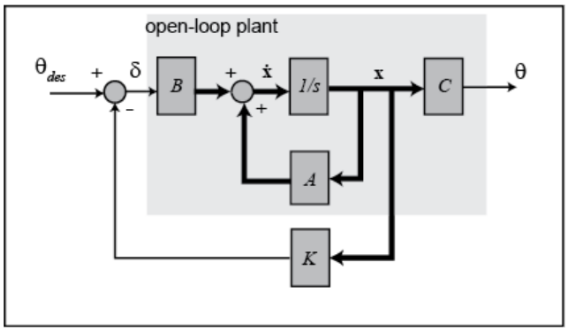


Figure 49) Physical setup of the system

We will start by opening up a blank Simulink model and we will insert a Step block from the Simulink ⟶ sources library as shown:

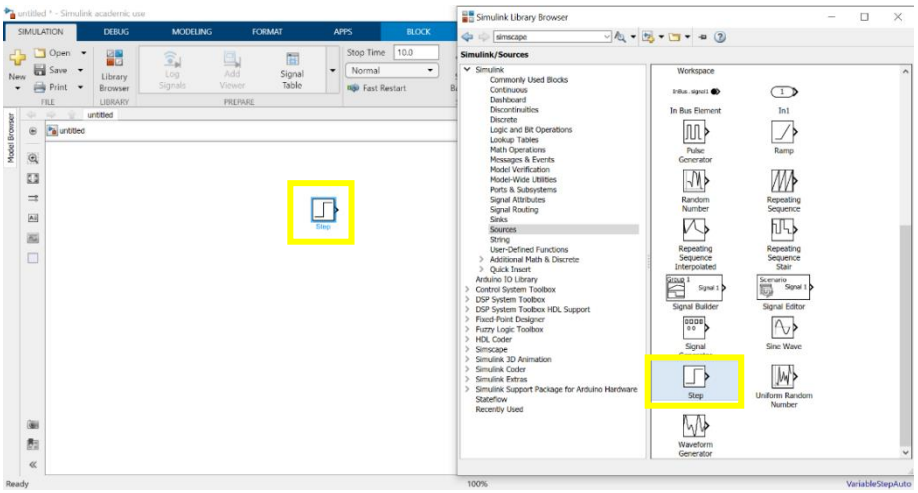


Figure 50) Step block implementation

We then open up the step block and insert the following parameters:

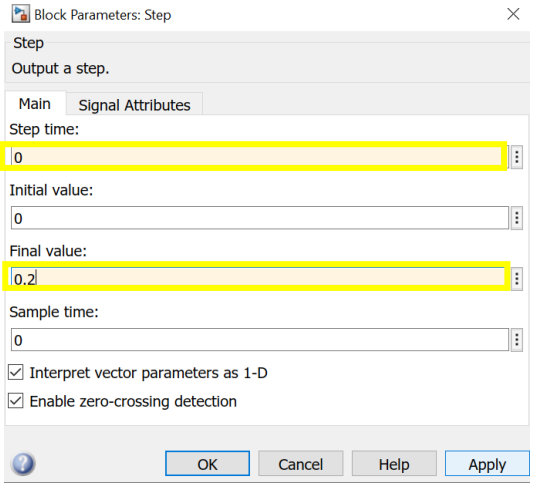


Figure 51) Step block parameters

Next, we insert a Demux block from the Simulink → Signal Routing library.

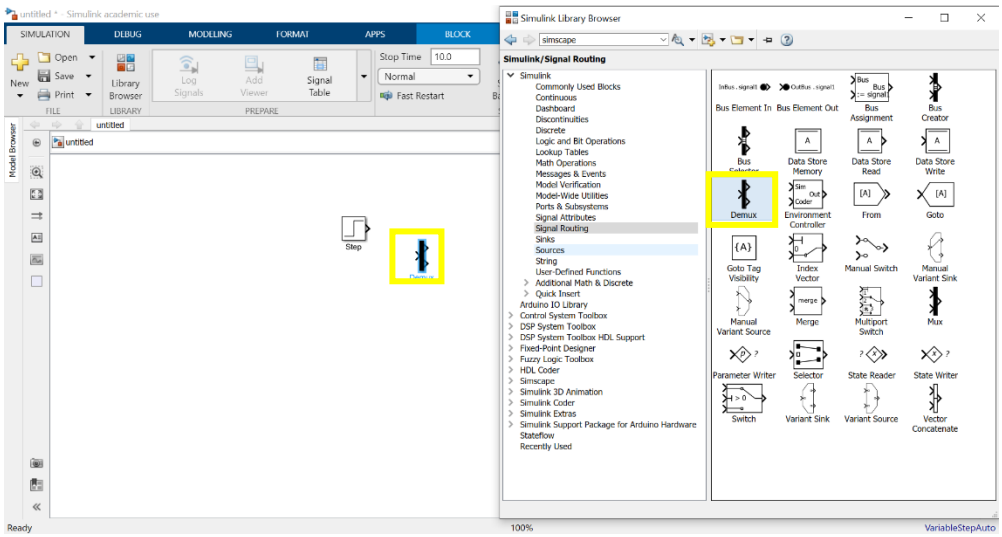


Figure 52) Demux implementation

Once we have the Demux block, we will double click the box and change the number of outputs from 2 to 3:

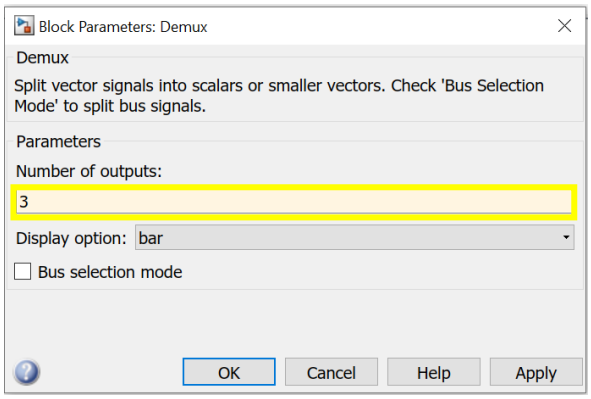


Figure 53) Demux parameters

Next, we insert a Scope block from the Simulink → Sinks library. With this block we will connect it to the lowest output of the Demux block as seen below:

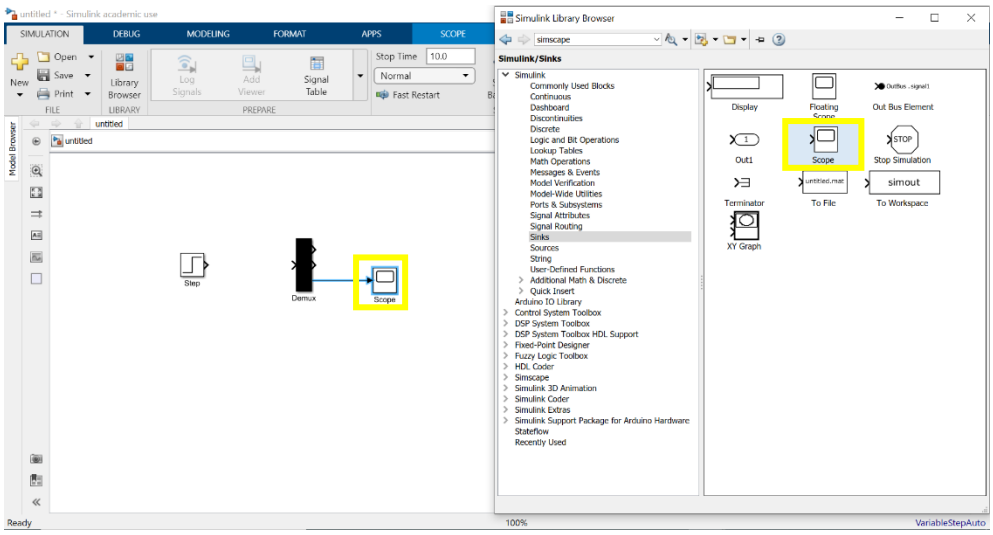


Figure 54) Scope Implementation

Next, we will add two Terminator blocks from the Simulink⟶ Sinks library and wire them to the other two outputs of the Demux block as shown below:

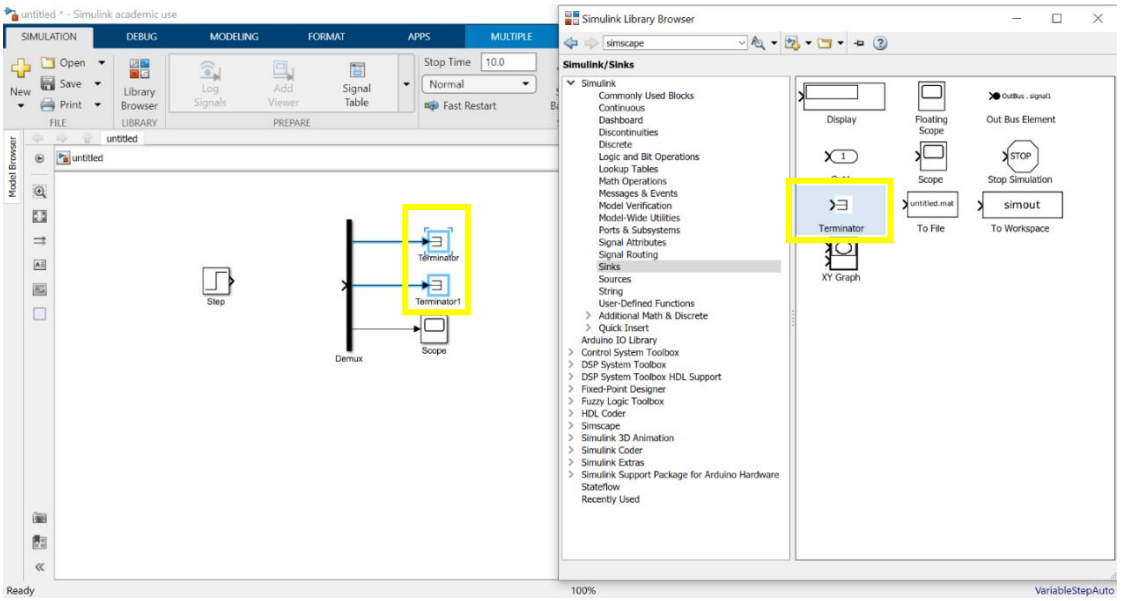


Figure 55) Terminator blocks implemented

Next, we will insert a State-Space block from the Simulink ⟶Continuous library and connect the output of the step block to the input of the State-Space block and the output of the State-Space block to the input of the Demux block as shown below:

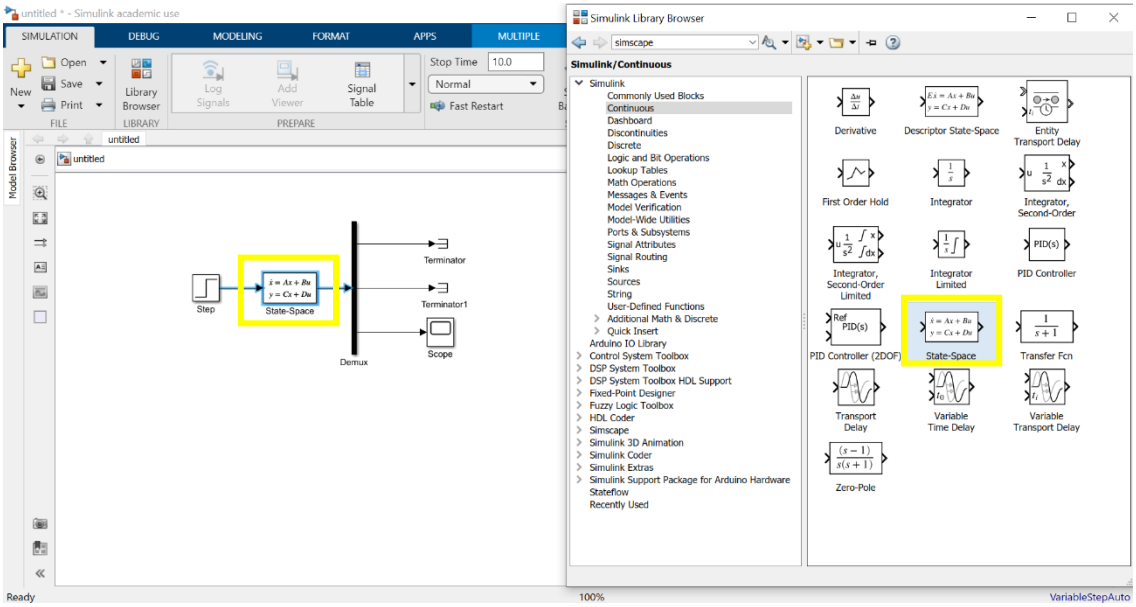


Figure 56) State-Space block implementation

We now need to input the parameters into the State-Space block, so we defined the parameters in the workspace in order to be able to use them in Simulink as shown below:

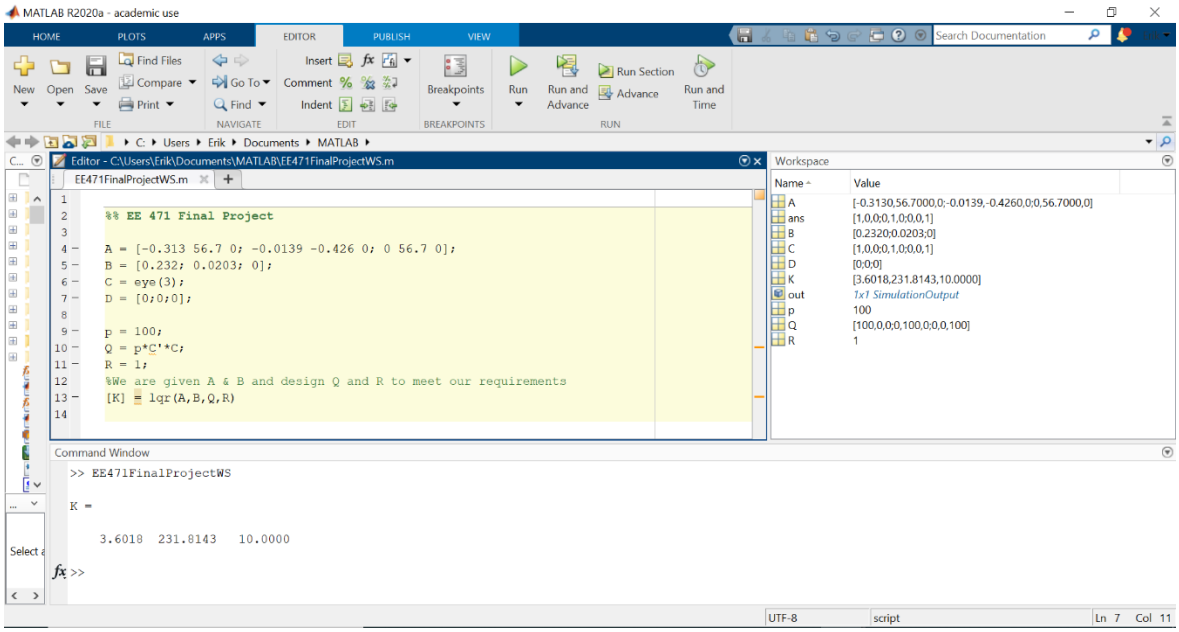


Figure 57) Creating variables in the workspace

Next, we input these values into the State-Space block by clicking the drop down menu in each parameter and choosing the appropriate variables defined in the workspace and labels:

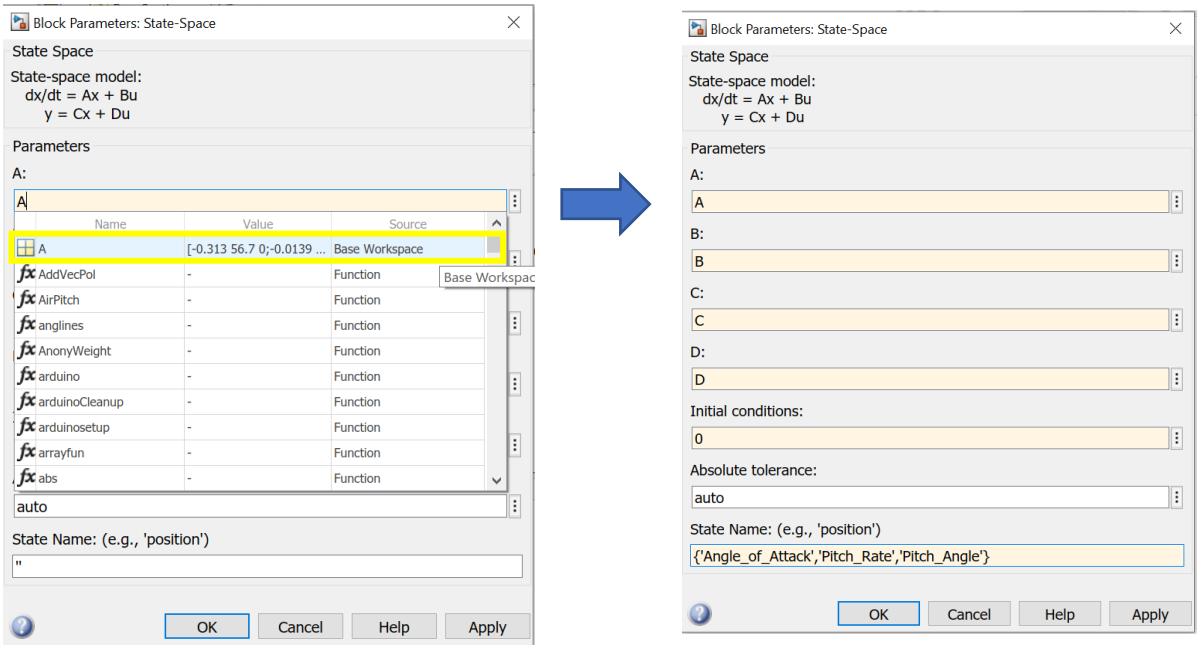


Figure 58) Defining State-Space parameters

Now the Model should appear as follows:

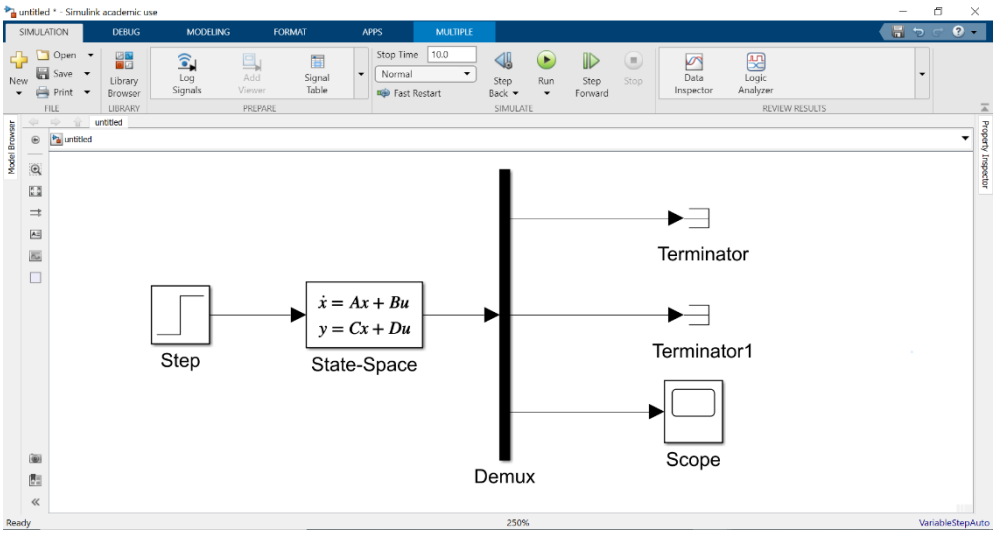


Figure 59) Open-loop model

Generating the Open-Loop and Closed-Loop Response

Once we have the model shown above, we run the code by clicking on the green button labeled "run" and once the code has been successfully executed, we double click the scope and see the following results. Since the output does not converge, we see that the system is unstable as previously noted in the first part of the lab. With that we will now use the K values from the LQR method worked out in part 1 in order to stabilize the response.

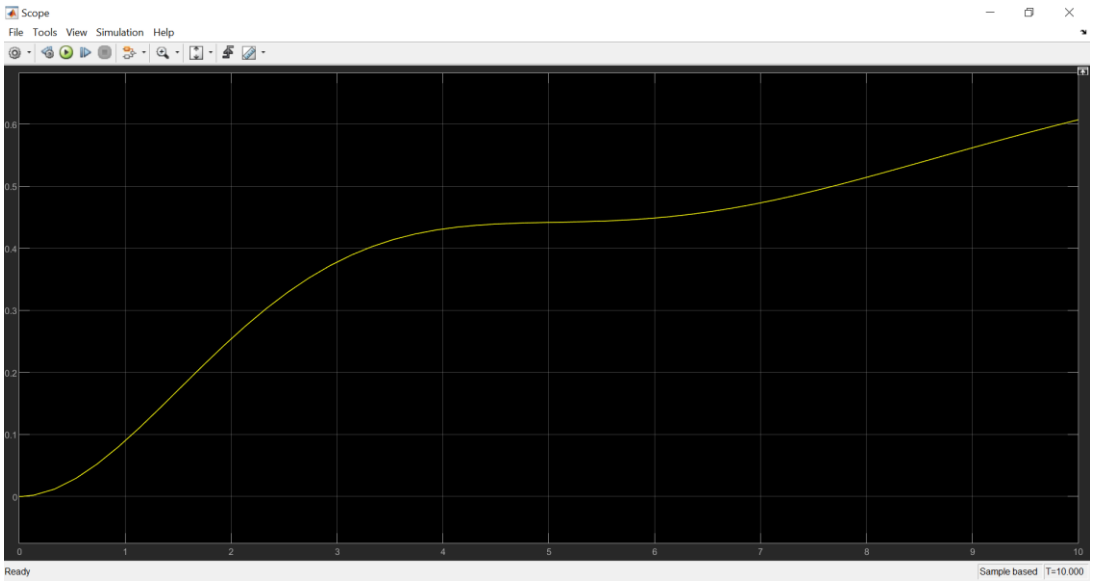


Figure 60) Open loop step response

We start by adding a sum block from the Simulink →Math library.

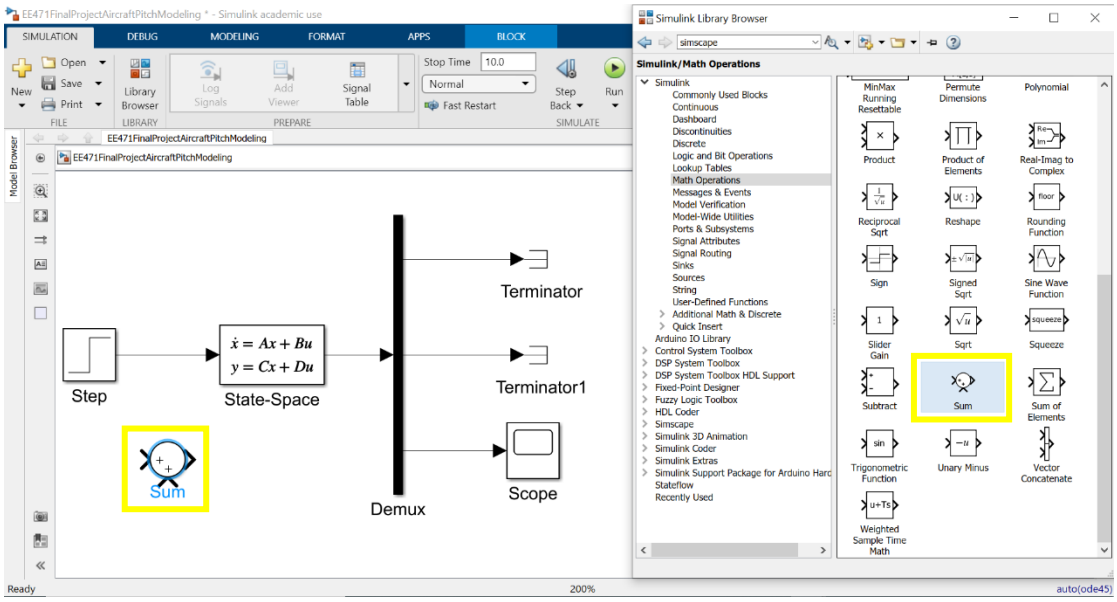


Figure 61) Sum Block implementation

Once we have the sum block added in, we connect it to the output of the Step block and input of the State-Space block and then double click the block to change the sign's from "|++" to "|+-" as shown in the figure below.

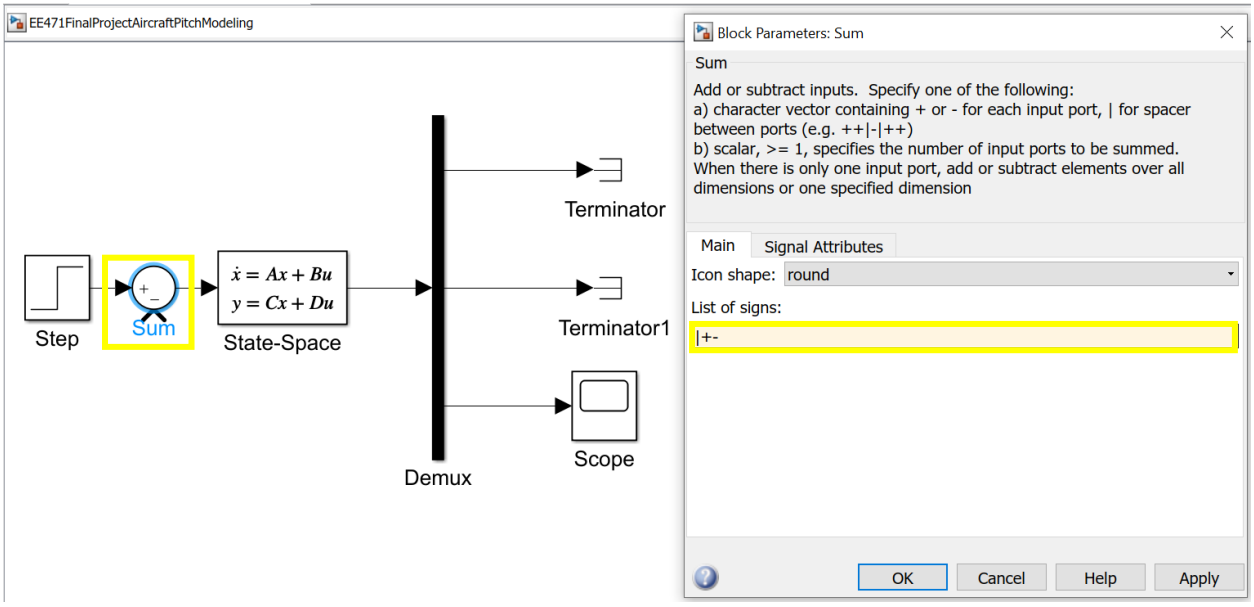


Figure 62) Sum block parameters

Next, we add a gain block from the Simulink →Math library and flip the block to create the feedback. We do this by right clicking the block and following the “Rotate & Flip” drop down menu and clicking on “Flip Block” option.

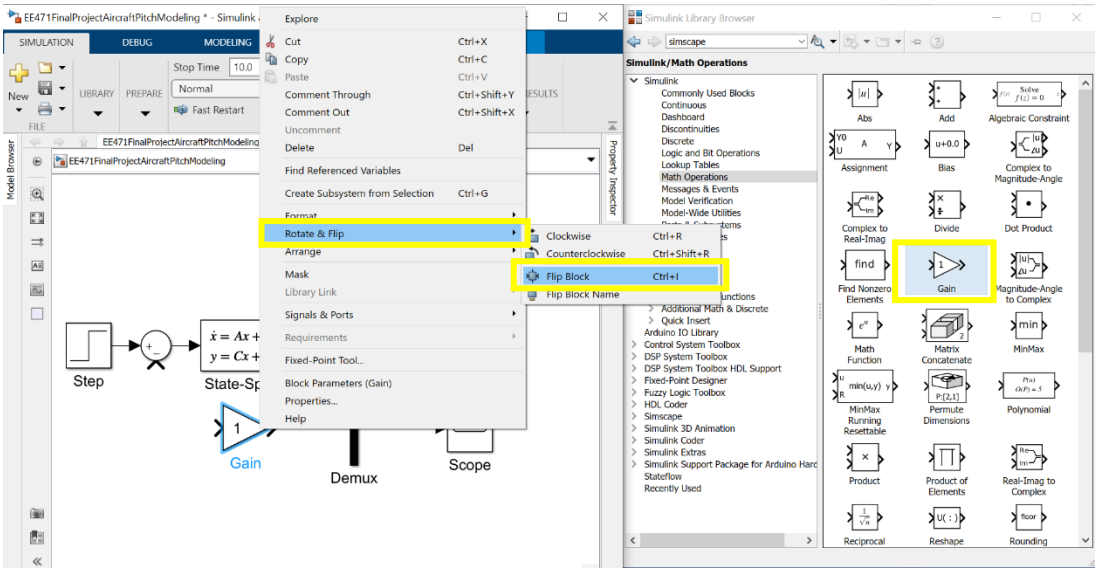


Figure 63) Gain Block implemented

Once we have the gain block facing the correct orientation, we double click on the block and add in the K values. We also click on the drop-down menu for Multiplication and select the “Matrix(K*u) option”. Next, we connect it to the output of the State-Space block and feed it into the sum block as shown below. Note that we have already defined K in the workspace and can just type in the variable name.

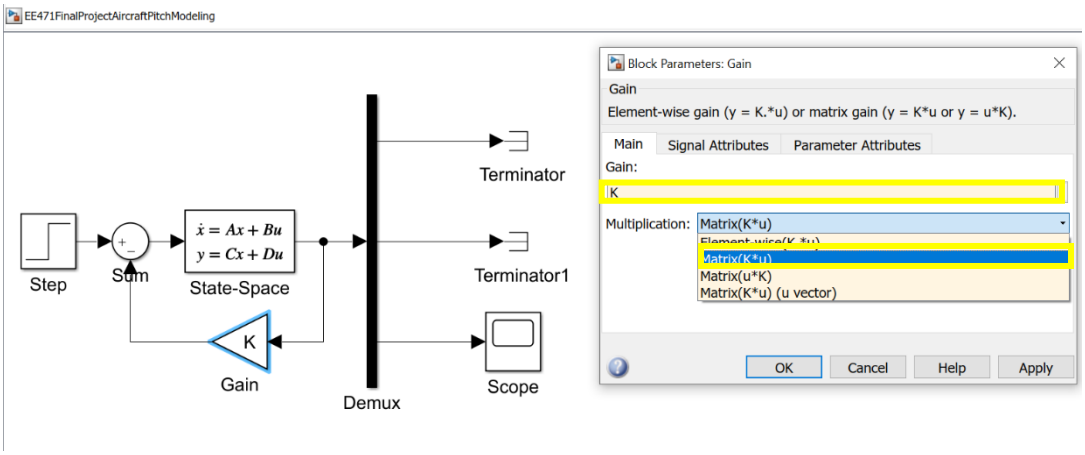


Figure 64) Gain block parameters

We will now run the simulation again and see that we get the following response.

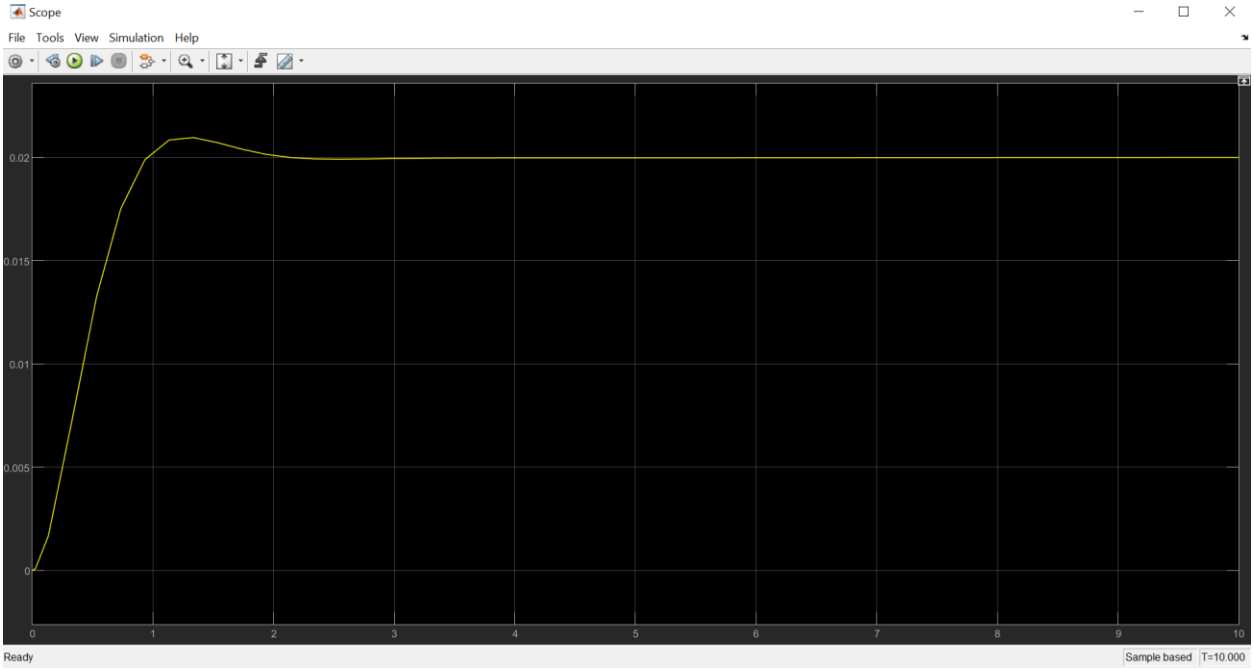


Figure 65) Closed-loop step response

Conclusion

In this section we saw that we were able to model the linearized state-space model that we derived in the first part of the lab and simulate the results onto Simulink. By using the different blocks in the libraries, we were able to create the open loop simulation to see that our system was not stable, and the response grew with time and hence needed to be stabilized. Once we created the closed loop model of the system, we were able to compare the two responses and see that the closed loop response with the calculated gain of K gave us the response that followed the 0.2 rad value that we initially were looking to follow. We obtained the K values using the Linearized Quadratic Regulator method that minimized the cost of the systems performance in order to create a more ideal situation. Overall, the results came out as expected and we were able to apply our control theory knowledge onto Simulink that can be applied to bigger projects in the future.

Aircraft Pitch: Simulink Controller Design

Introduction

In this section we will continue to work with the Simulink model in order to further compensate the response to meet our requirements. We note that the response is not within the full criteria we desired because the steady state error is not met so we will have to compensate that in order to get our system up to the specifications. We will be working with the Linear Quadratic Regulator results as well as with PID tuning.

Objectives

The Objective of this section is to test the systems output with external disturbances and we will find the best compensator for this system. First, we start by adding a precompensator to achieve the systems steady state error requirement. Once we have the precompensator, we test the systems robustness due to external disturbances. Noting that the systems precompensator and feedback gain are not enough to compensate for the introduced disturbance we have to switch the approach to a PID block. By adding the PID block into the system we see that the built-in toolbox for design will guide us to a stabilized system that we desire.

Implementation

State-Feedback Control with Pre-Compensation

Here we will begin by opening up the previous model which is now labeled "EE471FinalProjectAircraftPitchModeling.slx" and add in a gain block as shown before from the Simulink →Math library. This block will be the precompensator used to fix the steady state error

of the previous response. We will double click the block and set the gain = 7.0711. The gain block will have the Step block as an input and have its output connected to the summation block.

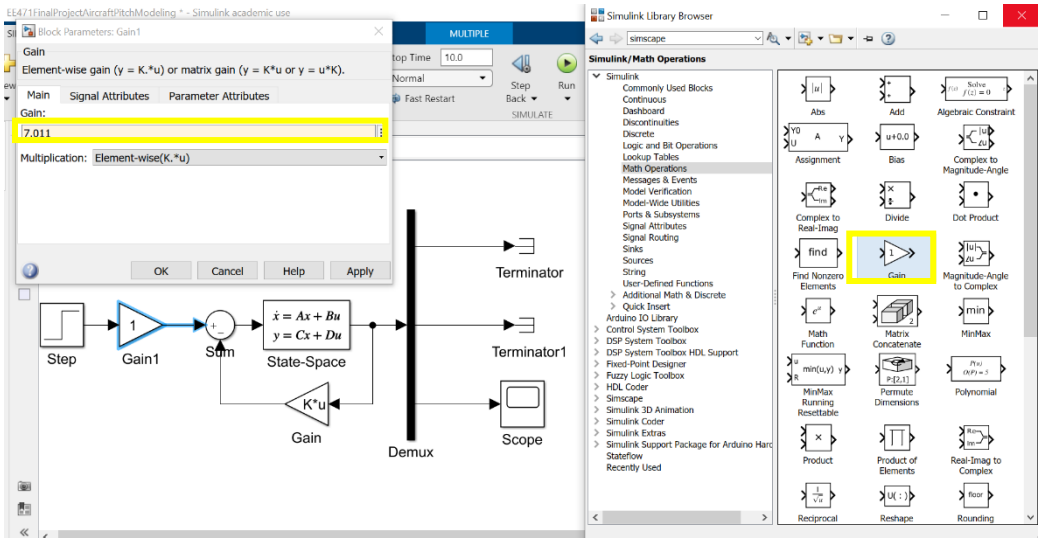


Figure 66) Precompensator Gain block implementation

Once we add the precompensation gain, we run the simulation and check the results.

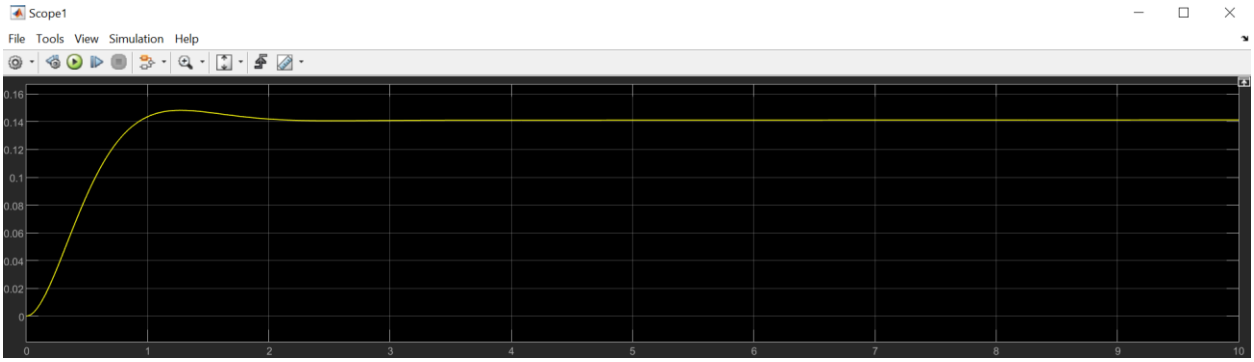


Figure 67) Step response with precompensator

Having these results, we note that the downside of precompensators is that they are implemented before the feedback loop and therefore will not take into consideration any outside disturbances.

System Robustness

Knowing that the precompensator does not take into consideration any outside disturbances, we will test the robustness of the system by adding in known disturbances into the system and analyzing how the system performs under such conditions. We begin by adding a step block to the model from the Simulink → sources library as shown:

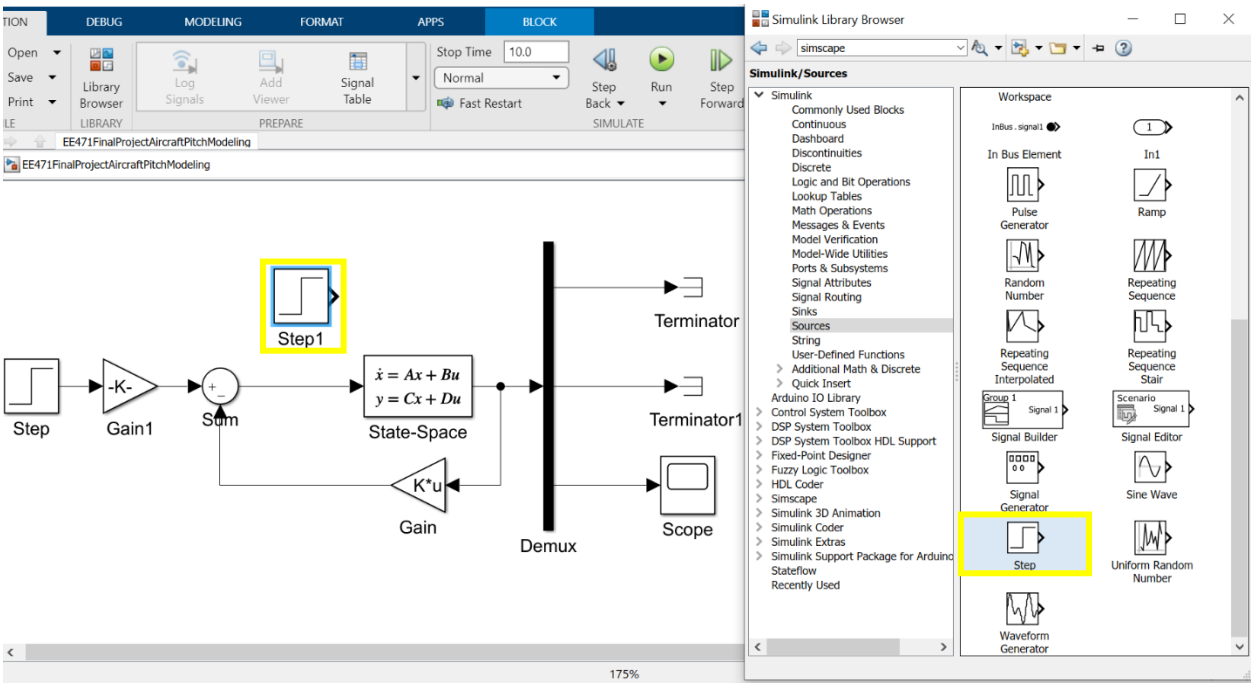


Figure 68) Disturbance block implementation

Next, we double tap the step block and input the following values, this block will be our disturbance input:

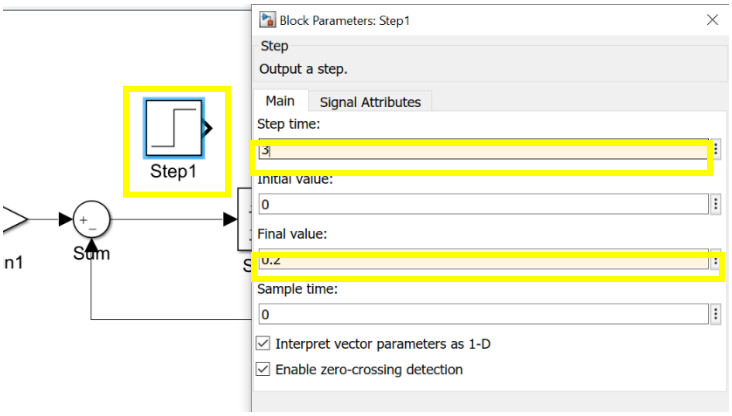


Figure 69) Disturbance block parameters

We will add in a sum block from the Simulink →Math library to add the disturbance. Once we add the sum block, we will have the output of the first sum block as an input to the second sum block and the disturbance block will be the second input and the output of the sum block will be into the state-space block as shown.

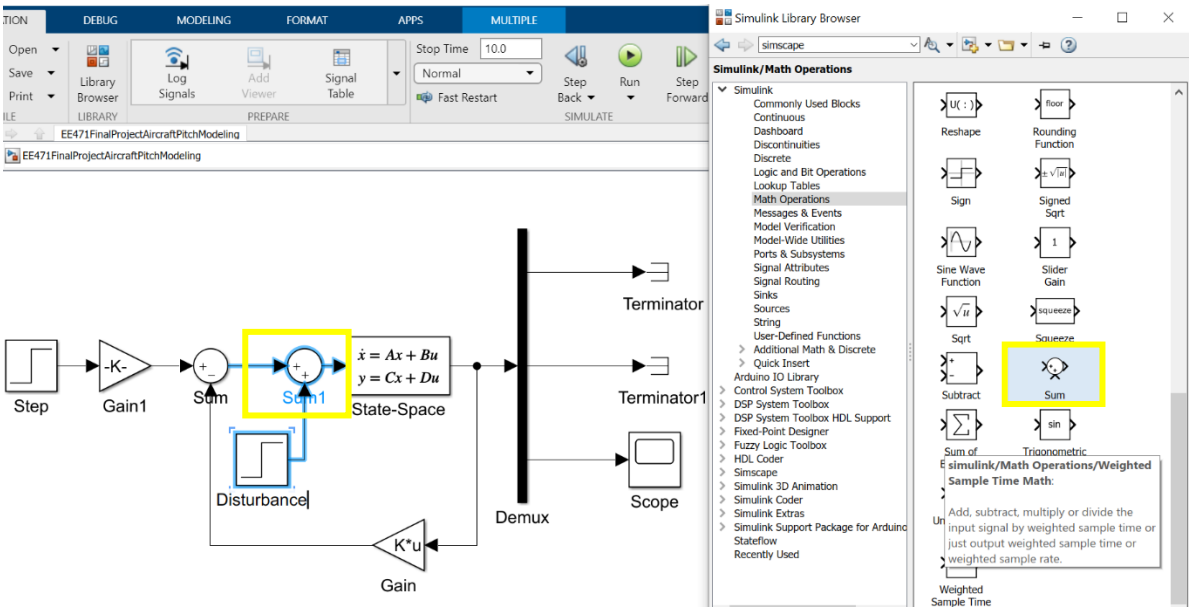


Figure 70) Disturbance block implementation

Next, we add the two wires as outputs to the scope block and labeling them as follows:

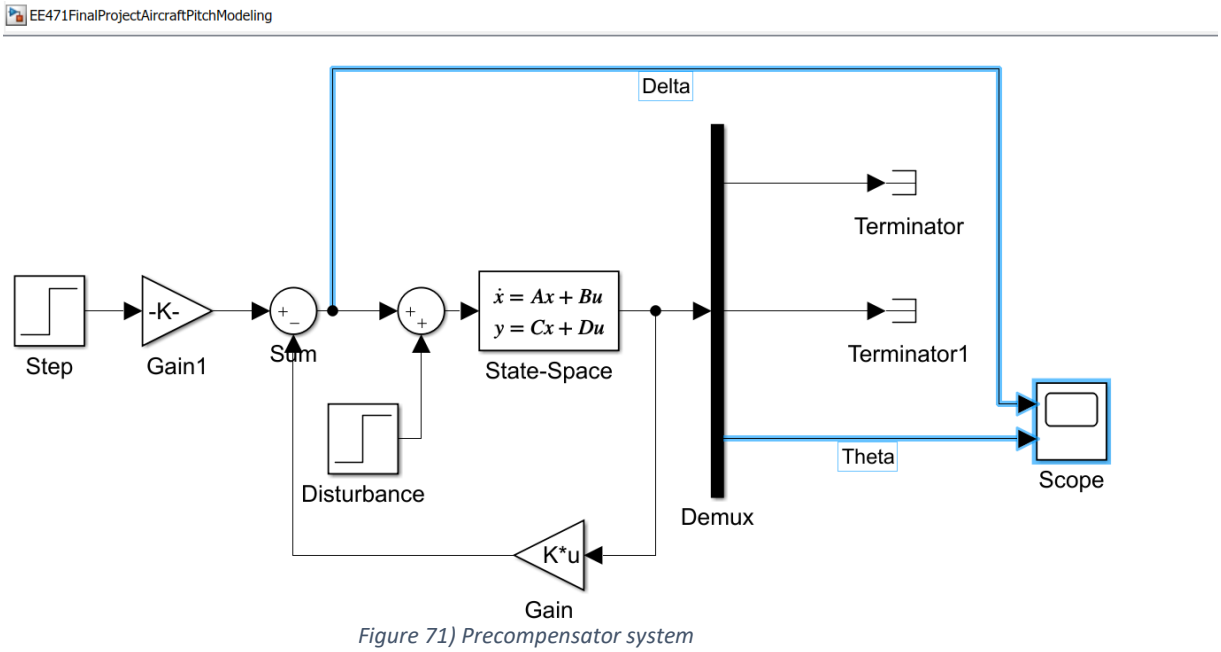


Figure 71) Precompensator system

We run the simulation and get the following results on the scope. Next, we click on the scope and press the View drop down menu and select the “Layout...” option.

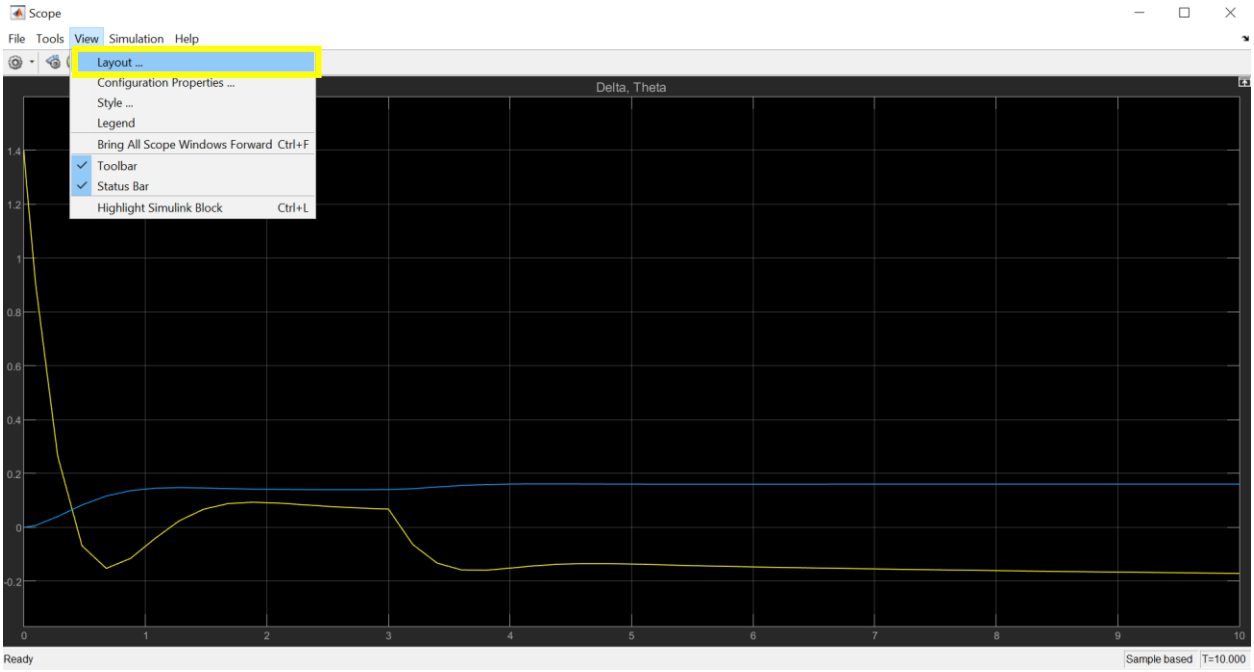


Figure 72) Precompensator system output

You will then be presented with the following option to lay out your scope and we chose the 1x2 layout in order to get the following response.

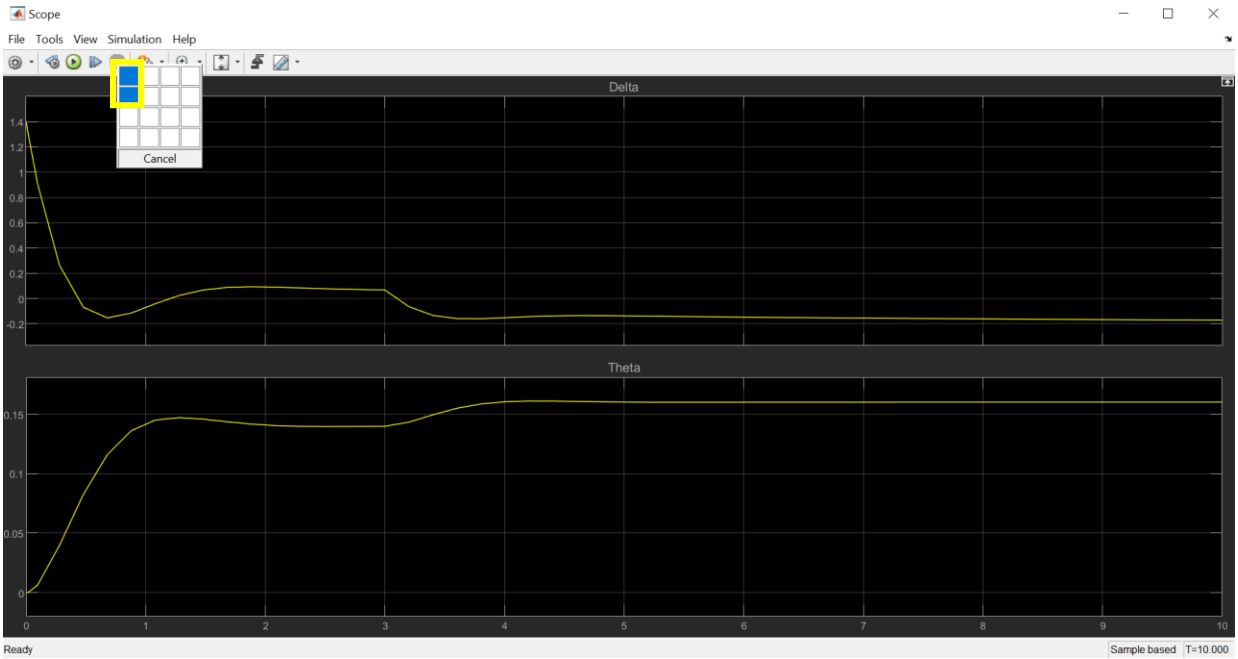


Figure 73) Precompensator output configured

From inspection, it is clear that the system was not able to recover from the effects of the disturbance as the desired final value was 0.2 radians and the disturbance at 3 seconds was not compensated for by the precompensation we have added. Now we will move on to the integral control that has proven to compensate disturbances and will be tested here accordingly.

Automated PID Tuning with Simulink

Here we will begin by noting that we will be removing the precompensation gain and feedback gain to add a more complex controller that can handle the disturbances we expect can happen. Here we must have the Simulink Control Design toolbox installed before continuing forward. Once we have the required toolbox, we insert the PID Controller block from the Simulink → Continuous library and place it between the two sum blocks as shown below. We leave Delta unchanged but now we have the feedback Theta from the output of the demux block.

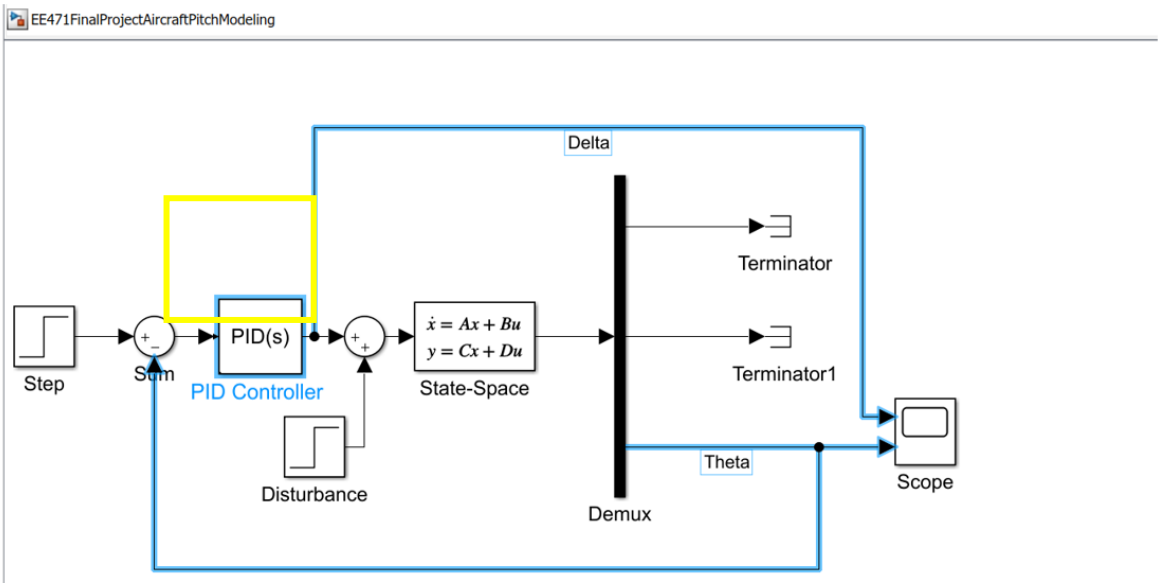


Figure 74) PID tuning system

Next, we double click on the PID block and click on the “Tune…” option in order to implement MATLAB’s automatic PID tuning:

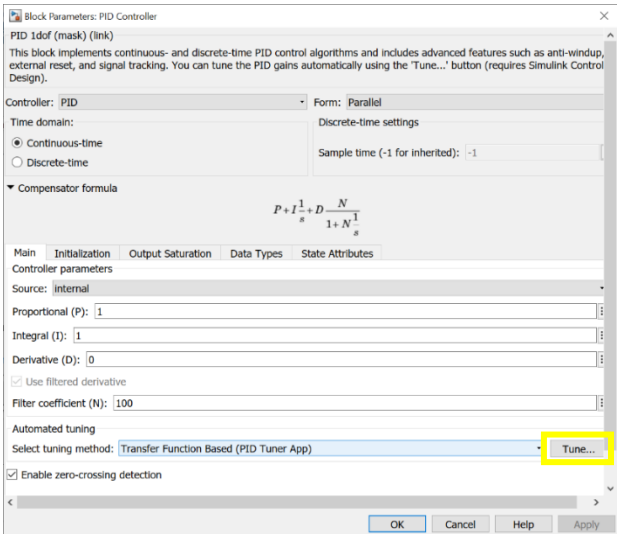


Figure 75) PID block parameters

Knowing that our model is already linearized, we will obtain the values though the automatic tuning option in Simulink’s Control system Design add on and have the following “step plot: Reference tracking” window appear after we click on the tuning option. We see that the initial response is the dotted line with oscillation but after the tuning, the response of the system is a lot smoother and stable which would be ideal for our system.

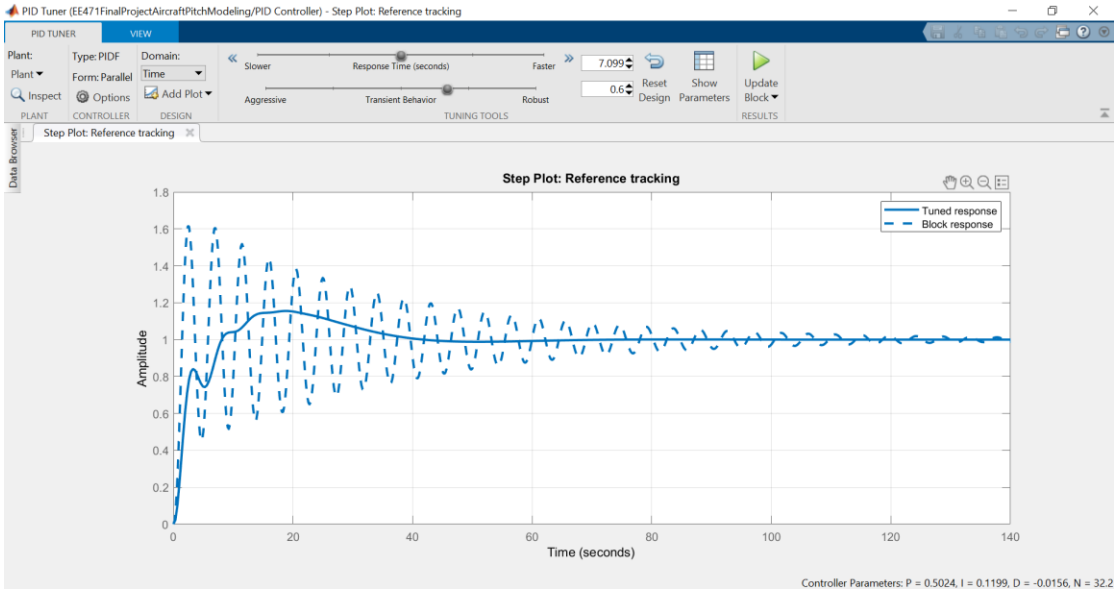


Figure 76) PID step output tuning window

Next, we select the “update block” option on the Results section to apply the changes onto our Simulink model whenever we change anything:

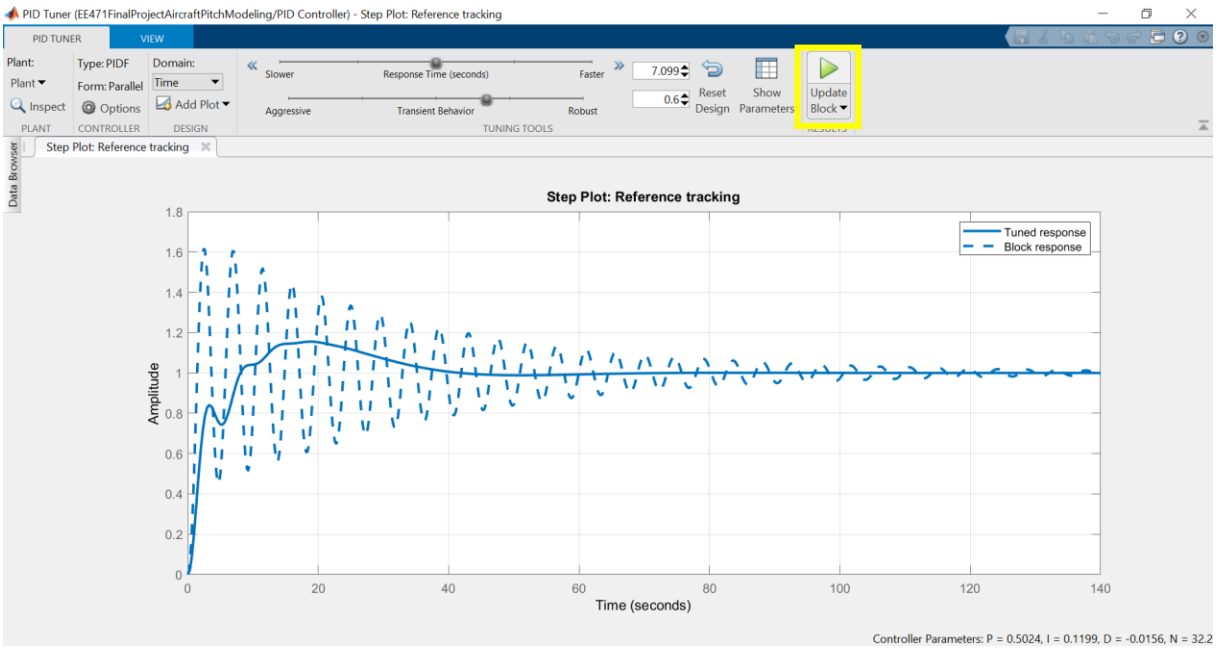


Figure 77) PID tuning update

Next, we click on the “Options” Drop down menu and uncheck the “Show block Response” option to get the following response as only the updated response.

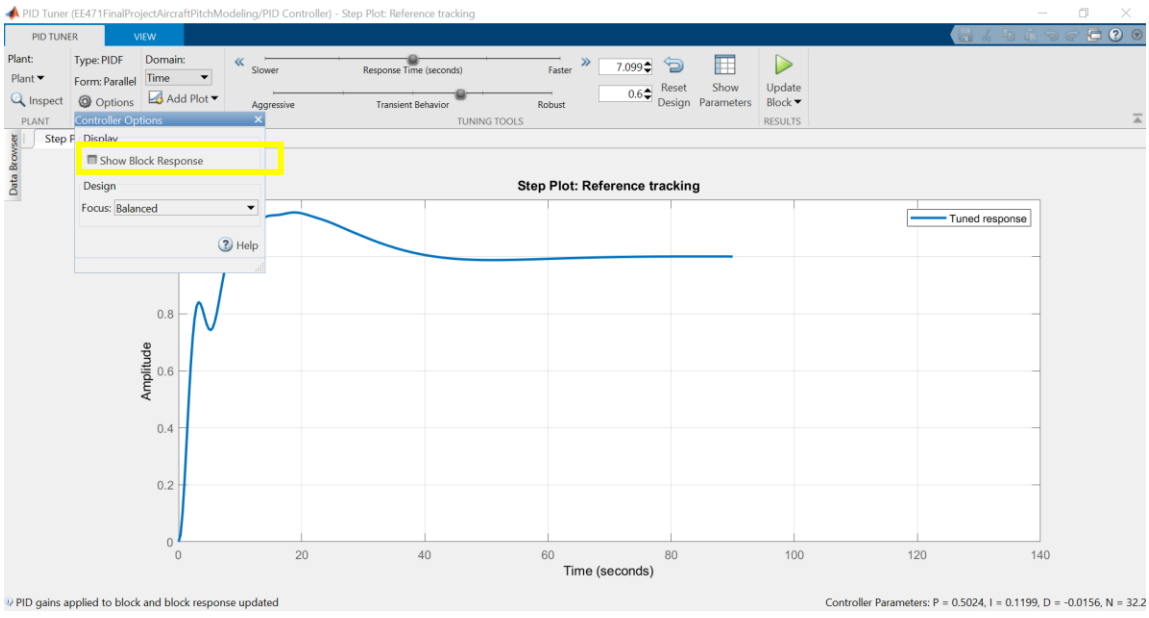


Figure 78) Navigating through appropriate response

Now changing the 7.099 response time in the “Tuning tools” to 0.089 to get the following figure.

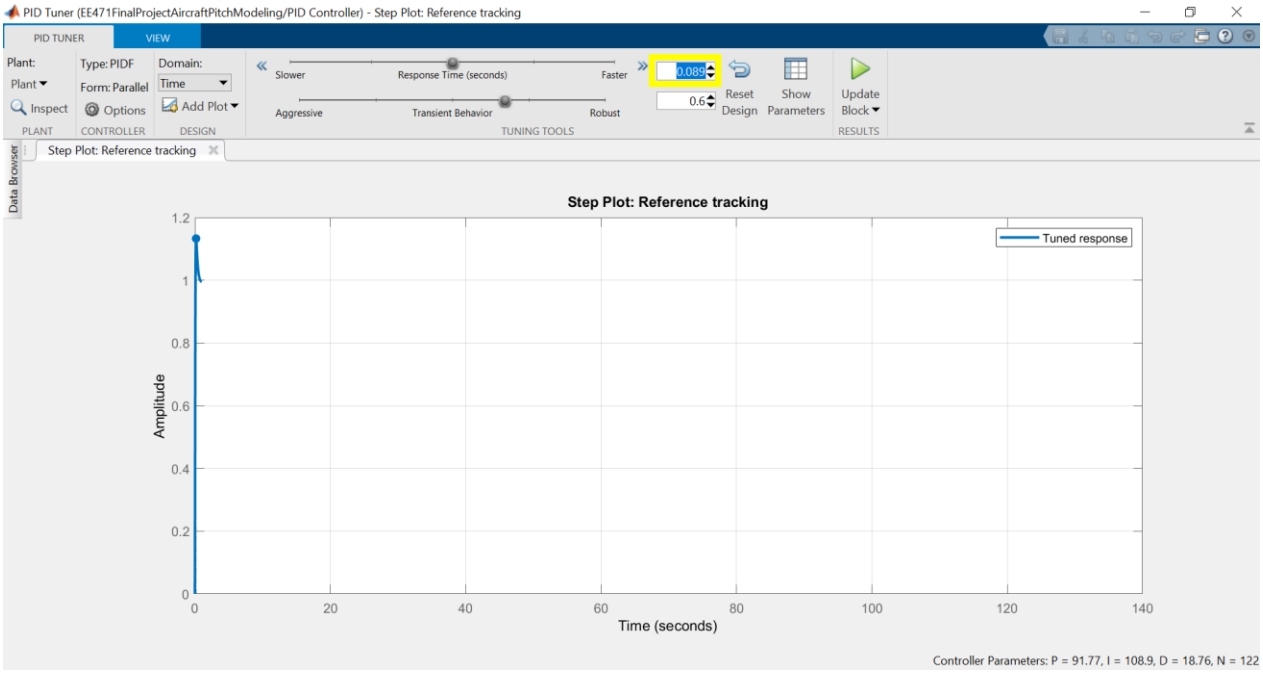


Figure 79) PID “Tuning tools”

Once we have the above figure, we double click the x-axis limits and change the upper limit to 1.

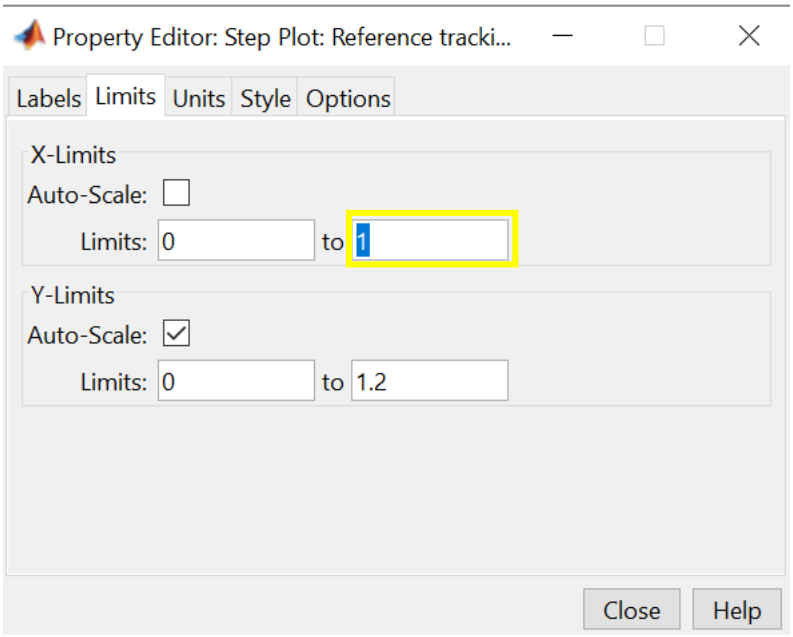


Figure 80) PID x-axis limit change

Then, we get the following response shown below:

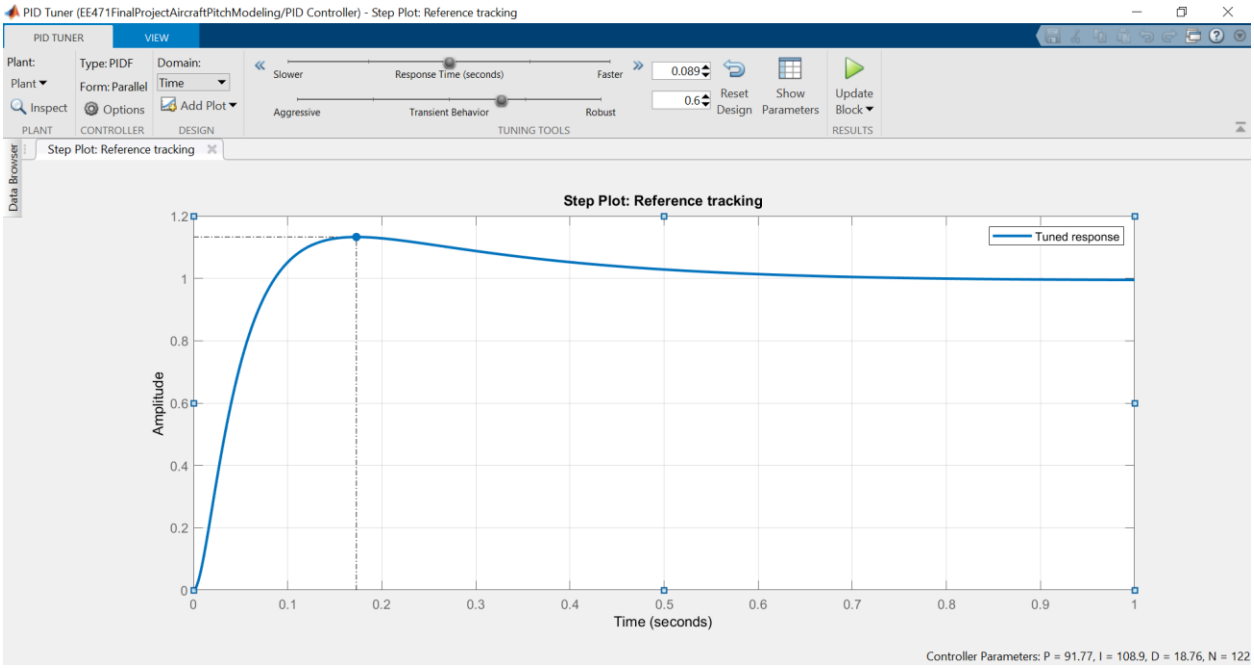


Figure 81) PID Tuned Response

Once we have the resulting response, we right click the graph and hoover over the Characteristics option so that we could display the “Peak Response” and “Settling Time”.

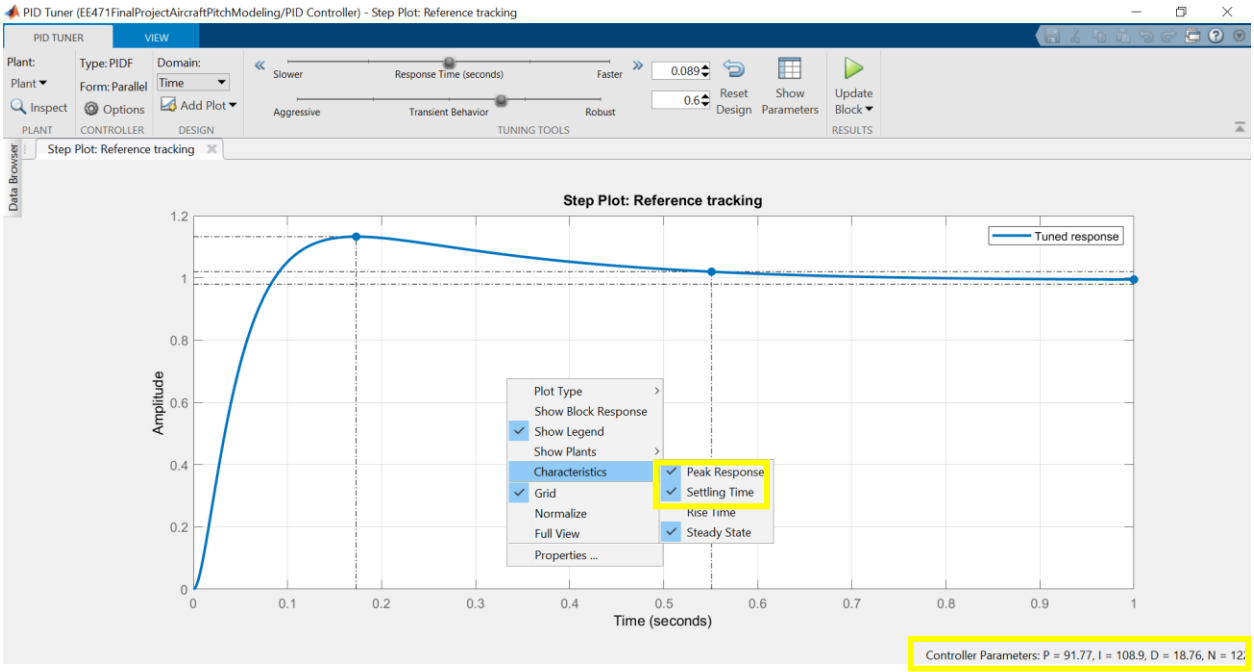


Figure 82) PID Tuned Response with characteristics

We see in the bottom right-hand corner of the figure above that the PID constants are shown. We once again click on the Update block option in the top right-hand corner and check the PID block characteristics to see that the PID has been updated as shown below.

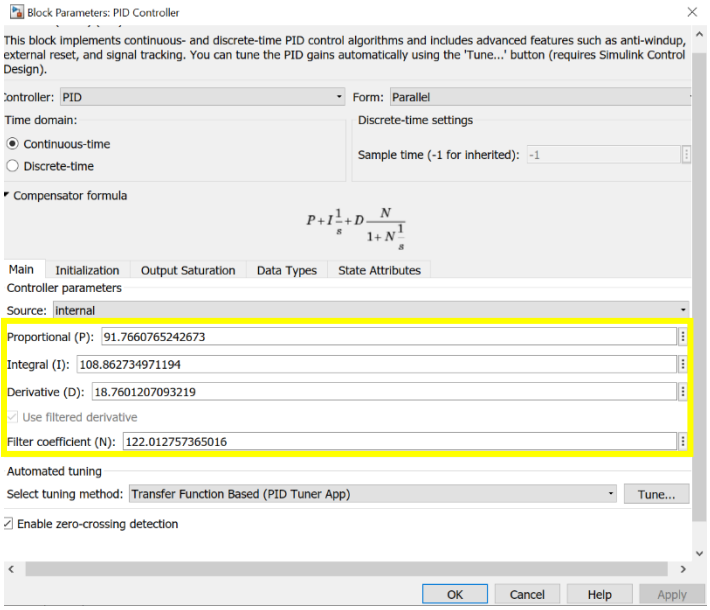


Figure 83) PID updated parameters

We now run our system's simulation in Simulink too see how the newly tuned PID coefficients change the response:

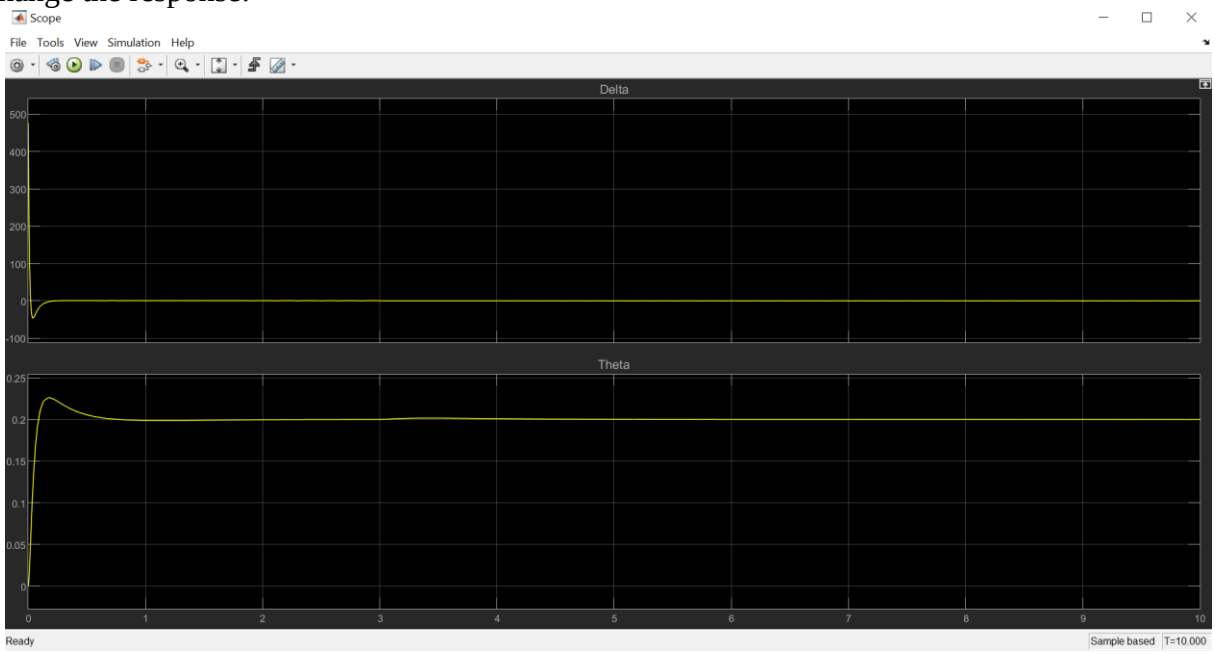


Figure 84) PID tuned Response

In the figure above we note that the disturbance is now accounted for and the system was able to compensate for the outside disturbance that was added. However, we see that this new integrated controller used up more energy to be able to compensate for the disturbance and that translates over to higher costs.

Next, we are to consider the elevator angle (δ) by further tuning the PID Block. We do this by double clicking the PID block once again and going to the "Output Saturation" section to click on the "Limit output" check bock. Once we check the box, we set the limits as follows:

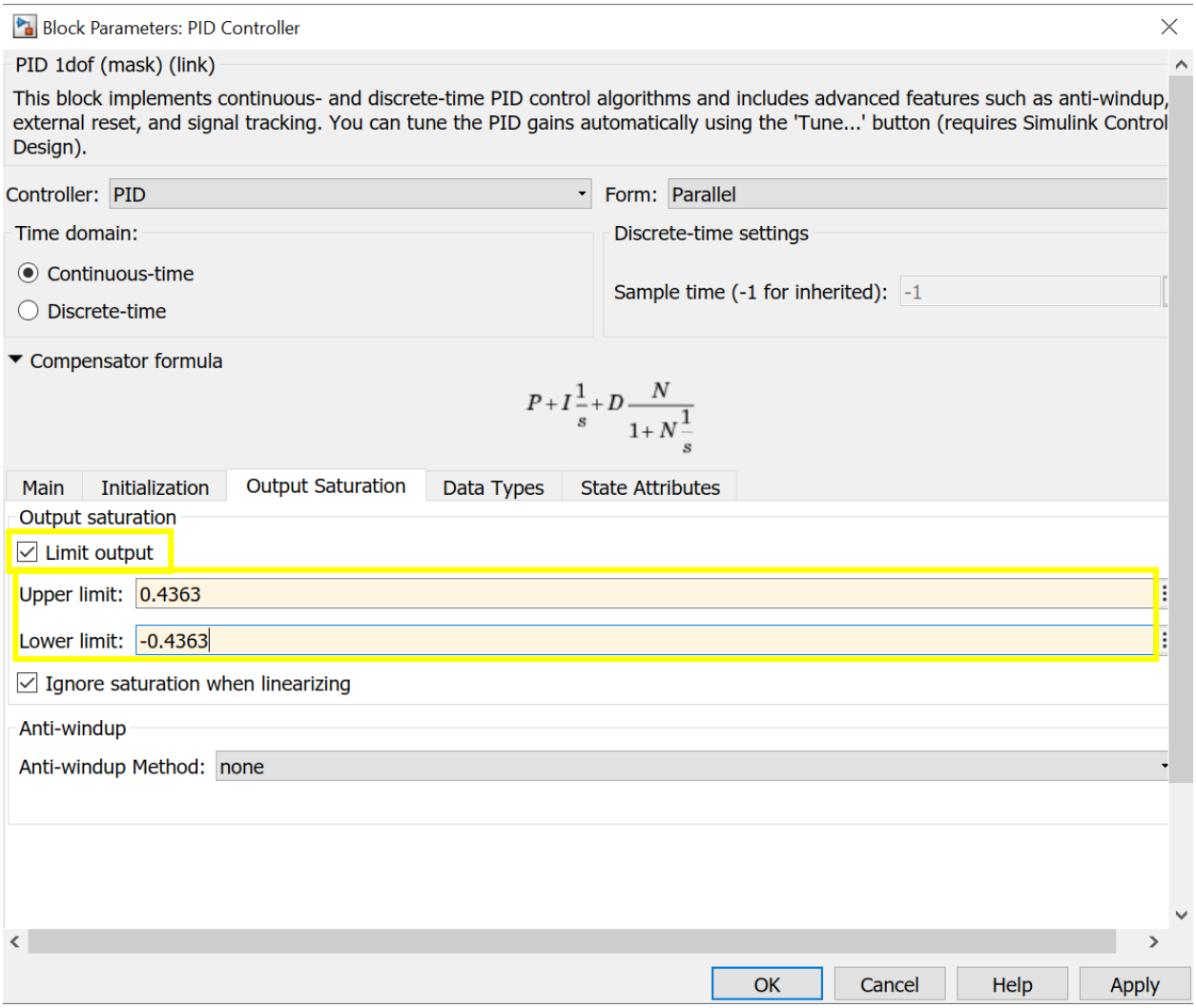


Figure 85) PID Saturation limits

Next, we must further modify the PID tuning block because as time goes on and the disturbance decreases the integrator builds up the error and falls into windup which is not good for the systems speed or performance. In order to combat this effect, we use the Clamp option in the PID tuning as an Anti-windup method to keep the system running at peak performance.

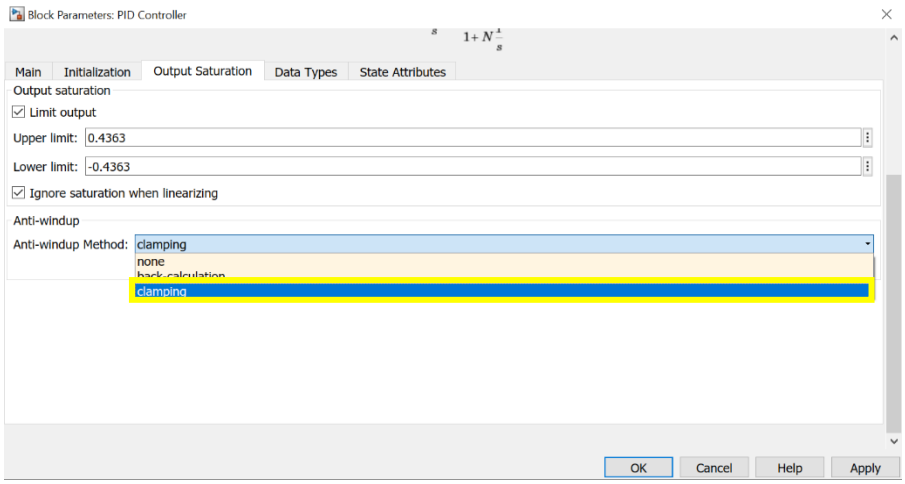


Figure 86) PID Clamping for Anti-windup Method

Now we run the model once more and check the scope to get the following results.

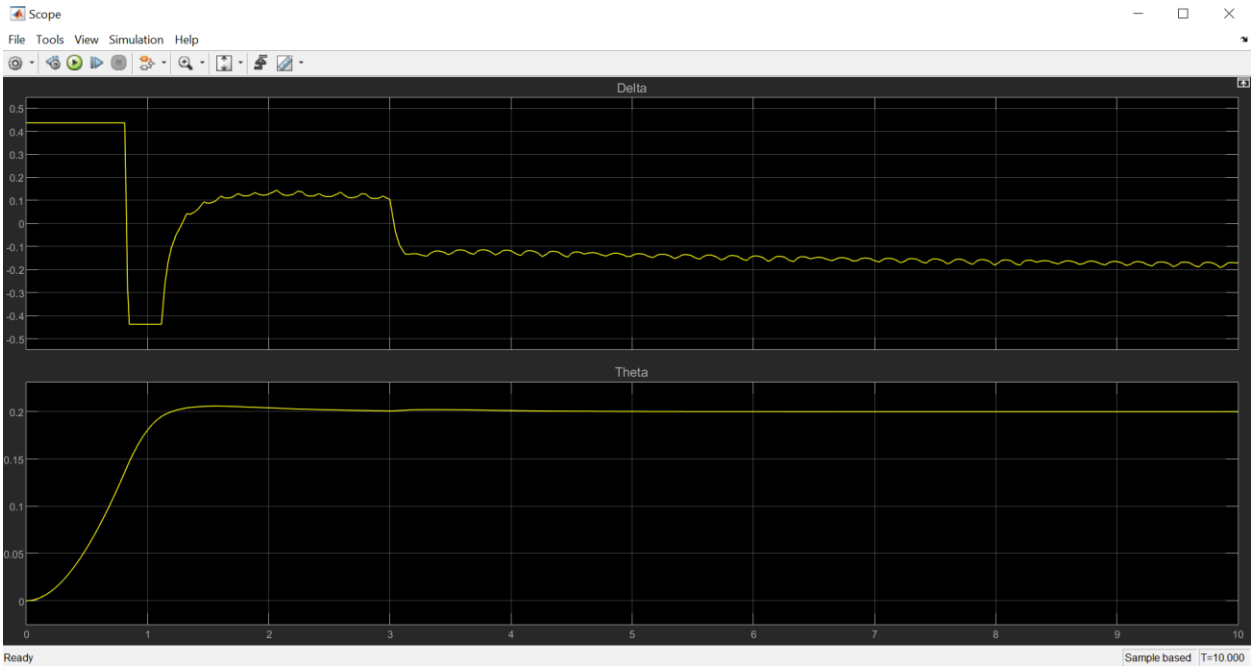


Figure 87) System final step response

Looking at the figure above we see that the system is now working under the conditions that we expected to end up with. The gains used in the PID along with the Anti-winding method have kept the systems efforts within the limits set for saturation and the step response is still meeting the requirements with a lower overshoot percentage this time.

Conclusion

After several tuning methods, we were able to see that different compensators have different functionalities. As we explored the different tuning methods, we saw that the precompensation gain was more cost efficient but did not consider the external disturbances. However, the PID tuning method was able to correct the outside disturbance but was used with higher energy which creates a more costly application to the design. These are the trade-offs that we will be dealing with in the future as we have to deal with problems where we will have the luxury of using more costly compensators or there will be times where we have to work within a projects budget. In the second part it was crucial to install the Simulink Control Design toolbox add-on which was able to use different tuning methods for the integral PID block. Here we noticed that the automatic tuning method used did not penalize the cost of the compensator because the energy used was significantly greater in comparison to the energy used by the precompensator gain. In the end, we observed that the step response from the PID automatic tuning did meet the requirements of the system we proposed and the PID block also accomplished the response with a faster settling time and lower percent overshoot then any of the previous compensation methods discussed which resulted in a successful simulation

Aircraft Pitch: MATLAB Animation Using State-Space

Introduction

The purpose of this experiment is to model our aircraft pitch system using state-space and utilize to design a visual aircraft animation that models the movement based on the pitch angle and weight factor.

Objective

We assume the aircraft is in steady cruise at constant altitude and velocity with lift, weight, thrust, and drag balancing out. Using state-space, the aircraft pitch system is represented as a set of linearized longitudinal and lateral equations. Using various MATLAB functions such as “lqr”, “rscale”, “ss”, and “lsim” we can animate the transient system behavior of the aircraft depending on the desired pitch and weight factor.

Problem

The user will be prompted to enter in the desired pitch angle (in radians) and weight factor, and dependent on those values we can determine if the following system requirements can be met:

- Settling time less than 5 seconds
- Rise time less than 1 seconds
- Overshoot less than 5%
- Steady-State error less than 2%

MATLAB Code Walkthrough

The aircraft pitch animation code begins with a prompt asking the user to enter in the desired pitch angle and weight factor, where the pitch angle value should range from -1 to 1 radians and the weight factor value should range from 1 to 100.

```
clear, clc, close all

prompt = {'Enter Pitch Angle in radians (-1 < Pitch Angle < 1)','Enter Weight Factor (1 < Weight
Factor < 100)'};
dis = 'User Input';
dims = [1 55];
r = inputdlg(prompt,dis,dims);
PA = str2double(r(1));                                  % pitch angle in radians
WF = str2double(r(2));                                  % pitch angle weight factor
```

Once the user has inputted the values, there is an "*if*" loop to confirm that those values are within the specified ranges, and if not, the aircraft pitch animation code will not run, and the following error message will be displayed:

Error: values for pitch angle and weight factor must be within the specified ranges

Once the user inputs acceptable values the code will continue to run.

```
if (PA >= -1) && (PA <= 1) && (WF >= 1) && (WF <= 100)
```

```
else
    disp('Error: values for pitch angle and weight factor must be within the specified ranges')
end
```

Next, the state space representation was derived to give us the system matrix, "*A*", input matrix, "*B*", output matrix, "*C*", and feedforward matrix, "*D*". The "*y0*" variable is defined early and will be used later on with the "*lsim*" function. After the state space was defined, we moved on to adding precompensation.

```
y0 = [0 0 0];
A = [-0.313 56.7 0; -0.0139 -0.426 0; 0 56.7 0];    % System Matrix
B = [0.232; 0.0203; 0];                             % Input Matrix
C = [0 0 1];                                        % Output Matrix
D = [0];
```

The pre-compensator was created by defining "*R*" as 1 and "*Q*" as a matrix the size of "*A*" with the user desired weight factor, "*WF*".

```
R = 1;
Q = [0 0 0;0 0 0;0 0 WF];
```

In order to find the best gain for the system, variables "*A*", "*B*", "*Q*", and "*R*" were plugged into the "*lqr*" function to output as the gain, "*k*". That gain was then used to find "*Nbar*" with the "*rscale*" function. The function "*rscale*" is used to calculate a scaling factor for the full-state system in order to decrease the final steady state error.

```
[k]= lqr(A,B,Q,R);
Nbar = rscale(A,B,C,D,k);
cl_sys = ss(A-B*k,B*Nbar,C,D);
```

The time variable, "*t*", was set to run for 6 seconds, and by using the user inputted pitch angle, "*PA*" we plugged the pervious variables into the "*lsim*" function. The "*lsim*" function simulated the response of the dynamic system and allowed us to find the output angle in radians, "*theta*". The step response of the pitch angle depending on the user desired pitch angle and weight factor was then plotted using "*theta*" and "*t*" variables. Due to the robust system design the steady state error should be negligible (about 0.001) while the rise time, settling time, and percent overshoot are all dependent on the weight factor.

```
t = 0:0.01:6;
PA = PA*ones(size(t));
[y,x] = lsim(A-B*k,B*Nbar,C,D,PA,t,y0);

theta = x(:,3);
theta = theta';
figure()
plot(t,theta)
title('Step Response of Pitch Angle')
ylabel('Pitch Angle (Radians)')
xlabel('Time (Seconds)')
```

After the system was designed, we moved on to creating the animation in order to visualize the systems physical behavior. We set the animation figure to fill the monitor screen with an "*xlim*" and "*ylim*" of -1 and 1 where the y axis represents the pitch angle in radians.

```
figure('units','normalized','outerposition',[0 0 1 1])
hold on
xlim([-1 1]);      % visual plot axis size
ylim([-1 1]);
ylabel('Pitch Angle (Radians)')
```

The following variables were defined to model the body length, tail 1&2 length, nose length, and body width for the aircraft. Several different angles also had to be calculated in order to consider the line orientation for creating the visual model ("*theta*" alone would work for the horizontal aircraft body line centered around zero, but in order to make the visual more realistic extra values were required).

```
L1 = 0.4;   % body length
L2 = 0.35;  % tail 1 length
L3 = 0.1;   % tail 2 length
L4 = 0.08;  % nose length
L5 = 0.03;  % body width

angle1 = atan(2*L5/L1);
angle2 = atan(2*L5/L2);
angle3 = atan(2*L5/L3);
```

Next, each of the line positions on the visual figure and movement were defined, with each individual line needing an x1,x2, y1, and y3 component. Note that “theta” was included in each command because that is the system output and how we model the movement. There were a total of 7 separate lines to make up the aircraft visual: top of the body, bottom of the body, top of the nose, bottom of the nose, back connection, front of the tail, and back of the tail.

```
% top of body
x1_Body1 = L1*cos(angle1+theta); x2_Body1 = -L1*cos(angle1-theta);
y1_Body1 = L1*sin(angle1+theta); y2_Body1 = L1*sin(angle1-theta);

% bottom of body
x1_Body2 = -L1*cos(pi-angle1+theta); x2_Body2 = L1*cos(pi-angle1-theta);
y1_Body2 = -L1*sin(pi-angle1+theta); y2_Body2 = -L1*sin(pi-angle1-theta);

% top of nose
x1_Nose1 = (L1+L4)*cos(theta); x2_Nose1 = L1*cos(angle1+theta);
y1_Nose1 = (L1+L4)*sin(theta); y2_Nose1 = L1*sin(angle1+theta);

% bottom of nose
x1_Nose2 = (L1+L4)*cos(theta); x2_Nose2 = -L1*cos(pi-angle1+theta);
y1_Nose2 = (L1+L4)*sin(theta); y2_Nose2 = -L1*sin(pi-angle1+theta);

% back connection
x1_Back = -L1*cos(angle1-theta); x2_Back = L1*cos(pi-angle1-theta);
y1_Back = L1*sin(angle1-theta); y2_Back = -L1*sin(pi-angle1-theta);

% front of tail
x1_Tail1 = -L3*cos(angle3-theta); x2_Tail1 = -L2*cos(angle2-theta) - (2*L3)*sin(theta);
y1_Tail1 = L3*sin(angle3-theta); y2_Tail1 = L2*sin(angle2-theta) + (2*L3)*cos(theta);

% back of tail
x1_Tail2 = -L2*cos(angle2-theta); x2_Tail2 = -L2*cos(angle2-theta) - (2*L3)*sin(theta);
y1_Tail2 = L2*sin(angle2-theta); y2_Tail2 = L2*sin(angle2-theta) + (2*L3)*cos(theta);
```

Each of the 7 aircraft lines (and one stagnant line to represent the 0-radian horizon) were then set to be plot using the x1, x2, y1, and y2 data with a “*for*” loop to model the animation movement for the specified time of “*t*”. Once everything was set for the aircraft animation, the “*drawnow*” function was used to combine all of the code to display the final product.

```
% Aircraft animation visual lines
plot([-1 1], [0 0])    % Pitch angle of 0
V1 = plot([x1_Body1(1) x2_Body1(1)], [y1_Body1(1) y2_Body1(1)], 'Color','k','LineWidth',2);
V2 = plot([x1_Body2(1) x2_Body2(1)], [y1_Body2(1) y2_Body2(1)], 'Color','k','LineWidth',2);
V3 = plot([x1_Nose1(1) x2_Nose1(1)], [y1_Nose1(1) y2_Nose1(1)], 'Color','k','LineWidth',2);
V4 = plot([x1_Nose2(1) x2_Nose2(1)], [y1_Nose2(1) y2_Nose2(1)], 'Color','k','LineWidth',2);
V5 = plot([x1_Back(1)  x2_Back(1)] , [y1_Back(1)  y2_Back(1)],  'Color','k','LineWidth',2);
V6 = plot([x2_Tail1(1) x1_Tail1(1)], [y2_Tail1(1) y1_Tail1(1)], 'Color','k','LineWidth',2);
V7 = plot([x2_Tail2(1) x1_Tail2(1)], [y2_Tail2(1) y1_Tail2(1)], 'Color','k','LineWidth',2);
```

```
% Plotting animation lines
for i = 2:length(t)-1
    set(V1,'XData',[x1_Body1(i) x2_Body1(i)],'YData',[y1_Body1(i) y2_Body1(i)]);
    set(V2,'Xdata',[x1_Body2(i) x2_Body2(i)],'Ydata',[y1_Body2(i) y2_Body2(i)]);
    set(V3,'XData',[x1_Nose1(i) x2_Nose1(i)],'YData',[y1_Nose1(i) y2_Nose1(i)]);
    set(V4,'XData',[x1_Nose2(i) x2_Nose2(i)],'YData',[y1_Nose2(i) y2_Nose2(i)]);
    set(V5,'XData',[x1_Back(i) x2_Back(i)]  ,'YData',[y1_Back(i) y2_Back(i)]);
    set(V6,'XData',[x2_Tail1(i) x1_Tail1(i)],'YData',[y2_Tail1(i) y1_Tail1(i)]);
    set(V7,'XData',[x2_Tail2(i) x1_Tail2(i)],'YData',[y2_Tail2(i) y1_Tail2(i)]);

    hold off
    drawnow
end
```

Animation Demonstration

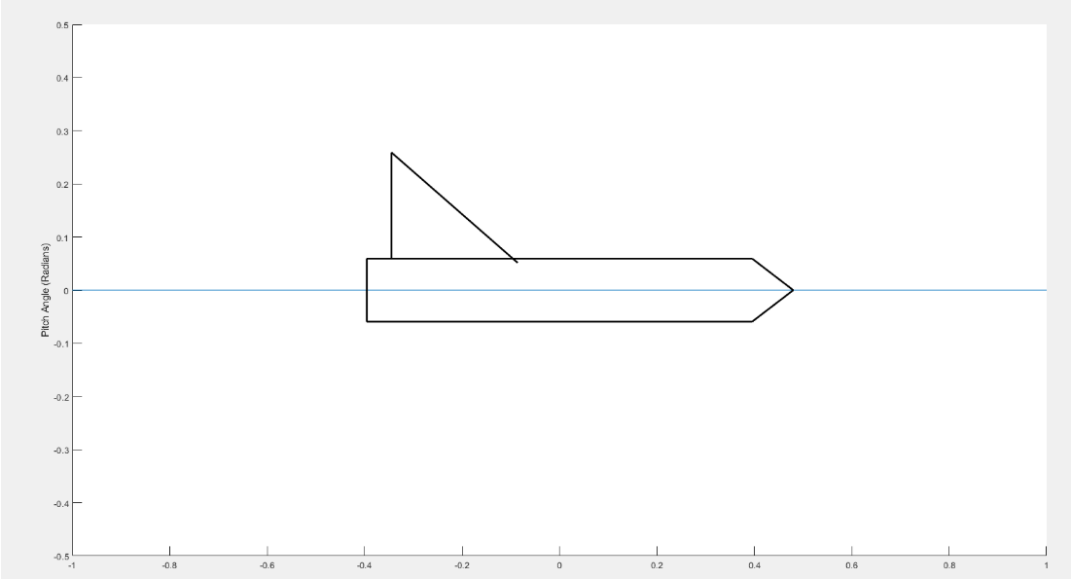


Figure 88) Aircraft animation starting position, where the pitch angle is set to 0 radians

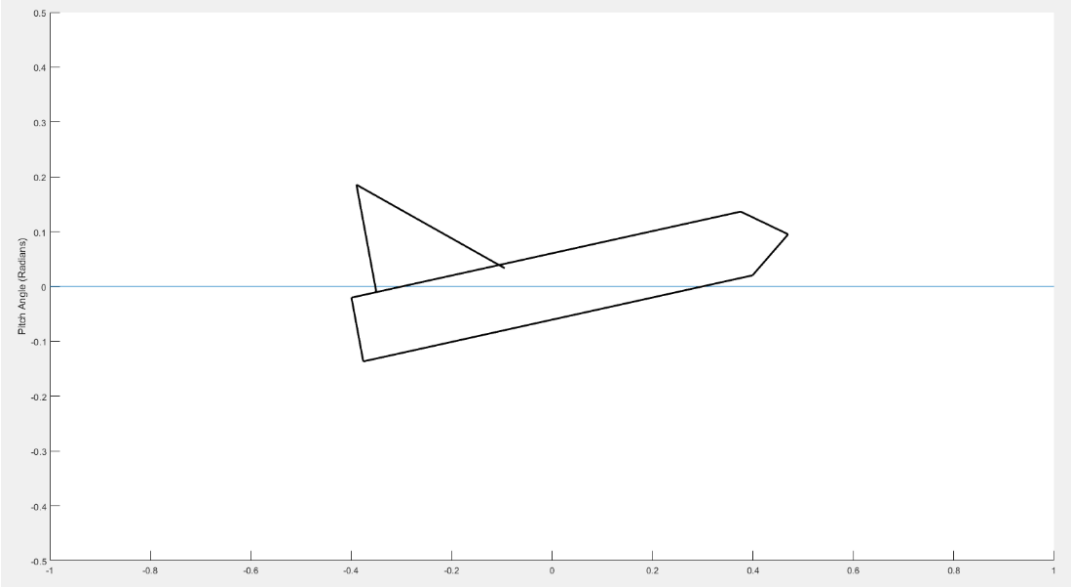


Figure 89) Aircraft animation ending position when pitch angle is set to 0.2 radians

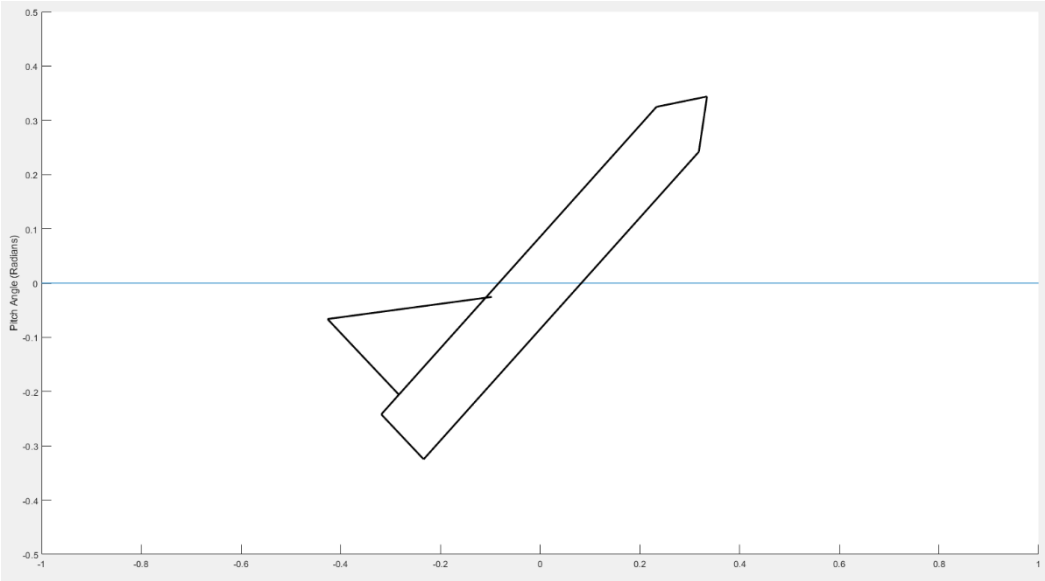


Figure 90) Aircraft animation ending position when pitch angle is set to 0.8 radians

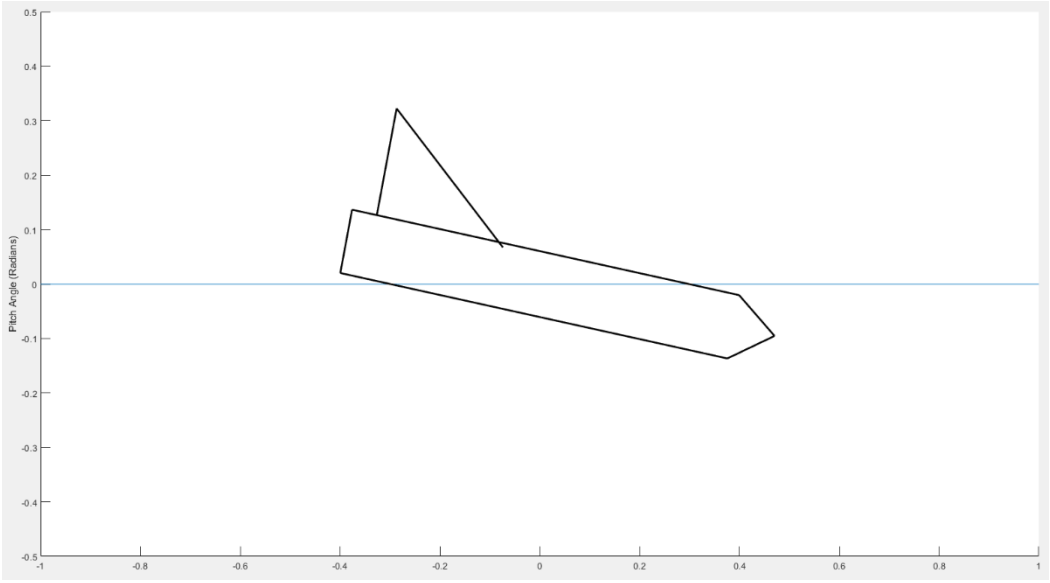
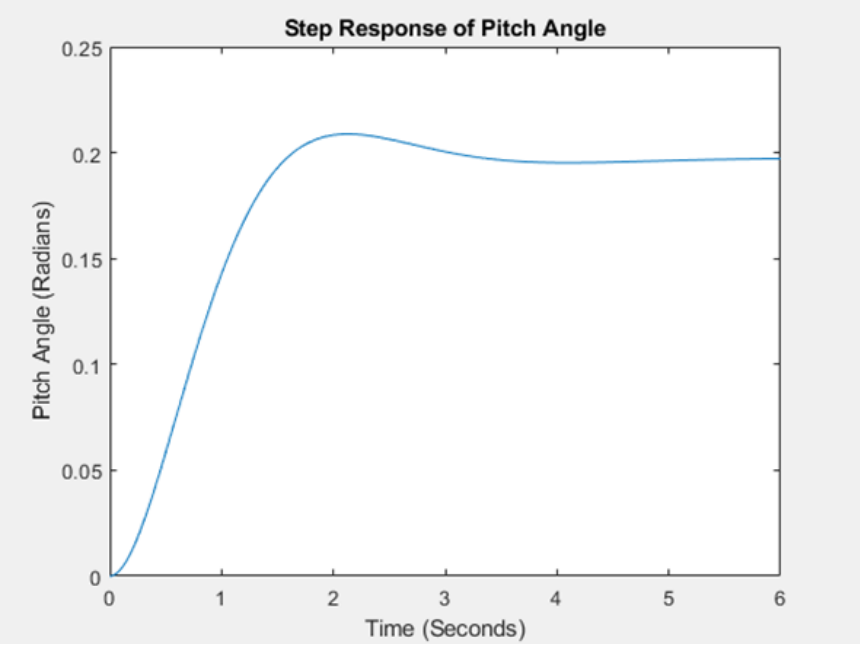


Figure 91) Aircraft animation ending position when pitch angle is set to -0.5 radians

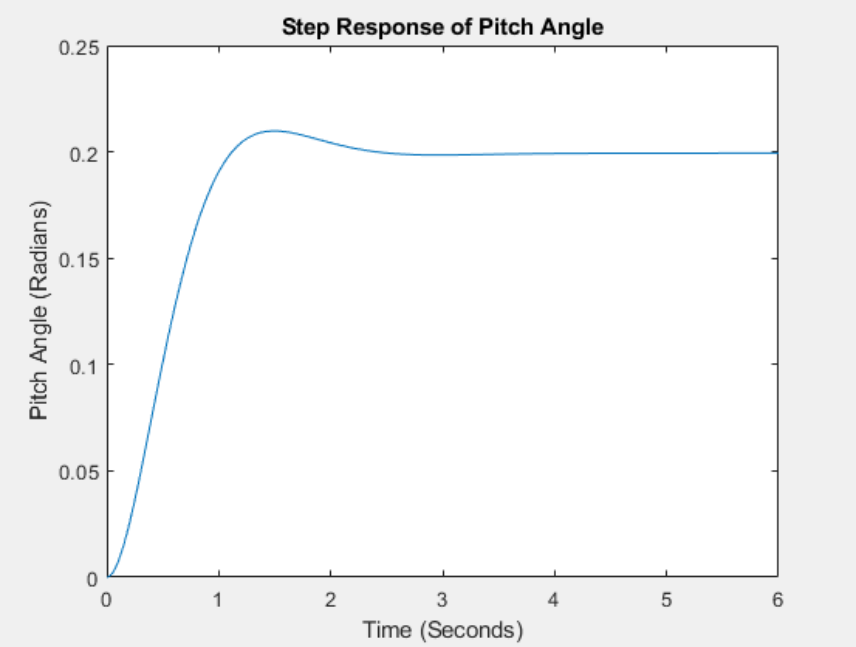
Figure 88 shows the starting position of the aircraft at 0 radians during each simulation of the animation. Depending on what the user enters as the desired pitch angle, the animation will adjust, as seen in Figure 89, Figure 90, and Figure 91 with the animation ending position when the desired pitch factor was set to 0.2, 0.8, and -0.5 radians. The weight factor will not affect the final position of the aircraft, as that is entirely dependent on the pitch angle, but it will affect the transient response the aircraft takes to reach the final position.

Aircraft Animation Step Response



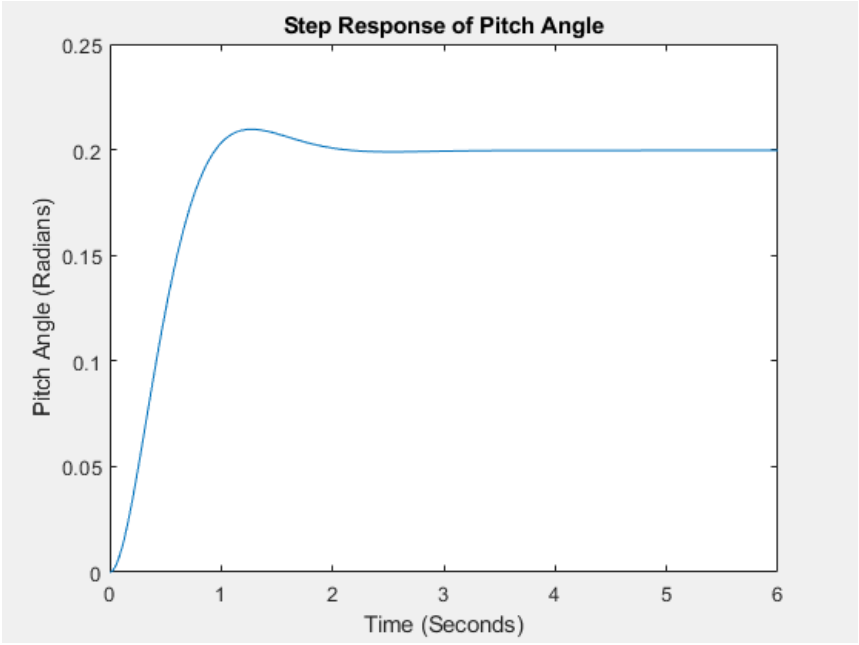

```
    RiseTime: 1.0374
SettlingTime: 2.9368
 SettlingMin: 0.1777
 SettlingMax: 0.2088
   Overshoot: 5.8906
  Undershoot: 0
        Peak: 0.2088
    PeakTime: 2.1300
```

Figure 92) Step response and "stepinfo" of pitch angle when PITCH ANGLE = 0.2 radians and WEIGHT FACTOR = 10



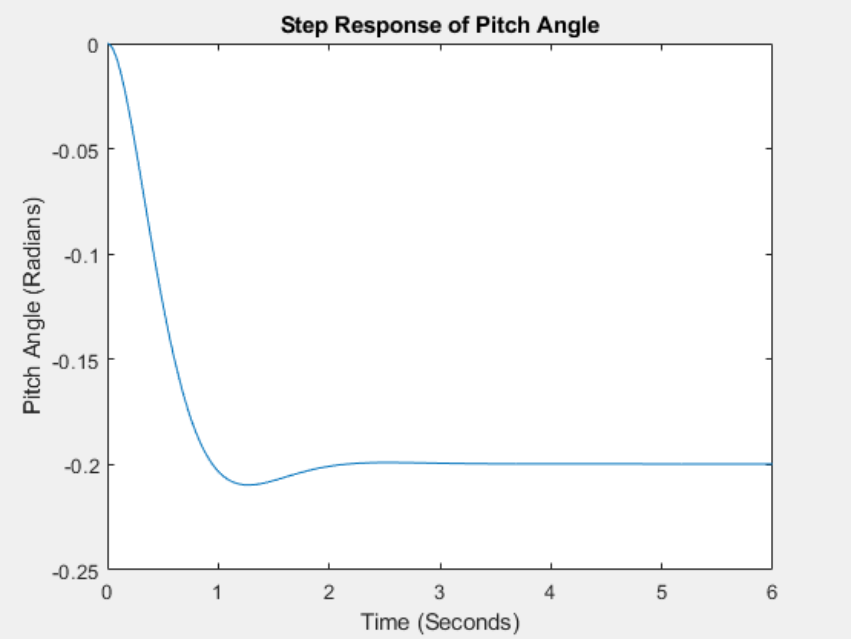

```
    RiseTime: 0.7243
SettlingTime: 2.0614
 SettlingMin: 0.1807
 SettlingMax: 0.2098
   Overshoot: 5.2099
  Undershoot: 0
        Peak: 0.2098
    PeakTime: 1.5000
```

Figure 93) Step response and "stepinfo" of pitch angle when PITCH ANGLE = 0.2 radians and WEIGHT FACTOR = 50



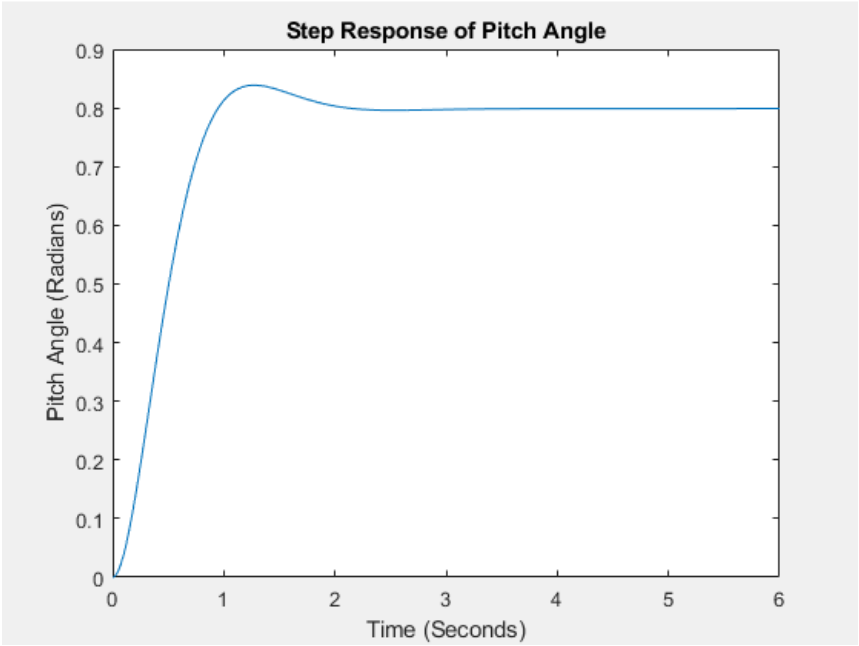

```
    RiseTime: 0.6158
SettlingTime: 1.7470
 SettlingMin: 0.1806
 SettlingMax: 0.2097
   Overshoot: 4.9859
  Undershoot: 0
        Peak: 0.2097
    PeakTime: 1.2700
```

Figure 94) Step response and "stepinfo" of pitch angle when PITCH ANGLE = 0.2 radians and WEIGHT FACTOR = 100




```
    RiseTime: 0.6158
SettlingTime: 1.7470
 SettlingMin: -0.2097
 SettlingMax: -0.1806
   Overshoot: 4.9859
  Undershoot: 0
        Peak: 0.2097
    PeakTime: 1.2700
```

Figure 95) Step response and "stepinfo" of pitch angle when PITCH ANGLE = -0.2 radians and WEIGHT FACTOR = 100




```
    RiseTime: 0.6158
SettlingTime: 1.7470
 SettlingMin: 0.7225
 SettlingMax: 0.8387
   Overshoot: 4.9859
  Undershoot: 0
        Peak: 0.8387
    PeakTime: 1.2700
```

Figure 96) Step response and "stepinfo" of pitch angle when PITCH ANGLE = 0.8 radians and WEIGHT FACTOR = 100

Viewing the step response and "stepinfo" of the pitch angle after the animation has run allows the user to get a different visualization of the same data. Recall back to the desired system requirements of:

$$Rise\ Time\ (T_r) < 1\ second$$

$$Settling\ Time\ (T_s) < 5\ seconds$$

$$Percent\ Overshoot\ (\%OS) < 5\%$$

Figure 92 shows the step response and "stepinfo" when the pitch angle is 0.2 radians, and the weight factor is 10. With those values, we get the following from the "stepinfo":

$$T_r = 1.0374\ seconds$$

$$T_s = 2.9368\ seconds$$

$$\%OS = 5.8906\%$$

We can note that the settling time is within the desired system requirements, however, the rise time and percent overshoot are too large.

We test the system again, where Figure 93 shows the step response and "stepinfo" when the pitch angle is 0.2 radians, and the weight factor is increased to 50. With those values, we get the following from the "stepinfo":

$$T_r = 0.7243\ seconds$$

$$T_s = 2.0614\ seconds$$

$$\%OS = 5.2099\%$$

We can note that now both the settling time and rise time are within the desired system requirements, however, the percent overshoot is still too large.

We test the system again, where Figure 94 shows the step response and “stepinfo” when the pitch angle is 0.2 radians, and the weight factor is increased even more to 100. With those values, we get the following from the “stepinfo”:

$$T_r = 0.6158\ seconds$$

$$T_s = 1.7470\ seconds$$

$$\%OS = 4.9859\%$$

We can note that now the settling time, rise time, and percent overshoot are within the desired system requirements.

We can also test the system with negative pitch angle, where Figure 95 shows the step response and “stepinfo” when the pitch angle is -0.2 radians, and the weight factor is 100. With those values, we get the following from the “stepinfo”:

$$T_r = 0.6158\ seconds$$

$$T_s = 1.7470\ seconds$$

$$\%OS = 4.9859\%$$

Note that the behavior is exactly the same as when a positive or negative pitch value is used.

We can also test the system with a much larger pitch angle and same weight factor, where Figure 96 shows the step response and “stepinfo” when the pitch angle is 0.8 radians, and the weight factor is 100. With those values, we get the following from the “stepinfo”:

$$T_r = 0.6158\ seconds$$

$$T_s = 1.7470\ seconds$$

$$\%OS = 4.9859\%$$

Note that the behavior is exactly the same as when the pitch angle is 0.2 radians is used, therefore we can assume that an increase or decrease in pitch angle does not change the transient system response, it is entirely dependent on the weight factor.

Conclusion

Using state space combined with MATLAB we were able to create an animated visual to show how the aircraft pitch angle adjusts to an input in radians. The systems requirements called for a settling time less than 5 seconds, a rise time less than 1 second, and a percent overshoot less than 5%. Thus, the user was asked to enter in the desired pitch angle and weight factor, and then run the code to observe the aircraft animation before viewing the data on the step input response graph and “stepinfo”. It was noted that any change in the input pitch angle, positive or negative, did not affect the system response. The strength of the weight factor, ranging from 1 to 100 decided the system transient response, and at the maximum weight factor of 100 we were able to meet the system requirements with a rise time of 0.6158 seconds, settling time of 1.7470 seconds, and percent overshoot of 4.9859%.

Project Conclusion

After analyzing the system response to different…

With the step response of the open-loop and closed-loop transfer functions of the aircraft pitch we determined that the open-loop system is unstable due to a pole lying on the imaginary axis. The closed-loop response of the system ended up being stable but failed to meet the design criteria of our experiment.

Using the P, PI, and PID controllers, the P met the rise time and steady-state error requirements but made our system oscillatory, the PI implementation further increased the overshoot and settling time of our system but reduced the steady-state error percentage to 0, and the PID reduced the overshoot and settling time. With trial and error by changing gains the PID eventually allowed us to meet all of the design requirements.

With the lead compensator, we were able to meet the requirements through analysis of the bode plot and phase margin to estimate how much phase to add to our system and predict what crossover frequency was necessary.

A compensator was designed using state-space methods of pole-placement and Linear Quadratic Regulation (LQR) while also applying a prefilter. Although pole-placement is very robust, we did not have a sense of the exact amount of energy needed to achieve complete pole-placement control. Thus, using LQR control design, we were able to better choose our gain matrix values and successfully design the controls to meet the desired aircraft pitch system requirements.

With Simulink and the linearized state-space model, open and closed loop simulations were run to test stability. Once we created the closed loop model of the system, gain (K) values were obtained using the Linearized Quadratic Regulator method to create a more ideal situation.

By animating the aircraft pitch system visually, we were able to show how the aircraft pitch angle adjusts to an input in radians and weight factor. It was noted that input pitch angle, positive or negative, did not affect the system response (in terms of meeting desired requirements for settling time, rise time, and percent overshoot), while the strength of the weight factor was what decided the system transient response. Different weight factors were tested with different pitch angles, and by using the maximum weight factor of 100 (limited in the code to be realistic, weight factor can theoretically go as large as desired if cost is not a factor) we were able to successfully meet the system requirements.

Proposed Future Work

Although it is possible to improve upon the design requirements further, it would require more complex and, in a real scenario, expensive controllers. On suggestion would be to account for the speed of the aircraft when the pitch angle changes, which could turn our system into MISO system. Doing so would complicate the problem further but allow for more control design methods to be implemented. Another suggestion would be factor in yaw and roll into equation and analyze how changing the elevator deflection and flight path angle affect the pitch, which turn our system into a MIMO system.

References

- "Aircraft Pitch: Simulink Modeling." *Control Tutorials for MATLAB and Simulink - Aircraft Pitch: Simulink Modeling*, MATLAB® 9.2, ctms.engin.umich.edu/CTMS/index.php?example=AircraftPitch&section=SimulinkModeling.
- "Aircraft Pitch: System Modeling." *Control Tutorials for MATLAB and Simulink - Aircraft Pitch: System Modeling*, MATLAB® 9.2, ctms.engin.umich.edu/CTMS/index.php?example=AircraftPitch&section=SystemModeling.
- Mathworks. "Linear Quadratic Regulator." *Linear-Quadratic Regulator (LQR) Design - MATLAB*, www.mathworks.com/help/control/ref/lqr.html.
- Tchamna, Rodrigue, et al. "Management of Linear Quadratic Regulator Optimal Control with Full-Vehicle Control Case Study." International Journal of Advanced Robotic Systems, Sept. 2016, doi:10.1177/1729881416667610.