Project: Vehicle Steering Control

Open-Loop System Analysis

Variable Values

We define the following variables based on the given values.

```
clear;
v = 25;
M = 1573;
J = 2873;
l_f = 1.10;
l_r = 1.58;
c_f = 80000;
c_r = 80000;
u = 0.9;
d_s = 1.96;
```

Define State Space Model

Matrix A:

```
A = [-u*(c_f+c_r)/(M*v) -1+u*(c_r*l_r-c_f*l_f)/(M*v^2);
    u*(c_r*l_r-c_f*l_f)/J -u*(c_f*l_f^2+c_r*l_r^2)/(J*v)];
```

Matrix B:

```
B = [u*c_f/(M*v); u*c_f*l_f/J];
```

Matrix C:

```
C = [u*((c_r*l_r-c_f*l_f)*d_s/J-(c_r+c_f)/M) (u/v)*(c_r*l_r-c_f*l_f)/M]
```

```
C = 1×2
  -67.9675    0.8788
```

Matrix D:

```
D = u*(d_s*c_f*l_f/J+c_f/M);
```

Define the state space model.

```
sys = ss(A, B, C, D);
```

Transfer Function of System

Compute the transfer function of the system.

```
s = tf('s');
G = tf(sys);
open_loop = G/s^2
open_loop =

    99.8 s^2 + 636.1 s + 3970
  ---------------------------
  s^4 + 7.377 s^3 + 25.21 s^2

Continuous-time transfer function.
Model Properties
```

Compute poles and zeros.

```
[z, p, k] = zpkdata(open_loop, 'v');
disp('Zeros: ');
```

```
Zeros:
```

```
disp(z);
```

```
  -3.1866 + 5.4425i
  -3.1866 - 5.4425i
```

```
disp('Poles: ');
```

```
Poles:
```

```
disp(p);
```

```
   0.0000 + 0.0000i
   0.0000 + 0.0000i
  -3.6886 + 3.4067i
  -3.6886 - 3.4067i
```

```
disp('Gain: ');
```

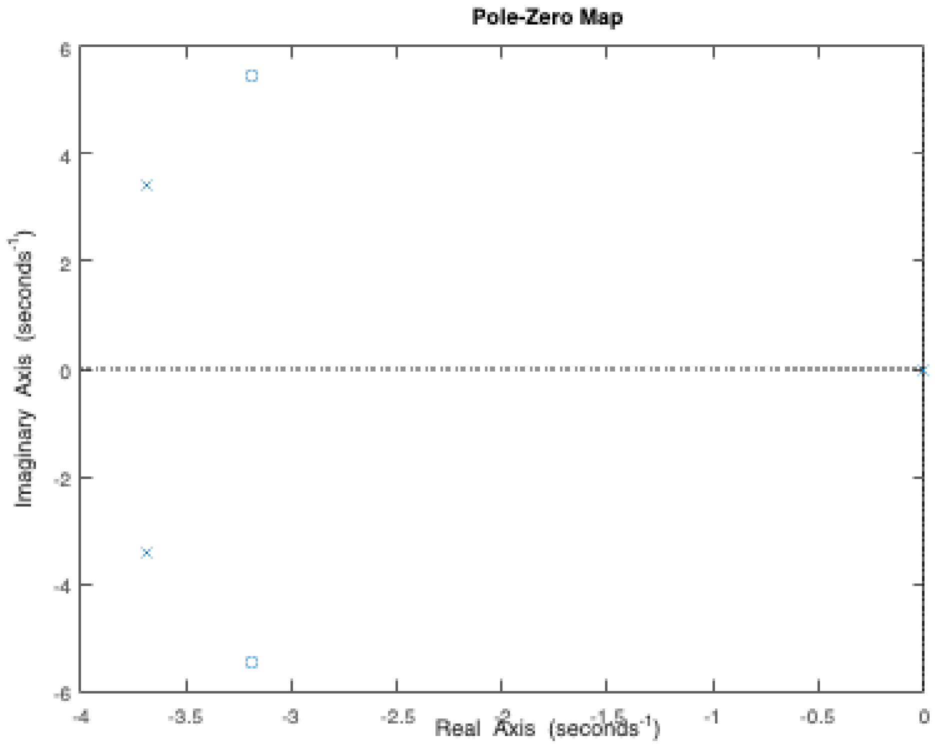
```
Gain:
```

```
disp(k);
```

```
   99.8037
```

Plot poles and zeros.

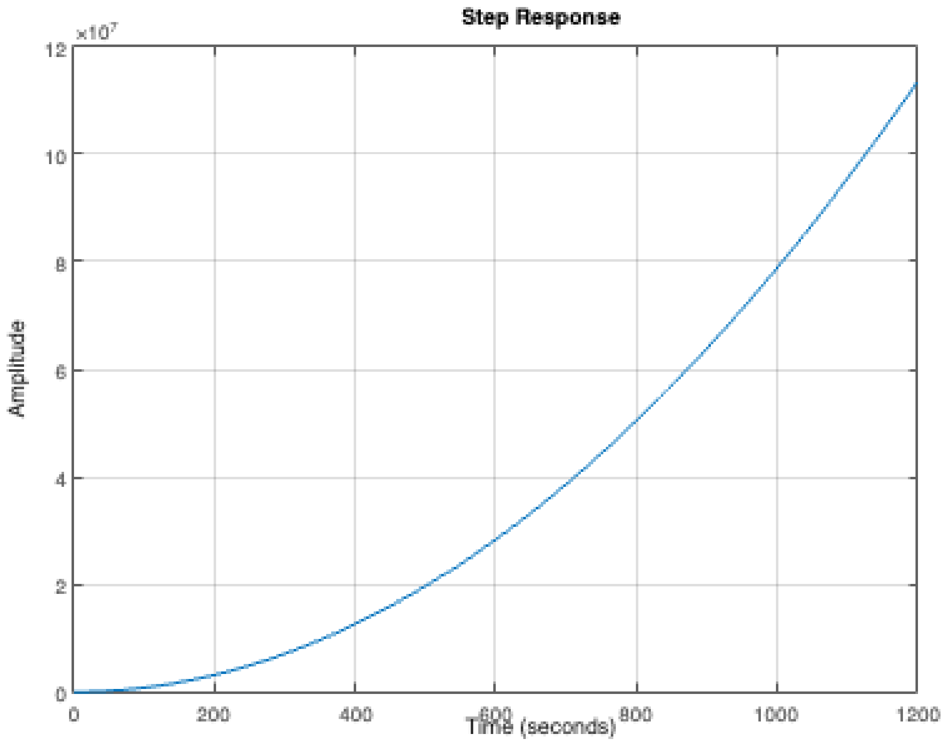
```
figure;
pzmap(open_loop);
```



Open Loop Step Response

Graph the open loop step response.

```
figure;
step(open_loop);
grid on;
```



```
disp("G(s) Step Response Specs:");
```

```
G(s) Step Response Specs:
```

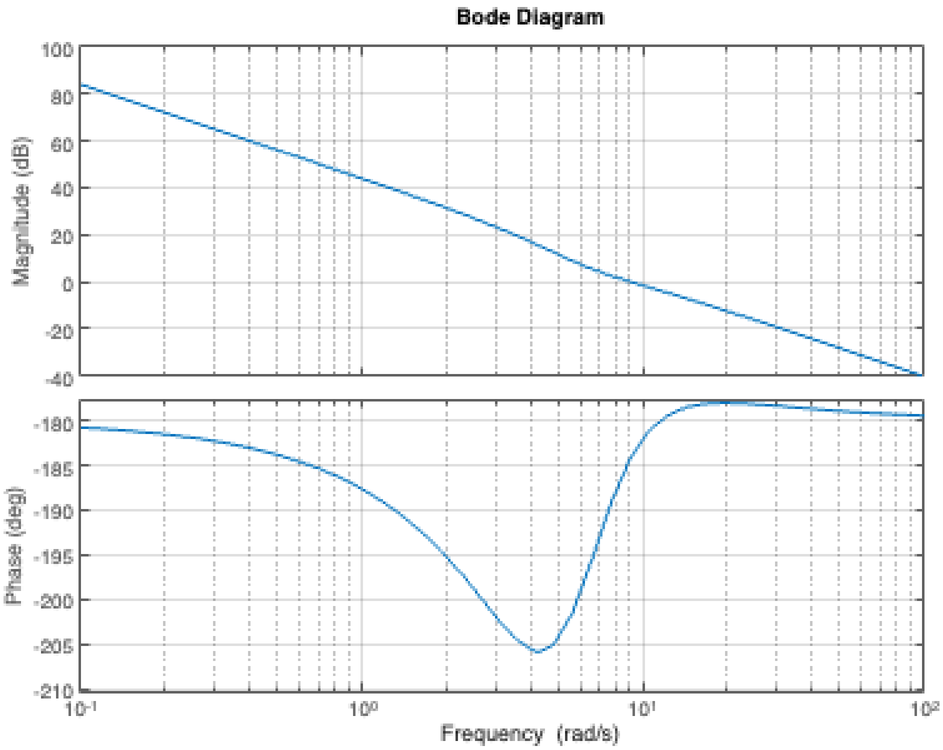
```
disp(stepinfo(open_loop));
```

```
Warning: Simulation did not reach steady state. Please specify YFINAL if this system is stable and
eventually settles.
        RiseTime: NaN
    TransientTime: NaN
     SettlingTime: NaN
      SettlingMin: NaN
      SettlingMax: NaN
        Overshoot: NaN
       Undershoot: NaN
             Peak: Inf
         PeakTime: Inf
```

Open Loop Frequency Response

Compute the open loop frequency response.

```
figure;
bode(open_loop);
grid on;
```



Observations

The open loop transfer function is marginally stable. We have a pole at 0, making it marginally stable. We can see in the step response that it diverges to infinity.

PID Control Design

The objective here is to design a PID controller that minimizes the system output $\Delta Y_S(s)$ for any input reference lateral acceleration $\rho_{ref}(s)$. We note that since we are inputting $v^2\rho_{ref}(s)$, where v^2 is a constant, in our design and analysis, we can denote the input to the whole system with the variable $\ddot{Y}_{ref}(s)$. By the linearity property of LTI systems, any $\rho_{ref}(s)$ that we input will just be scaled by $1/v^2$.

To tune the controller to perform well, we first obtain the transfer function from $\ddot{Y}_{ref}(s)$ to $\ddot{Y}_S(s)$. This gives us the system response

$$H(s) = \frac{\ddot{Y}_S(s)}{\ddot{Y}_S(s)} = \frac{FG}{s^2 + FG}.$$

Then, the task is to design a controller Fthat produces the desired output $\ddot{Y}_S(s)$given the vehicle G and the integrator $1/s^2$. Once we have calibrated it to produced the desire output, we can just redefine the system response with $\Delta Y_S(s)$ as the output. This transfer function is given by

$$L(s) = \frac{\Delta Y_S(s)}{\ddot{Y}_{ref}(s)} = -\frac{1}{s^2 + FG}.$$

Reference Path

We first define a reference path as an input. We are using a Lissajous curve in the shape of the infinity sign. The parametric equations of a Lissajous curve are given by

$$(x, y) = (A\sin(at + \delta), B\sin(bt)).$$

Here, we set $\delta = \pi/2$, $A = a = 1$, $B = 1$, and $b = 2$.

```
t = linspace(1, 100, 1000);
delta = pi/2;
a = 1;
b = 2;

rx_d = cos(t+delta);
rx_dd = -sin(t+delta);

ry_d = b*cos(b*t);
ry_dd = -b*b*sin(b*t);

p_ref = (rx_d.*ry_dd-ry_d.*rx_dd)./((rx_d.^2+ry_d.^2).^1.5);
y_ref = p_ref*v^2; % Time domain expression of path
```

Proportional Control Design

We first start with designing the proportional control, assigning a relatively large value of K_P and small values for the derivative and integral constants.

```
Kp = 5;
Ki = 0.1;
Kd = 0.1;
F = pid(Kp, Ki, Kd); % Controller
```

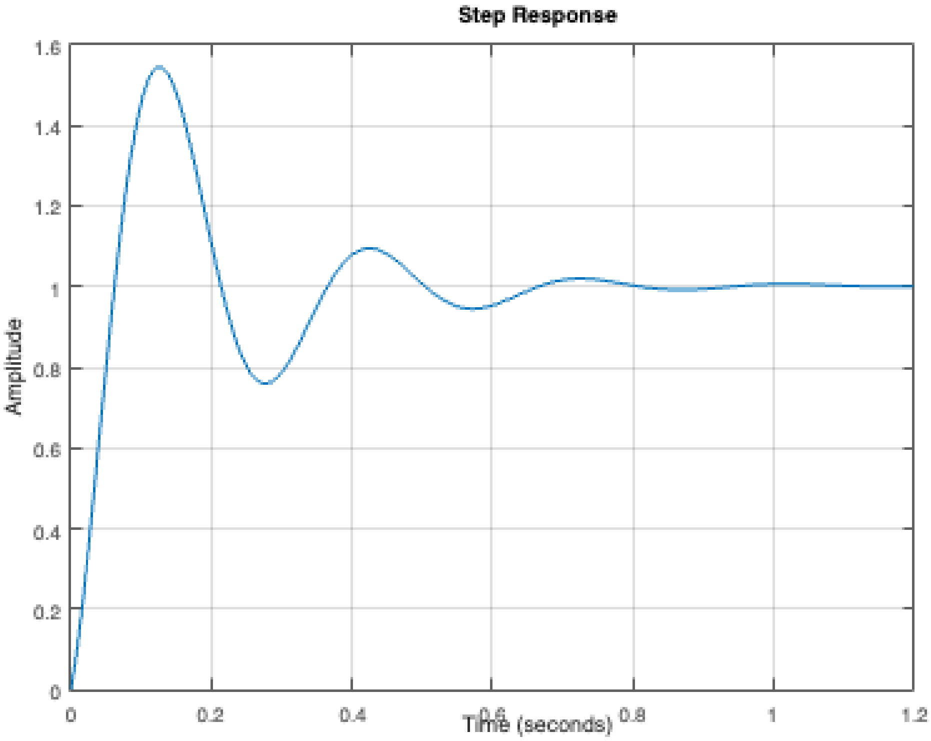
We then formed the closed loop system.

```
s = tf('s');
H = (F*G)/(s^2+F*G)
```

```
H =
 
       9.98 s^7 + 636.3 s^6 + 7989 s^5 + 6.056e04 s^4 + 2.377e05 s^3 + 5.049e05 s^2 + 1.001e04 s
  ------------------------------------------------------------------------------------------------
    s^8 + 24.73 s^7 + 741.1 s^6 + 8361 s^5 + 6.12e04 s^4 + 2.377e05 s^3 + 5.049e05 s^2 + 1.001e04 s
 
Continuous-time transfer function.
Model Properties
```

Afterward, we then tested the step input. This corresponds to constant lateral acceleration, meaning that the reference path is a circle.

```
figure;
step(H);
grid on;
```



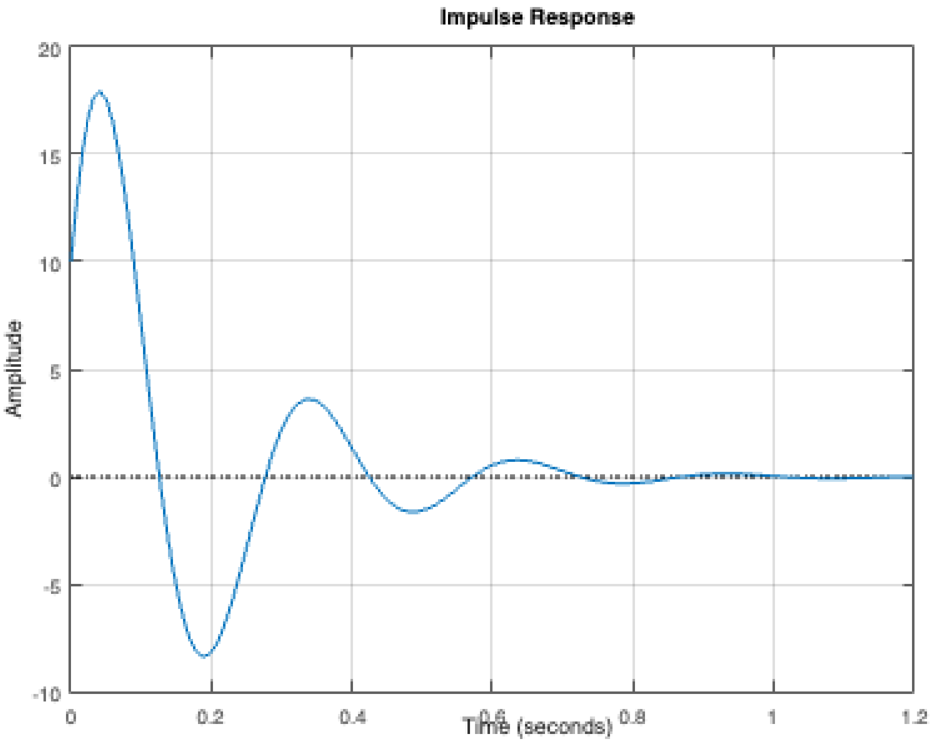
```
disp(stepinfo(H));
```

```
         RiseTime: 0.0479
    TransientTime: 0.6389
     SettlingTime: 0.6389
      SettlingMin: 0.7597
      SettlingMax: 1.5452
        Overshoot: 54.5164
       Undershoot: 0
             Peak: 1.5452
         PeakTime: 0.1285
```

In the step response, we can see that the settling time of less than a second is acceptable. However, the 54% overshoot and oscillations are not great for a vehicle steering system, as this will cause the oscillations in the wheel before it finishes turning. As such, the main task in tuning the PID coefficients is to dampen the oscillations and minimize the overshoot while maintaing a relatively quick settling time.

For reference, we also checked the system's impulse response. We can see that it is stable and that the system response dies off to zero as time goes to infinity.

```
figure;
impulse(H);
grid on;
```



Before moving onto tuning the PID coefficients, we must also ensure system stability.

```
[z, p, k] = zpkdata(H, 'v');
disp('Zeros: ');
```

```
Zeros:
```

```
disp(z);
```

```
   0.0000 + 0.0000i
 -49.9800 + 0.0000i
  -3.1866 + 5.4425i
  -3.1866 - 5.4425i
  -3.6886 + 3.4067i
  -3.6886 - 3.4067i
  -0.0200 + 0.0000i
```

```
disp('Poles: ');
```

```
Poles:
```

```
disp(p);
```

```
   0.0000 + 0.0000i
  -5.3775 +21.1187i
  -5.3775 -21.1187i
  -3.2913 + 5.5628i
  -3.2913 - 5.5628i
  -3.6886 + 3.4067i
  -3.6886 - 3.4067i
  -0.0200 + 0.0000i
```

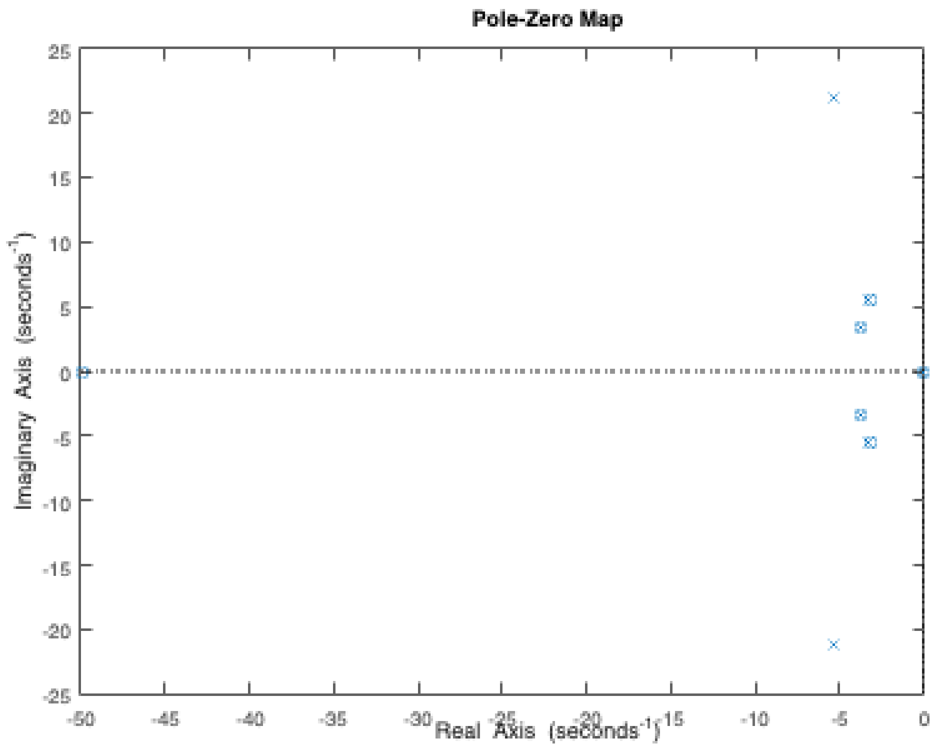
```
disp('Gain: ');
```

```
Gain:
```

```
disp(k);
```

```
    9.9804
```

```
figure;
pzmap(H);
```



The pole and zero in the origin cancel each other out. All remaining poles have negative real components, so our system is stable.

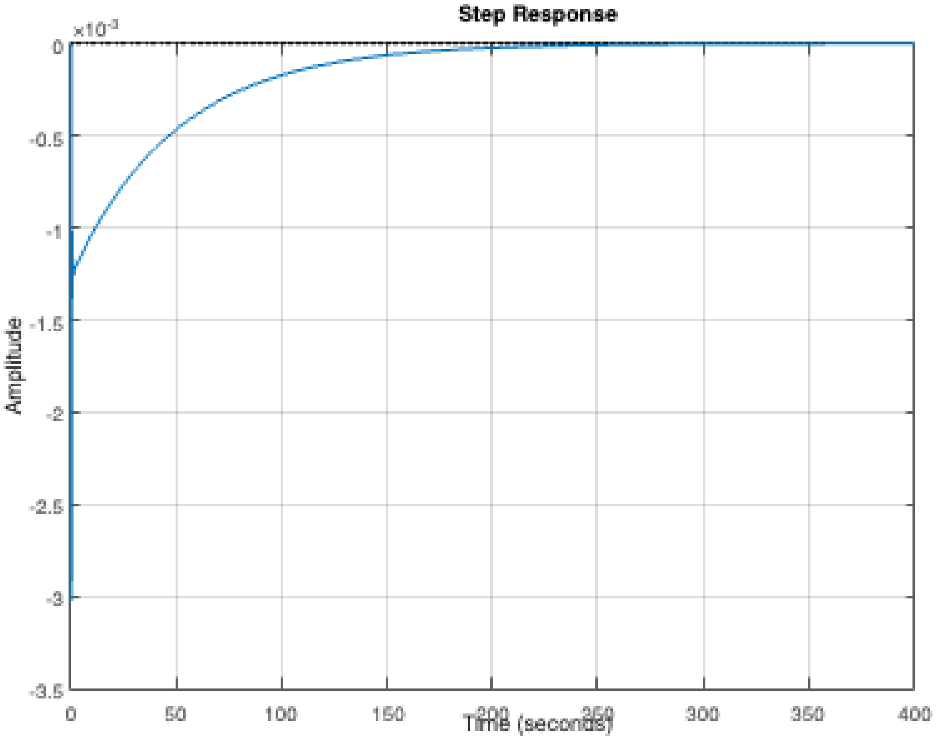
Lastly, we can run a preliminary test of $L(s)$, taking $\Delta Y_S(s)$ as the output. It is important to note that since the steady state of this system is supposed to be zero, the rise time, settling time, and over shoot times cannot be computed due to division by zero. As such, when evaluating the performance of this system, we look to $H(s)$.

```
L_test = -1/(s^2+F*G);
figure;
step(L_test);
stepinfo(L_test)
```

```
ans = struct with fields:
         RiseTime: 0
    TransientTime: 152.1152
     SettlingTime: NaN
      SettlingMin: -0.0030
```

```
        SettlingMax: 0
          Overshoot: Inf
         Undershoot: Inf
               Peak: 0.0030
           PeakTime: 0.1370
```

```
grid on;
```



We can see that the system first goes negative instead of positive. This is expected, as $H(s)$ first has a rise time before reaching unity, causing the $\ddot{Y}_S(s) - \ddot{Y}_{ref}(s)$ to be negative at first.

Tuning PID Coefficients

We aim to achieve a settling time of less than a second, an overshoot of no more than 10-percent, and a rise time of less than half a second. These characteristics will allow the vehicle to quickly response to changes along the rode and change its course in an efficient manner.

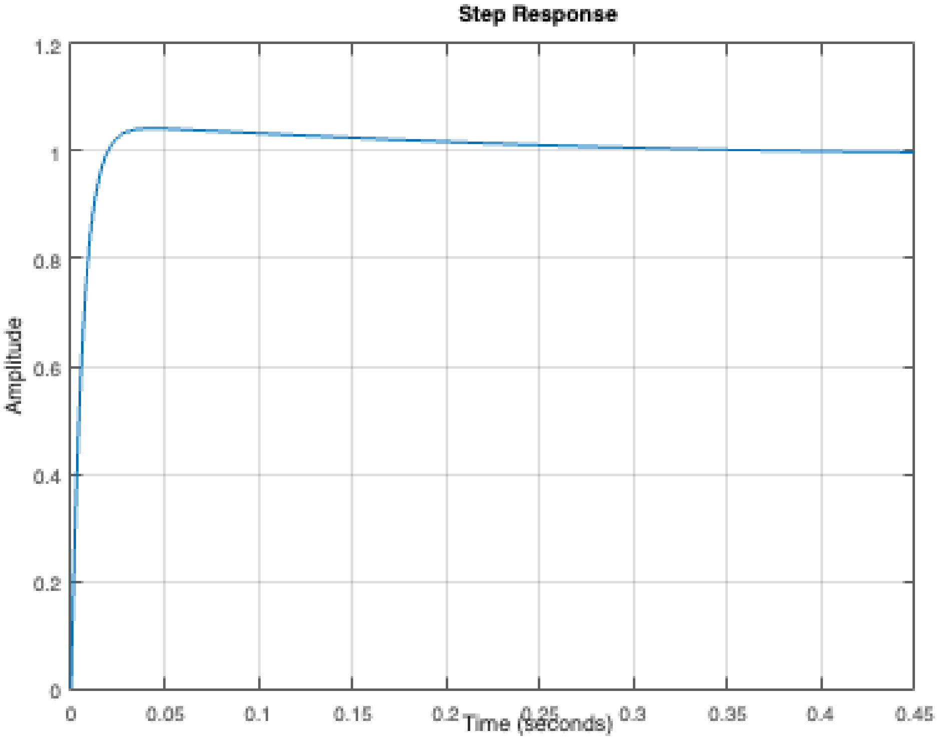
We used MATLAB's pidTuner to help us find the following coefficients, using G/s^2 as the system plant. We optimized first for robustness and then speed, ensuring the overshoots were not too extreme. In the process, we set robustness as high as possible, then increased speed until oscillations began to occur. We then adjusted the unit step response in this manner until we achieved a rise time of 0.012 seconds, a settling time of 0.184 seconds, and an overshoot of 4.23-percent. This will allow the vehicle to respond quickly and stably to changes in the road.

```
%pidTuner(G/s^2, 'PID');
Kp_new = 14.1065;
```

```
Ki_new = 26.9496;
Kd_new = 1.6286;

F_new = pid(Kp_new, Ki_new, Kd_new); % New controller
H_new = (F_new*G)/(s^2+F_new*G);
```

We can see that our coefficients perform well in the new step response.

```
figure;
step(H_new);
grid on;
```

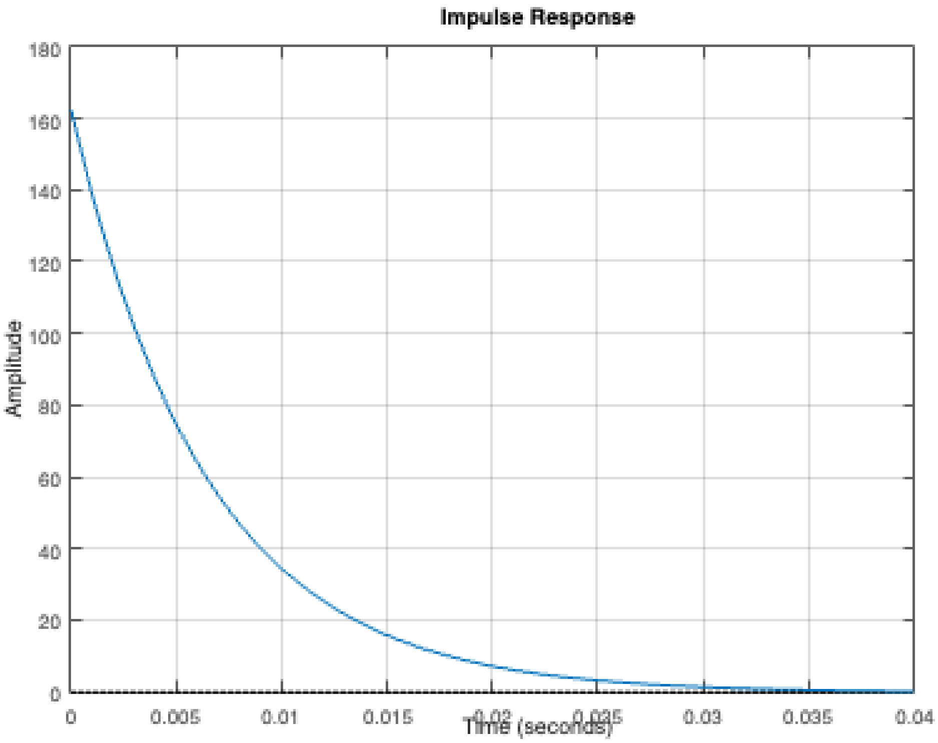


```
disp(stepinfo(H_new));
```

```
        RiseTime: 0.0120
   TransientTime: 0.1842
    SettlingTime: 0.1842
     SettlingMin: 0.9043
     SettlingMax: 1.0423
       Overshoot: 4.2327
      Undershoot: 0
            Peak: 1.0423
        PeakTime: 0.0438
```

The new impulse response is also stable.

```
figure;
impulse(H_new);
grid on;
```



The poles and zeros also confirm stability. Again, the pole and zero at the origin cancel each other out.

```
[z, p, k] = zpkdata(H_new, 'v');
disp('Zeros: ');
```

```
Zeros:
```

```
disp(z);
```

```
   0.0000 + 0.0000i
  -3.1866 + 5.4425i
  -3.1866 - 5.4425i
  -5.8170 + 0.0000i
  -3.6886 + 3.4067i
  -3.6886 - 3.4067i
  -2.8447 + 0.0000i
```

```
disp('Poles: ');
```

```
Poles:
```

```
disp(p);
```

```
   1.0e+02 *

   0.0000 + 0.0000i
  -1.5470 + 0.0000i
  -0.0319 + 0.0553i
  -0.0319 - 0.0553i
  -0.0601 + 0.0000i
  -0.0369 + 0.0341i
  -0.0369 - 0.0341i
  -0.0283 + 0.0000i
```

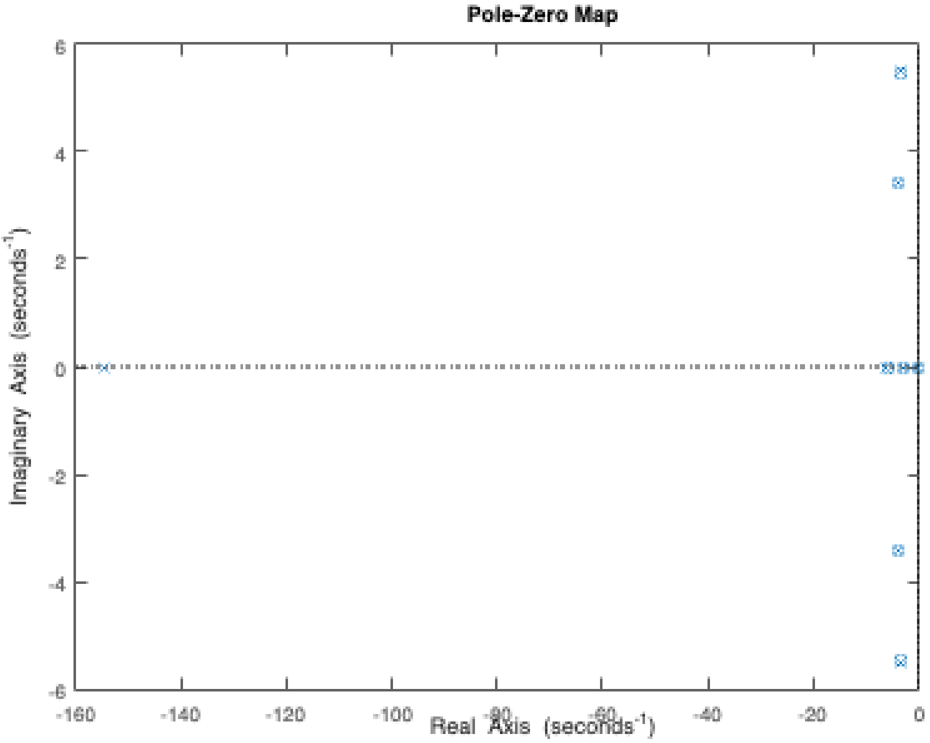
```
disp('Gain: ');
```

```
Gain:
```

```
disp(k);
```

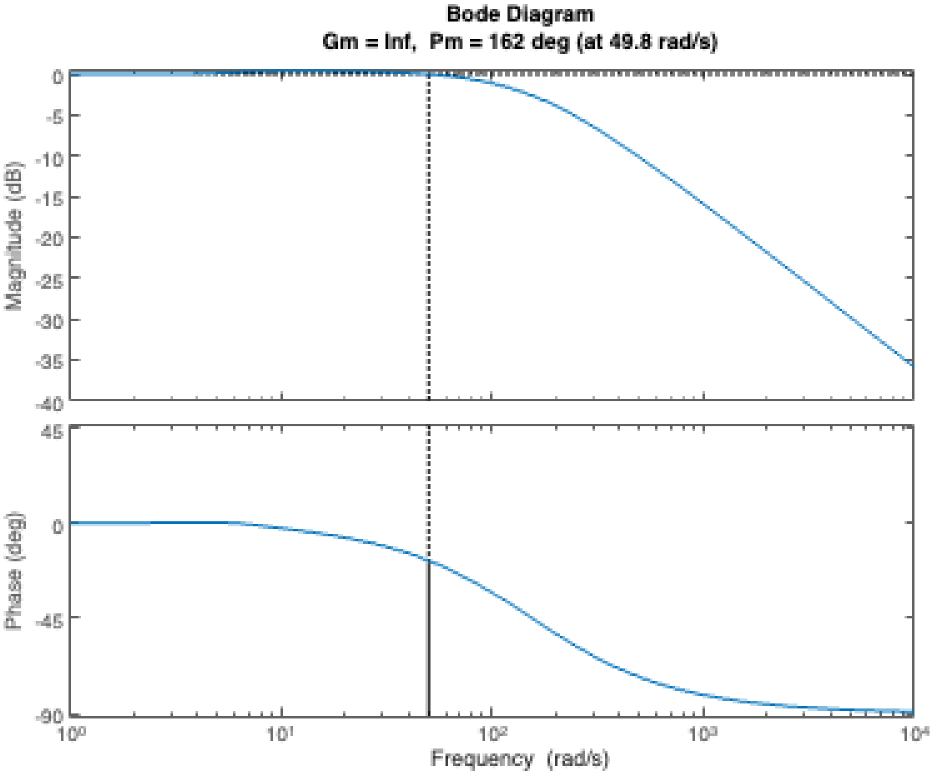
```
  162.5404
```

```
figure;
pzmap(H_new);
```



In the bode plot, the gain margin of infinity suggests that no matter how much the gain increases, our system will always be BIBO stable.

```
figure;
margin(H_new)
```

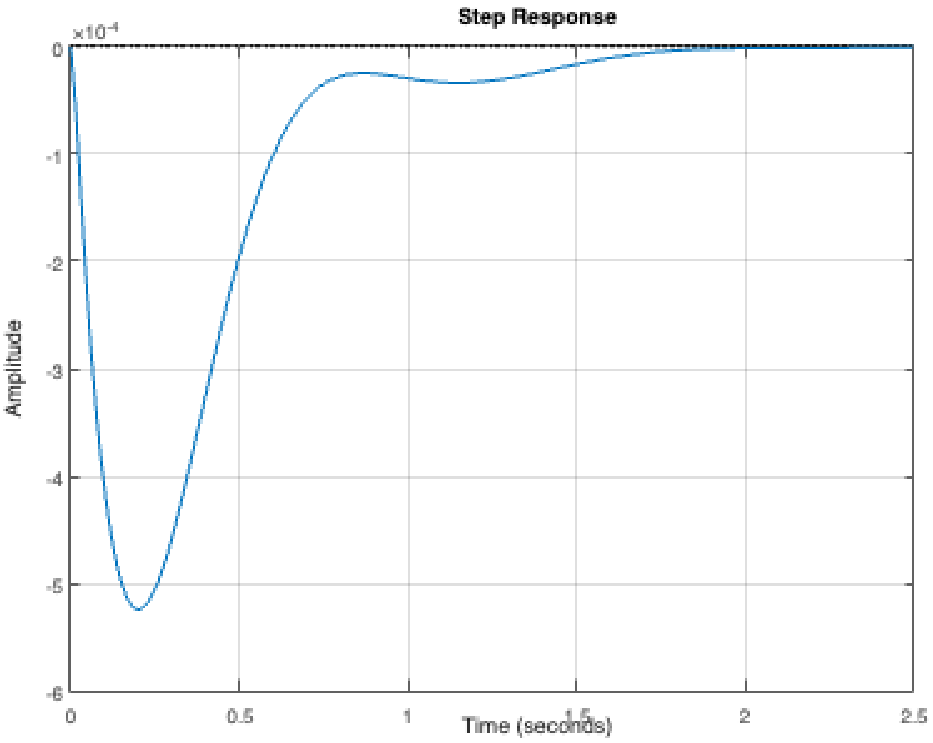


We also defined the system $L(s)$ again using this new controller.

```
L_new = -1/(s^2+F_new*G);
figure;
step(L_new);
stepinfo(L_new)
```

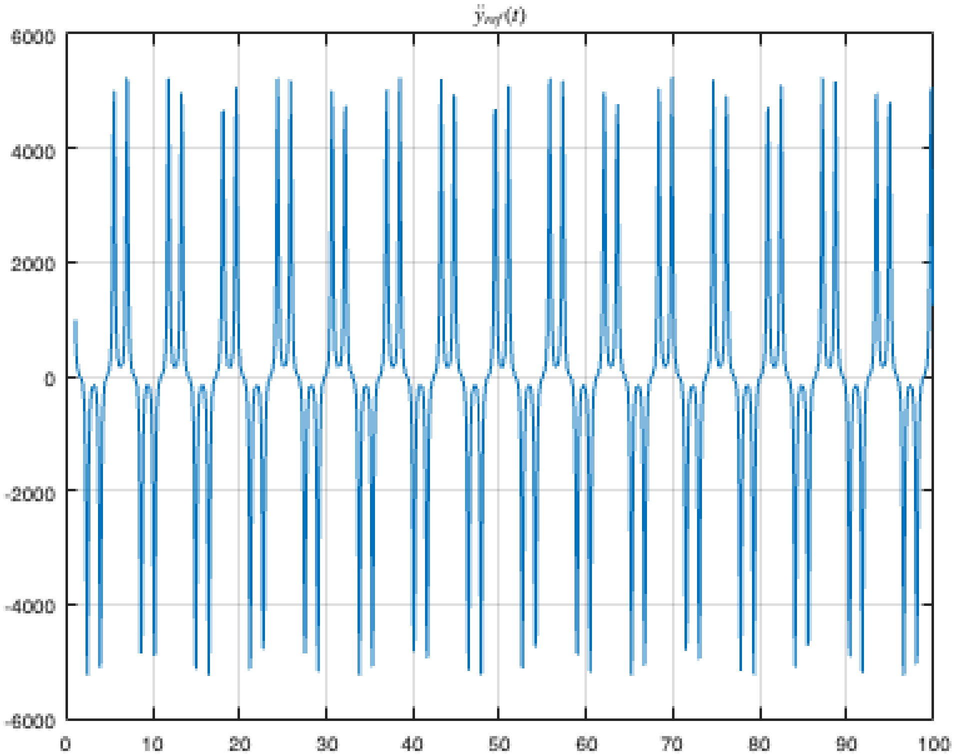
```
ans = struct with fields:
         RiseTime: 0
    TransientTime: 1.6094
     SettlingTime: NaN
      SettlingMin: -5.2416e-04
      SettlingMax: 0
        Overshoot: Inf
       Undershoot: Inf
             Peak: 5.2416e-04
         PeakTime: 0.1993
```

```
grid on;
```

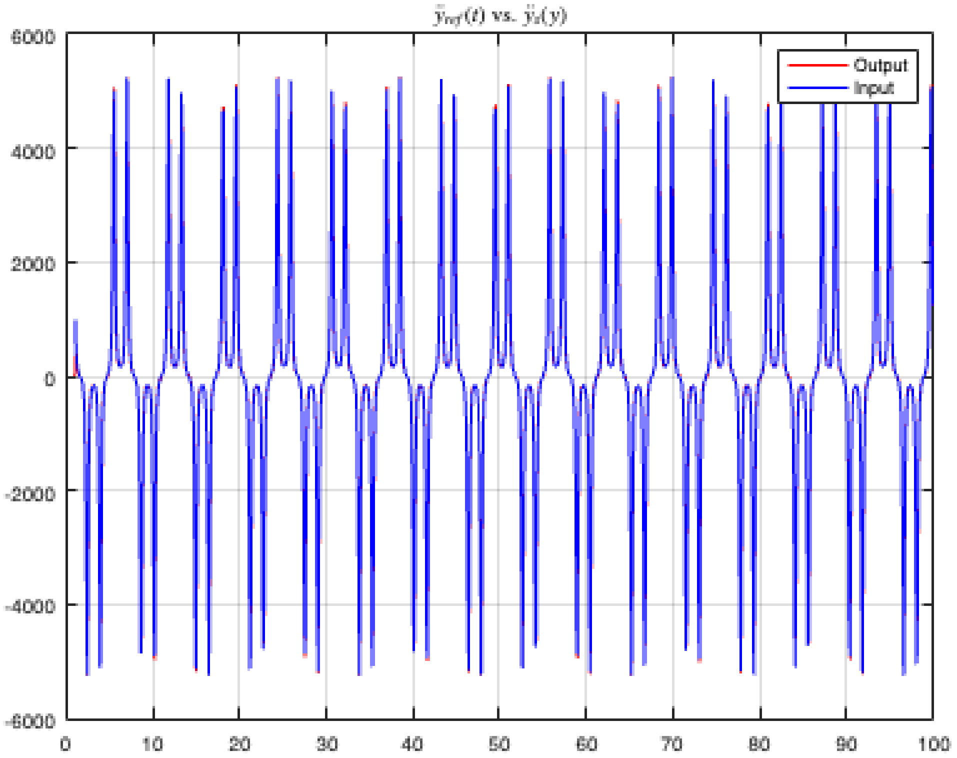


To test this design, we inputted the Lissajous curve defined earlier into the new system response. The input is blue, and the output is red. We can see that our control system's steady state response follows the reference accurately.

```
[y, t_out] = lsim(H_new, y_ref, t);
figure;
plot(t, y_ref);
grid on;
title("$\ddot y_{ref}(t)$", 'Interpreter', 'latex');
```

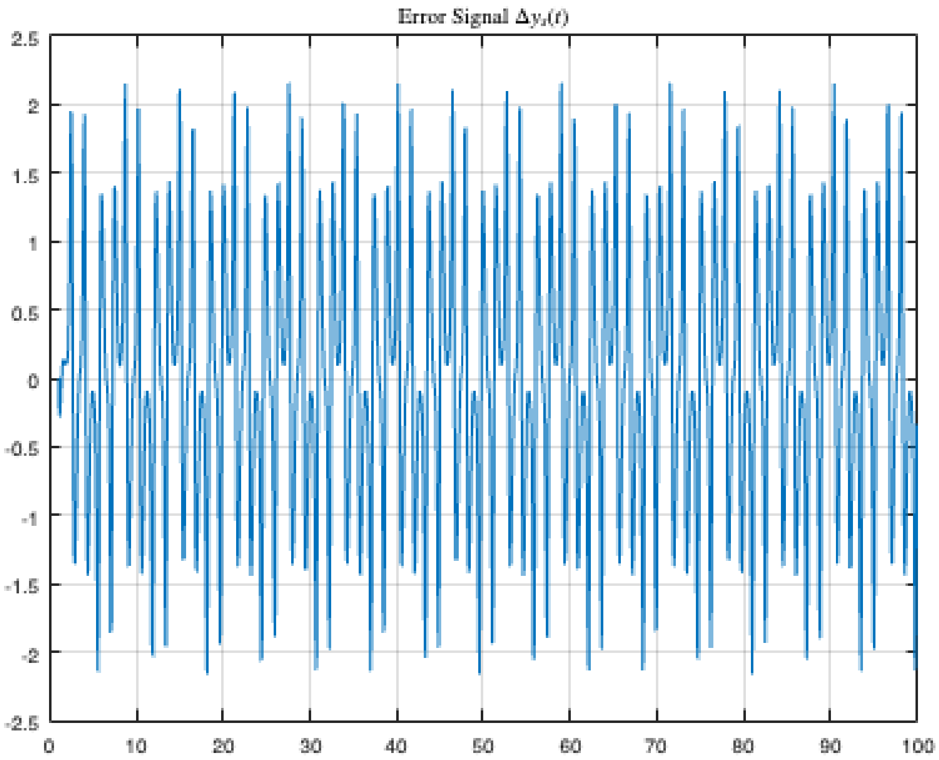


```
figure;
%plot(t, y_ref, 'b', 'DisplayName', 'Input');   % Plot input in blue
plot(t, y, 'r', 'DisplayName', 'Output'); % Plot output in red

hold on;
%plot(t, y, 'r', 'DisplayName', 'Output'); % Plot output in red
plot(t, y_ref, 'b', 'DisplayName', 'Input');   % Plot input in blue

legend('show');
grid on;
title('$\ddot y_{ref}(t)$ vs. $\ddot y_s(y)$', 'Interpreter', 'latex')
hold off;
```



We also plotted the system error for the Lissajous curve.

```
[error, t_out] = lsim(L_new, y_ref, t);
figure;
plot(t, error);
grid on;
title('Error Signal $\Delta y_s(t)$', 'Interpreter', 'latex');
```



Given the input's large magnitude in comparison to the error signal, our error signal is extremely small, and our system follows the reference signal well. It is always within ± 2.5 of the reference signal.

```
% Compute median percent error
disp("Median percent error:");
```

```
Median percent error:
```

```
disp(median(abs(y'-y_ref)/abs(y_ref)));
```

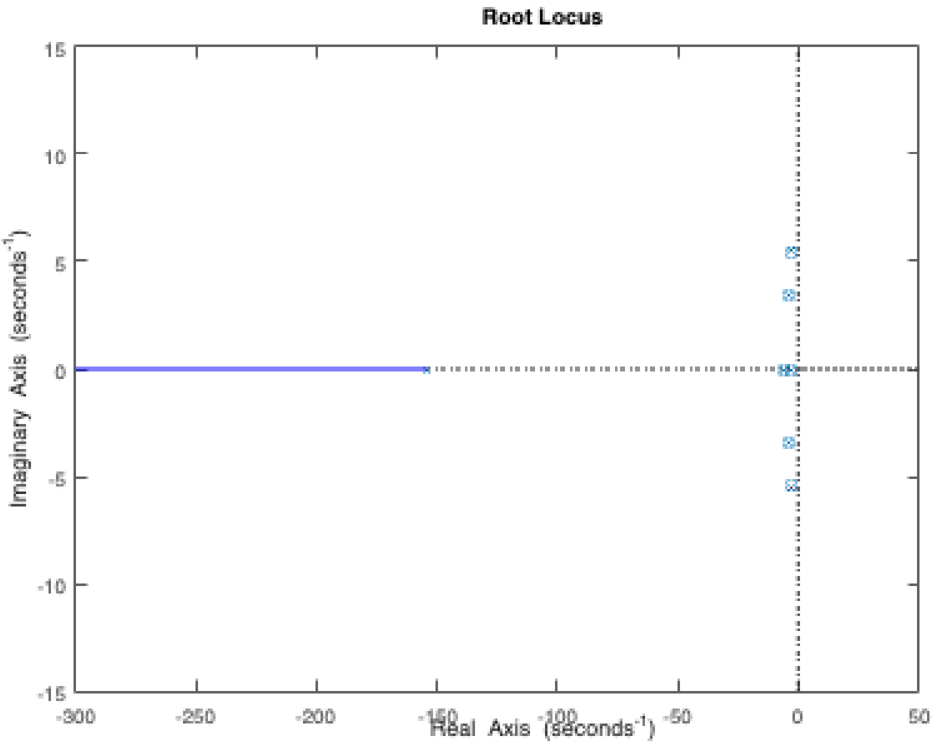
```
    0.0132
```

Root Locus and Lead-Lag Control Design

Root Locus Plot

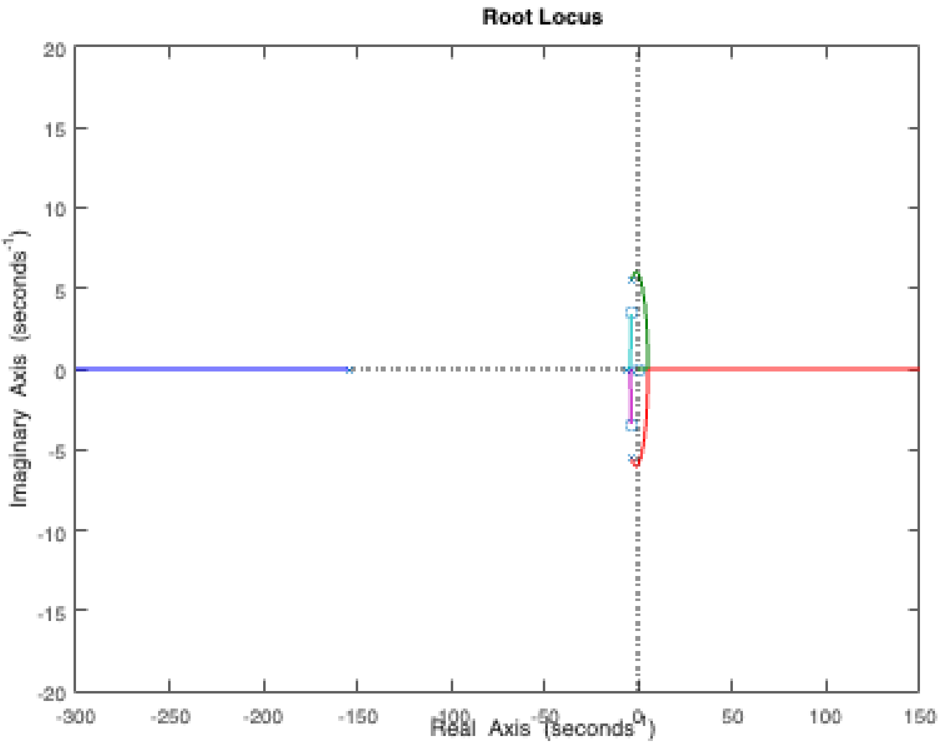
We first generate the root locus plot of our new $H(s)$ with the tuned PID coefficients. We can see that regardless of how high the gain is, the root locus never crosses into the right half plane.

```
figure;
rlocus(H_new);
```



However, for the main system $L(s)$, we see that increasing the gain too much can lead to instability, as seen by the red locus.

```
figure;
rlocus(L_new);
```



Thus, one way we can ensure stability for the main system is by adding some zeros to the left half plane to draw the poles toward it. This can prevent them from going into the right half plane as the gain increases.

Lead-Lag Compensation

We then designed a lead lag compensator.

```
lead = (s+10)/(s+5);
lag = (s+0.1)/(s+0.01);
F_ll = F_new*lead*lag;
H_ll = (F_ll*G)/(s^2+F_ll*G);
```

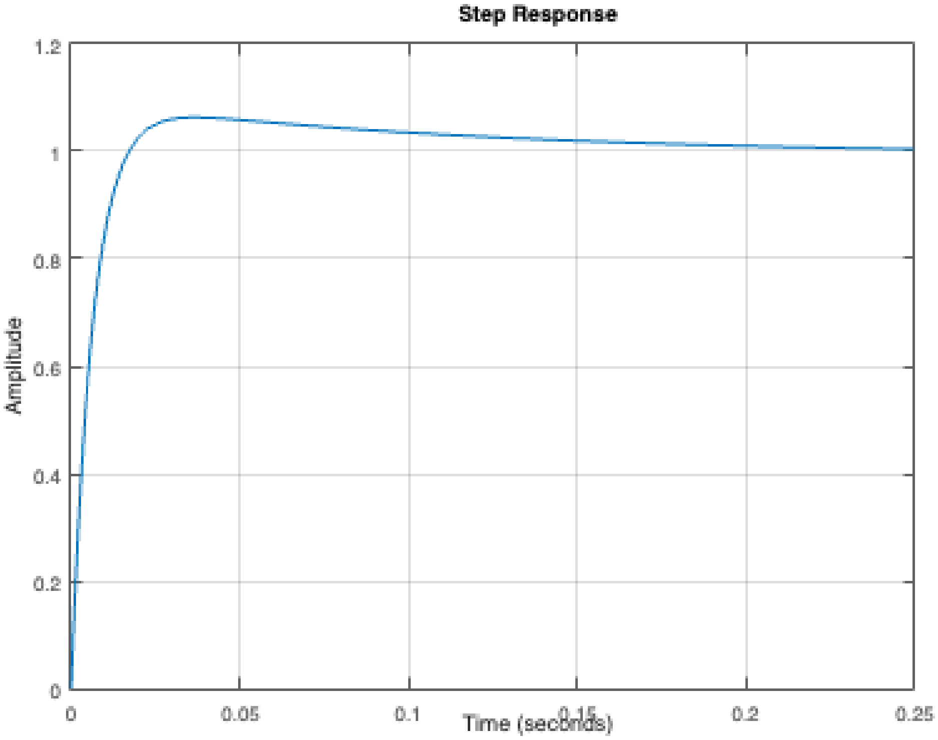
The step response of $H_{LL}(s)$ decays to the steady state value much more quickly, at a slightly higher percent overshoot.

```
figure;
step(H_ll);
stepinfo(H_ll)
```

```
ans = struct with fields:
         RiseTime: 0.0112
    TransientTime: 0.1479
     SettlingTime: 0.1479
      SettlingMin: 0.9045
      SettlingMax: 1.0628
        Overshoot: 6.2823
       Undershoot: 0
```

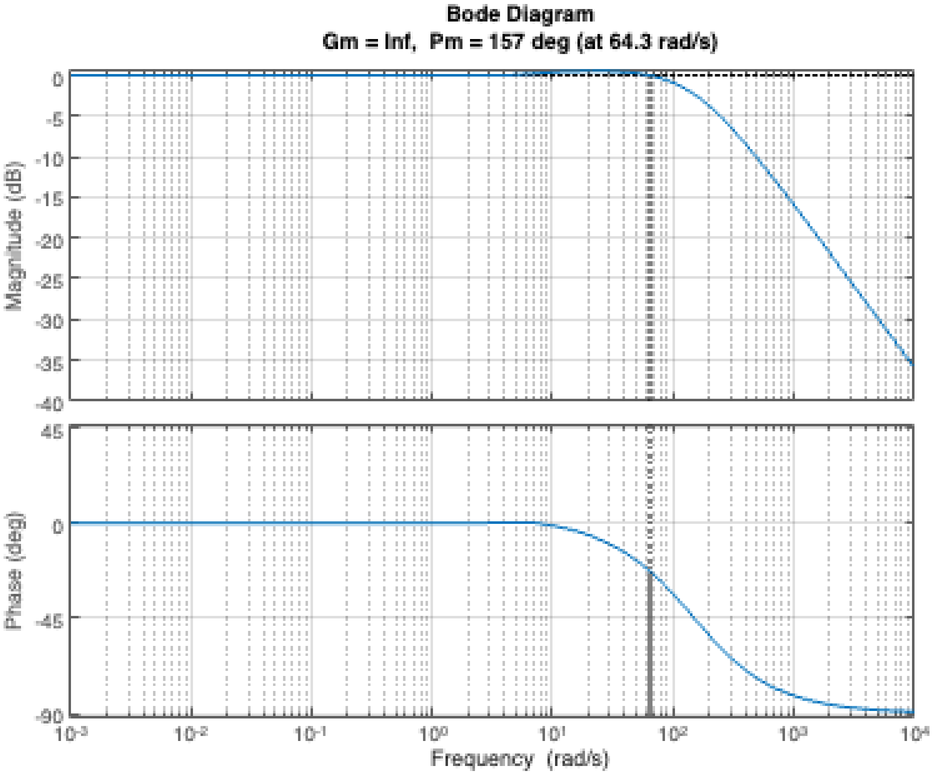
```
            Peak: 1.0628
        PeakTime: 0.0368
```

```
grid on;
```



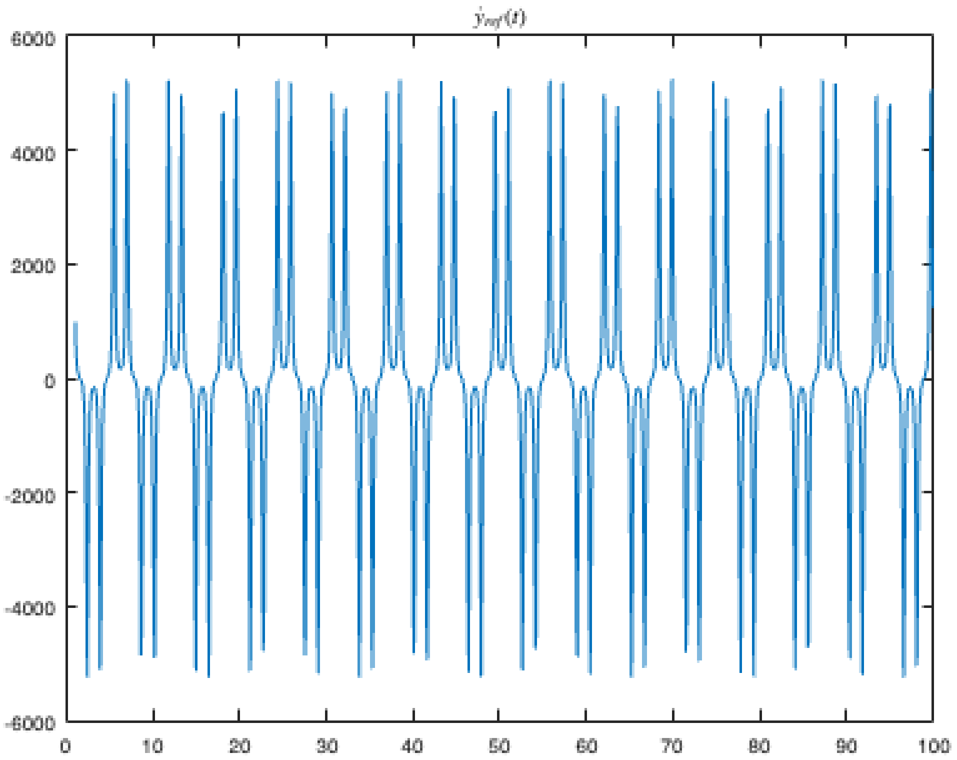
The bode plot indicates a much large phase margin, indicating more stability.

```
margin(H_ll);
grid on;
```

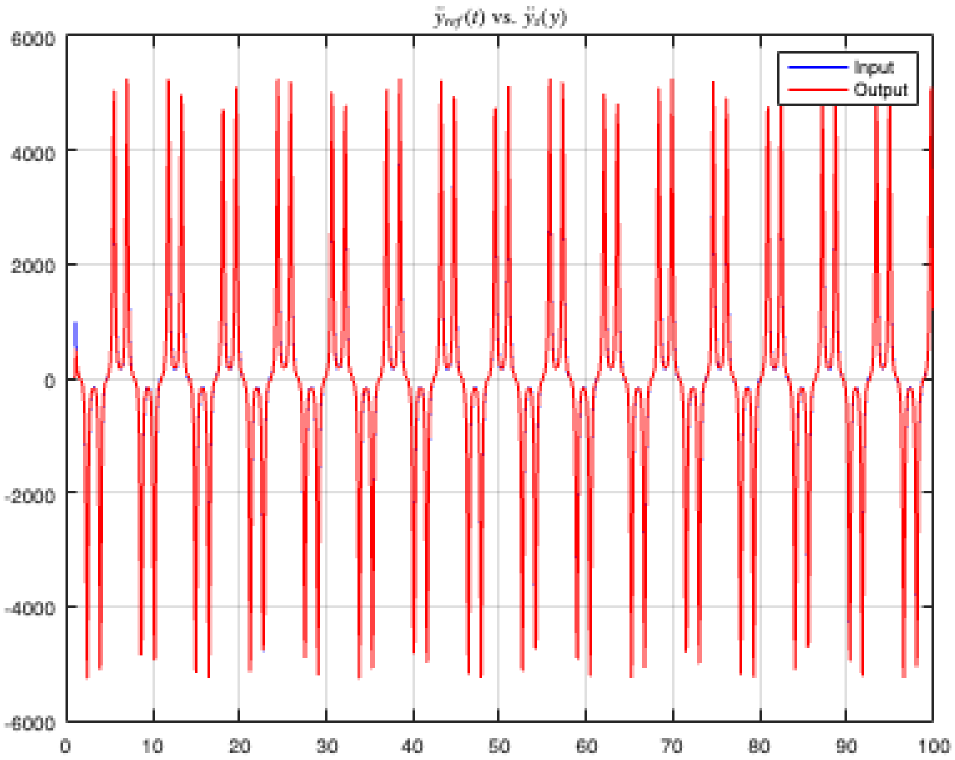


We then simulated it with the reference path.

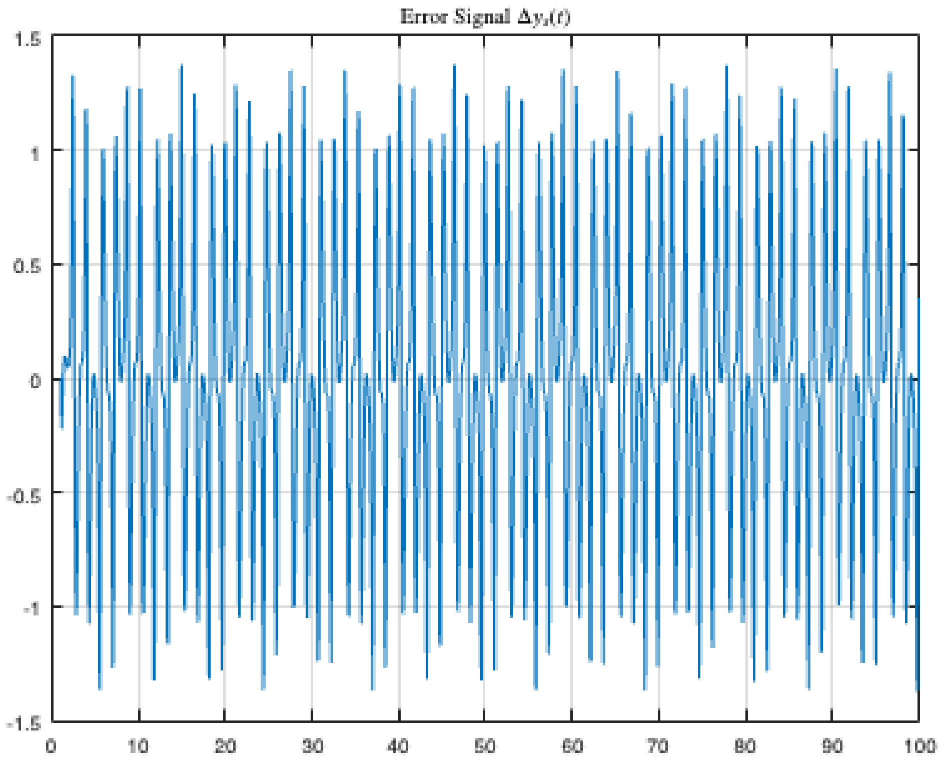
```
[y, t_out] = lsim(H_ll, y_ref, t);
figure;
plot(t, y_ref);
title("$\dot y_{ref}(t)$", 'Interpreter', 'latex');
```



```
figure;
plot(t, y_ref, 'b', 'DisplayName', 'Input');   % Plot input in blue
hold on;
plot(t, y, 'r', 'DisplayName', 'Output'); % Plot output in red
legend('show');
grid on;
title('$\ddot y_{ref}(t)$ vs. $\ddot y_s(y)$', 'Interpreter', 'latex')
hold off;
```



```
L_ll = -1/(s^2+F_ll*G);

[error, t_out] = lsim(L_ll, y_ref, t);
figure;
plot(t, error);
grid on;
title('Error Signal $\Delta y_s(t)$', 'Interpreter', 'latex');
```



```
disp("Median percent error:");
```

```
Median percent error:
```

```
disp(median(abs(y'-y_ref)/abs(y_ref)));
```

```
    0.0077
```

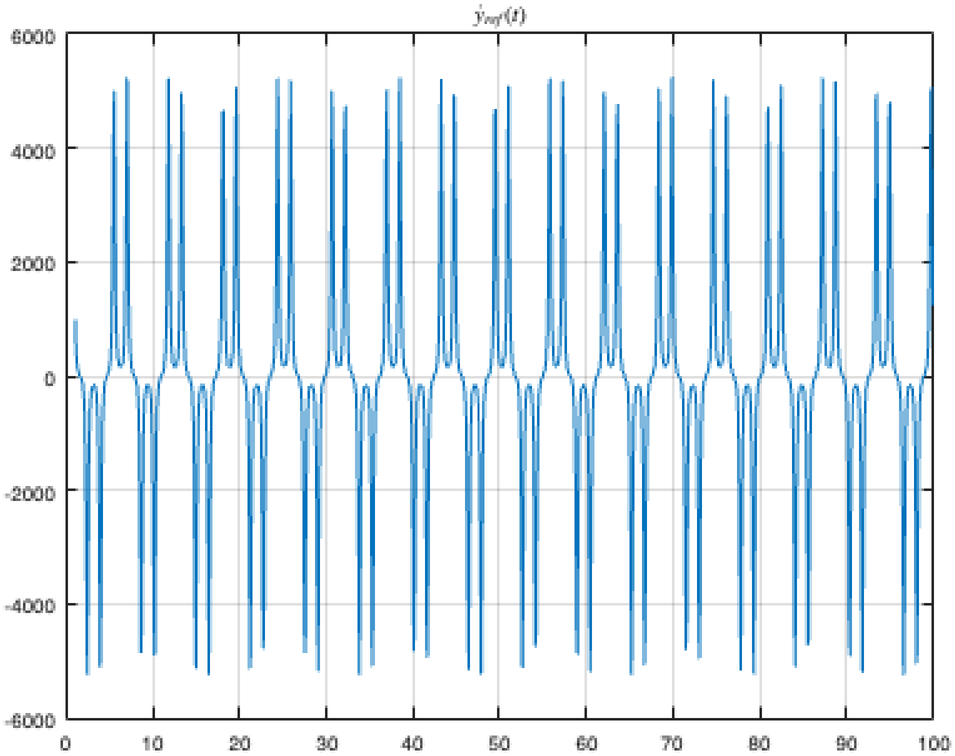
Testing Against Lissajous Curves

Test the system against different Lissajous Curves.

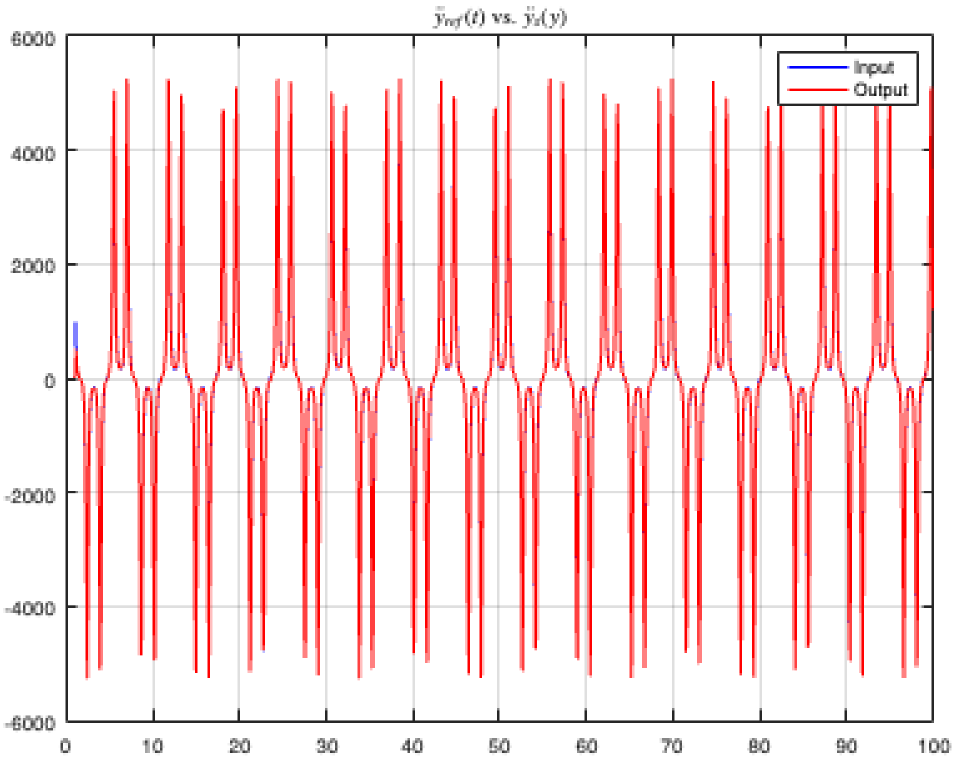
Curve 1: $a = 3,\ \ b = 2$

```
t = linspace(1, 100, 1000);
delta = pi/2;
a = 3;
b = 2;

rx_d = cos(t+delta);
rx_dd = -sin(t+delta);

ry_d = b*cos(b*t);
ry_dd = -b*b*sin(b*t);

p_ref = (rx_d.*ry_dd-ry_d.*rx_dd)./((rx_d.^2+ry_d.^2).^1.5);
```

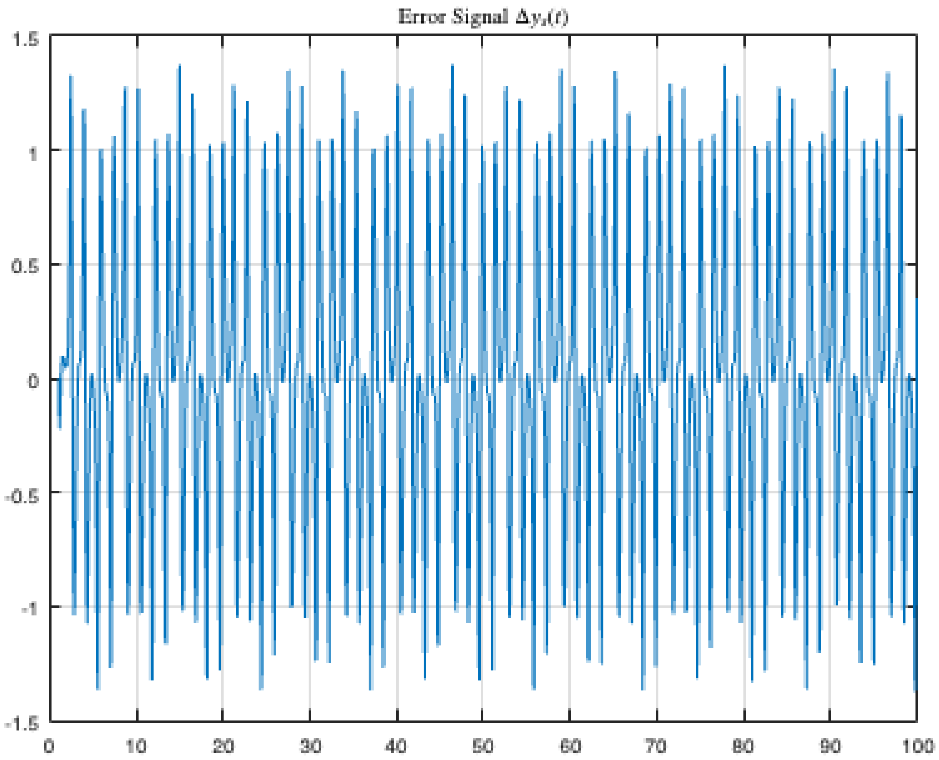
```
y_ref = p_ref*v^2;

[y, t_out] = lsim(H_ll, y_ref, t);
figure;
plot(t, y_ref);
title("$\dot y_{ref}(t)$", 'Interpreter', 'latex');
grid on;
```



```
figure;
plot(t, y_ref, 'b', 'DisplayName', 'Input');   % Plot input in blue
hold on;
plot(t, y, 'r', 'DisplayName', 'Output'); % Plot output in red
legend('show');
grid on;
title('$\ddot y_{ref}(t)$ vs. $\ddot y_s(y)$', 'Interpreter', 'latex')
hold off;
```



```
[error, t_out] = lsim(L_ll, y_ref, t);
figure;
plot(t, error);
grid on;
title('Error Signal $\Delta y_s(t)$', 'Interpreter', 'latex');
```



```
disp("Median percent error:");
```

```
Median percent error:
```

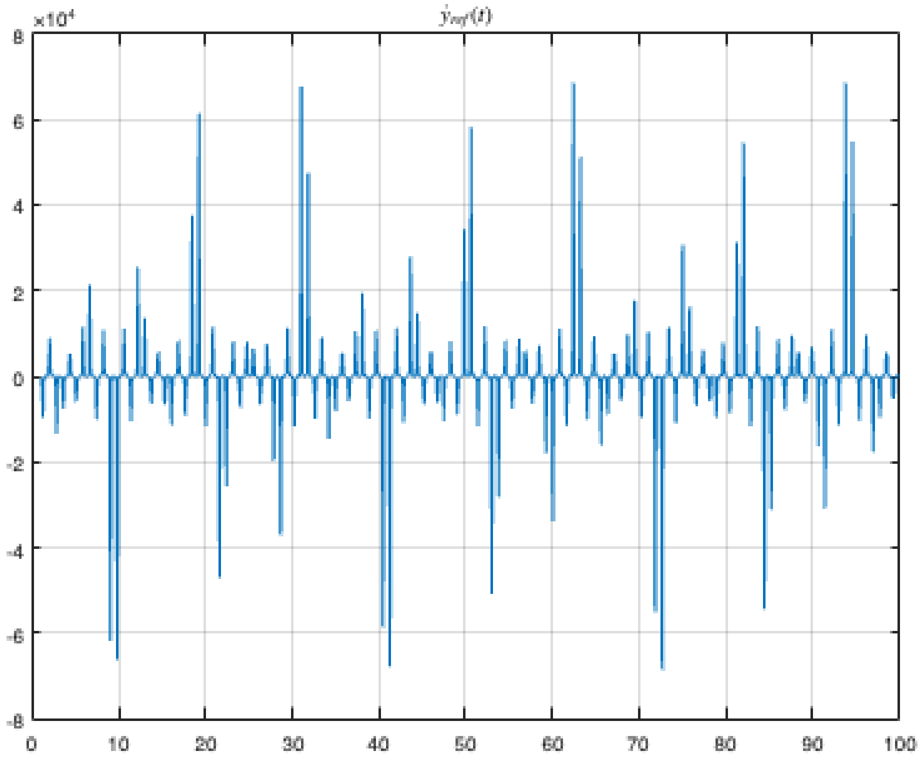
```
disp(median(abs(y'-y_ref)/abs(y_ref)));
```

```
    0.0077
```

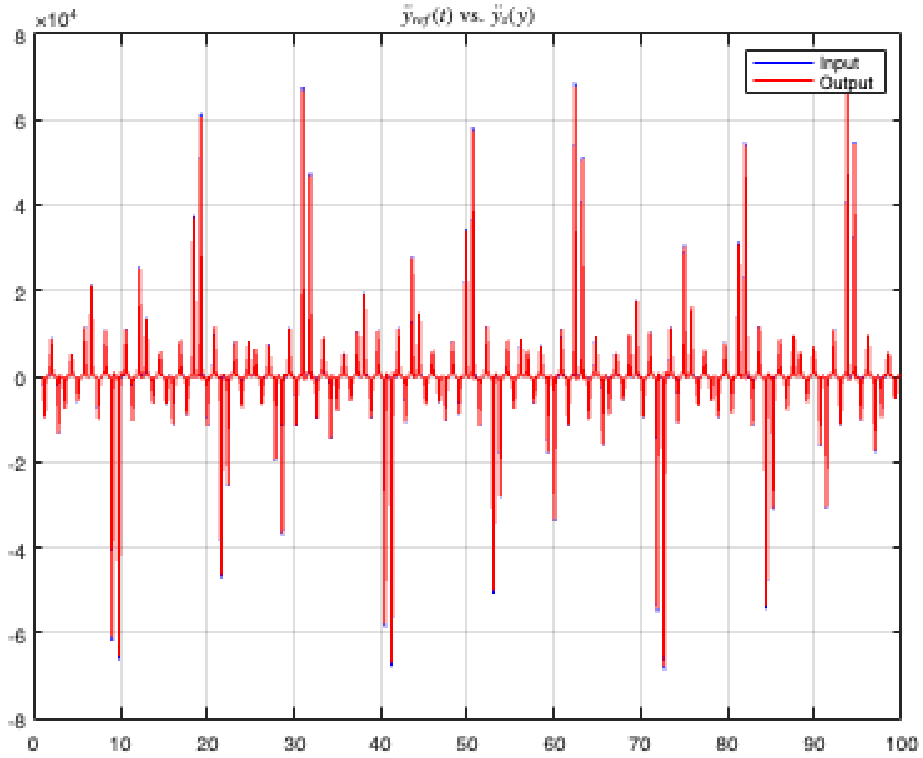
Curve 2: $a = 3,\ b = 4$

```
t = linspace(1, 100, 1000);
delta = pi/2;
a = 3;
b = 4;

rx_d = cos(t+delta);
rx_dd = -sin(t+delta);

ry_d = b*cos(b*t);
ry_dd = -b*b*sin(b*t);

p_ref = (rx_d.*ry_dd-ry_d.*rx_dd)./((rx_d.^2+ry_d.^2).^1.5);
y_ref = p_ref*v^2;

[y, t_out] = lsim(H_ll, y_ref, t);
figure;
plot(t, y_ref);
```

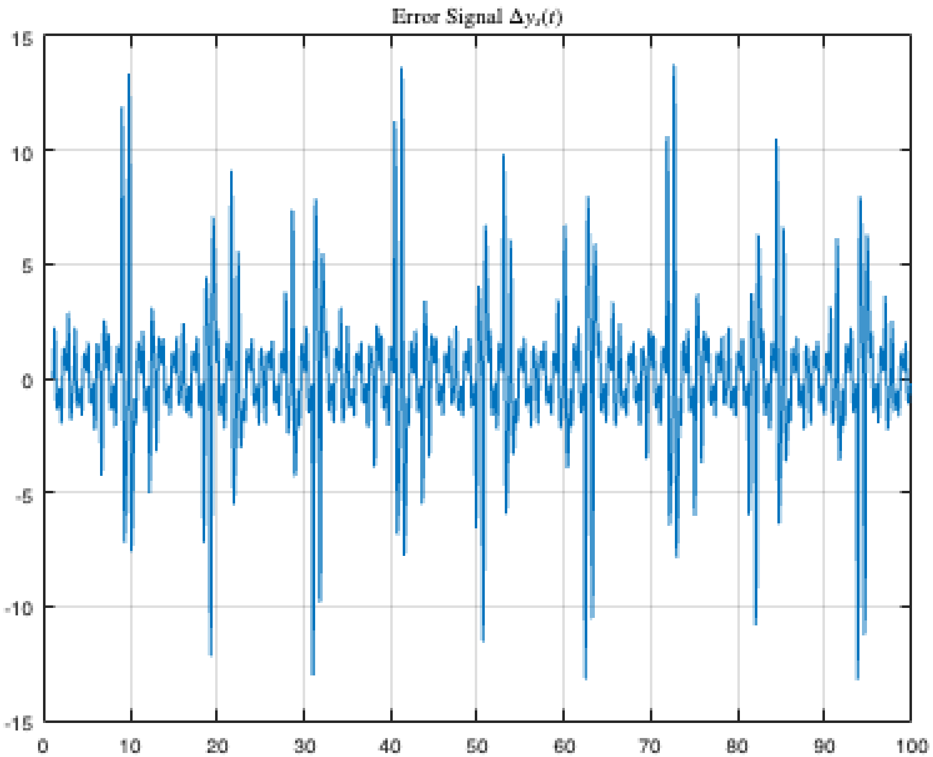
```
title("$\dot y_{ref}(t)$", 'Interpreter', 'latex');
grid on;
```



```
figure;
plot(t, y_ref, 'b', 'DisplayName', 'Input');   % Plot input in blue
hold on;
plot(t, y, 'r', 'DisplayName', 'Output'); % Plot output in red
legend('show');
grid on;
title('$\ddot y_{ref}(t)$ vs. $\ddot y_s(y)$', 'Interpreter', 'latex')
hold off;
```



```
[error, t_out] = lsim(L_ll, y_ref, t);
figure;
plot(t, error);
grid on;
title('Error Signal $\Delta y_s(t)$', 'Interpreter', 'latex');
```



```
disp("Median percent error:");
```

```
Median percent error:
```

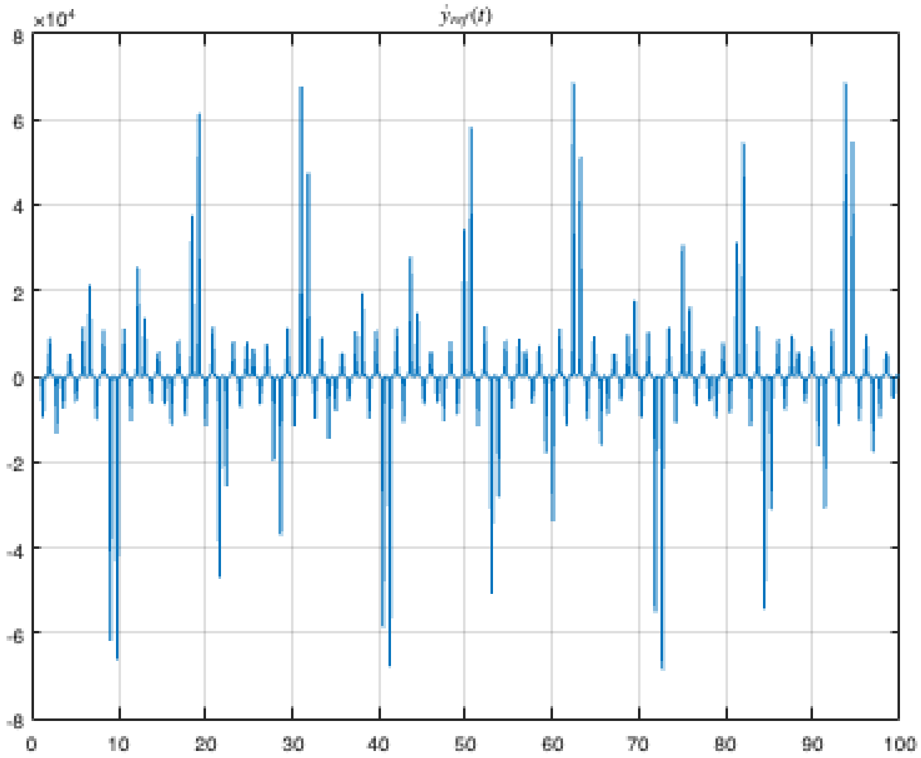
```
disp(median(abs(y'-y_ref)/abs(y_ref)));
```

```
    0.0127
```

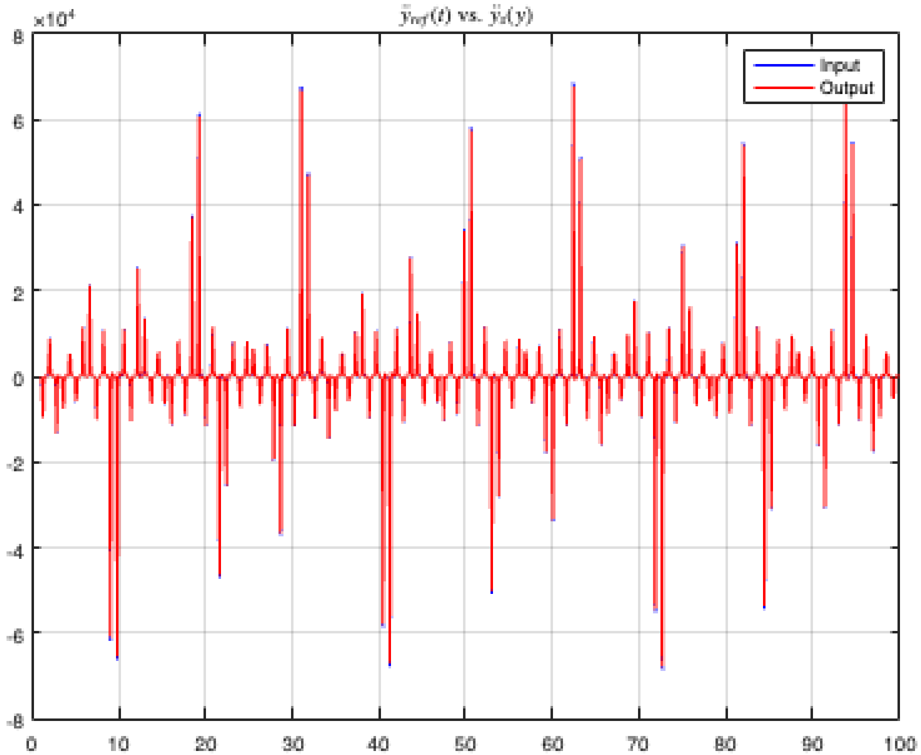
Curve 3: $a = 5,\ \ b = 4$

```
t = linspace(1, 100, 1000);
delta = pi/2;
a = 5;
b = 4;

rx_d = cos(t+delta);
rx_dd = -sin(t+delta);

ry_d = b*cos(b*t);
ry_dd = -b*b*sin(b*t);

p_ref = (rx_d.*ry_dd-ry_d.*rx_dd)./((rx_d.^2+ry_d.^2).^1.5);
y_ref = p_ref*v^2;

[y, t_out] = lsim(H_ll, y_ref, t);
figure;
```

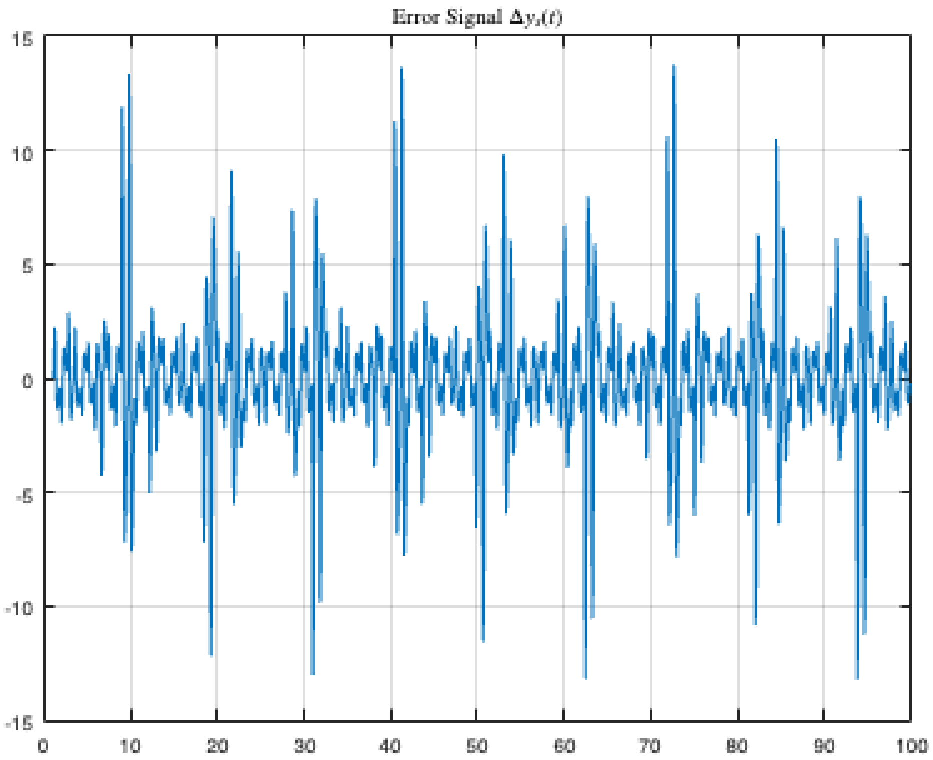
```
plot(t, y_ref);
title("$\dot y_{ref}(t)$", 'Interpreter', 'latex');
grid on;
```



```
figure;
plot(t, y_ref, 'b', 'DisplayName', 'Input');   % Plot input in blue
hold on;
plot(t, y, 'r', 'DisplayName', 'Output'); % Plot output in red
legend('show');
grid on;
title('$\ddot y_{ref}(t)$ vs. $\ddot y_s(y)$', 'Interpreter', 'latex')
hold off;
```



```
[error, t_out] = lsim(L_ll, y_ref, t);
figure;
plot(t, error);
grid on;
title('Error Signal $\Delta y_s(t)$', 'Interpreter', 'latex');
```



```
disp("Median percent error:");
```

```
Median percent error:
```

```
disp(median(abs(y'-y_ref)/abs(y_ref)));
```

```
    0.0127
```