Capstone: Modern Control Design and Analysis

Please provide a report on the analysis of the given transformation function based on the following questions, accompanied by thorough analysis and simulation using MATLAB:

$$G(s) = \frac{1.8s + 2.67}{s^3 + 9.25s^2 + 28.36s + 28.81}$$

1. System Analysis

1.1. Poles and Zeros

To fully understand the behavior of a linear system, it is essential to identify its poles and zeros. The poles and zeros of a system provide critical information about its stability, frequency response, and impulse response. We can gain insight into the system's characteristics and design appropriate control strategies by analyzing the location and multiplicity of the poles and zeros.

In MATLAB, the `tf` function can be used to compute the transfer function of a system, which can then be used to obtain the system's poles and zeros using the `pole` and `zero` functions.

```
% system transfer function
num = [1.8, 2.67];
den = [1, 9.25, 28.36, 28.81];
sys = tf(num, den);


% zeros and poles of the system
sys_zeros = zero(tf(num, den))
sys_poles = pole(tf(num, den))
nStates = max(size(sys_poles))
```

```
zeros = -1.4833
poles = 3×1
   -3.4599
   -3.1285
   -2.6616
nStates = 3
```

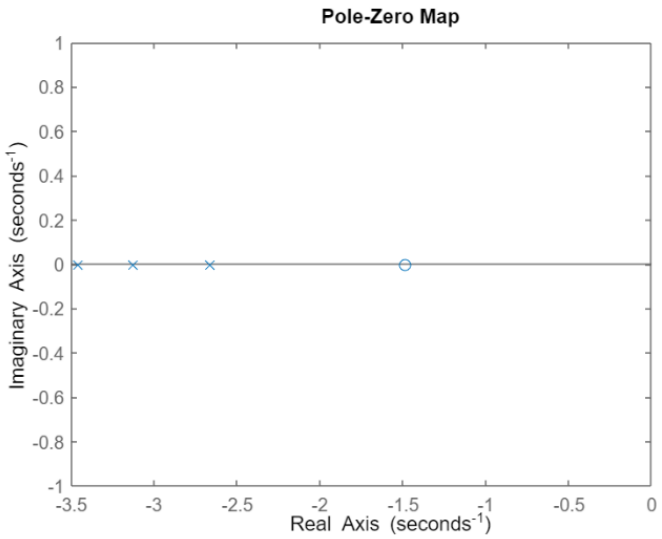
1.2. Stability

The input-output stability, also known as BIBO stability, of a linear system is a critical property that determines whether the output of the system remains bounded for any bounded input signal. We can determine the system's stability by

analyzing the location of its poles in the complex plane. Since all of the poles of the system have negative real parts or are located in the left-half plane (LHP), the system is input-output stable, and its output is guaranteed to remain bounded for any bounded input signal. This property is essential in control systems design, where stability is fundamental for the system's proper functioning.

The `pzplot` function in MATLAB can be used to visualize the location of the poles and zeros of a linear system in the complex plane.

```
% zeros and poles locations
pzplot(sys);
```

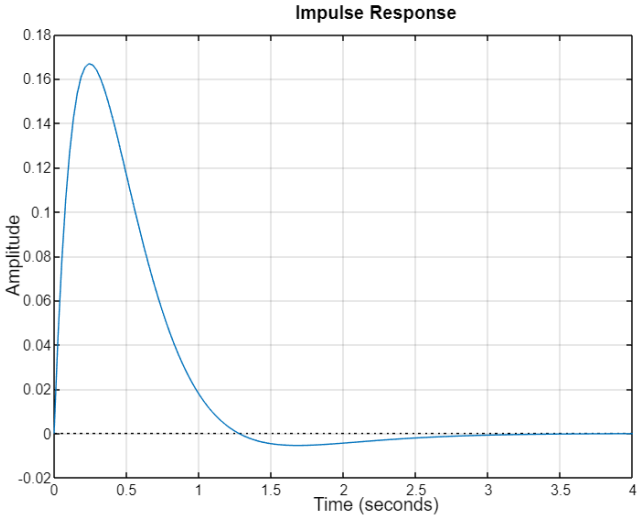


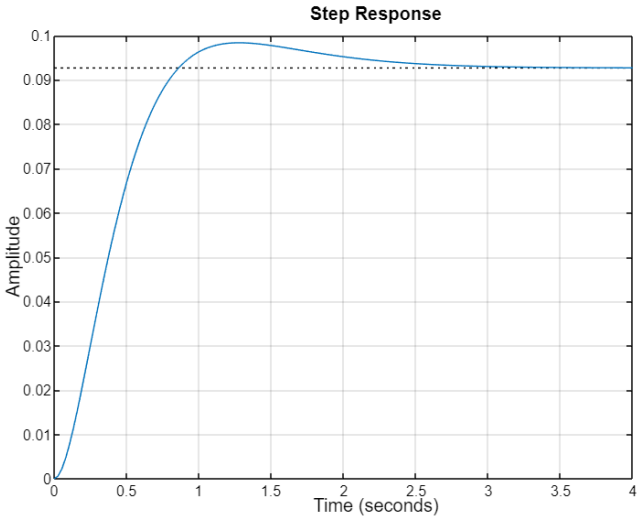
1.3. Phase Dynamics

In mathematical terms, a linear time-invariant system is said to be minimum-phase if the system and its inverse are causal and stable. A non-minimum phase system, on the other hand, is characterized by the presence of one or more zeros in the right-half plane (RHP). This property is diagnostic, as it allows us to identify non-minimum phase systems by looking at their zeros. Another interesting feature of non-minimum phase systems is that their step or impulse response initially moves in the opposite direction before settling toward the final output. This characteristic introduces a time delay, which can limit the response time and impact controller design and robustness. Consequently, non-minimum phase systems are generally more challenging to control and require specialized techniques such as inverse control or feedforward control.

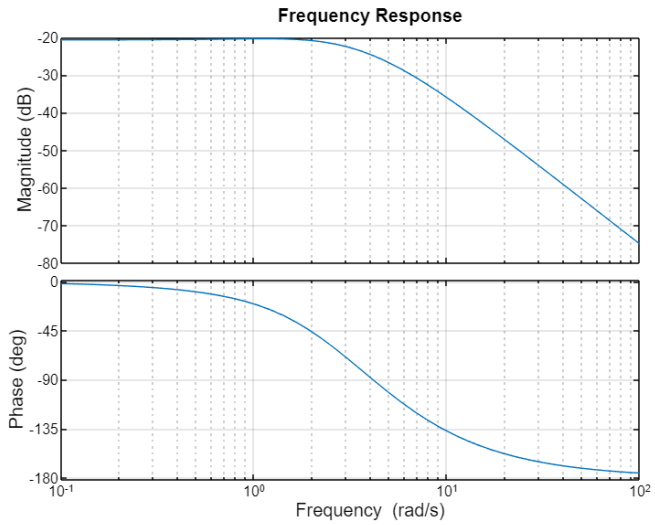
Based on the analysis of the poles and zeros of the system, we can conclude that the system is minimum-phase. we found that it has no zeros in the right-half plane (RHP), which is a necessary condition for minimum-phase systems. The absence of RHP zeros implies that the system and its inverse are both causal and stable. This property is desirable as it ensures that the system is well-behaved and can be effectively controlled using standard techniques.

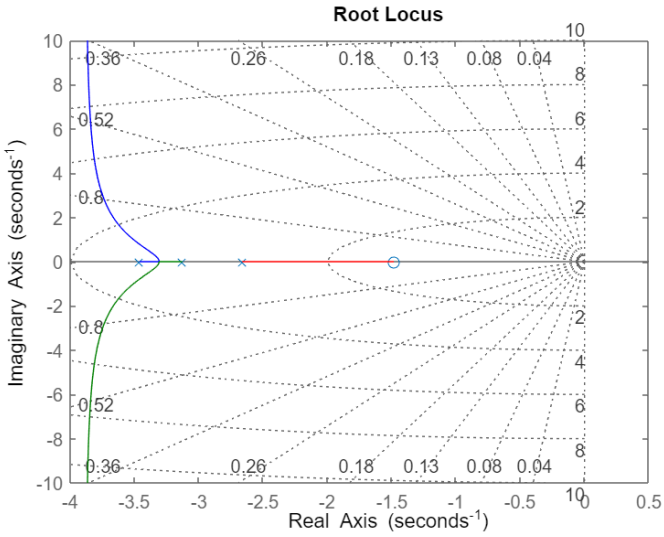
1.4. System's Characteristics

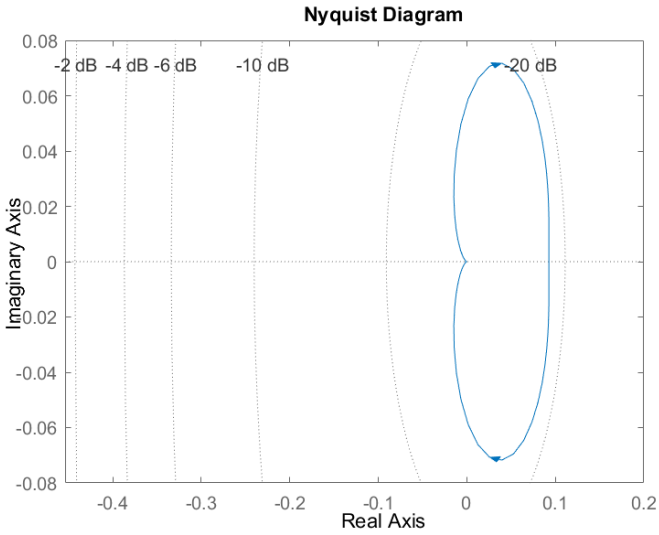





```
     RiseTime: 0.5627
TransientTime: 2.1762
 SettlingTime: 2.1762
   SettlingMin: 0.0842
   SettlingMax: 0.0984
     Overshoot: 6.1385
    Undershoot:      0
          Peak: 0.0984
      PeakTime: 1.2804
```

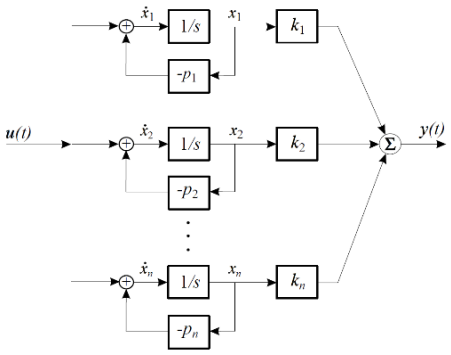

Frequency Response
Magnitude (dB)
Phase (deg)
Frequency (rad/s)


Root Locus
Imaginary Axis (seconds-1)
Real Axis (seconds-1)
0.36
0.26
0.18
0.13
0.08
0.04
0.52
0.8


Nyquist Diagram
-2 dB
-4 dB
-6 dB
-10 dB
-20 dB
Imaginary Axis
Real Axis

2. State-Space Representation

It is important to note that there is no unique state-space representation for a transfer function. The diagonal (Modal) canonical form is a particular representation that is useful for eigenvalue analysis and reduced order representation with physical interpretation. This form decomposes the transfer function into a diagonal matrix of poles and a matrix of residues that capture the system's response to each pole.



The following are the steps to realize the diagonal canonical form of the system's transfer function:

To determine the value of the feedforward matrix D in a state-space representation, we can rewrite the system's transfer function $G(s)$ in the following form. To obtain the value of D, we can evaluate the transfer function $G(s)$ as s approaches infinity.

$$G(s) = \hat{G}(s) + D$$

$$D = \lim_{s\to\infty} G(s) \Rightarrow \boxed{D = 0}$$

Find the residues of the transfer function $\hat{G}(s)$ using the partial fraction expansion.

$$G(s) = \frac{1.8s + 2.67}{s^3 + 9.25s^2 + 28.36s + 28.81} = \frac{a}{s + 3.4599} + \frac{b}{s + 3.1285} + \frac{c}{s + 2.6616}$$

In MATLAB, we can apply the partial fraction expansion to a transfer function using the `residue` function. The `residue` function takes the numerator and denominator coefficients of the transfer function as inputs and returns the residues, poles, and direct terms of the partial fraction expansion as outputs.

```
% calculating the residues in partial fraction expansion
[residues, ~, ~] = residue(num, den)
```

```
residues = 3×1
  -13.4512
   19.1417
   -5.6904
```

Thus, the partial fraction expansion of the system G(s) is shown below:

$$G(s) = \frac{Y(s)}{U(s)} = \frac{-13.4512}{s + 3.4599} + \frac{19.1417}{s + 3.1285} + \frac{-5.6904}{s + 2.6616}$$

Multiply the transfer function by the input signal $U(s)$ in the Laplace domain to obtain the output signal $Y(s)$.

$$\Rightarrow Y(s) = -13.4512 \cdot \boxed{\frac{U(s)}{s + 3.4599}} + 19.1417 \cdot \boxed{\frac{U(s)}{s + 3.1285}} - 5.6904 \cdot \boxed{\frac{U(s)}{5 + 2.6616}}$$

The state variables of the system can be identified as the outputs of the integrator blocks in the system block diagram. These state variables can be expressed in terms of the poles of the system and the input signal.

$$X_1(s) = \frac{U(s)}{s + 3.4599} \quad X_2(s) = \frac{U(s)}{s + 3.1285} \quad X_3(s) = \frac{U(s)}{s + 2.6616}$$

Using an inverse Laplace transform, we can obtain the state space representation of the system in the diagonal canonical form.

$$X_1(s) = \frac{U(s)}{s + 3.4599} \Rightarrow sX_1(s) + 3.4599X_1(s) = U(s) \xrightarrow{\mathcal{L}^{-1}} \boxed{\dot{x}_1(t) = -3.4599x_1(t) + u(t)}$$

$$X_2(s) = \frac{U(s)}{s + 3.1285} \Rightarrow sX_2(s) + 3.1285X_2(s) = U(s) \xrightarrow{\mathcal{L}^{-1}} \boxed{\dot{x}_2(t) = -3.1285x_2(t) + u(t)}$$

$$X_3(s) = \frac{U(s)}{s + 2.6616} \Rightarrow sX_3(s) + 2.6616X_3(s) = U(s) \xrightarrow{\mathcal{L}^{-1}} \boxed{\dot{x}_3(t) = -2.6616x_3(t) + u(t)}$$

$$Y(s) \xrightarrow{\mathcal{L}^{-1}} \boxed{y(t) = -13.4512 \cdot x_1(t) + 19.1417 \cdot x_2(t) - 5.6904 \cdot x_3(t)}$$

Thus, the state-space representation in diagonal canonical form is as follows:

$$\dot{x}(t) = \begin{pmatrix} -3.4599 & & \\ & -3.1285 & \\ & & -2.6616 \end{pmatrix} x(t) + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} u(t)$$

$$y(t) = (-13.4512 \quad 19.1417 \quad 5.6904)x(t)$$

As expected, in the Modal canonical form of the system $G(s)$, the system matrix A is a diagonal matrix of the poles of the system, while the output matrix C is the residue matrix obtained from the partial fraction expansion.

In MATLAB, we can use the `ss` function to construct a state-space representation of a linear time-invariant system as a system model. The `ss` function takes the state matrix A, input matrix B, output matrix C, and feedforward matrix D as inputs and returns a system model object that can be used for analysis and control design.

```
% obtaining diagonal representation
A = diag(sys_poles);
B = [1; 1; 1];
C = residues';
D = 0;

% concluding the state-space equations
sys_ss = ss(A, B, C, D)
```

```
sys_ss =

  A =
          x1       x2       x3
   x1   -3.46        0        0
   x2        0  -3.129        0
   x3        0        0  -2.662

  B =
       u1
   x1   1
   x2   1
   x3   1

  C =
          x1       x2       x3
   y1  -13.45    19.14    -5.69

  D =
       u1
   y1   0

Continuous-time state-space model.
```

However, the diagonal canonical form is just one of several possible representations. Other common alternative approaches include the observable canonical form and the controllable canonical form, which are widely used in control systems engineering. The observable canonical form emphasizes the observability of the system, while the controllable canonical form emphasizes the controllability of the system. Choosing the appropriate canonical form depends on the specific application and the desired properties of the system being analyzed. Therefore, it is important to carefully consider the advantages and limitations of each canonical form before making a choice.

MATLAB provides several built-in functions and tools for obtaining a state-space representation of a transfer function. This can be achieved using the `tf2ss` or the `canon` functions.

It is important to note that functions such as `tf2ss` in MATLAB assume that the given system is controllable. Therefore, before using these functions to convert between transfer function and state-space representations, the controllability of the

system should be checked using the controllability matrix. If the system is not fully controllable, then there may be some states that cannot be controlled by any input, and the state-space representation obtained from these functions may not be valid.

If there are uncontrollable modes in the system, we need to either modify the system to make it fully controllable or use alternative techniques to obtain a suitable state-space model. Modifying the system to make it controllable can involve adding additional inputs or outputs to the system or using other control design techniques to achieve controllability. Alternatively, we can use alternative state-space representations, such as the observable canonical form or the Jordan canonical form, which do not require the system to be fully controllable.

```
% state-space representation using tf2ss
[A, B, C, D] = tf2ss(num, den)
```

```
A = 3×3
   -9.2500  -28.3600  -28.8100
    1.0000         0         0
         0    1.0000         0
B = 3×1
     1
     0
     0
C = 1×3
         0    1.8000    2.6700
D =  0
```

```
% obtaining modal canonical form using canon
matlab_ss = canon(sys, "modal")
```

```
matlab_ss =

  A =
          x1       x2       x3
   x1   -3.46        0        0
   x2        0  -3.129        0
   x3        0        0  -2.662

  B =
          u1
   x1    -118
   x2  -181.2
   x3   63.92

  C =
             x1         x2         x3
   y1     0.114    -0.1056   -0.08902

  D =
       u1
   y1   0

Continuous-time state-space model.
```

3. Controllability and Observability

Controllability and observability are two important concepts in control theory that describe the ability to control and observe the states of a system, respectively.

3.1. Controllability

Controllability refers to the ability to steer the system from any initial state to any desired final state in a finite time. A system is said to be controllable if all its states can be controlled using the inputs available. The controllability of a system can be checked using the controllability matrix, which is a matrix that is constructed from the system's state matrix and input matrix. If the rank of the controllability matrix is equal to the number of states in the system, then the system is fully controllable.

The controllability matrix is defined as bellow:

$$\Phi_C = (B \quad AB \quad A^2B \quad \cdots)_n$$

In MATLAB, we can use the `ctrb` function to evaluate the controllability matrix of a linear time-invariant system. We can then use the `rank` function to check whether the controllability matrix is full rank, which indicates that the system is fully controllable.

```
% Controllability matrix
Co = ctrb(A, B)

%   Alternatively, we could calculate the controllability
%   matrix directly instead of using built-in functions
%   Co = [B, A*B, A^2*B]

rank_co = rank(Co)
```

```
Co = 3×3
    1.0000   -9.2500   57.2025
         0    1.0000   -9.2500
         0         0    1.0000
rank_co = 3
```

Since the rank of the controllability matrix is equal to the number of state variables, then the system is fully controllable, and there are no uncontrollable modes. Thus, the fully controllable system cannot be decomposed into controllable and uncontrollable modes. If a system is fully controllable, then all its states can be controlled by applying suitable inputs, and there are no uncontrollable modes in the system.

3.2. Observability

Observability, on the other hand, refers to the ability to determine the internal states of the system by measuring its outputs. A system is said to be observable if all its states can be observed from the outputs of the system. The observability of a system can be checked using the observability matrix, which is a matrix that is constructed from the system's state matrix and output matrix. If the rank of the

observability matrix is equal to the number of states in the system, then the system is fully observable.

The observability matrix is defined as bellow:

$$\Phi_O = \begin{pmatrix} C \\ CA \\ CA^2 \\ \vdots \end{pmatrix}_n$$

In MATLAB, we can use the `obsv` function to evaluate the observability matrix of a linear time-invariant system. We can then use the `rank` function to check whether the observability matrix is full rank, which indicates that the system is fully observable.

```
% Observability matrix
Ob = obsv(A, C)

%   Alternatively, we could calculate the observability
%   matrix directly instead of using built-in functions
%   Ob = [C; C*A; C*A^2]

rank_ob = rank(Ob)
```

```
Ob = 3×3
         0    1.8000    2.6700
    1.8000    2.6700         0
  -13.9800  -51.0480  -51.8580

rank_ob = 3
```

Since the rank of the observability matrix is equal to the number of state variables, then the system is fully observable, and there are no unobservable modes. Thus, the fully observable system cannot be decomposed into observable and unobservable modes. If a system is fully observable, then all its states can be observed from the output, and there are no unobservable modes in the system.

Duality

It is important to note that controllability and observability are regarded as duals in the conventional input-state-output formulation of dynamic systems. This means that there is a concrete relationship between the two. We can say that a system (A, B) is controllable if and only if the system (A', C, B', D) is observable. The duality between controllability and observability is expressed mathematically by the Kalman's decomposition theorem. According to this theorem, any linear time-invariant system can be decomposed into a controllable subsystem and an observable subsystem. The controllable subsystem can be controlled to any desired state, while the observable subsystem can be uniquely determined from the output.

4. Minimal State-Space Representation

One way to check whether a state-space model is minimal is to examine its Jordan form. The Jordan form of a matrix is obtained by transforming the matrix into a block diagonal form with each block being a Jordan block. A Jordan block is a square matrix with a constant diagonal and 1's on the super diagonal.

A state-space representation is said to be minimal if its state matrix A has a Jordan form containing as many Jordan blocks as the number of state variables. This implies that the system has been represented with the smallest possible number of state variables that fully capture its dynamics, which can lead to more efficient and accurate analysis and control design.

To obtain the Jordan form of a state-space model, you can use the `jordan` function in MATLAB. The `jordan` function computes the Jordan form of a matrix and returns two matrices: V and J. The matrix J is the Jordan form of the matrix, and the matrix V is a matrix of generalized eigenvectors that transforms the original matrix into its Jordan form. Here, because the state matrix of the system was diagonalizable, the Jordan and the Modal forms are equivalent. Thus, both forms result in a diagonal matrix.

```
% obtaining the jordan form
[V, J] = jordan(A)
```

```
V = 3×3
     1     0     0
     0     1     0
     0     0     1

J = 3×3
   -3.4599         0         0
         0   -3.1285         0
         0         0   -2.6616
```

To conclude, according to the state matrix A and its Jordan form, the obtained state-space realization is *minimal*. The minimal characteristic function is as bellow:

$$|\lambda I - A| = (\lambda + 3.4599) \cdot (\lambda + 3.1285) \cdot (\lambda + 2.6616)$$

```
% evaluating the eigenvalues of the system
[eigvectors, eigvalues] = eig(A)
```

```
eigvectors = 3×3
     1     0     0
     0     1     0
     0     0     1

eigvalues = 3×3
   -3.4599         0         0
         0   -3.1285         0
         0         0   -2.6616
```

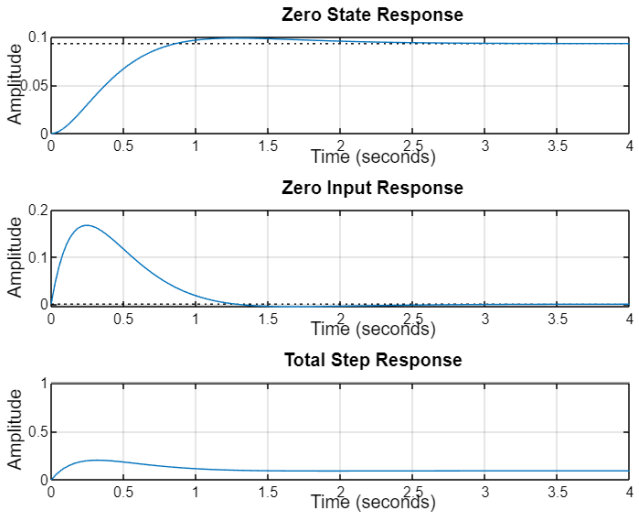
5. Step Response of the Open-Loop System

The step response of a state-space system in its zero state, i.e., with all initial conditions set to zero, is equivalent to the step response of the corresponding transfer function. This is because they both exhibit the same underlying system characteristics.

To observe the system response, we will plot the step response with arbitrary initial conditions $x_0 = (1 \quad 1 \quad 1)$. This will allow us to witness the behavior of the system under these conditions.

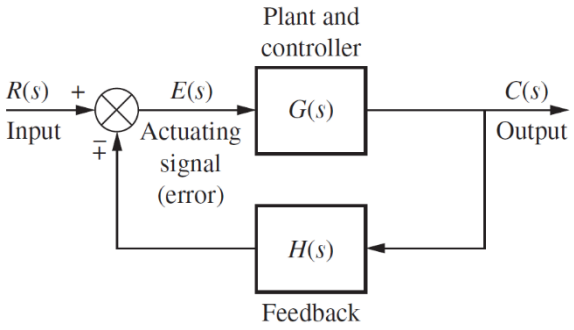
In MATLAB, we can utilize the `initial` and `lsim` functions to plot both the zero-input response and total response of a system. The `initial` function is used to set the initial conditions of the state-space model, while the `lsim` function is used to simulate the system response to an arbitrary input signal.

```
tFinal = 4;
t = 0:0.01:tFinal;    % time vector t = [0 ~ tFinal]
u = ones(size(t));    % step input u(t) = 1; for t>0
x0 = [1 1 1];         % arbitrary initial conditions

subplot(3, 1, 1);
step(sys_ss, tFinal, plotoptions);
title('Zero State Response');

subplot(3, 1, 2);
initial(sys_ss, x0, tFinal, plotoptions)
title('Zero Input Response');

subplot(3, 1, 3);
lsim(sys_ss, u, t, x0, plotoptions);
title('Total Step Response');
```



6. Negative Unity Feedback

Negative unity feedback, also known as negative feedback, is a control system configuration in which the output of a system is fed back to the input with a negative sign. In other words, the feedback signal is subtracted from the input signal to form the error signal, which is then used to adjust the system's behavior.

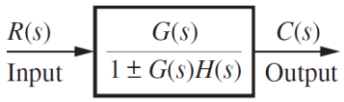


In MATLAB, the `feedback` function allows us to specify the plant transfer function and the feedback transfer function, and it returns the closed-loop transfer function for the entire system.

```
% negative feedback gain
K = 1;

% negative unity feedback loop
sys_cl = feedback(sys_ss, K);
```

```
sys_cl =

  A =
          x1       x2       x3
   x1   9.991  -19.14     5.69
   x2   13.45  -22.27     5.69
   x3   13.45  -19.14    3.029

  B =
       u1
   x1   1
   x2   1
   x3   1

  C =
          x1       x2       x3
   y1 -13.45    19.14    -5.69

  D =
       u1
   y1   0

Continuous-time state-space model.
```

Additionally, instead of using the `feedback` function, we can mathematically derive the closed-loop transfer function:



Where G(s) is the transfer function of the plant or process being controlled and H(s) is the transfer function of the feedback loop.

```
% negative unity feedback transfer function
g_cl = sys / (1 + (K * sys))
```

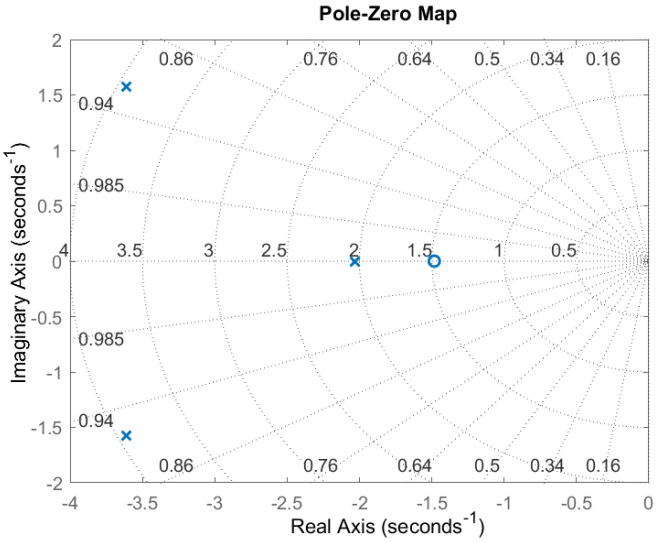
```
g_cl =

            1.8 s + 2.67
  --------------------------------
  s^3 + 9.25 s^2 + 30.16 s + 31.48

Continuous-time transfer function.
```

When comparing the mathematically derived closed-loop system and the closed-loop system obtained using the `feedback` function, we observe that both have the same poles. This indicates that they share the same system characteristics.

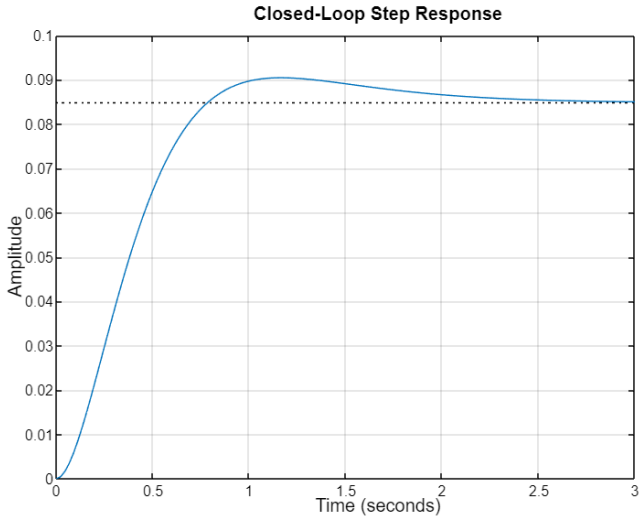
```
% poles and zeros of the closed-loop system
poles_cl = pole(sys_cl)
zeros_cl = zero(sys_cl)
```

```
poles_cl = 3×1 complex
  -3.6092 + 1.5712i
  -3.6092 - 1.5712i
  -2.0316 + 0.0000i
zeros_cl = -1.4833
```

```
pzmap(sys_cl);
```



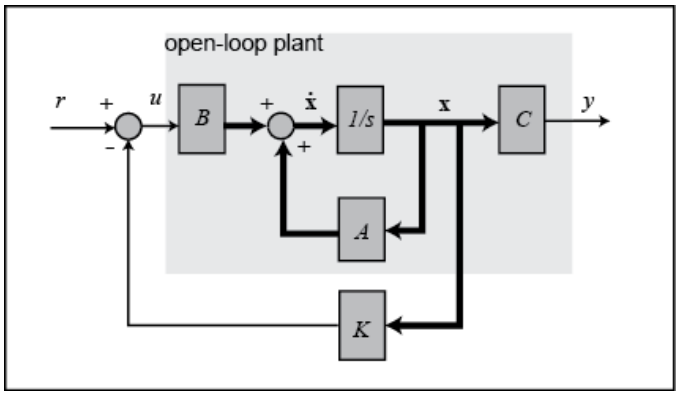
```
% closed-loop system's step response
figure;
step(sys_cl, plotoptions);
title('Closed-Loop Step Response');
```




```
        RiseTime: 0.5157
    TransientTime: 2.0355
     SettlingTime: 2.0355
      SettlingMin: 0.0768
      SettlingMax: 0.0905
        Overshoot: 6.6718
       Undershoot:      0
             Peak: 0.0905
         PeakTime: 1.1560
```

7. State Feedback Control Design

State feedback is a control technique that uses the states of a system to generate a control signal. In state feedback, the states of the system are measured and fed back to a controller, which uses this information to generate a control input that drives the system towards a desired state or output.



Mathematically, we can represent the state feedback control law as follows:

$$u(t) = -kx(t)$$

where $u(t)$ is the control input at time t, $x(t)$ is the state vector of the system at time t, and K is the state feedback gain matrix.

The state feedback gain matrix K is typically determined using pole placement techniques, which involve selecting the poles of the closed-loop system to achieve desired performance characteristics, such as stability and response time. The gain matrix is then computed such that the resulting closed-loop system has the desired pole locations. The desired characteristic function Δ_d can be written as:

$$\Delta_d = (s - p_1)(s - p_2) \dots (s - p_n)$$

Where p_i represents the desired pole locations of the closed-loop system.

The state feedback control law can be obtained by substituting $u(t)$ from the first equation into the state-space model, which yields:

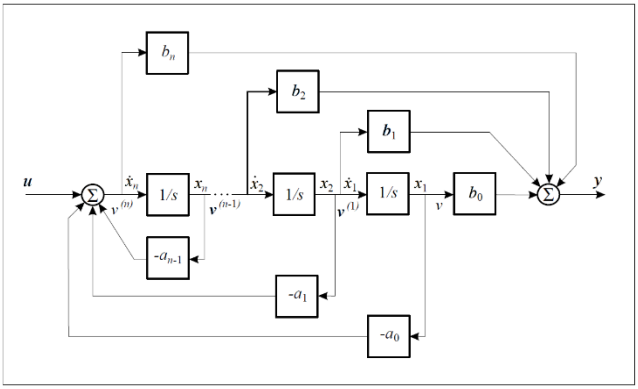
$$\dot{x}(t) = (A - BK)\, x(t)$$

$$y(t) = Cx(t) + Du(t)$$

Where $(A - BK)$ is the closed-loop system matrix A_c. Once we have the desired pole locations, we can then solve the following equation to find the state feedback gain matrix K:

$$|sI - (A - BK)| = \Delta_d$$

If a system is in controllable canonical form, designing a state feedback controller becomes easier because the controllability of the system can be easily verified, and the feedback gain matrix can be directly computed from the system matrix. Thus, we obtain the Controllable canonical form for the system.



Since we have already verified that our system is controllable, we can use the `tf2ss` function to obtain the controllable canonical form of the state-space representation.

```
% controllable canonical form
[Ac, Bc, Cc, Dc] = tf2ss(num, den);
sys_controllable = ss(Ac, Bc, Cc, Dc)
```

```
sys_controllable =

  A =
         x1       x2       x3
   x1  -9.25   -28.36   -28.81
   x2      1        0        0
   x3      0        1        0

  B =
       u1
   x1   1
   x2   0
   x3   0

  C =
         x1     x2     x3
   y1     0    1.8   2.67

  D =
       u1
   y1   0

Continuous-time state-space model.
```

Firstly, we will perform a close pole placement by selecting desired poles that are relatively close to the origin. The choice of close pole placement is appropriate when we want to achieve a response with minimal overshoot. Suppose the criteria for the controller were $T_s \leq 1s$ and $\%O.S. \leq 5\%$; where approximately $\xi \geq 0.7$ and $\omega_n \geq 5$.

We might try to place two dominant poles at $-5 \pm 4i$, and the third pole sufficiently fast that it won't have much effect on the response. However, the zero of the system would increase the overshoot. To solve this issue, we employed the third pole close to the zero at -1.4833.

$$P_d = \{-5 - 4i, -5 + 4i, -1.5\}$$

$$\Delta_d = \prod_{i=1}^{n}(s - P_i) = s^3 + 11.5s^2 + 56s + 61.5$$

Then, we evaluate the closed-loop system matrix A_c:

$$A = \begin{pmatrix} -9.25 & -28.36 & -28.81 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad K = (k_1 \quad k_2 \quad k_3)$$

$$\Rightarrow \quad A - BK = \begin{pmatrix} -9.25 - k_1 & -28.36 - k_2 & -28.81 - k_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

To achieve state feedback gain matrix K we need to ensure that the new system characteristic function is equal to the desired system characteristic function with the desired poles.

$$|sI - A_c| = \begin{vmatrix} s + (9.25 + k_1) & 28.36 + k_2 & 28.81 + k_3 \\ -1 & s & 0 \\ 0 & -1 & s \end{vmatrix}$$

$$\Rightarrow |sI - A_c| = s^3 + (9.25 + k_1)s^2 + (28.36 + k_2)s + (28.81 + k_3)$$

$$|sI - A_c| = \Delta_d \quad \Rightarrow \quad \boxed{K = (2.25 \quad 27.64 \quad 32.69)}$$

In MATLAB, we can use the `place` function to calculate the state feedback gain matrix for a given system. This function uses pole placement techniques to determine the gain matrix that places the closed-loop system poles at the desired locations. Additionally, the function `acker` achieves the same goal, but the method can become computationally expensive for higher-order systems, and the accuracy of the desired pole locations may be affected by numerical issues.

```
% the desired poles
p_desired_close = [-5-4i, -5+4i, -1.5];

% the feedback gain matrix
K_close = place(Ac, Bc, p_desired_close)
```

```
K_close = 1×3
    2.2500   27.6400   32.6900
```

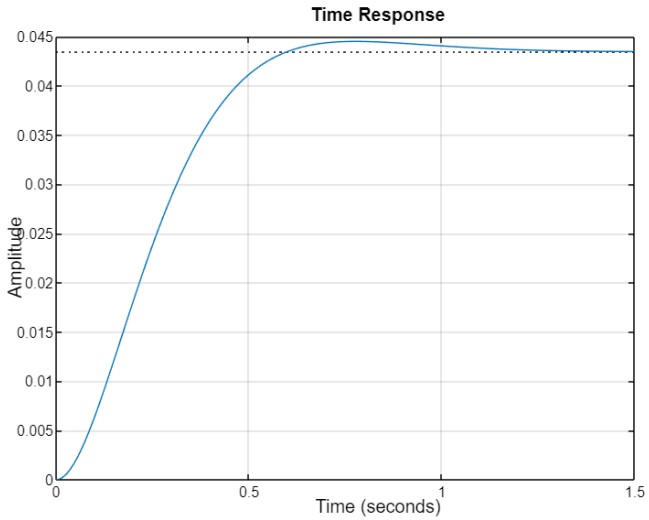
We can verify the closed-loop pole locations using the `eig` function.

```
p_close = eig(Ac - Bc*K_close)
```

```
p_close = 3×1 complex
  -5.0000 + 4.0000i
  -5.0000 - 4.0000i
  -1.5000 + 0.0000i
```

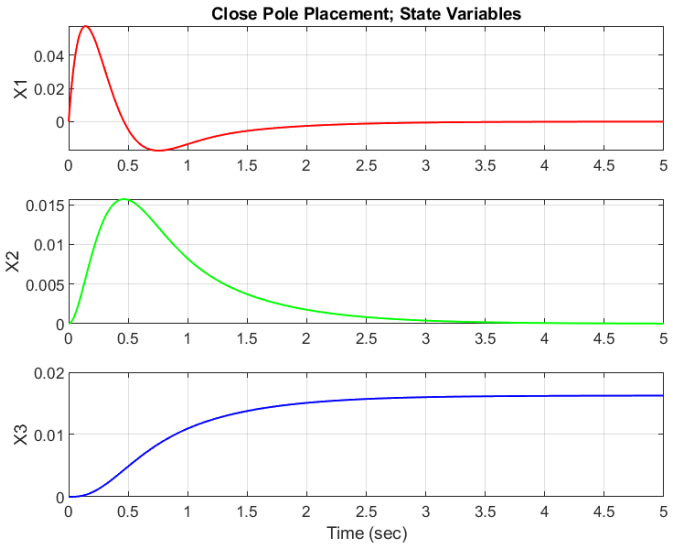
We now plot the step response of the closed-loop system. Although the new system has a slower response compared to the open-loop system, it exhibits no overshoot and is overdamped. Additionally, it is interesting to note that the zeros of the system did not change even when state feedback was applied. This indicates that the state feedback controller did not affect the system's transmission zeros.

```
% the closed-loop system with state feedback
sys_close = ss(Ac - Bc*K_close, Bc, Cc, Dc);

% the closed-loop system's step response
figure;
step(sys_close, plotoptions);
```




```
     RiseTime: 0.3680
TransientTime: 0.9097
  SettlingTime: 0.9097
   SettlingMin: 0.0393
   SettlingMax: 0.0445
     Overshoot: 2.5081
    Undershoot: 0
          Peak: 0.0445
      PeakTime: 0.7737
```

```
tFinal = 5;
t = 0:0.01:tFinal;      % time vector t = [0 ~ tFinal]
u = ones(size(t));      % step input u(t) = 1; for t>0
x0 = [0; 0; 0];         % zero state initial condition

[~, ~, x_close] = lsim(sys_close, u, t, x0);

figure;
subplot(3,1,1);
plot(t, x_close(:,1), 'r');
ylabel('X1');

subplot(3,1,2);
plot(t, x_close(:,2), 'g');
ylabel('X2');

subplot(3,1,3);
plot(t, x_close(:,3), 'b');
ylabel('X3');
xlabel('Time (sec)');
```



In general, the behavior of the state variables depends on the system dynamics, the controller design, and the initial conditions. If the system is controllable and observable, it is possible to design a state feedback controller that stabilizes the system and drives all the state variables to zero. However, in some cases, it may not be possible to drive all the state variables to zero due to system asymmetry.

System asymmetry or coupling refers to situations where the system's behavior depends on the direction in which a particular input is applied. In other words, the system does not behave symmetrically with respect to its inputs. Asymmetry can arise due to various reasons, such as the presence of physical asymmetries in the system, nonlinearities, or measurement noise.

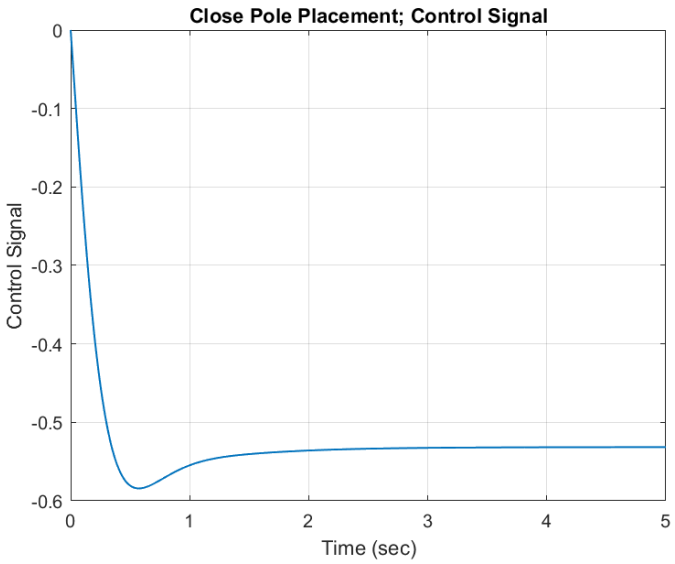
System asymmetry can pose challenges to the design of a state feedback controller. If the system is inherently asymmetric, it may be difficult to stabilize all the state variables simultaneously, and some may converge to non-zero values. In such cases, the state feedback controller may need to be designed to stabilize certain state variables while allowing others to converge to non-zero values.

In our case, in the open-loop system, all state variables converged to a non-zero value. However, in the state feedback controller design, the state variable x_3 appears to still converge to a non-zero value, while the other state variables converge to zero. This could be attributed to the system's inherent asymmetry, as explained earlier.

As mentioned earlier, the state variable x_3 converges to a non-zero value, while the other variables converge to zero. Additionally, we established that the control signal $u = -kx$ is a function of the state variables and the state feedback gains. Therefore, as time approaches infinity, the only parameter affecting the control signal is the state variable x_3. We expect the control signal to converge to a non-zero value, which is the product of k_3 and x_3 at infinity.

We can observe the behavior of the control signal in the plot below:

```
figure;
plot(t, -K_close*x_close');
xlabel('Time (sec)');
ylabel('Control Signal');
```

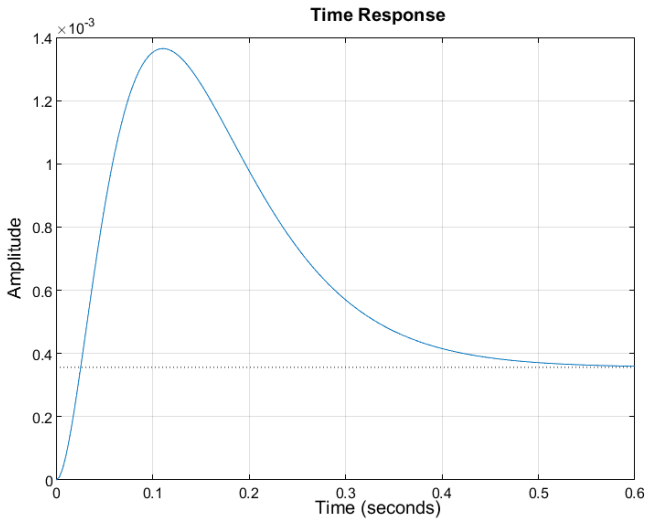


In the next section, we compare the results of a far pole placement with those of a close pole placement. Far pole placement resulted in a faster response, although the response may be adversely oscillatory.

```
% the desired poles
p_desired_far = [-25, -20, -15];

% the feedback gain matrix
K_far = place(Ac, Bc, p_desired_far)
```

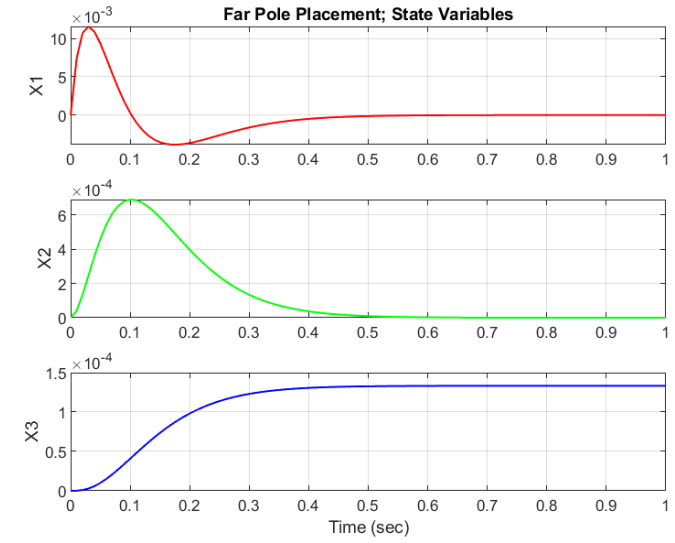
```
K_far = 1×3
10^3 ×
    0.0508    1.1466    7.4712
```

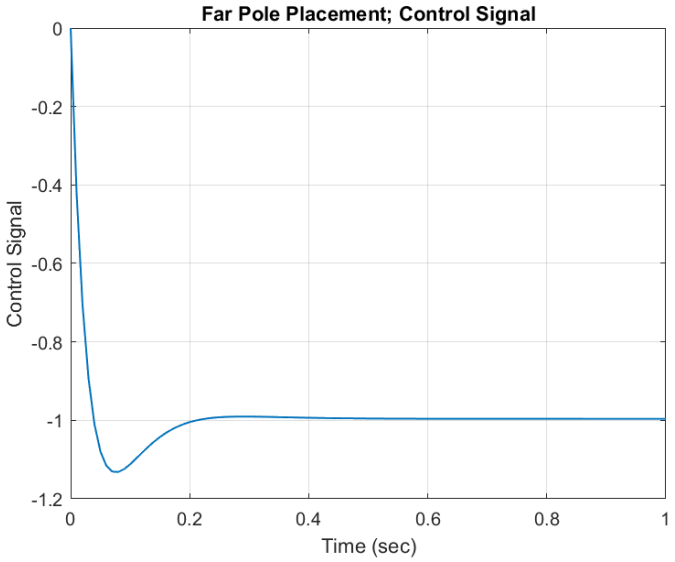
```
sys_far = ss(Ac - Bc*K_far, Bc, Cc, Dc);

% Plot the closed-loop system's step response
figure;
step(sys_far, plotoptions);
```




```
     RiseTime:     0.0171
TransientTime:     0.4791
 SettlingTime:     0.5527
  SettlingMin: 3.3720e-04
  SettlingMax:     0.0014
    Overshoot:   283.3890
   Undershoot:          0
         Peak:     0.0014
     PeakTime:     0.1105
```

After performing the far pole placement, we observed that the state feedback gain matrix values significantly increased. The faster response came at the cost of a transient response with some detrimental overshoot. We traced this adverse oscillation to the zero that is relatively close to the jω-axis. To mitigate this phenomenon, a pole close to the zero is suggested. While this might increase the settling time, it reduces the overshoot. Additionally, the control signal for a far pole placement required more effort compared to close pole placement.



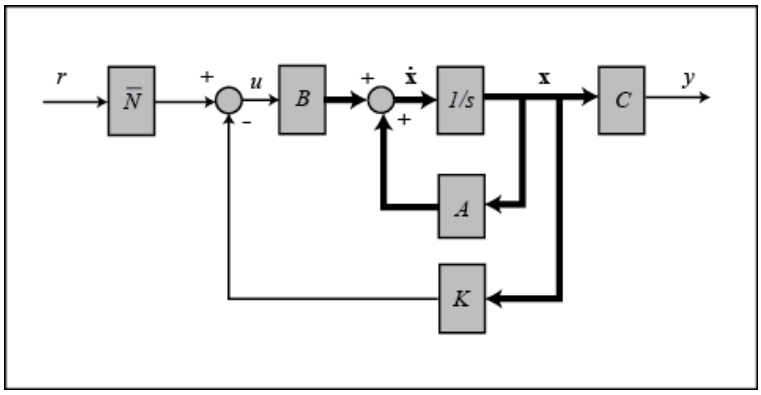


Overall, far pole placement can result in a faster response however might require greater effort and may have a transient response with unacceptable oscillation. Close pole placement can result in a smoother response with less effort but may have a longer settling time.

8. Static Tracker

After achieving a satisfactory transient response and analyzing the system output based on the close and far pole placement, the next step is to address the steady-state response. The step response of the close pole placement state-feedback controller indicates poor step tracking performance.

This is because the output is not compared to the reference directly; instead, all states are measured, multiplied by the gain vector K, and then subtracted from the reference. Consequently, there is no guarantee that $u = -Kx$ will be equal to the desired output. To address this issue, we can introduce a scaling factor $\bar{N}$ in the reference input to ensure that it matches the steady-state Kx. This scaling factor is illustrated in the following diagram:



To obtain the scaling factor $\bar{N}$ for the reference input, we need to find the inverse of the closed-loop transfer function as s approaches zero.

$$\bar{N} = \lim_{s \to 0} {G_{cl}}^{-1}(s)$$

$$G_{cl}(s) = \frac{1.8s + 2.67}{s^3 + 11.5s^2 + 56s + 61.5}$$

$$\Rightarrow \bar{N} = \lim_{s \to 0} \frac{s^3 + 11.5s^2 + 56s + 61.5}{1.8s + 2.67} = \frac{61.5}{2.67}$$

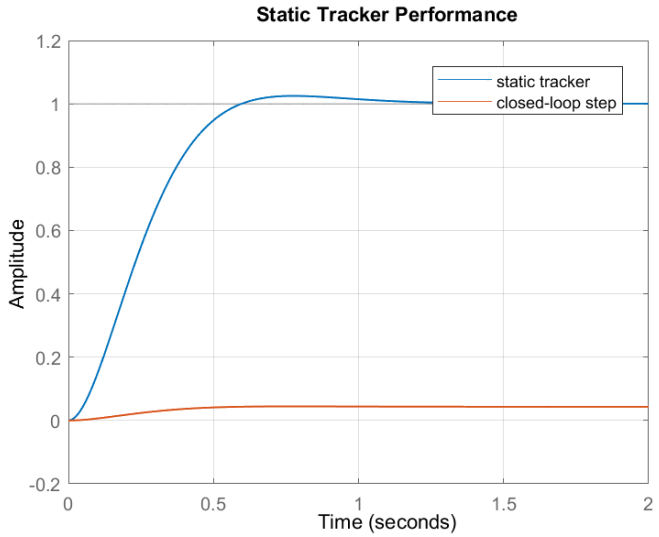
$$\Rightarrow \boxed{\bar{N} = 23.0337}$$

It is important to note that the solution to the static tracker equation we have provided is valid only if the transfer function of the system has no zeros at the origin.

```
% Calculate the static scaling factor
desired_output = 1;
n_bar = desired_output / (Cc*inv(-(Ac-Bc*K_close))*Bc)
```

```
n_bar = 23.0337
```

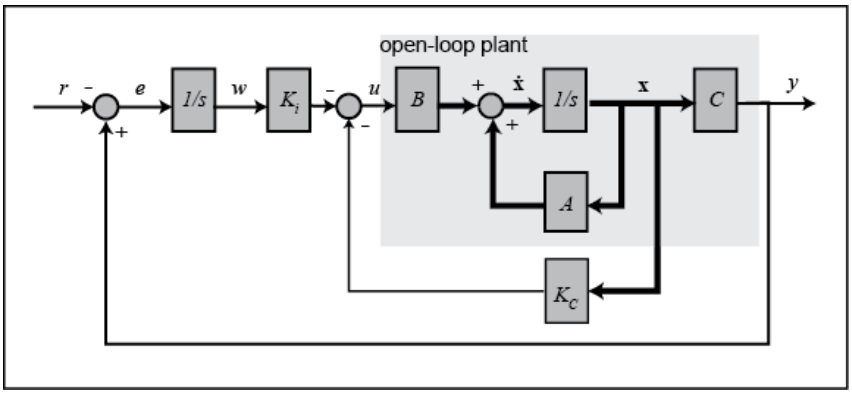
```
% Plot the static tracker performance
figure;
hold on;
lsim(sys_close, n_bar*u, t, x0);
lsim(sys_close, u, t, x0);
legend('static tracker','closed-loop system');
title('Static Tracker Performance');
grid on;
```



It is worth noting that the transient response of the system remains unchanged in the static tracker design. Only the steady-state value is scaled to the desired output level.

9. Integral (Robust) Tracker

The integral tracker works by integrating the error signal between the reference input and the system output over time and adding the resulting integral term to the control signal. This ensures that any steady-state error is eliminated, and the system output accurately tracks the reference input.



An integral tracker offers several advantages over a static scaling factor for improving the steady-state performance of a control system. It can handle time-varying reference inputs, while a static scaling factor can only handle constant reference inputs. Secondly, an integral tracker can handle constant disturbances, such as a constant offset or bias in the system. Furthermore, an integral tracker can handle changes in the system dynamics, such as changes in the load or environment, while a static scaling factor cannot.

Worth noting that the design of an integral tracker is only possible if the following augmented matrix is full rank. In other words, we can design an integral tracker if and only if the system has controllability and observability properties, which ensure that the system's state can be fully determined by the input and output signals.

$$\begin{pmatrix} B & A \\ 0 & -C \end{pmatrix}_{(n+p)\times(n+m)}$$

In MATLAB, we can determine the rank of a matrix by using the `rank` function.

```
% Check the condition to design the tracker
rank_aug = rank([Bc, Ac; 0, -Cc])
```

```
rank_aug = 4
```

Then, to incorporate the integral tracker into the system, we add a new state variable that corresponds to the integral of the error signal. This new state variable is added to the state vector. Also, pay close attention to the control signal equation.

$$\begin{pmatrix}\dot{x}\\ \dot{w}\end{pmatrix} = \begin{pmatrix}A & 0\\ -C & 0\end{pmatrix}\begin{pmatrix}x\\ w\end{pmatrix} + \begin{pmatrix}B\\ 0\end{pmatrix}u + \begin{pmatrix}0\\ I\end{pmatrix}r$$

$$y = \begin{pmatrix}C & 0\end{pmatrix}\begin{pmatrix}x\\ w\end{pmatrix}$$

$$where \quad u(t) = k_i w - k_c x$$

Suppose we want to apply pole placement to the robust tracker in the same manner as the static tracker, but with an additional pole dynamic for the integral term. In this case, we need to select the pole to be sufficiently fast such that it does not significantly affect the overall system response.

$$P_d = \{-5-4i, -5+4i, -1.5, -10\}$$

Choosing a fast pole dynamic for the integral term allows the integral tracker to respond quickly to changes in the system dynamics, while still maintaining stability and avoiding oscillations.

```
% Obtain the augmented state-space model
A_aug = [Ac, zeros(n, m); -Cc, zeros(p, m)];
B_aug = [Bc; zeros(m, m)];
C_aug = [Cc, zeros(p, m)];
D_aug = 0;

B_r = [zeros(n, m); ones(p, m)];

% Construct the system dynamics for the tracker
sys_aug = ss(A_aug, B_aug, C_aug, D_aug)
```

```
sys_aug =

  A =
          x1       x2       x3       x4
   x1   -9.25   -28.36   -28.81        0
   x2        1        0        0        0
   x3        0        1        0        0
   x4        0     -1.8    -2.67        0

  B =
       u1
   x1   1
   x2   0
   x3   0
   x4   0

  C =
         x1     x2     x3     x4
   y1     0    1.8   2.67      0

  D =
       u1
   y1   0

Continuous-time state-space model.
```

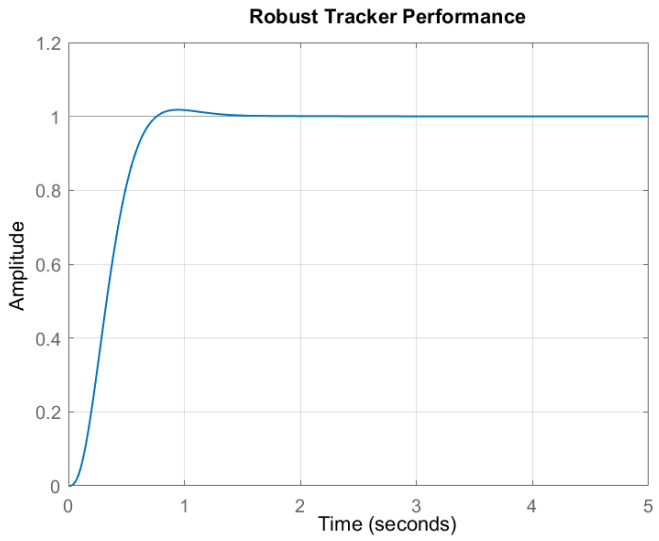
We can now compute the gain vector K for the desired pole locations, using the same poles as before for the close pole placement, and a relatively fast pole for the integral tracker.

```
p_desired_tracker = [p_desired_close, -10];
K_tracker = place(A_aug, B_aug, p_desired_tracker)
```

```
K_tracker = 1×4
   12.2500  142.6400  178.0833 -230.3371
```

Note the evaluated gain of the tracker pole at -10 thoroughly. There appears to be a connection between the static and robust tracker, as the gain of the tracker pole is dependent on the static tracker scale factor. It is noteworthy that this gain is equal to the product of the static tracker scale factor and the integral tracker pole.

```
% Obtain the robust tracker state-space model
sys_tracker = ss(A_aug - B_aug*K_tracker, B_r, C_aug, D_aug);

tFinal = 5;
t = 0:0.01:tFinal;      % time vector t = [0 ~ tFinal]
r = ones(size(t));      % step input r(t) = 1; for t>0
x0 = [0; 0; 0; 0];      % zero state initial condition

figure;
lsim(sys_tracker, r, t, x0);
title('Robust Tracker Performance');
grid on;
```



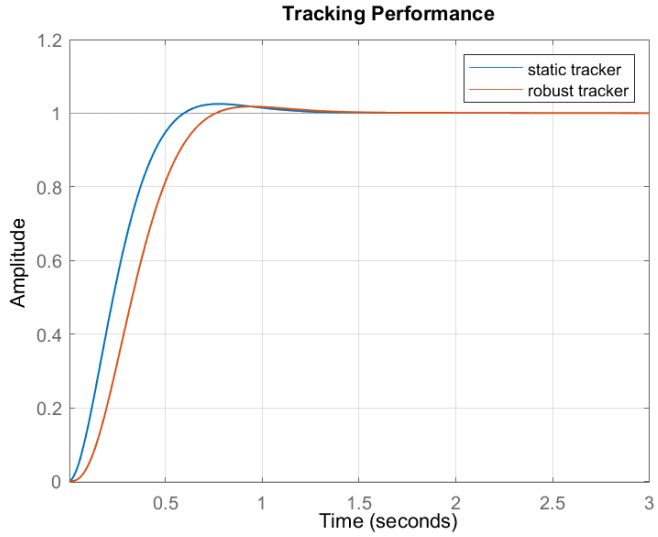

```
          RiseTime: 0.4335
    TransientTime: 0.6992
     SettlingTime: 0.6992
      SettlingMin: 0.9034
      SettlingMax: 1.0180
        Overshoot: 1.8001
       Undershoot:      0
             Peak: 1.0180
         PeakTime: 0.9395
```

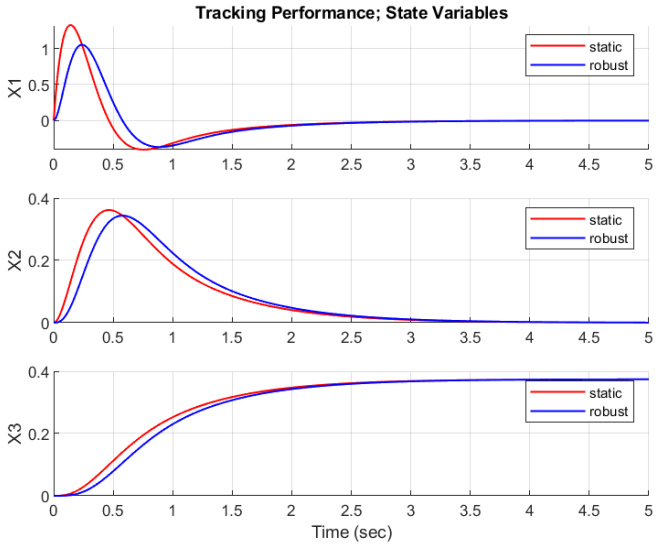
10. Static and Robust Tracker Comparison

In this section, we will compare the performance of the static tracker with that of the integral tracker based on three criteria:

10.1. Tracking Performance

As can be observed, both the static and robust tracker designs exhibit acceptable tracking performance. As previously mentioned, the gain of the integral pole in the robust tracker is dependent on the static scale factor and the location of the integral pole. Therefore, under conditions of no disturbance, the tracking performance of both controllers is comparable. However, it is evident that while the robust tracker displays a lower overshoot, it has a slower response. The transient response can be modified by relocating the integral pole. As the pole gets farther from the jω-axis, it provides a faster response but with a higher overshoot.





10.2. Model Parameter Variation

Here, we aim to evaluate the performance of the designed trackers by changing the parameters of the system matrix A. However, it is crucial to note that altering the system matrix parameters can lead to changes in important system characteristics such as controllability, observability, and stability. Additionally, it may affect the feasibility of designing a tracker for the system.

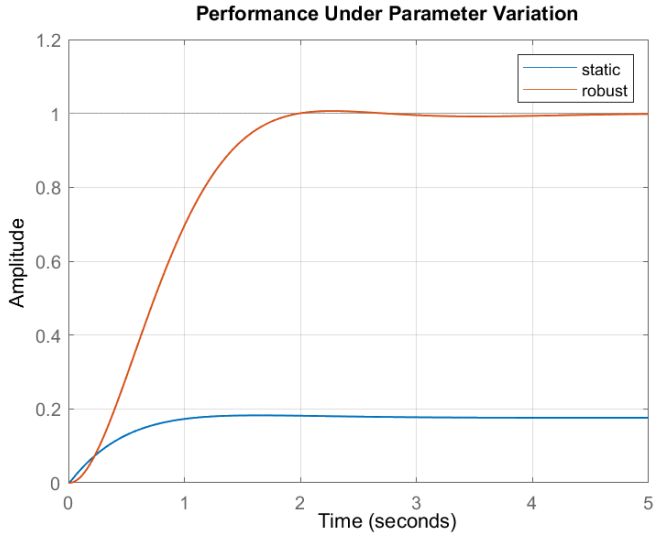
Therefore, before making any changes to the system matrix parameters, it is necessary to carefully analyze the system's characteristics and determine the potential effects of the changes on the system's behavior.

```
% Alter the system matrix parameters
A_var = Ac + 10*[Ac(1,:); zeros(n-1,n)]
```

```
A_var = 3×3
 -101.7500 -311.9600 -316.9100
    1.0000         0         0
         0    1.0000         0
```

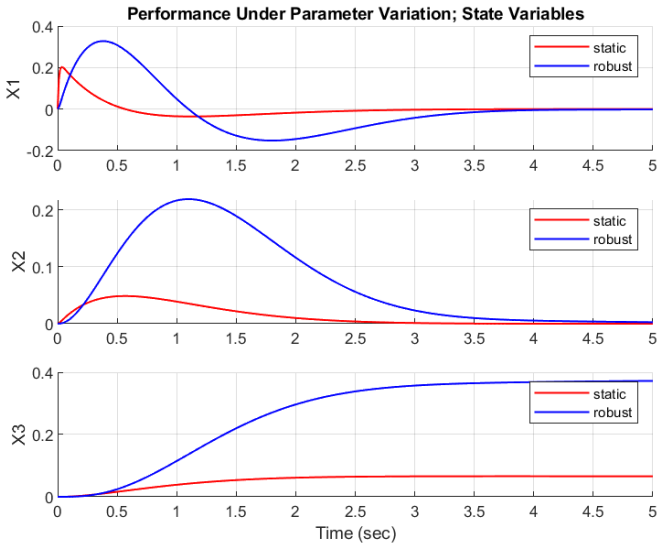
```
% new robust tracker model
A_aug_var = [A_var, zeros(n, m); -Cc, zeros(p, m)];
sys_tracker_var = ss(A_aug_var - B_aug*K_tracker, B_r, C_aug, D_aug);

% new static tracker model
sys_close_var = ss(A_var - Bc*K_close, Bc, Cc, Dc);
```

We utilized the previously developed codes to obtain and plot the new static and robust tracker designs. Our analysis revealed that modifying the system parameters had a significant impact on the performance of the designed trackers. Specifically, the static tracker failed to regulate the output under the new system conditions. In contrast, the integral tracker demonstrated robust tracking of the input signal despite the modifications to the system parameters.



During this experiment, we only modified the system parameters while keeping the static scale factor and state feedback gain matrix unchanged. Our analysis showed that the transient response of both trackers was affected by these changes. Specifically, the settling time increased, resulting in a slower response.

To gain further insights into the system's behavior, we analyzed the state variables and plotted them for both trackers. These plots provide a visual representation of how the system responds to the changes in the system parameters.



To sum up, the performance of the two trackers differed significantly under the new system conditions. The static tracker was unable to track the reference signal, indicating an unacceptable response. On the other hand, the integral tracker was able to robustly track the input signal despite the changes in the system parameters.

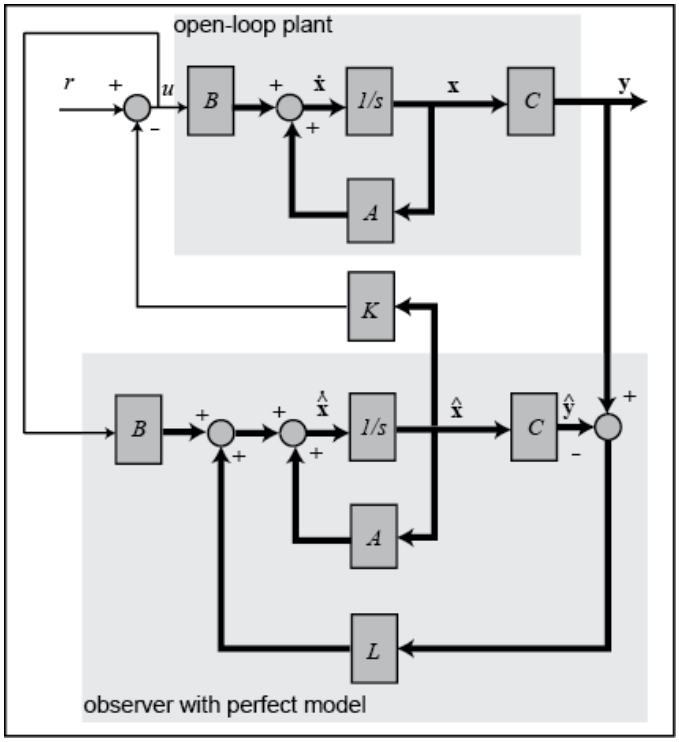
10.3. Constant Disturbance

Eventually, we employed a constant disturbance to the output signal to investigate the system's behavior under such conditions. Specifically, we wanted to observe how the designed trackers would respond to a constant bias applied to the output signal.

To conduct the experiment, we first applied a constant bias of 0.5 level to the output signal. We then gradually increased the disturbance to compare the tracking performance of the controllers under different levels of disturbance.

11. Full-Order Observer Design

A full-order observer is a state estimator that is used to estimate the full state of a dynamical system from the available measurements of its output. In practice, it is important to check the observability of the system before designing a full-order observer. If the system is not observable, then the observer may provide inaccurate estimates of the state variables, which can lead to poor system performance.



The observer consists of two parts: an observer equation and an estimator equation. The observer equation is used to calculate the derivative of the estimated state variables, while the estimator equation is used to update the estimated state variables based on the output measurements.

$$\dot{\hat{x}}(t) = A\hat{x}(t) + Bu(t) + L(y(t) - C\hat{x}(t))$$

Also, the estimation error dynamics is shown as follows:

$$\dot{e}(t) = (A - LC)e(t)$$

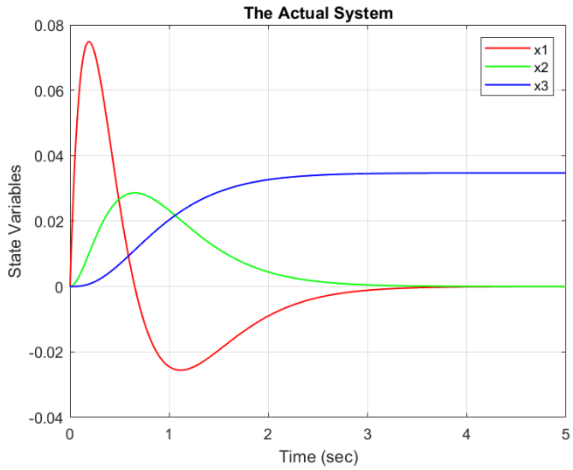
One common approach for selecting the observer poles is to place them at a faster rate than the system poles but not too fast to avoid amplifying noise. In this case, we can use the dominant poles of the system as a guideline for the observer poles.

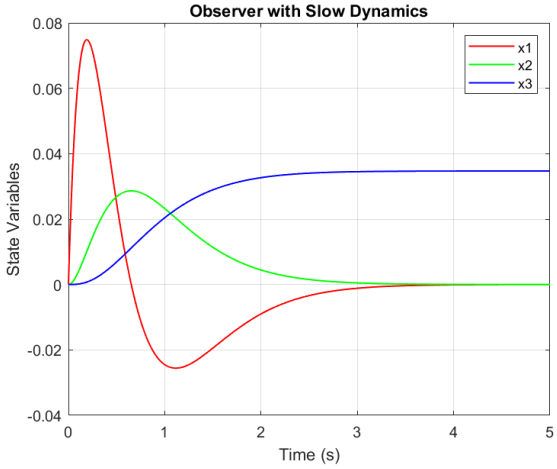
We will place the observer poles at two different sets of values, one with faster poles and another with slower poles.

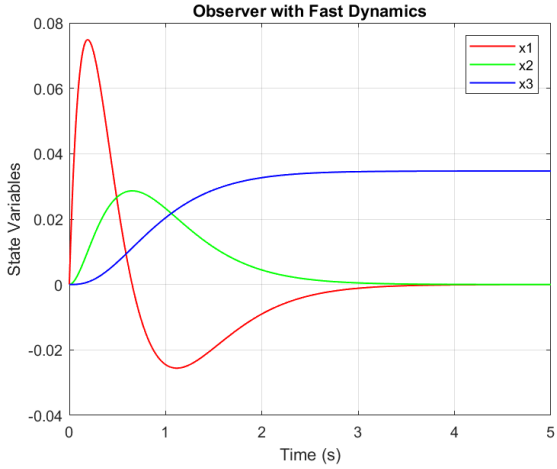
```
% Observer with faster poles
obs_poles_fast = [100*sys_poles(1), 100*sys_poles(2), 100*sys_poles(3)];
L_fast = place(A', C', obs_poles_fast)';

A_obs_fast =    [     A         zeros(size(A))
                zeros(size(A))    A-L_fast*C ];
B_obs_fast =    [     B
                zeros(size(B)) ];
C_obs_fast =    [     C       zeros(size(C)) ];

obs_fast = ss(A_obs_fast, B_obs_fast, C_obs_fast, 0);

% Observer with slower poles
obs_poles_slow = [10*sys_poles(1), 10*sys_poles(2), 10*sys_poles(3)];
L_slow = place(A', C', obs_poles_slow)';

A_obs_slow =    [     A         zeros(size(A))
                zeros(size(A))    A-L_slow*C ];
B_obs_slow =    [     B
                zeros(size(B)) ];
C_obs_slow =    [     C       zeros(size(C)) ];

obs_slow = ss(A_obs_slow, B_obs_slow, C_obs_slow, 0);

% Evaluate the State Variables
[~, ~, x_fast] = lsim(obs_fast, ones(size(t)), t, [0 0 0 0 0 0]);
[~, ~, x_slow] = lsim(obs_slow, ones(size(t)), t, [0 0 0 0 0 0]);
[~, ~, x] = lsim(sys_ss, ones(size(t)), t, [0 0 0]);

% Evaluate the estimation error
err_fast = [x(:,1)-x_fast(:,1), x(:,2)-x_fast(:,2), x(:,3)-x_fast(:,3)];
err_slow = [x(:,1)-x_slow(:,1), x(:,2)-x_slow(:,2), x(:,3)-x_slow(:,3)];
```

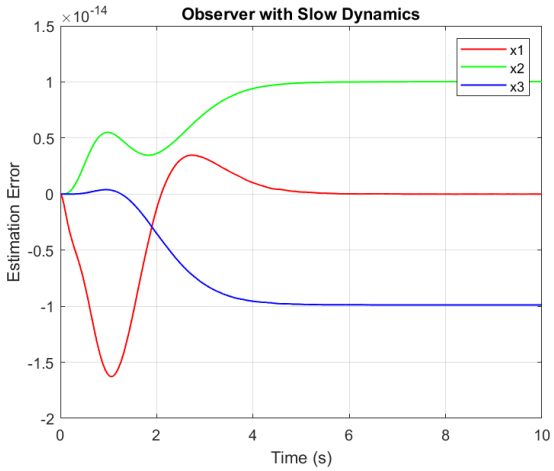
After designing both fast and slow observers, we can conclude that both exhibit acceptable performance and are capable of accurately estimating the system output. Our study revealed that the observer with farther poles had a faster estimation. However, we also observed that this faster estimation came at the cost of increased estimation error when compared to the observer with slower dynamics. It is worth noting this error was found to be negligible in practical terms.

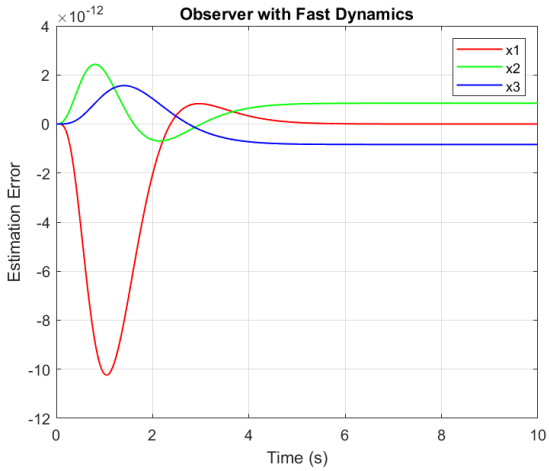
Please find attached plots of the state variables and estimation error for both observers, which provides a visual representation of the difference in estimation error between the two observers.


The Actual System
State Variables
Time (sec)
x1
x2
x3


Observer with Slow Dynamics
State Variables
Time (s)
x1
x2
x3


Observer with Fast Dynamics
State Variables
Time (s)
x1
x2
x3


Observer with Slow Dynamics
×10^-14
Estimation Error
Time (s)
x1
x2
x3

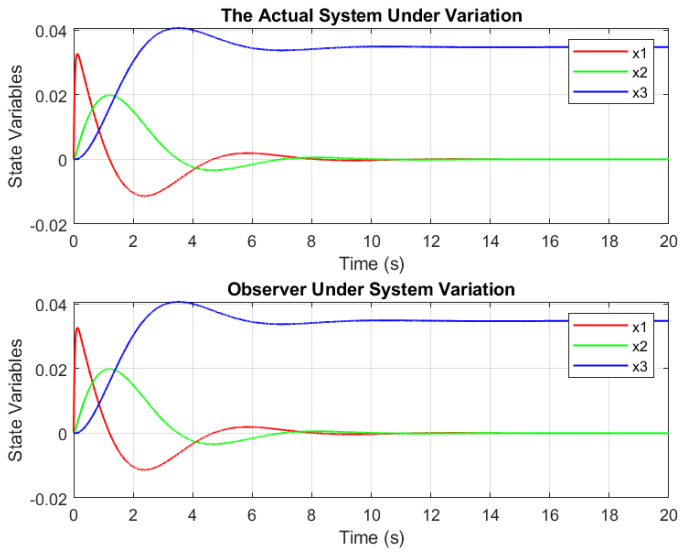

Observer with Fast Dynamics
×10^-12
Estimation Error
Time (s)
x1
x2
x3

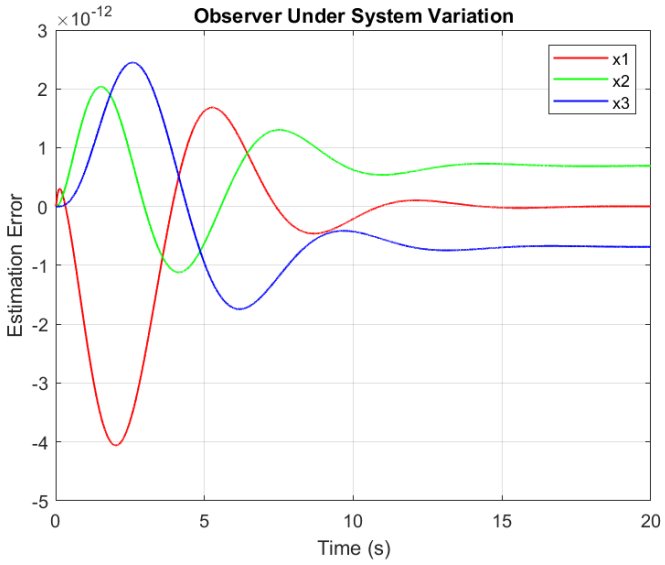
12. Observer Performance Under System Variation

In this section, we aim to investigate the performance of both observers under varying system parameters. By changing these parameters, we can simulate real-world scenarios where system load or environmental factors may fluctuate and evaluate the robustness of the observers in such situations. This analysis is crucial for assessing the practical applicability of the observers, as it allows us to determine their effectiveness in dynamic and unpredictable environments.

In this example, we alter only one of the system parameters. This modification has a significant impact on the poles of the transfer function, as it effectively increases the coefficient of s^2 in the denominator by a factor of 3. As a result, the system becomes more sensitive to changes in this matrix index.

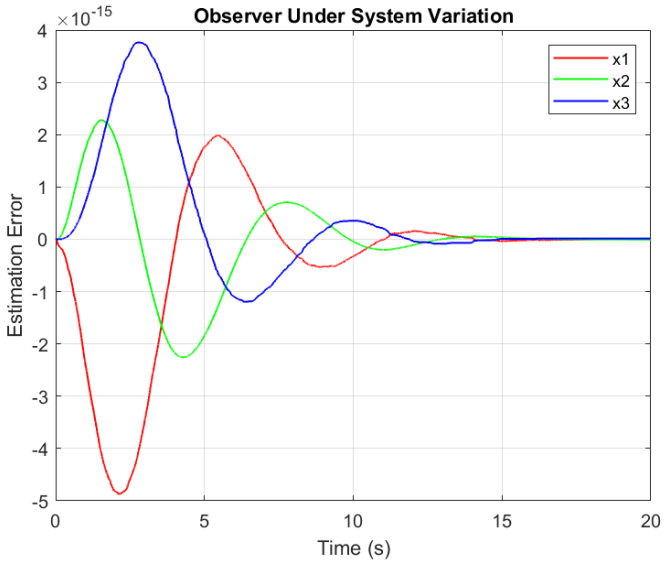
```
A_var = Ac + 2*[Ac(1,1) zeros(1, n-1); zeros(n-1,n)];

A_obs_var =    [     A_var          zeros(size(A_var))
                zeros(size(A_var))   A_var-L_fast*Cc ];
B_obs_var =    [      Bc
                zeros(size(Bc)) ];
C_obs_var =    [      Cc             zeros(size(Cc)) ];

% Obtain the observer model
obs_var = ss(A_obs_var, B_obs_var, C_obs_var, 0);

% Evaluate the State Variables
[~, ~, x_var] = lsim(ss(A_var,Bc,Cc,0), ones(size(t)), t, [0 0 0]);
[~, ~, x_obs_var] = lsim(obs_var, ones(size(t)), t, [0 0 0 0 0 0]);

% Evaluate the Estimation Error
err_var = [ x_var(:,1)-x_obs_var(:,1), ...
            x_var(:,2)-x_obs_var(:,2), ...
            x_var(:,3)-x_obs_var(:,3) ];
```





After modifying one of the system parameters, we observed a significant increase in estimation error and a decrease in system stability. Furthermore, in the previous section, we found that the error dynamics stabilized after a certain period of time. However, due to the modification in the system, it now takes a longer time for the error dynamics to converge and stabilize.

Next, we will test the slower observer with the applied modifications, we obtained an interesting result: the estimation error converged to zero after a certain amount of time. This finding suggests that slower observers may be less sensitive to system variations and noise, compared to faster observers. It is worth noting that even in the original system, the slower observer had a higher estimation error than the modified observer under varying system conditions.



13. Reduced-Order Observer Design

A reduced-order observer is a type of state estimator that predicts the state variables of a system using a reduced number of measurements, or outputs, of the system. In some cases, obtaining full measurements of all states of a system may be difficult or impossible, making it necessary to estimate the missing states. The design of a reduced-order observer involves selecting a subset of the states to be estimated and designing an observer to estimate those states using the available measurements.

To design a reduced-order observer, we first check that the output matrix C is in the standard form:

$$C = (I_q \quad 0)$$

If not, we calculate the rank of matrix C to determine the value of q. Using C, we then form a singularity matrix $P_{n\times n}$. It is important to pay close attention to the equation below and select matrix R such that P is full rank and invertible.

$$q = rank(C)$$

$$P_{n\times n} = \begin{pmatrix} C_{q\times n} \\ R_{(n-q)\times n} \end{pmatrix}$$

Once we have formed the singularity matrix P, we can obtain a new model.

$$\begin{matrix} \dot{\bar{x}}(t) = PAP^{-1}\,\bar{x}(t) + PBu(t) \\ y(t) = CP^{-1}x(t) \end{matrix} \quad \Rightarrow \quad \boxed{\begin{matrix} \dot{\bar{x}}(t) = \bar{A}\,\bar{x}(t) + \bar{B}u(t) \\ y(t) = \bar{C}x(t) = (I_q \quad 0)x(t) \end{matrix}}$$

We can express the new state-space model in the following decomposed form:

$$\begin{pmatrix} \dot{\bar{x}}_1 \\ \dot{\bar{x}}_2 \end{pmatrix} = \begin{pmatrix} \bar{A}_{11} & \bar{A}_{12} \\ \bar{A}_{21} & \bar{A}_{22} \end{pmatrix} x(t) + \begin{pmatrix} \bar{B}_1 \\ \bar{B}_2 \end{pmatrix} u(t)$$

Worth noting that $\dot{\bar{x}}_1$ and $\dot{\bar{x}}_2$ are vectors of size n and $(n-q)$, respectively, where $\dot{\bar{x}}_2$ represents the states that will be estimated.

The dynamics of the estimated values and estimation error can then be analyzed as follows:

$$\dot{z}(t) = (\bar{A}_{22} - \bar{L}\bar{A}_{12})z(t) + [(\bar{A}_{22} - \bar{L}\bar{A}_{12})\bar{L} - (\bar{A}_{21} - \bar{L}\bar{A}_{11})]y(t) + (\bar{B}_2 - \bar{L}\bar{B}_1)u(t)$$

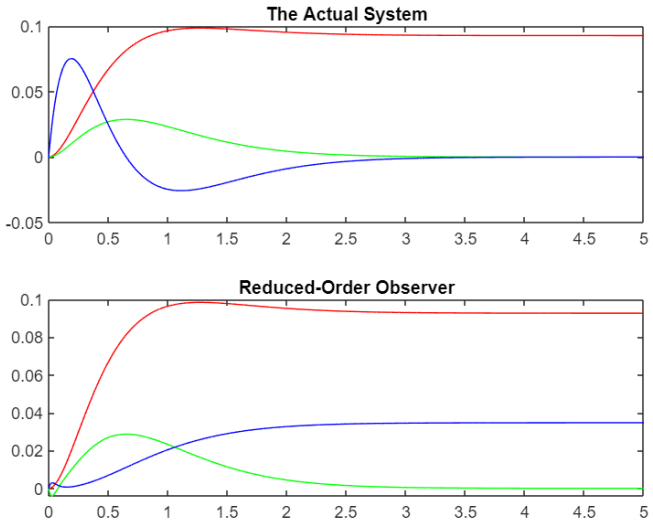
$$\dot{e}(t) = (\bar{A}_{22} - \bar{L}\bar{A}_{12})e(t)$$

Where $z = \hat{x}_2 - \bar{L}x_1$, thus we conclude:

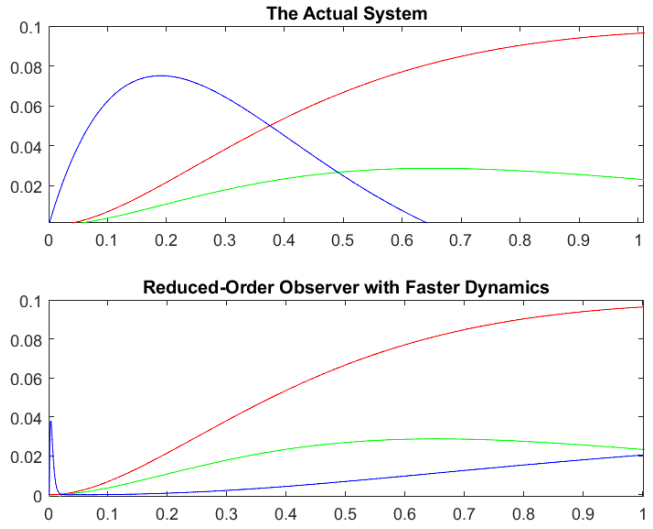
$$\hat{\bar{x}}(t) = \begin{pmatrix} y \\ \bar{L}y + z \end{pmatrix} \Rightarrow \hat{x}(t) = P^{-1}\hat{\bar{x}}(t)$$

```
% Evaluate the value q
q = rank(C);

% Construct the R matrix
R = [   0   1   0   ;
        1   0   0   ];

% P must be full-rank and invertible
P = [   C   ;
        R   ];
% Verify the obtained P matrix
rank(P);
inv(P);

A_bar = P*A*inv(P);
B_bar = P*B;
C_bar = C*inv(P);
D_bar = 0;

sys_singular = ss(A_bar, B_bar, C_bar, D_bar);
sys_poles = pole(sys_singular);

% Decompose the system matrix A_bar
A11 = A_bar(1:q, 1:q);
A12 = A_bar(1:q, q+1:n);
A21 = A_bar(q+1:n, 1:q);
A22 = A_bar(q+1:n, q+1:n);

% Decompose the input matrix B_bar
B1 = B_bar(1:q, :);
B2 = B_bar(q+1:n, :);

% Desired Observer Poles
p_reduced_obs = 5*sys_poles(1:n-q, 1);

% Evaluate the obserever coefficients L
L_bar = place(A22', A12', p_reduced_obs)';

x_bar(:,1) = [0; 0; 0];
z(:,1) = [0; 0];

for k = 2:size(t,2)
    y_red(k) = C_bar * x_bar(:,k-1);
    x_bar(:,k) = x_bar(:, k-1) + dt*(A_bar* x_bar(:,k-1) + B_bar*1);
    z(:,k) = z(:,k-1) + dt*((A22 - L_bar*A12)*z(:,k-1) + ...
    ((A22 - L_bar*A12)*L_bar + (A21 - L_bar*A11))*y_red(k-1) + (B2-L_bar*B1)*1);
end

x_red = inv(P)*([[1 0 0 ];[L_bar eye(2)]])*[y_red; z];
```

As expected, the computation cost has significantly reduced due to the reduction in the number of variables. However, it is worth noting that the observer still exhibits a certain degree of error, which may impact its overall accuracy.



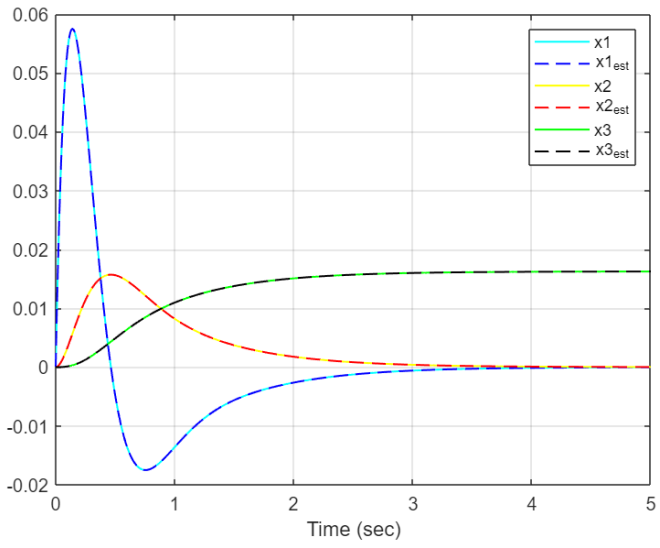
In addition to the standard dynamics, we can also experiment with different pole values to determine the optimal performance of the reduced-order observer. Our findings suggest that increasing the pole values of the observer leads to a faster response, which generally tends to better mimic the actual signals. However, it's important to note that there is a limit to the effectiveness of this method, as increasing the pole values beyond a certain threshold can result in diminishing returns. Despite these efforts, it's worth mentioning that the estimated variables may still exhibit a extensive degree of error.



14. Regulator with Full-Order Observer

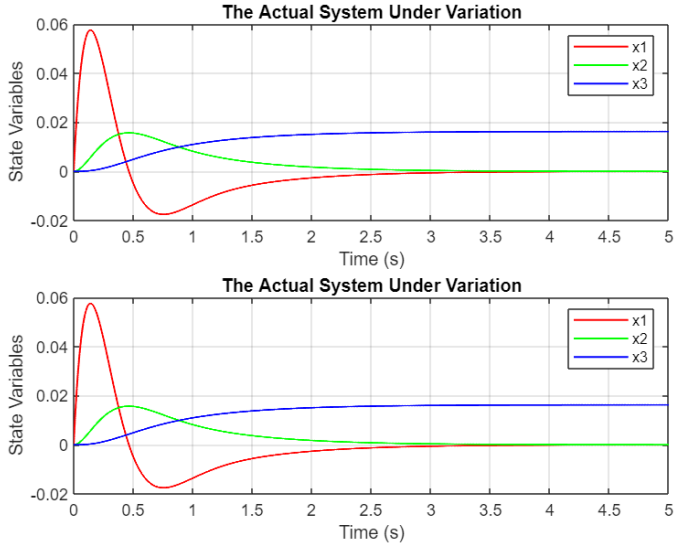
This section aims to integrate the findings from Sections 7 and 11 of this report. Specifically, we aim to design a state feedback controller for a system where the state variables are not directly accessible, thereby requiring the design of an observer. To accomplish this, we evaluate the state feedback gain matrix K and the observer gain matrix L separately, and then coordinate the results to establish an effective control system. Please find the relevant code provided below for your reference.

```
p_desired_regulator = [-5-4i, -5+4i, -1.5];
K = place(A, B, p_desired_regulator);

p_desired_observer = 10*p_desired_regulator;
L = place(A', C', p_desired_observer)';

A_reg = [   A-B*K          B*K
        zeros(size(A))    A-L*C ];

B_reg = [     B
        zeros(size(B)) ];

C_reg = [     C    zeros(size(C)) ];

t = 0:0.01:5;
x0 = [0 0 0];
[~,~,x_reg] = lsim(ss(A_reg,B_reg,C_reg,0),ones(size(t)),t,[x0 x0]);

n = 3;
e = x_reg(:,n+1:end);
x_reg = x_reg(:,1:n);
x_est = x_reg - e;

% Save state variables explicitly to aid in plotting
x1 = x_reg(:,1); x2 = x_reg(:,2); x3 = x_reg(:,3);
x1_est = x_est(:,1); x2_est = x_est(:,2); x3_est = x_est(:,3);
```



15. Regulator with Reduced-Order Observer

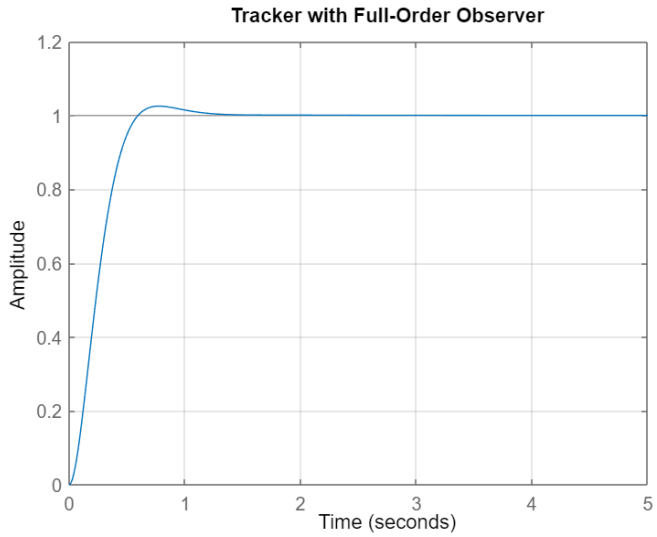
```
p_desired_regulator = [-5-4i, -5+4i, -1.5];
K = place(A, B, p_desired_regulator);

q = rank(C);
R = [   0   1   0   ;
        1   0   0   ];
P = [   C   ;
        R   ];

rank(P);
Q = inv(P);
Q1 = Q(1:3,1);
Q2 = Q(1:3,2:3);

A_bar = P*A*inv(P);
B_bar = P*B;
C_bar = C*inv(P);
D_bar = 0;

sys_singular = ss(A_bar, B_bar, C_bar, D_bar);

A11 = A_bar(1:q, 1:q);
A12 = A_bar(1:q, q+1:n);
A21 = A_bar(q+1:n, 1:q);
A22 = A_bar(q+1:n, q+1:n);

B1 = B_bar(1:q, :);
B2 = B_bar(q+1:n, :);

p_reduced_observer = 10*p_desired_regulator(1:n-q);
L_bar = place(A22', A12', p_reduced_observer)';

% Coefficients
cz = A22 - L_bar*A12;
cy = (A22 - L_bar*A12)*L_bar + (A21 - L_bar*A11);
cb = B2 - L_bar*B1;

A_est = [   A-B*K*Q1*C-B*K*Q2*L_bar*C          -B*K*Q2;
       cy*C-cb*K*Q1*C-cb*K*Q2*L_bar*C    cz-cb*K*Q2 ];
B_est = [B; cb];
C_est = [C zeros(1,2)];

sys_est = ss(A_est,B_est,C_est,0);

tFinal = 5;
t = 0:0.001:tFinal;
u = ones(size(t));

[y, ~, z_est] = lsim(sys_est, u, t, [0 0 0 0 0]);
x_est = inv(P)*[y'; L_bar.*y'+z_est(:,4:end)'];
```


The Actual System Under Variation
State Variables
Time (s)
x1
x2
x3
0.06
0.04
0.02
0
-0.02
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

The Actual System Under Variation
State Variables
Time (s)
x1
x2
x3
0.06
0.04
0.02
0
-0.02
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

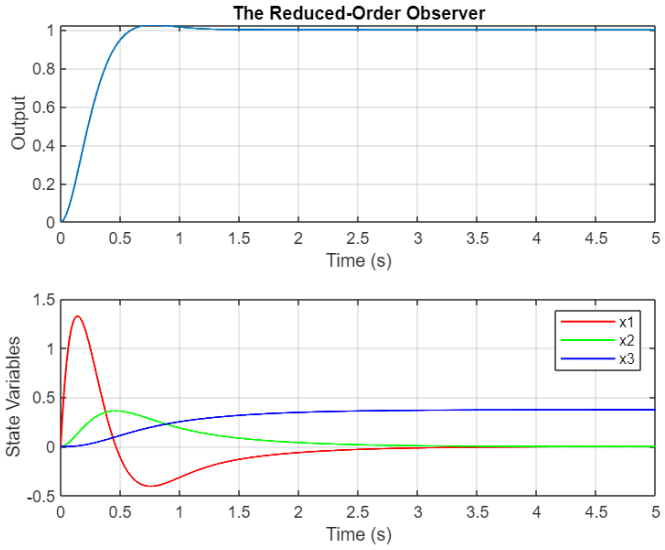
16. Tracker with Full-Order Observer

```
p_desired_regulator = [-5-4i, -5+4i, -1.5];
K = place(A, B, p_desired_regulator);

p_desired_observer = 10*p_desired_regulator;
L = place(A', C', p_desired_observer)';


A_reg = [   A-B*K          B*K;
        zeros(size(A))    A-L*C ];

B_reg = [     B*n_bar;
        zeros(size(B)) ];

C_reg = [     C    zeros(size(C)) ];


tFinal = 5;
t = 0:0.01:tFinal;      % time vector t = [0 ~ tFinal]
u = ones(size(t));      % step input u(t) = 1; for t>0
x0 = [0; 0; 0];         % zero state initial condition

figure;
hold on;
lsim(ss(A_reg,B_reg,C_reg,0), u, t, [0 0 0 0 0 0]);
title('Tracker with Full-Order Observer');
grid on;
```



17. Tracker with Reduced-Order Observer

```
A_est = [   A-B*K*Q1*C-B*K*Q2*L_bar*C          -B*K*Q2;
       cy*C-cb*K*Q1*C-cb*K*Q2*L_bar*C    cz-cb*K*Q2 ];
B_est = n_bar*[B; cb];
C_est = [C zeros(1,2)];
```



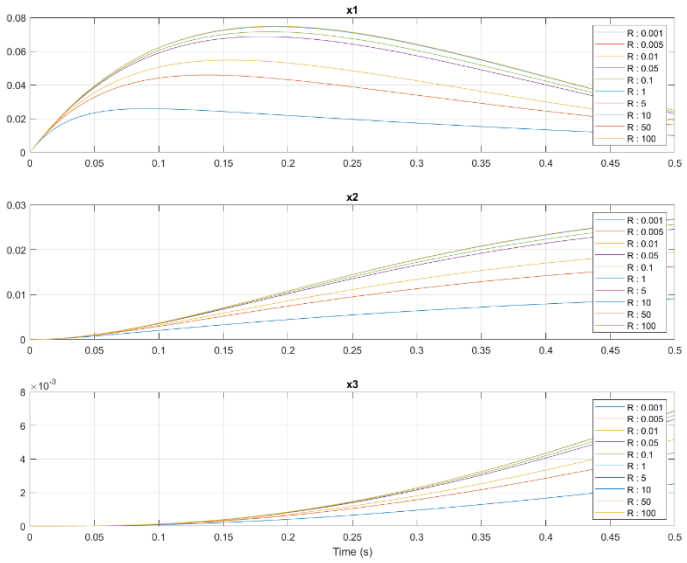
18. LQR Controller Design

LQR (Linear Quadratic Regulator) is a control technique used to design linear state feedback controllers for linear time-invariant (LTI) systems. The objective of LQR control is to minimize a quadratic cost function that measures the deviation of the system's state variables from their desired values and the control input from its optimal value.

The LQR technique computes the optimal state feedback gain matrix K that minimizes the cost function. The optimal gain matrix is obtained by solving the algebraic Riccati equation, which is a matrix equation that describes the steady-state behavior of the system. The Riccati equation can be solved analytically or numerically using MATLAB or other software tools.

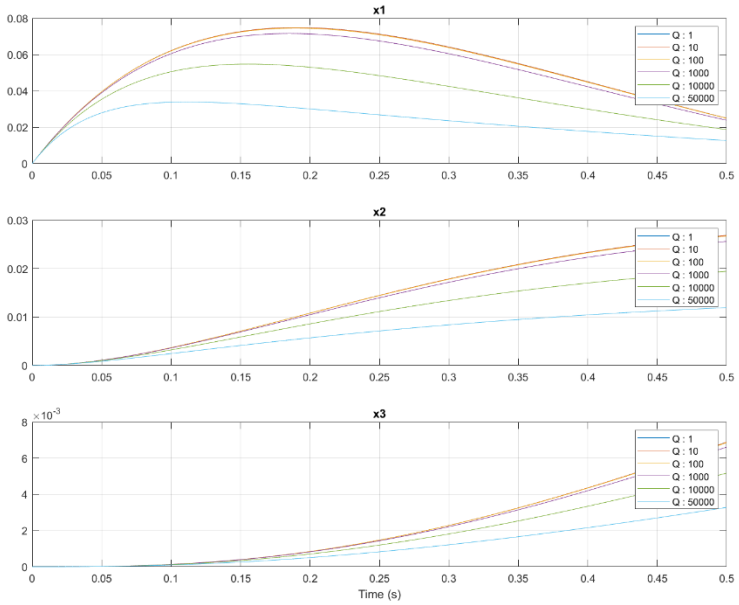
18.1. Constant Q

```
% Define the Q and R matrices
Q = eye(3); % a 3x3 identity matrix
R = 1; % a scalar value

% Obtain the optimal state feedback gain K
[K, S, e] = lqr(sys_controllable, Q, R);


% Plot the output response
figure;

% Simulate the system with different values of R
% different values of R to be tested
R_values = [ 0.001 0.005 0.01 0.05 0.1 1 5 10 50 100];
for i = 1:length(R_values)
    R = R_values(i);
    [K, S, e] = lqr(sys_controllable, Q, R);
    sys_lqr = ss(A-B*K, B, C, 0);

    t = 0:0.001:0.5;
    x0 = [0; 0; 0]; % initial state vector
    [y, t, x] = lsim(sys_lqr, ones(size(t)), t, x0);

    subplot(3,1,1);
    plot(t, x(:, 1));
    title('x1')
    hold on;
    subplot(3,1,2);
    plot(t, x(:, 2));
    title('x2')
    hold on;
    subplot(3,1,3);
    plot(t, x(:, 3));
    title('x3')
    hold on;
    xlabel('Time (s)');
end
```



18.2. Constant R

```
Q_values = [ 1*eye(3); 10*eye(3); 100*eye(3); 1000*eye(3); 10000*eye(3);
50000*eye(3)];
for i = 1:length(Q_values)/3
    Q = Q_values(3*i-2:3*i,:);
    [K, S, e] = lqr(sys_controllable, Q, R);
    sys_lqr = ss(A-B*K, B, C, 0);

    Q_labels(i) = {["Q : " + num2str(Q_values(3*i-2,1))]};

    [y, t, x] = lsim(sys_lqr, ones(size(t)), t, x0);
```



Thanks,

Erfan Riazati