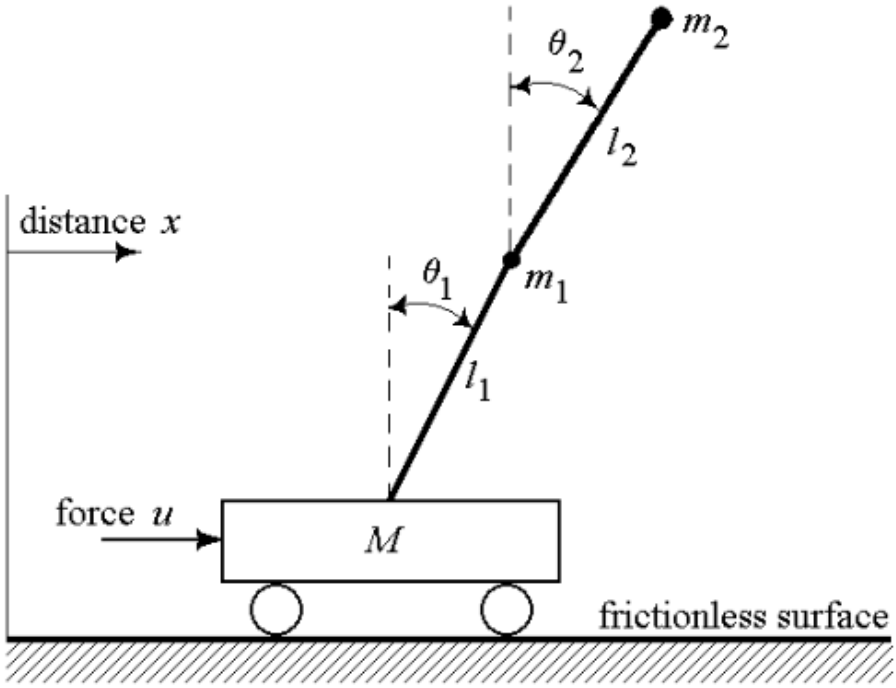
FunWork #3

Victoria Nagorski

Prelude

In Funwork #1 the problem statement was defined with the following diagram:



With the given properties of the system:

- $m_1 = 0.5kg$
- $l_1 = 0.5m$
- $m_2 = 0.75kg$
- $l_2 = 0.75m$
- $M = 1.5kg$
- $g = 9.81m/sec^2$

We were also given that the state variables follow the following definition: $x_1 = x$, $x_2 = \theta_1$, $x_3 = \theta_2$, $x_4 = \dot{x}$, $x_5 = \dot{\theta_1}$, $x_6 = \dot{\theta_2}$. These were substituted much later since it was easier to think in terms of x, θ_1, and θ_2.

With the problem statement, the following assumptions were be made:

- The body is rigid
- The rods are mass-less
- We are working with a simplified point-mass system

A the end of the problem we had ended up with the 3 equations:
Equation 1:

$$(M+m_1+m_2)\ddot{x}+(m_1+m_2)l_1\ddot{\theta}_1\cos\theta_1+m_2l_2\ddot{\theta}_2\cos\theta_2-(m_1+m_2)l_1\dot{\theta}_1^2\sin\theta_1-m_2l_2\dot{\theta}_2^2\sin\theta_2=Q_1$$

Equation 2:

$$(m_1+m_2)l_1\ddot{x}\cos\theta_1+(m_1+m_2)\ddot{\theta}_1l_1^2+m_2l_1l_2\ddot{\theta}_2\cos(\theta_1-\theta_2)+m_2l_1l_2\dot{\theta}_2^2\sin(\theta_1-\theta_2) \\ -m_1l_1g\sin\theta_1-m_2l_1g\sin\theta_1=Q_2$$

Equation 3:

$$m_2l_2\ddot{x}\cos\theta_2+m_2l_1l_2\ddot{\theta}_1\cos(\theta_1-\theta_2)+m_2\ddot{\theta}_2l_2^2-m_2l_1l_2\dot{\theta}_1^2\sin(\theta_1-\theta_2)-m_2gl_2\sin\theta_2=Q_3$$

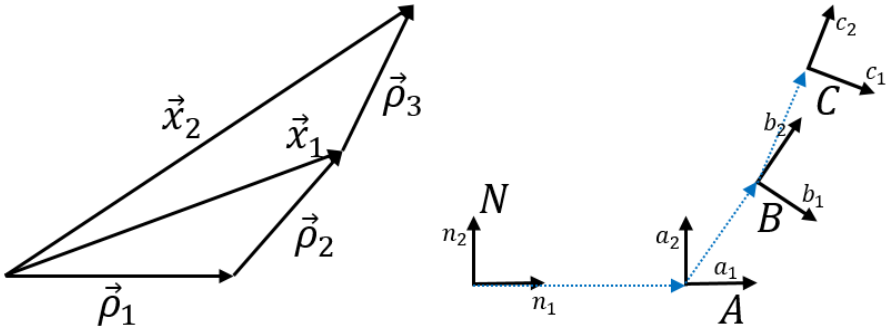
Where Q_1, Q_2, and Q_3 are called the generalized forces. Where 1, 2, and 3 relate to the generalized coordinates: $q_1=x$, $q_2=\theta_1$, and $q_3=\theta_2$. The generalized forces can be derived with the following equations:

$$\dot{W}=\sum\vec{F}\cdot\vec{v}+\sum\vec{\tau}\cdot\vec{\omega}$$

F is force, v is velocity, τ is torque, and ω is angular velocity. Now the generalized forces:

$$Q_i=\frac{\partial\dot{W}}{\partial\dot{q}_i}$$

The generalized forces will depend on which inputs are added into the system. I might reference this diagram when deriving the generalized forces in the following sections. The frames and vectors of the problem are then described in the following figure:



One Input Equilibrium

In problem 1, we are only given one force in the problem. We can start by solving the prelude's equations:

$$\dot{W} = \sum \vec{F} \cdot \vec{v} + \sum \vec{\tau} \cdot \vec{\omega} =< F, 0, 0 > \cdot < \dot{x}, 0, 0 > +0 = F\dot{x}$$

F is force, v is velocity, τ is torque, and ω is angular velocity. Now the generalized forces:

$$Q_1 = \frac{\partial(F\dot{x})}{\partial \dot{x}} = F$$

$$Q_2 = \frac{\partial(F\dot{x})}{\partial \dot{\theta_1}} = 0$$

$$Q_3 = \frac{\partial(F\dot{x})}{\partial \dot{\theta_2}} = 0$$

The end result are these non-linear equations:
Equation 1:

$$(M+m_1+m_2)\ddot{x} + (m_1+m_2)l_1\ddot{\theta}_1 \cos\theta_1 + m_2 l_2 \ddot{\theta}_2 \cos\theta_2 - (m_1+m_2)l_1\dot{\theta}_1^2 \sin\theta_1 - m_2 l_2 \dot{\theta}_2^2 \sin\theta_2 = F$$

Equation 2:

$$(m_1+m_2)l_1\ddot{x}\cos\theta_1 + (m_1+m_2)\ddot{\theta}_1 l_1^2 + m_2 l_1 l_2 \ddot{\theta}_2 \cos(\theta_1-\theta_2) + m_2 l_1 l_2 \dot{\theta}_2^2 \sin(\theta_1-\theta_2) - m_1 l_1 g \sin\theta_1 - m_2 l_1 g \sin\theta_1 = 0$$

Equation 3:

$$m_2 l_2 \ddot{x}\cos\theta_2 + m_2 l_1 l_2 \ddot{\theta}_1 \cos(\theta_1-\theta_2) + m_2 \ddot{\theta}_2 l_2^2 - m_2 l_1 l_2 \dot{\theta}_1^2 \sin(\theta_1-\theta_2) - m_2 g l_2 \sin\theta_2 = 0$$

After some manipulation that was done in Funwork #1, we have the following matrices that will be inputted into Matlab. I will first define some constants and exchange those into the matrices. It makes it easier to track in my code.

- $r_1 = M + m_1 + m_2$
- $r_2 = (m_1 + m_2)l_1$
- $r_3 = m_2 l_2$
- $r_4 = (m_1 + m_2)l_1^2$
- $r_5 = m_2 l_1 l_2$
- $r_6 = m_2 l_2^2$
- $f_1 = (m_1 l_1 + m_2 l_2)g$
- $f_2 = m_2 l_2 g$

Rewrite with the correct state variables:

$$M = \begin{bmatrix} r_1 & r_2\cos x_2 & r_3\cos x_3 \\ r_2\cos x_2 & r_4 & r_5\cos(x_2 - x_3) \\ r_3\cos x_3 & r_5\cos(x_2 - x_3) & r_6 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & -r_2 x_5 \sin x_2 & -r_3 x_6 \sin x_3 \\ 0 & 0 & r_5 x_6 \sin(x_2 - x_3) \\ 0 & -r_5 x_5 \sin(x_2 - x_3) & 0 \end{bmatrix}$$

$$G = \begin{bmatrix} 0 \\ -f_1 \sin x_2 \\ -f_2 \sin x_3 \end{bmatrix}$$

$$H = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \\ \dot{x}_5 \\ \dot{x}_6 \end{bmatrix} = \begin{bmatrix} 0 & I_{3x3} \\ 0 & -M^{-1}C \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} + \begin{bmatrix} 0 \\ -M^{-1}G \end{bmatrix} + \begin{bmatrix} 0 \\ M^{-1}H \end{bmatrix} u$$

Now that we have the nonlinear equations, we can begin to solve for the equilibrium pair. The equilibrium pair can be found solving the following equation:

$$f(x_e, u_e) = 0$$

We are given the following equilibrium state vector:

$$x_e = \begin{bmatrix} 0.1 & 60^\circ & 45^\circ & 0 & 0 & 0 \end{bmatrix}^T$$

To have Matlab solve for having the arm on the right side, we will have the following in the code:

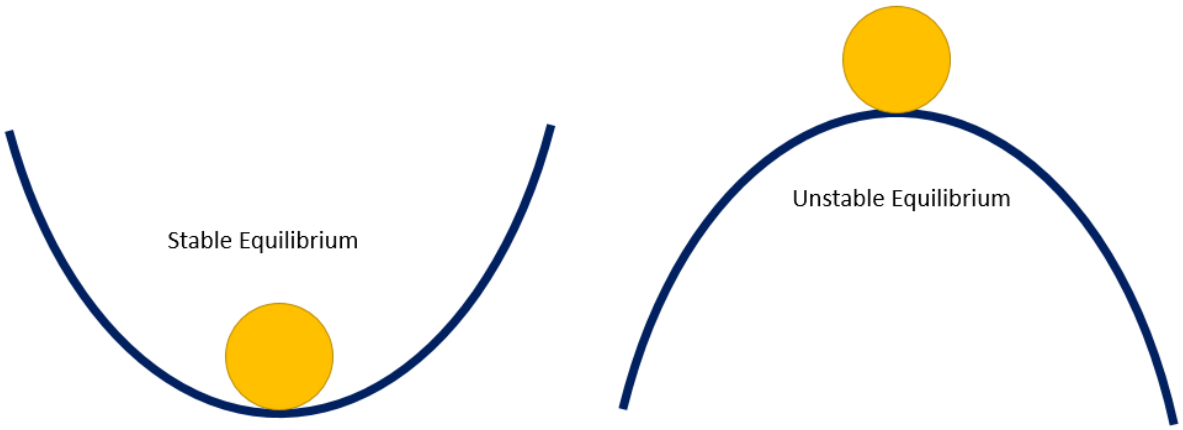
$$x_e = \begin{bmatrix} 0.1 & -60^\circ & -45^\circ & 0 & 0 & 0 \end{bmatrix}^T$$

To solve using Matlab, I will first exchange the equilibrium state vector into the symbolic nonlinear equations stored in Matlab. Matlab has built in solvers for symbolic equations that will allow me to ask it to solve for u_e.

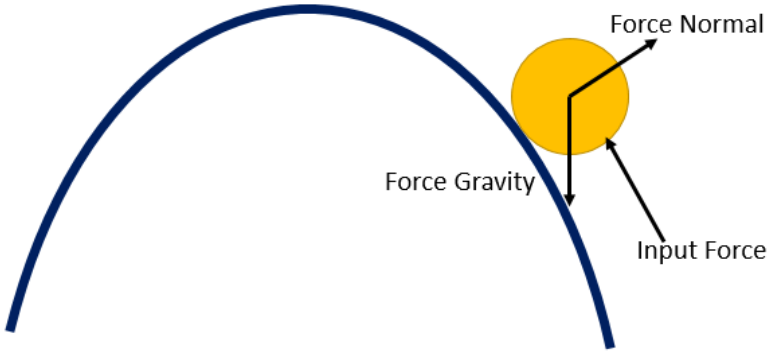
The following is the code used:

```
clear; close all; clc;                  % Make sure everthing is closed
load('Values.mat');                     % From vars from Homework # 1
syms u                                  % Define variable u as symbolic

% Start problem
f = simplify(f);                        % First simplify non-linear terms

theta1 = -60*pi/180;                    % Convert degrees to radians
theta2 = -45*pi/180;                    % Convert degrees to radians

x_e = [0.1;theta1;theta2;0;0;0];        % Create the x state
g = subs(f,x,x_e);                      % Plug in x_e to non-linear terms

f_e = g == [0;0;0;0;0;0];               % Set the equation f(x_e,u_e) = 0
u_e = solve(f_e,u)                      % Solve equations for u_e
```

After running the previous code, the warning “Warning: Unable to find explicit solution." is given, and u_e comes out as “Empty Sym". This tells us that we do not have enough inputs in the system to “force" an equilibrium state that is not naturally there. What do I mean by this? In some systems (not all), there are sometimes points where natural equilibrium points can occur- whether stable or not. They do not require system inputs to maintain position if no external forces are applied. The image below shows how natural equilibrium points can be visualized in a simple system.



Stable equilibrium means that the system will naturally go back to its spot no matter the disturbance. Our double inverted arm is at a stable equilibrium at $180°$. Unstable equilibrium is where the system will stay if there are no disturbances to the system. Our double inverted pendulum is at an unstable equilibrium at $0°$. However, if we were to try to keep the ball at the position as shown below:



We would need to apply some force to keep the ball from falling into its natural position at the bottom of the hill. In the case of our system, we do not have enough inputs to make a non-natural point of equilibrium that is not at angles $0°$ or $180°$.

Two Input Equilibrium

In problem 2, we are given a force applied to the mass M, and torque applied where l_1 connects to mass M in the problem. When referencing the frames diagram in the prelude, we see that the torque changes the angle between frame A and B. We can start solving for $\dot{W}$:

$$\dot{W} = \sum \vec{F} \cdot \vec{v} + \sum \vec{\tau} \cdot \vec{\omega} =< F, 0, 0 > \cdot < \dot{x}, 0, 0 > + < 0, 0, \tau > \cdot < 0, 0, \dot{\theta_1} >$$

$$\dot{W} = F\dot{x} + \tau\dot{\theta_1}$$

F is force, v is velocity, τ is torque, and ω is angular velocity. Now the generalized forces:

$$Q_1 = \frac{\partial \dot{W}}{\partial \dot{x}} = F$$

$$Q_2 = \frac{\partial \dot{W}}{\partial \dot{\theta_1}} = \tau$$

$$Q_3 = \frac{\partial \dot{W}}{\partial \dot{\theta_2}} = 0$$

The end result are these non-linear equations:
Equation 1:

$$(M+m_1+m_2)\ddot{x} + (m_1+m_2)l_1\ddot{\theta}_1\cos\theta_1 + m_2l_2\ddot{\theta}_2\cos\theta_2 - (m_1+m_2)l_1\dot{\theta}_1^2\sin\theta_1 - m_2l_2\dot{\theta}_2^2\sin\theta_2 = F$$

Equation 2:

$$(m_1+m_2)l_1\ddot{x}\cos\theta_1 + (m_1+m_2)\ddot{\theta}_1l_1^2 + m_2l_1l_2\ddot{\theta}_2\cos(\theta_1-\theta_2) + m_2l_1l_2\dot{\theta}_2^2\sin(\theta_1-\theta_2) - m_1l_1g\sin\theta_1 - m_2l_1g\sin\theta_1 = \tau$$

Equation 3:

$$m_2l_2\ddot{x}\cos\theta_2 + m_2l_1l_2\ddot{\theta}_1\cos(\theta_1-\theta_2) + m_2\ddot{\theta}_2l_2^2 - m_2l_1l_2\dot{\theta}_1^2\sin(\theta_1-\theta_2) - m_2gl_2\sin\theta_2 = 0$$

The only thing that changes from problem 1 is the H matrix and the u vector.

$$H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$$

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} F \\ \tau \end{bmatrix}$$

Now that we have the nonlinear equations, we can begin to solve for the equilibrium pair. The equilibrium pair can be found solving the following equation:

$$f(x_e, u_e) = 0$$

We are given the following equilibrium state vector:

$$x_e = \begin{bmatrix} 0.1 & 60^\circ & 45^\circ & 0 & 0 & 0 \end{bmatrix}^T$$

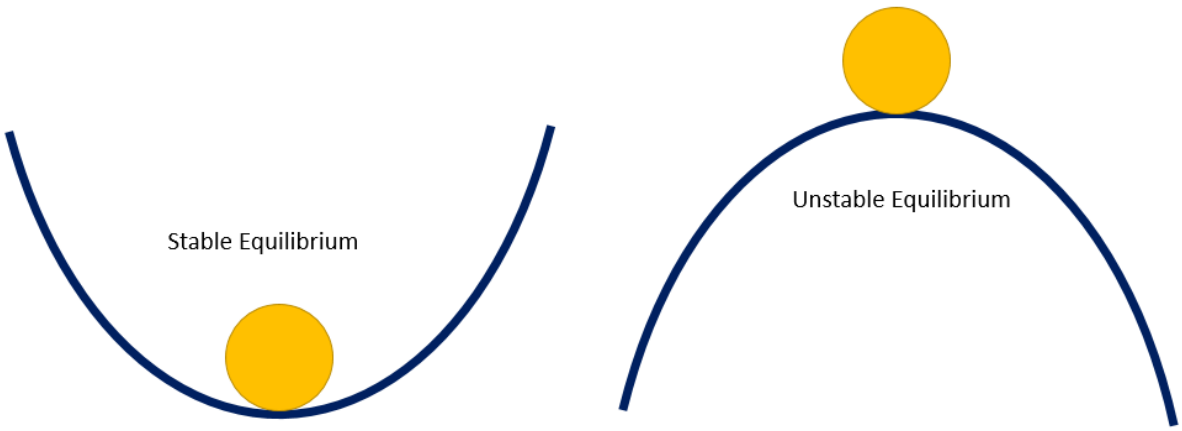
To have Matlab solve for having the arm on the right side, we will have the following in the code:

$$x_e = \begin{bmatrix} 0.1 & -60^\circ & -45^\circ & 0 & 0 & 0 \end{bmatrix}^T$$

To solve using Matlab, I will first exchange the vector into the symbolic non-linear equations stored in Matlab. Matlab has built in solvers for symbolic equations that will allow me to ask it to solve for u_e. The following is the code used:

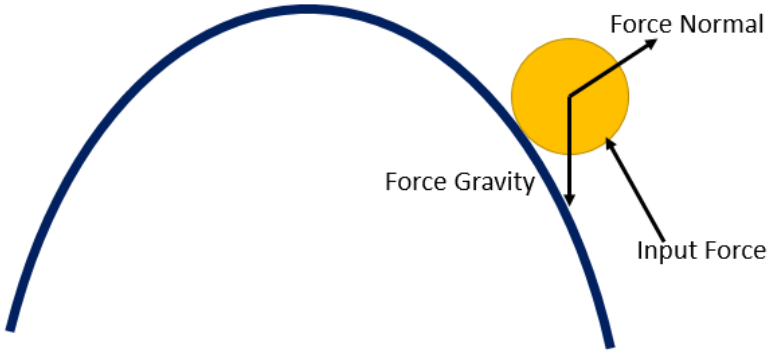
```
clear; close all; clc;                  % Make sure everything is closed
load('Values_Rev2.mat');                % From vars from Homework # 2
syms u                                  % Define variable u as symbolic

% Start problem
theta1 = -60*pi/180;                    % Convert degrees to radians
theta2 = -45*pi/180;                    % Convert degrees to radians

x_e = [0.1;theta1;theta2;0;0;0];        % Create the x state
g = subs(f,x,x_e);                      % Plug in x_e to non-linear ↩
    equations

f_e = g == [0;0;0;0;0;0];               % Set the equation f(x_e,u_e) = 0
u_e = solve(f_e,u)                      % Solve equations for u_e
```

After running the previous code, the warning “Warning: Unable to find explicit solution." is given, and u_e comes out as “Empty Sym". This tells us that we do not have enough inputs in the system to “force" an equilibrium state that is not naturally there. What do I mean by this? In some systems (not all), there are sometimes points where natural equilibrium points can occur- whether stable or not. They do not require system inputs to maintain position if no external forces are applied. The image below shows how natural equilibrium points can be visualized in a simple system.



Stable equilibrium means that the system will naturally go back to its spot no matter the disturbance. Our double inverted arm is at a stable equilibrium at 180°. Unstable equilibrium is where the system will stay if there are no disturbances to the system. Our double inverted pendulum

is at an unstable equilibrium at $0°$. However, if we were to try to keep the ball at the position as shown below:



We would need to apply some force to keep the ball from falling into its natural position at the bottom of the hill. In the case of our system, we do not have enough inputs to make a non-natural point of equilibrium that is not at angles $0°$ or $180°$.

Three Input Equilibrium

In problem 3, we are given a force applied to the mass M, a torque applied where l_1 connects to mass M in the problem, and now a torque between l_1 and l_2. We will call the first torque τ_1 and the second torque τ_2. When referencing the frames diagram in the prelude, we see that τ_1 changes the angle between frame A and B. However, τ_2 changes the reference frames between B and C. We can start solving for $\dot{W}$:

$$\dot{W} = \sum \vec{F} \cdot \vec{v} + \sum \vec{\tau} \cdot \vec{\omega}$$

$$\dot{W} =< F, 0, 0 > \cdot < \dot{x}, 0, 0 > + < 0, 0, \tau_1 > \cdot < 0, 0, \dot{\theta_1} > + < 0, 0, \tau_2 > \cdot < 0, 0, \dot{\theta_3} >$$

$$\dot{W} = F\dot{x} + \tau_1\dot{\theta_1} + \tau_2\dot{\theta_3}$$

Where $\dot{\theta_3} = \dot{\theta_2} - \dot{\theta_1}$. So we can write:

$$\dot{W} = F\dot{x} + \tau_1\dot{\theta_1} + \tau_2(\dot{\theta_2} - \dot{\theta_1})$$

To do a little proof that τ_2 is with respect to $\dot{\theta_3}$ is the following:

- Newton's Third Law: For every action, there is an equal and opposite reaction. This would still hold true for torques which means we can expect two terms to show up due to this torque. The reason this law does not matter in question 2 is because mass M is constrained to not rotate by the ground.
- The torque, τ_2, changes the angle between the reference frames B and C. This angle is θ_3, and **not** θ_2. θ_2 is w.r.t the inertial frame. τ_2 is not taken w.r.t the inertial frame- this would be incorrect. τ_2 acts within the reference frames attached to the body.

Now the generalized forces:

$$Q_1 = \frac{\partial \dot{W}}{\partial \dot{x}} = F$$

$$Q_2 = \frac{\partial \dot{W}}{\partial \dot{\theta_1}} = \tau_1 - \tau_2$$

$$Q_3 = \frac{\partial \dot{W}}{\partial \dot{\theta_2}} = \tau_2$$

The end result are these non-linear equations:
Equation 1:

$$(M+m_1+m_2)\ddot{x} + (m_1+m_2)l_1\ddot{\theta}_1\cos\theta_1 + m_2l_2\ddot{\theta}_2\cos\theta_2 - (m_1+m_2)l_1\dot{\theta}_1^2\sin\theta_1 - m_2l_2\dot{\theta}_2^2\sin\theta_2 = F$$

Equation 2:

$$(m_1+m_2)l_1\ddot{x}\cos\theta_1 + (m_1+m_2)\ddot{\theta}_1l_1^2 + m_2l_1l_2\ddot{\theta}_2\cos(\theta_1-\theta_2) + m_2l_1l_2\dot{\theta}_2^2\sin(\theta_1-\theta_2) - m_1l_1g\sin\theta_1 - m_2l_1g\sin\theta_1 = \tau_1 - \tau_2$$

Equation 3:

$$m_2l_2\ddot{x}\cos\theta_2 + m_2l_1l_2\ddot{\theta}_1\cos(\theta_1-\theta_2) + m_2\ddot{\theta}_2l_2^2 - m_2l_1l_2\dot{\theta}_1^2\sin(\theta_1-\theta_2) - m_2gl_2\sin\theta_2 = \tau_2$$

We now change the H matrix and u vector to look like:

$$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} F \\ \tau_1 \\ \tau_2 \end{bmatrix}$$

Now that we have the nonlinear equations, we can begin to solve for the equilibrium pair. The equilibrium pair can be found solving the following equation:

$$f(x_e, u_e) = 0$$

We are given the following equilibrium state vector:

$$x_e = \begin{bmatrix} 0.1 & 60^\circ & 45^\circ & 0 & 0 & 0 \end{bmatrix}^T$$

To have Matlab solve for having the arm on the right side, we will have the following in the code:

$$x_e = \begin{bmatrix} 0.1 & -60^\circ & -45^\circ & 0 & 0 & 0 \end{bmatrix}^T$$

To solve using Matlab, I will first exchange the equilibrium state vector into the symbolic nonlinear equations stored in Matlab. Matlab has built in solvers for symbolic equations that will allow me to ask it to solve for u_e. The result for u_e:

$$u_e = \begin{bmatrix} 0 & 9.2117 & 3.9019 \end{bmatrix}^T$$

The following is the code used:

```
clear; close all; clc;                  % Clear all
syms u                                  % Define variable as symbolic

% Problem contants
m1 = 0.5;                               % kg
l1 = 0.5;                               % m
m2 = 0.75;                              % kg
l2 = 0.75;                              % m
M = 1.5;                                % kg
g = 9.81;                               % m/sec^2

% Solve for matrix contants
r1 = M + m1 + m2;
r2 = (m1 + m2) * l1;
r3 = m2*l2;
r4 = (m1 + m2) * l1^2;
r5 = m2 * l1 * l2;
r6 = m2 * l2^2;
```

```
f1 = (m1 * l1 + m2 * l1) * g;
f2 = m2 * l2 * g;

% Define Non-Linear Matrices
syms x1 x2 x3 x4 x5 x6 u1 u2 u3
M = [  r1         r2*cos(x2)      r3*cos(x3);
    r2*cos(x2)      r4          r5*cos(x2-x3);
    r3*cos(x3) r5*cos(x2-x3)        r6];
C = [0 -r2*x5*sin(x2)     -r3*x6*sin(x3);
     0        0             r5*x6*sin(x2-x3);
     0 -r5*x5*sin(x2-x3)        0];
G = [       0;
    -f1 * sin(x2);
    -f2 * sin(x3)];
H = [1 0 0;0 1 -1;0 0 1];

% Solve for the Non-Linear Functions
matrix1 = [zeros(3,3)   eye(3);      % Break up equation
           zeros(3,3) -M^-1 * C];
matrix2 = [zeros(3,1); -M^-1 * G];  % Break up equation
matrix3 = [zeros(3,3); M^-1 * H];   % Break up equation
x = [x1; x2; x3; x4; x5; x6];       % Define x vector
u = [u1; u2; u3];                   % Define state vector
f = simplify(matrix1 * x + matrix2 + matrix3 * u);

% Find the Equilibrium Input
theta1 = -60*pi/180;                % Convert degrees to radians
theta2 = -45*pi/180;                % Convert degrees to radians

x_e = [0.1;theta1;theta2;0;0;0];    % Create the x state
g = subs(f,x,x_e);                  % Plug in x_e to non-linear ↩
    equations

f_e = g == [0;0;0;0;0;0];           % Set the equation f(x_e,u_e) = 0

u_temp = struct2cell(solve(f_e,u)) % Currently in a struct >> cell ↩
    array
u_e = zeros(3,1);                   % Initialize u_e variable
for i = 1:3                         % Loop through cell array
   u_e(i,1) = double(u_temp{i});    % Covert to double data type
end
u_e                                 % Print out u_e for publishing

% Save Important Values
save('Values_Rev3.mat','f','x','x_e','u_e')
```

Linearize Three Input System

Linearization through Taylor's expansion utilizes perturbations around an equilibrium point to approximate a linear model. The variables can be described as $x = x_e + \delta x$ or $u = u_e + \delta u$. These equations say that state and inputs are an accumulation of both equilibrium point and the small perturbations around that equilibrium. Taylor's expansion is the following:

$$\frac{d}{dt}x = f(x_e + \delta x, u_e + \delta u) = f(x_e, u_e) + \frac{\partial f}{\partial x}(x_e, u_e)\delta x + \frac{\partial f}{\partial u}(x_e, u_e)\delta u + H.O.T.$$

The differentials can then be lumped into the following matrix form (Jacobian matrices):

$$A = \frac{\partial f}{\partial x}(x_e, u_e) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

$$B = \frac{\partial f}{\partial u}(x_e, u_e) = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial u_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

The end result is $\frac{d}{dt}\delta x = A\delta x + B\delta u$.
In the code, the equilibrium points for states and inputs are put in the same vector. This was a preference choice. The equilibrium state as put into the code (make thetas negative and in radians):

$$EquilibriumVector = \begin{bmatrix} x_e \\ u_e \end{bmatrix} = \begin{bmatrix} x_{1e} \\ x_{2e} \\ x_{3e} \\ x_{4e} \\ x_{5e} \\ x_{6e} \\ u_{1e} \\ u_{2e} \\ u_{3e} \end{bmatrix} = \begin{bmatrix} 0.1 \\ -1.0472 \\ -0.7854 \\ 0 \\ 0 \\ 0 \\ 0 \\ 9.2117 \\ 3.9019 \end{bmatrix}$$

The previously derived $\dot{x}$ is set equal to $f(x, u)$. It is this $f(x, u)$ that is then passed into the 'jacobian' function in Matlab. Once the Jacobian matrix is solved for, the equilibrium points need to be passed into the matrix. What results is the A and B matrices. The following is the code used to linearize the non-linear equations of motion:

```
% Linearize with the Jacobian
equil = [x_e;u_e];                  % Equilibrium point
a = jacobian(f,x);                  % Create Jacobian matrix for A
b = jacobian(f,u);                  % Create Jacobian matrix for B
A = double(subs(a,[x;u],equil))     % Plug in equilibrium points
B = double(subs(b,[x;u],equil))     % Plug in equilibrium points
C = [1 0 0 0 0 0;                   % Define the C matrix for Sys
     0 1 0 0 0 0;
```

```
      0 0 1 0 0 0]
D = zeros(3,3)                          % Define the D matrix for Sys

% Check Observability and Controllability
Co = rank(cnrb(A,B))                    % Controllability
O = rank(obsv(A,C))                     % Observability

% Save Matrices for Easy Access
save('Values_Rev3.mat','A','B','C','D','f','x','x_e','u_e')
```

The resulting matrices that are used for the rest of this report:

$$A = \begin{bmatrix} 0 & 0 & 0 & 1.0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.0 \\ 0 & -0.534 & -1.126 & 0 & 0 & 0 \\ 0 & 22.505 & -17.804 & 0 & 0 & 0 \\ 0 & -13.988 & 21.776 & 0 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0.425 & -0.174 & -0.114 \\ -0.174 & 7.341 & -11.904 \\ -0.289 & -4.563 & 10.144 \end{bmatrix}$$

We are also going to define matrices C and D at this point. We only have 3 outputs, so we should only have 3 rows within C. D will be a zero matrix. In most instances (at least in aerospace engineering), D will be a zero matrix.

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$$D = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Now to quickly check the controllabilty and observability of the new system. Matlab's functions 'cntb()' and 'obsv()' will be used, respectively, to solve for each characteristic. It was found that $rank(cntb(A,B)) = 6$, and $rank(obsv(A,C)) = 6$. This means we should be able to place the poles were desired through feedback control. However, this is not the only criteria used when dealing with output feedback control. This will be touched on more later.

Design a Controller using LMIs

We will start with the Lyapunov equation:

$$A^T P + PA \prec 0$$

$$P \succ 0$$

We will use the closed-looped A_c for this design, which is:

$$A_c = A - BK$$

Plug in:

$$(A - BK)^T P + P(A - BK) \prec 0$$

$$P \succ 0$$

Expand:

$$A^T P + PA - K^T B^T P - PBK \prec 0$$

However, this not solvable using LMI solvers since it is a BMI in K and P. We will define some new variables:

$$Z = KS \rightarrow S^{-1} Z = K$$

$$S = P^{-1}$$

Now multiply both sides of the equation by P^{-1}, and simplify:

$$P^{-1}(A^T P + PA - K^T B^T P - PBK)P^{-1} \prec 0$$

$$P^{-1}A^T + AP^{-1} - P^{-1}K^T B^T - BKP^{-1} \prec 0$$

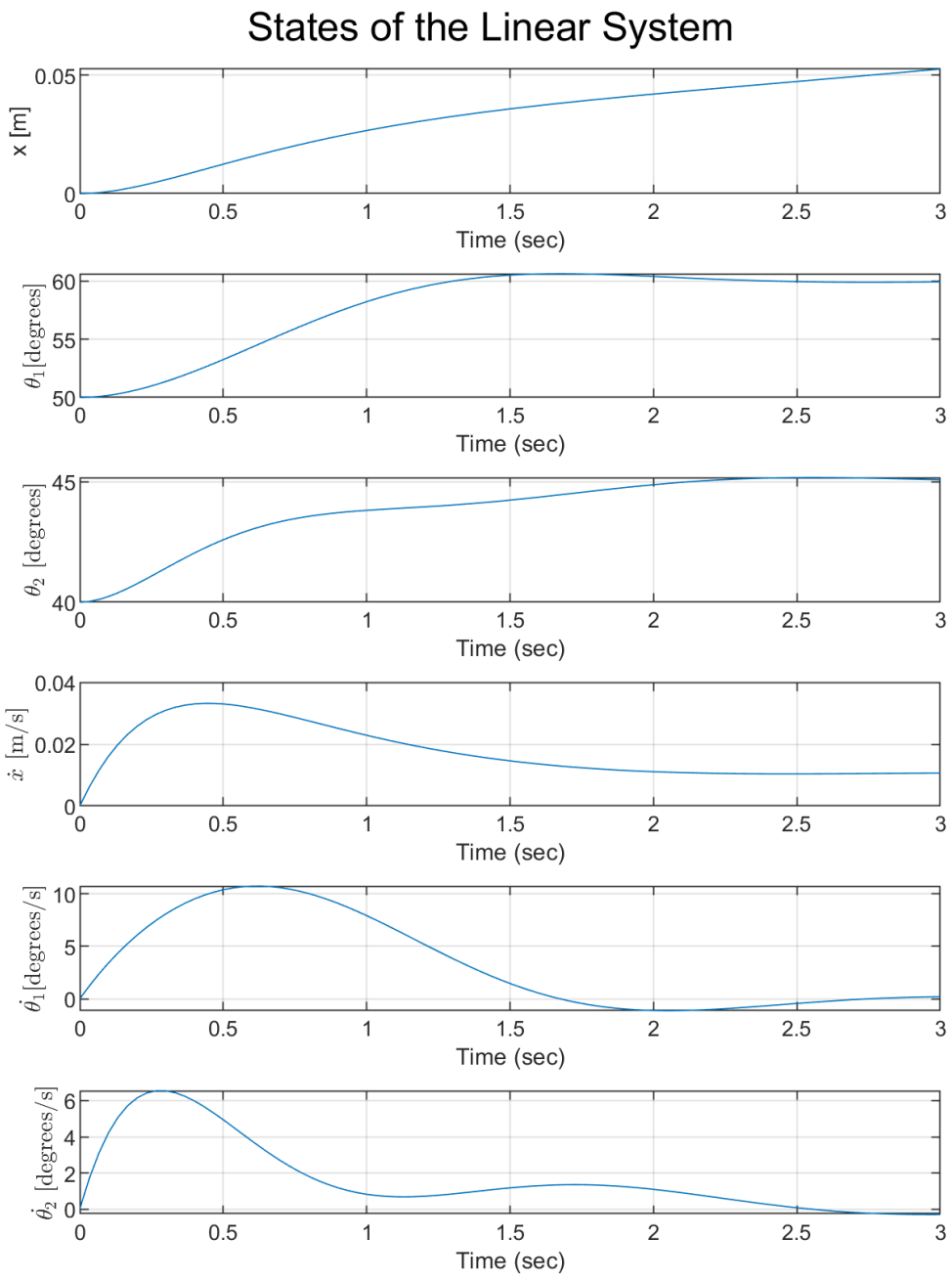
$$SA^T + AS - SK^T B^T - BKS \prec 0$$

$$SA^T + AS - Z^T B^T - BZ \prec 0$$

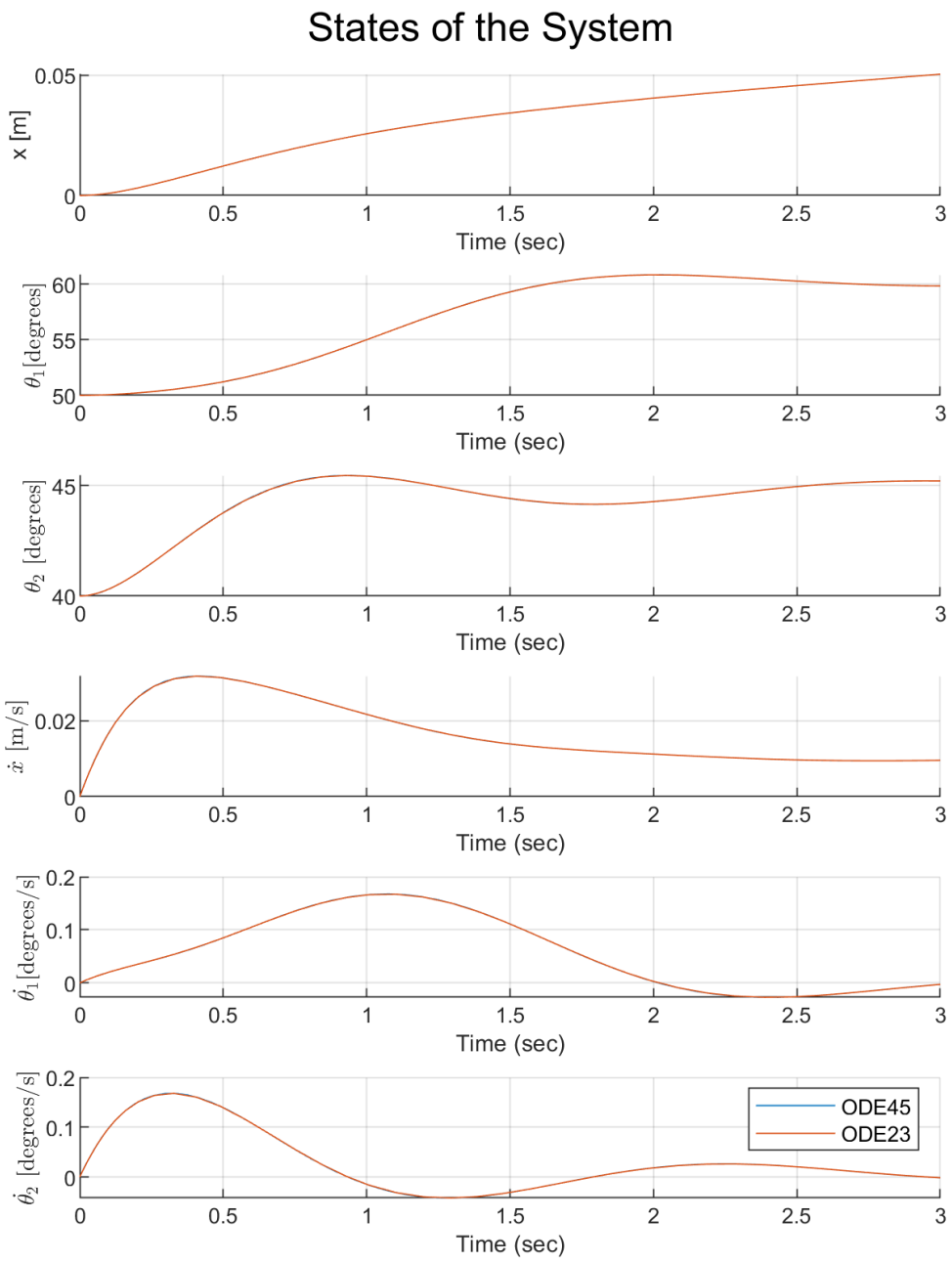
Now the first half the equation is linear in S, and the second half is linear in Z. With this, we can now use the powerful LMI solvers.

It was found that this initial equation only gave alright results. Nothing too impressive since everything converged pretty slowly. I will quickly show pictures on the next page of the linear and nonlinear results before changing the above equation.

The linear response:



The nonlinear response:



To improve these responses we can add a constant α term to the equation to shift the poles further to the left of the complex plane. Going back to the original equations and adding the α term:

$$(A - BK)^T P + P(A - BK) + \alpha P \prec 0$$

$$P \succ 0$$

This α would go through the following variable change:

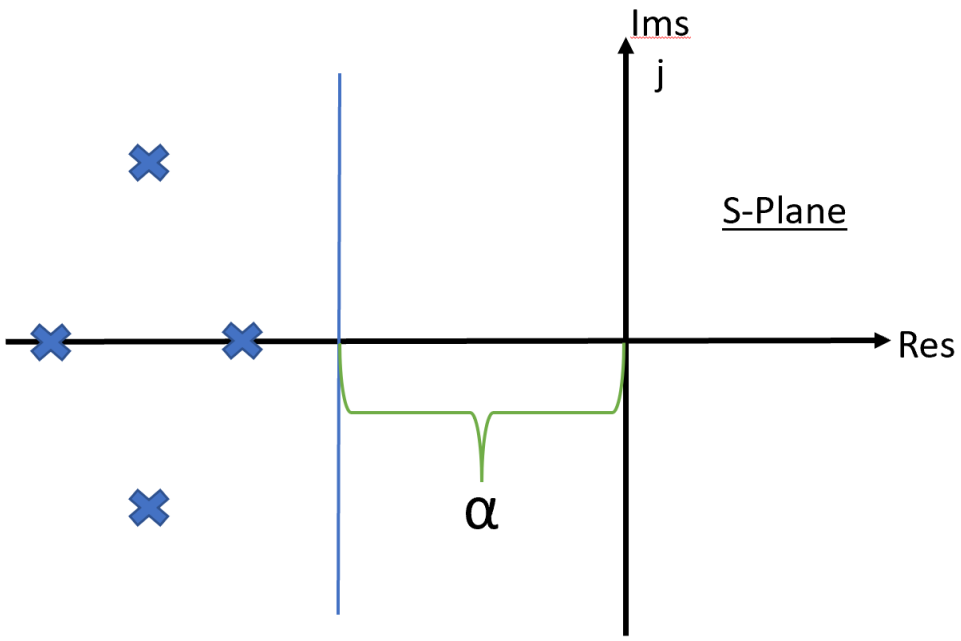
$$\alpha P \rightarrow P^{-1} \alpha P P^{-1} \rightarrow \alpha S$$

The final equations would be the following:

$$SA^T + AS - Z^T B^T - BZ + \alpha S \prec 0$$

$$P \succ 0$$

What this alpha term essentially does is shift the poles further back by a distance of α as shown below:



To solve for α, we will use the Matlab command 'gevp()' to solve the generalized eigenvalue problem. Minimize λ subject to:

$$C(x) \prec D(x)$$

$$0 \prec B(x)$$

$$A(x) \prec \lambda B(x)$$

To use 'gevp()', I set up the equation $A^T P + PA \prec \alpha P$ using the open-loop A. I used the open-loop because K had not been solved for yet, and I was going to use the α to solve for the given gain matrix K. And ultimately, the choice of α is up to the designer. The value of α used to solve for the gain matrix is $\alpha = 12.3326$.
The LMI solver gives us the following K:

$$K = \begin{bmatrix} 1104.080 & 182.686 & 232.719 & 123.609 & 21.505 & 27.178 \\ 279.420 & 129.878 & 160.712 & 31.124 & 17.987 & 22.075 \\ 157.104 & 62.701 & 95.965 & 17.494 & 8.952 & 13.089 \end{bmatrix}$$

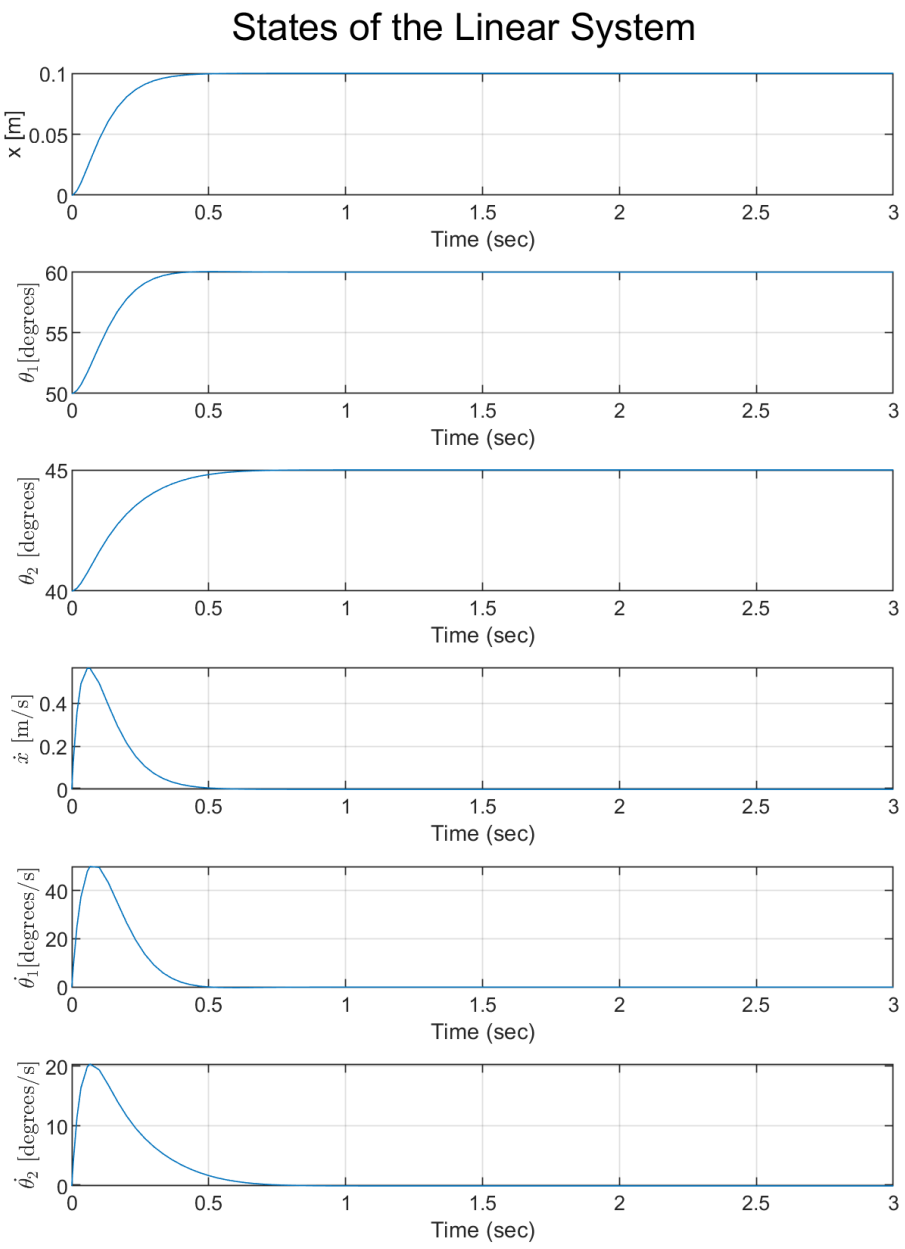
The code that solves for the gain matrix K is below. The linear Simulink model can be found in the published print-out at the end- since it wasn't required for this assignment.

```
% Preliminaries
clear x
```

```
theta1 = -60*pi/180;                % Convert state to radians
theta2 = -45*pi/180;                % Convert state to radians
span = [0 3];
m = 3;
n = 6;

% Solve for Alpha
setlmis([]);
p = lmivar(1,[6 1])

lmiterm([1 1 1 0],1)                % P > I : I
lmiterm([-1 1 1 p],1,1)             % P > I : P
lmiterm([2 1 1 p],1,A,'s')          % LFC (lhs)
lmiterm([-2 1 1 p],1,1)             % LFC (rhs)
lmis = getlmis
[alpha,~]=gevp(lmis,1)

% LMI Solve
cvx_begin sdp

% Variable definiton
variable S(n,n) symmetric
variable Z(m,n)

% LMIs
A*S + S*A' - B*Z - Z'*B' <= -eps * eye(n)
S >= eps * eye(n)

cvx_end

K = Z*S^-1;                         % Solve for K matrix

% Simulate the Controller
clear x0
% Initial States
x0.x = 0;
x0.theta1 = -50*pi/180;
x0.theta2 = -40*pi/180;
x0.x_dot = 0;
x0.theta1_dot = 0;
x0.theta2_dot = 0;

% Reference Input
v.one = 0;
v.two = 0;
sim('Linear_Controller_Design.slx',3);

```

```
% Pull out Data
data = ans.logsout{1}.Values.Data;
time = ans.logsout{1}.Values.Time';

% Plot State vs Time
figure
hold on
sgtitle('States of the System')
subplot(6,1,1)
plot(time,data(:,1)')
xlabel('Time (sec)')
ylabel('x [m]')
grid
subplot(6,1,2)
plot(time,-data(:,2)'*180/pi)
xlabel('Time (sec)')
ylabel('$\theta_1$[degrees]','Interpreter','latex')
grid
subplot(6,1,3)
plot(time,-data(:,3)'*180/pi)
xlabel('Time (sec)')
ylabel('$\theta_2$ [degrees]','Interpreter','latex')
grid
subplot(6,1,4)
plot(time,data(:,4)')
xlabel('Time (sec)')
ylabel('$\dot{x}$ [m/s]','Interpreter','latex')
grid
subplot(6,1,5)
plot(time,-data(:,5)'*180/pi)
xlabel('Time (sec)')
ylabel('$\dot{\theta_1}$[degrees/s]','Interpreter','latex')
grid
subplot(6,1,6)
plot(time,-data(:,6)'*180/pi)
xlabel('Time (sec)')
ylabel('$\dot{\theta_2}$ [degrees/s]','Interpreter','latex')
grid
hold off

% Save Matrices for Easy Access
save('Values_Rev3.mat','A','B','C','D','f','x','x_e','u_e','K')
```

The next page contains the simulation graphs for the linear states to do a quick check. We notice that the states all converge much more rapidly that the previous iteration. In the previous iteration, the states did not necessarily converge before three seconds. However, with this new design, all my states converged around 0.5 seconds. For this assignment, this is desirable since we do not care if the controller is admissible. In this controller, the stepper motors providing torque might be unrealistically moving to get this desired response. Since the focus of this assignment is to design the controller for desirable states, we will keep this design for the rest of the report.



We are then asked to simulate the system next using ode45 and ode23, and then compare the results between the two. Some facts about the solvers to note outright:

- They are both the Runge-Kutta method
- ode23 uses a 3 stage, third-order method. It usually takes larger steps when solving differential equations.
- ode45 uses a 6 stage, fifth-order method. Meaning these solutions tend to be smoother and more accurate than its ode23 counterpart.

I am going to take some time to explain the code a little. Matlab's ode functions take the form of $[t, y] = ode\#\#(function, span, IC)$. For the function part, I created a local function which lives in the same script as the main code instead of a separate script. Local functions are put at the bottom of the script. I want to specifically note how I input u into the nonlinear equations. The linear equations were derived through perturbation theory and resulted in:

$$\frac{d}{dt}\delta x = A\delta x + B\delta u$$

$$\delta u = -K\delta x$$

The deltas come from the fact we are looking at the perturbations around an equilibrium point to use linear control techniques. While the nonlinear equations, however, look at the whole picture.

$$\delta x = x - x_e$$

$$\delta u = u - u_e$$

The plain x and u in these equations is what the nonlinear equations use. So when we go to write the inputs that the nonlinear equations will use, we write:

$$u = -K(x - x_e) + u_e$$

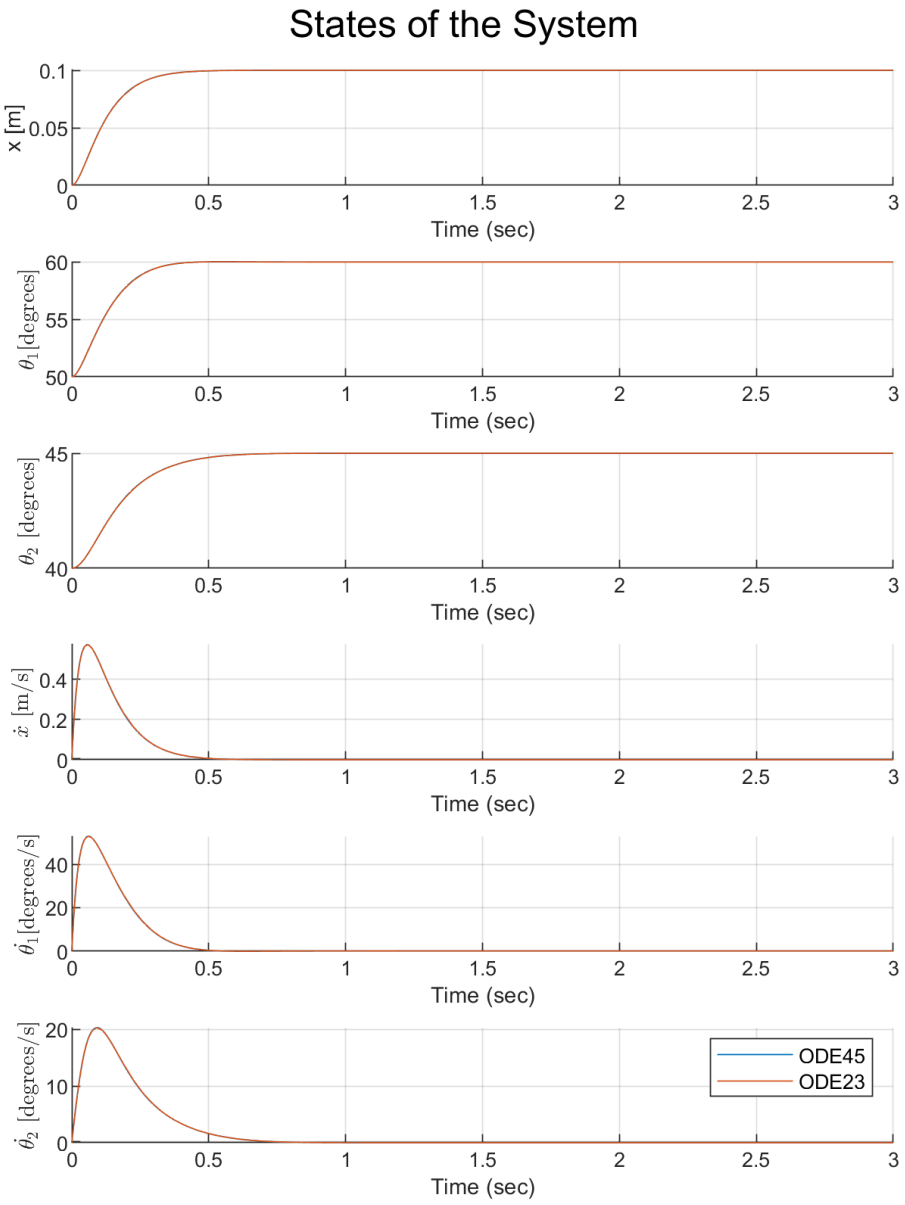
The problem defined the span to be an interval of $[0, 3]$ secs. ICs were left for me to choose. I used the initial conditions:

$$x_0 = \begin{bmatrix} 0 & 50^\circ & 40^\circ & 0 & 0 & 0 \end{bmatrix}^T$$

To have Matlab solve for having the arm on the right side, we will have the following in the code:

$$x_0 = \begin{bmatrix} 0 & -50^\circ & -40^\circ & 0 & 0 & 0 \end{bmatrix}^T$$

The graphs that come out of the odes are on the next page. As we can see, this controller design performed much better with the nonlinear equations when compared to the first iteration. As with the linear model, the states converged around the 0.5 second mark. This response is desirable since we are focused on best state responses in this assignment.
As far as comparing the two odes against each-other, we will notice there is no visual difference between the two graphs. However, if we look at the vectors themselves that get outputted by the function, we will notice ode45 produces a longer vector. Meaning the step sizes are smaller, and thus will provide better accuracy. I will expand on this with the output controller problem where the function is rapidly changing.



The code for the odes:

```
x0 = [0;-50*pi/180;-40*pi/180;0;0;0];                    % Create the x0 state

% Compare odes
[t1,y1] = ode45(@(t1,y1) nonlinear(y1,x_e,u_e,K),span,x0);  % ODE45
[t2,y2] = ode23(@(t2,y2) nonlinear(y2,x_e,u_e,K),span,x0);  % ODE23

% Plot graphs
figure
hold on
sgtitle('States of the System')
```

```
subplot(6,1,1)
hold on
plot(t1,y1(:,1))
plot(t2,y2(:,1))
hold off
xlabel('Time (sec)')
ylabel('x [m]')
grid
subplot(6,1,2)
hold on
plot(t1,-y1(:,2)*180/pi)
plot(t2,-y2(:,2)*180/pi)
hold off
xlabel('Time (sec)')
ylabel('$\theta_1$[degrees]','Interpreter','latex')
grid
subplot(6,1,3)
hold on
plot(t1,-y1(:,3)*180/pi)
plot(t2,-y2(:,3)*180/pi)
hold off
xlabel('Time (sec)')
ylabel('$\theta_2$ [degrees]','Interpreter','latex')
grid
subplot(6,1,4)
hold on
plot(t1,y1(:,4))
plot(t2,y2(:,4))
hold off
xlabel('Time (sec)')
ylabel('$\dot{x}$ [m/s]','Interpreter','latex')
grid
subplot(6,1,5)
hold on
plot(t1,-y1(:,5)*180/pi)
plot(t2,-y2(:,5)*180/pi)
hold off
xlabel('Time (sec)')
ylabel('$\dot{\theta_1}$[degrees/s]','Interpreter','latex')
grid
subplot(6,1,6)
hold on
plot(t1,-y1(:,6)*180/pi)
plot(t2,-y2(:,6)*180/pi)
hold off
xlabel('Time (sec)')
ylabel('$\dot{\theta_2}$ [degrees/s]','Interpreter','latex')
```

```
58 legend('ODE45','ODE23')
59 grid
60 hold off
61
62 %% ============ Define Local Functions ===============
63 function[x_dot]= nonlinear(x,x_e,u_e,K)
64         u = -K * (x-x_e) + u_e;
65         x2 = x(2);
66         x3 = x(3);
67         x4 = x(4);
68         x5 = x(5);
69         x6 = x(6);
70         u1 = u(1);
71         u2 = u(2);
72         u3 = u(3);
73     % Insert Equations
74     x_dot = double([
75                         x4;
76                         x5;
77                         x6;
78                         -(420*u1 - (2943*sin(2*x2))/2 + 150*x5^2*sin(x2) ↩
                               + (135*x6^2*sin(x3))/2 - 180*u1*cos(2*x2 - 2*↩
                               x3) + 400*u3*cos(2*x2 - x3) - 840*u2*cos(x2) +↩
                                840*u3*cos(x2) - 400*u3*cos(x3) + (135*x6^2*↩
                               sin(2*x2 - x3))/2 + 360*u2*cos(x2 - 2*x3) - ↩
                               360*u3*cos(x2 - 2*x3))/(30*(5*cos(2*x2) + 9*↩
                               cos(2*x2 - 2*x3) - 26));
79                         (22800*u3 - 22800*u2 - 26487*sin(x2 - 2*x3) - ↩
                               91233*sin(x2) + 3600*u2*cos(2*x3) - 3600*u3*↩
                               cos(2*x3) + 4725*x6^2*sin(x2 - x3) + 750*x5^2*↩
                               sin(2*x2) - 4000*u3*cos(x2 + x3) + 4200*u1*cos↩
                               (x2) + 1350*x5^2*sin(2*x2 - 2*x3) - 1800*u1*↩
                               cos(x2 - 2*x3) + 13600*u3*cos(x2 - x3) + 675*↩
                               x6^2*sin(x2 + x3))/(150*(5*cos(2*x2) + 9*cos↩
                               (2*x2 - 2*x3) - 26));
80                         -(2700*sin(x2 - x3)*x5^2 + 1215*sin(2*x2 - 2*x3)*↩
                               x6^2 + 13600*u3 - 26487*sin(2*x2 - x3) + ↩
                               26487*sin(x3) - 4000*u3*cos(2*x2) + 1800*u1*↩
                               cos(2*x2 - x3) + 3600*u2*cos(x2 + x3) - 3600*↩
                               u3*cos(x2 + x3) - 1800*u1*cos(x3) - 12240*u2*↩
                               cos(x2 - x3) + 12240*u3*cos(x2 - x3))/(135*(5*↩
                               cos(2*x2) + 9*cos(2*x2 - 2*x3) - 26))
81                 ]);
82 end
```

Animate the Controller

I wanted to mix up the initial conditions- so I tried angles greater than the equilibrium point this time. My initial state:

$$x_0 = \begin{bmatrix} 0 & 80 & 60 & 0 & 0 & 0 \end{bmatrix}^T$$

Video of the state-feedback controller driving nonlinear model:

`https://youtu.be/9sabr2cWCIA`

As seen in the animation, the double pendulum still settles very quickly at the equilibrium state. I was hoping that changing the ICs would give a more interesting reaction, but everything still converged quickly.

The interface script:

```
%% Pre-Load Before Starting
load('Values_Rev3.mat')
Bus_Architecture()

%% Begin Script
Sim_Time = 15;                                  % Initialize Time

% Initial States
x0.x = 0;                                       % Meters
x0.theta1 = -80*pi/180;                         % Radians
x0.theta2 = -60*pi/180;                         % Radians
x0.x_dot = 0;
x0.theta1_dot = 0;
x0.theta2_dot = 0;

sim('Three_Input_Model.slx',Sim_Time)           % Simulate Model
logsout = ans.logsout;                          % Pull Out Data

% Plot state vs time
figure(1)
hold on
sgtitle('States of the System')
subplot(3,1,1)
hold on
plot(logsout{1}.Values.Time',logsout{1}.Values.Data)
hold off
xlabel('Time (sec)')
ylabel('x [m]')
grid
subplot(3,1,2)
hold on
plot(logsout{2}.Values.Time',-logsout{2}.Values.Data*180/pi)
hold off
```

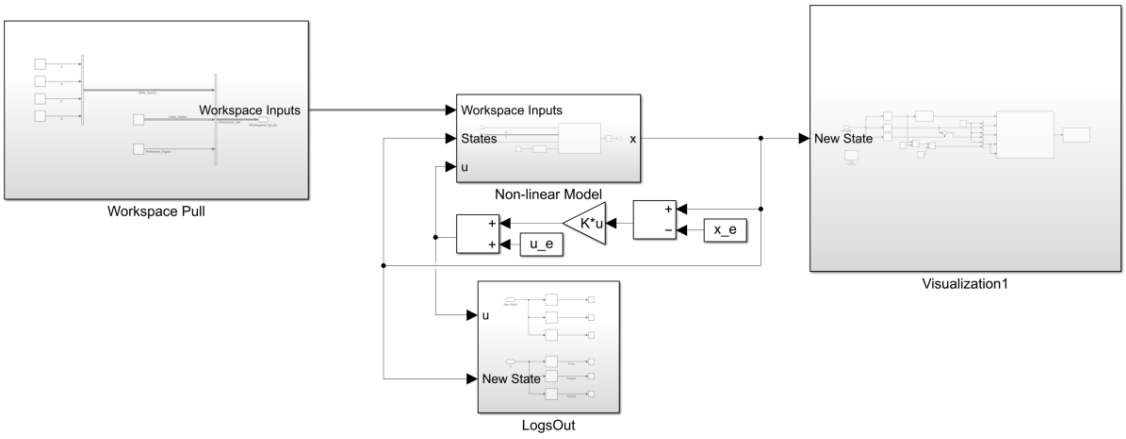
```
xlabel('Time (sec)')
ylabel('$\theta_1$[degrees]','Interpreter','latex')
grid
subplot(3,1,3)
hold on
plot(logsout{3}.Values.Time',-logsout{3}.Values.Data*180/pi)
hold off
xlabel('Time (sec)')
ylabel('$\theta_2$ [degrees]','Interpreter','latex')
grid
hold off
```

Bus architecture script (this will be more important when we start tracking later):

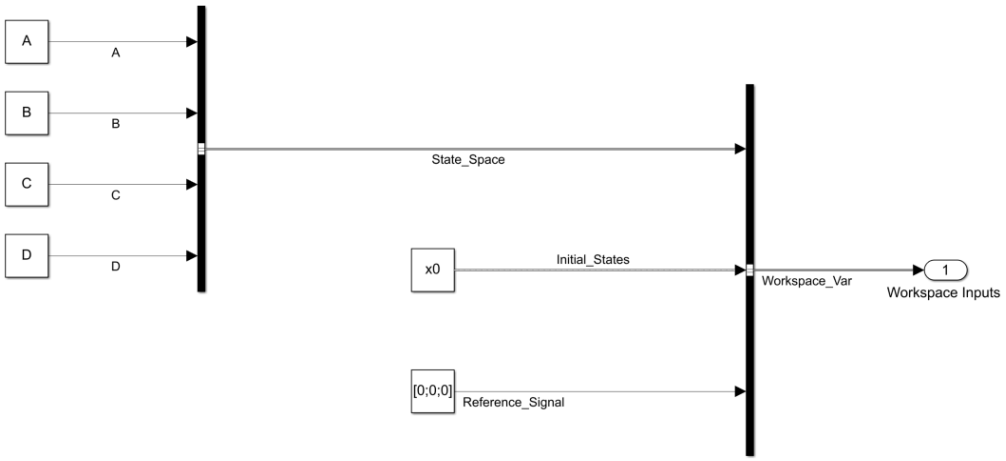
```
function Bus_Architecture()
    % Give Definition to Bus of States
    state_names = {'x','theta1','theta2',...
        'x_dot','theta1_dot','theta2_dot'};
    state_types = {'double','double','double',...
        'double','double','double'};
    for i = 1:length(state_names)
        temp(i) = Simulink.BusElement;
        temp(i).Name = state_names{i};
        temp(i).SampleTime = -1;
        temp(i).Complexity = 'real';
        temp(i).Dimensions = 1;
        temp(i).DataType = state_types{i};
    end
    State_bus = Simulink.Bus;
    State_bus.Elements = temp;
    assignin('base','State_bus',State_bus)
    clear temp state_names state_types

    % Give Definition to Bus of Refence
    ref_names = {'one','two',};
    ref_types = {'double','double'};
    for i = 1:length(ref_names)
        temp(i) = Simulink.BusElement;
        temp(i).Name = ref_names{i};
        temp(i).SampleTime = -1;
        temp(i).Complexity = 'real';
        temp(i).Dimensions = 1;
        temp(i).DataType = ref_types{i};
    end
    Reference_bus = Simulink.Bus;
    Reference_bus.Elements = temp;
```

```
33      assignin('base','Reference_bus',Reference_bus)
34      clear temp ref_names ref_types
35  end
```

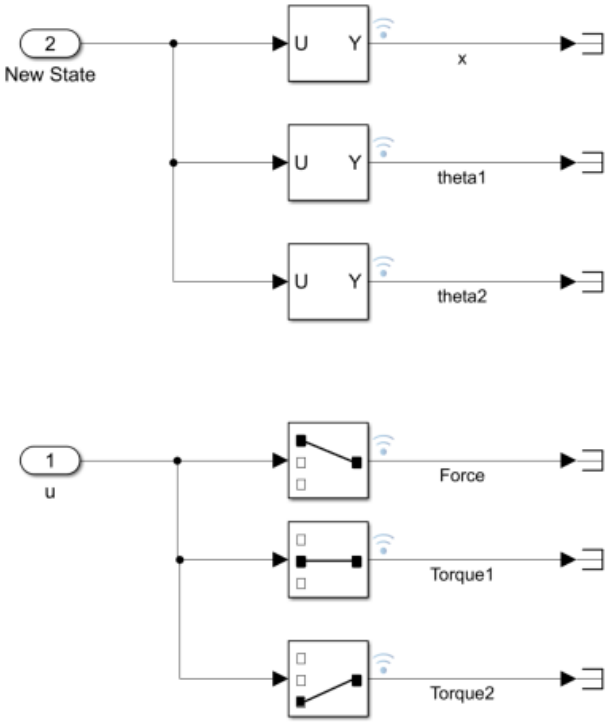
I will quickly walk through of the Simulink model. Below is the high level of the model that is able to break down further:



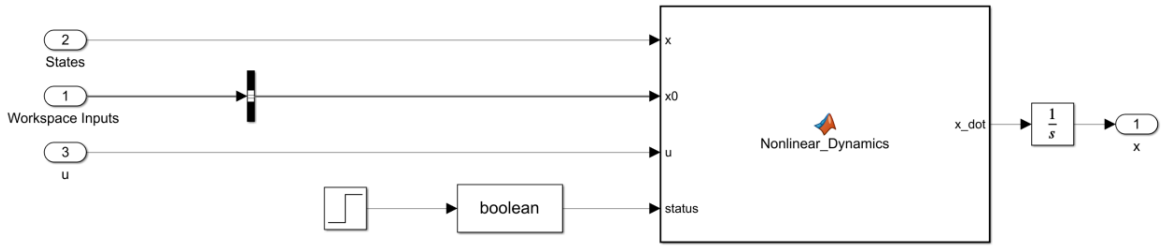
The block “Workspace Pull" pulls in workspace variables to the model.



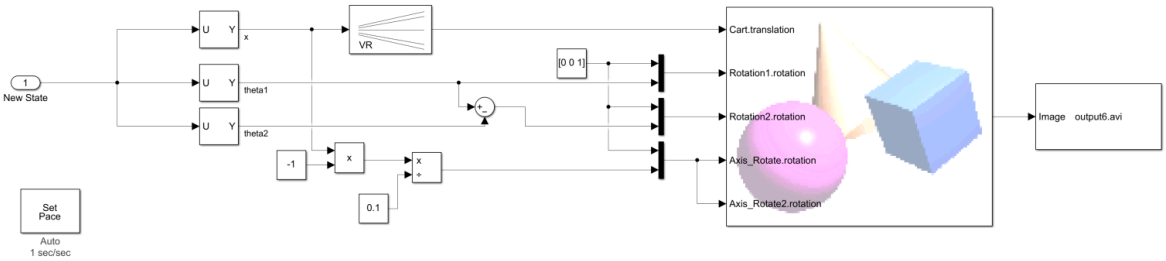
This is where I save all data so I can plot graphs later if desired.



The nonlinear dynamics of the model is contained within this block.



This is the visualization block that drives the animations. This block does not change from iteration to iteration.



Design an Output-Feedback Controller

We are now moving onto an output-feedback controller. This means our system and controller look like:

$$\dot{x} = Ax + Bu$$

$$y = Cx$$

$$u = -Ky$$

We notice that our controller no longer has the full state available for which to calculate the input, u. The closed-loop state matrix is now as follows:

$$A_c = A - BKC$$

Going to the Lyapunov equation:

$$A^T P + PA \prec 0$$

$$P \succ 0$$

Then combining the two:

$$(A - BKC)^T P + P(A - BKC) \prec 0$$

$$A^T P + PA - C^T K^T B^T P - PBKC \prec 0$$

We will notice that we have a BMI in K and P, and therefore cannot use LMI solvers. The next step is to define $BM = PB$ and $N = MK$. (Which I, personally, found interesting to note M and P are similar matrices since they have the same eigenvalues. $M^T = M$.) We begin exchanging with $BM = PB$:

$$A^T P + PA - C^T K^T M B^T - BMKC \prec 0$$

After exchanging with $N = MK$:

$$A^T P + PA - C^T N^T B^T - BNC \prec 0$$

$$BM = PB$$

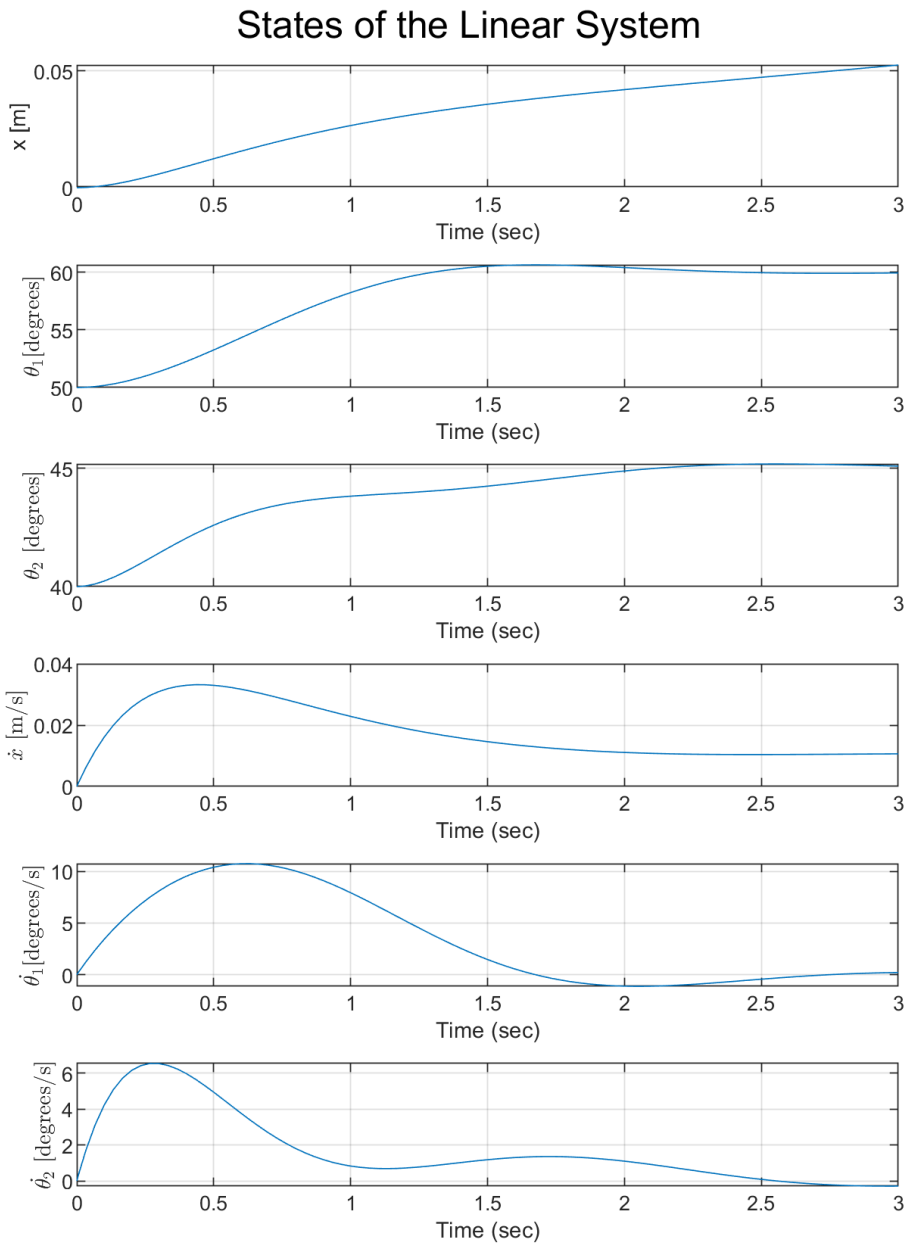
$$P \succ 0$$

These equations are LMIs. The first half of the first equation is linear in P and the second half is linear in N. We can now use the powerful LMI solver. The gain matrix K that we get:

$$K = \begin{bmatrix} -51.711 & 0.234 & -61.838 \\ 3.827 & 3.063 & 4.437 \\ 0.846 & -0.009 & 2.490 \end{bmatrix}$$

I will compare at the very end what happens if an αP term is added to this LMI equation.

The linear system graphs that follow:



It is interesting to note that the graphs look the same as the linear state controller (without the α term added). This is surprising because I expected results to depreciate a little. Still need to test the nonlinear model. The following is code used:

```
% Preliminaries
p = 3;
clear K
% LMI Solve
cvx_begin sdp quiet

```

```
% Variable definitons
variable P(n,n) symmetric
variable N(m,p)
variable M(m,m)

% LMIs
P*A + A'*P - B*N*C - C'*N'*B' <= -eps*eye(n)
B*M == P*B
P >= eps*eye(n)

cvx_end

K = M^-1*N;                                    % Solve for K matrix

% Initial States
clear x0
x0.x = 0;
x0.theta1 = -50*pi/180;
x0.theta2 = -40*pi/180;
x0.x_dot = 0;
x0.theta1_dot = 0;
x0.theta2_dot = 0;
sim('Linear_Controller_Design_Output.slx',3);

% Pull out Data
data = ans.logsout{1}.Values.Data;
time = ans.logsout{1}.Values.Time';

% Plot State vs Time
figure
hold on
sgtitle('States of the System')
subplot(6,1,1)
plot(time,data(:,1)')
xlabel('Time (sec)')
ylabel('x [m]')
grid
subplot(6,1,2)
plot(time,-data(:,2)'*180/pi)
xlabel('Time (sec)')
ylabel('$\theta_1$[degrees]','Interpreter','latex')
grid
subplot(6,1,3)
plot(time,-data(:,3)'*180/pi)
xlabel('Time (sec)')
ylabel('$\theta_2$ [degrees]','Interpreter','latex')
grid
```

```
54 subplot(6,1,4)
55 plot(time,data(:,4)')
56 xlabel('Time (sec)')
57 ylabel('$\dot{x}$ [m/s]','Interpreter','latex')
58 grid
59 subplot(6,1,5)
60 plot(time,-data(:,5)'*180/pi)
61 xlabel('Time (sec)')
62 ylabel('$\dot{\theta_1}$[degrees/s]','Interpreter','latex')
63 grid
64 subplot(6,1,6)
65 plot(time,-data(:,6)'*180/pi)
66 xlabel('Time (sec)')
67 ylabel('$\dot{\theta_2}$ [degrees/s]','Interpreter','latex')
68 grid
69 hold off
```

Looking at the following graphs on the next page, we will notice that the nonlinear model performs considerably worse with the output feedback controller compared to the state feedback controller. This makes intuitive sense since the gain matrix K went from a 3x6 matrix to a 3x3 matrix. The output controller now has to estimate the appropriate inputs with only three states available. With six states, the estimate can be much more refined. For this reason, one could guess the observer controller might perform better. The observer would provide six estimated states to determine an output.

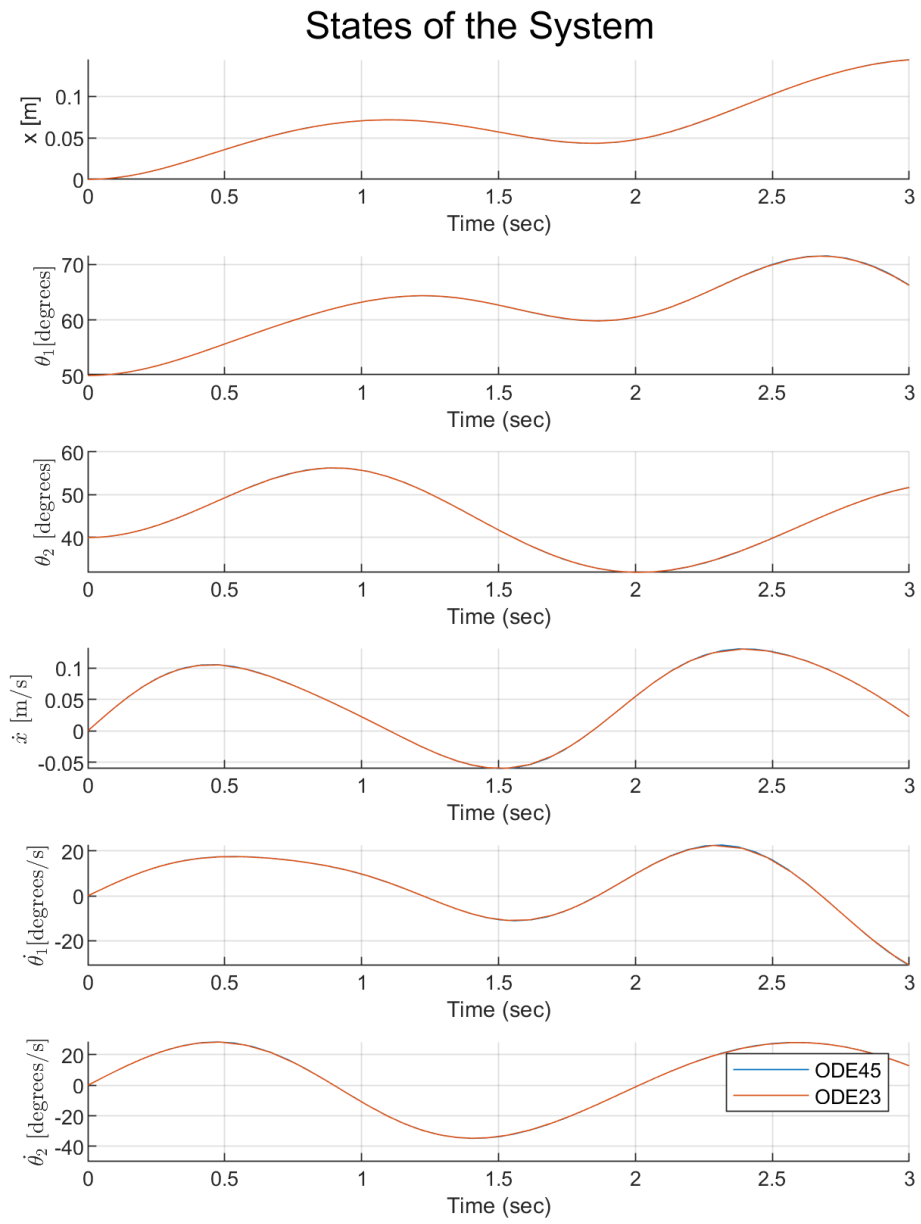
There exist two conditions for an output feedback controller to be able to place poles as desired. First, the system must be controllable and observable. We tested these properties in the linearization section, and found that the system was both controllable and observable. The second test is whether the system meets the condition $p + m \geq n + 1$. p is the rank of C, m is the rank of B, and n is the length of one side of the square matrix A. When plugging into the equation, we get $3 + 3 \geq 6 + 1 \rightarrow 6 \geq 7$, which is false. Therefore, we cannot guarantee the placement of all the poles with the gain matrix K.

Based on these outcomes of linear vs nonlinear responses, it is probably a good emphasis reason why we should look at how well the linear controller actually controls the nonlinear model (the real world model). I find this particularly interesting since my aerospace courses never really tied together the linear model with the nonlinear model.

As far as comparing the two odes against each-other, we notice that we can start to see a little of the blue sticking out from behind the orange. This becomes more noticeable as the simulation runs on, and errors are allowed to accumulate. From this, and observing that the state functions are not as smooth, we can say ode23 is more appropriate for smoother functions. If the function is constantly changing, as below, the ode45 will better capture the changes with its smaller step size.

There is only a small change of code from the previous problem. The controller now has the form of $\delta u = -K\delta y$, where $\delta y = C\delta x$ and the full x is not available to the designer. To fully

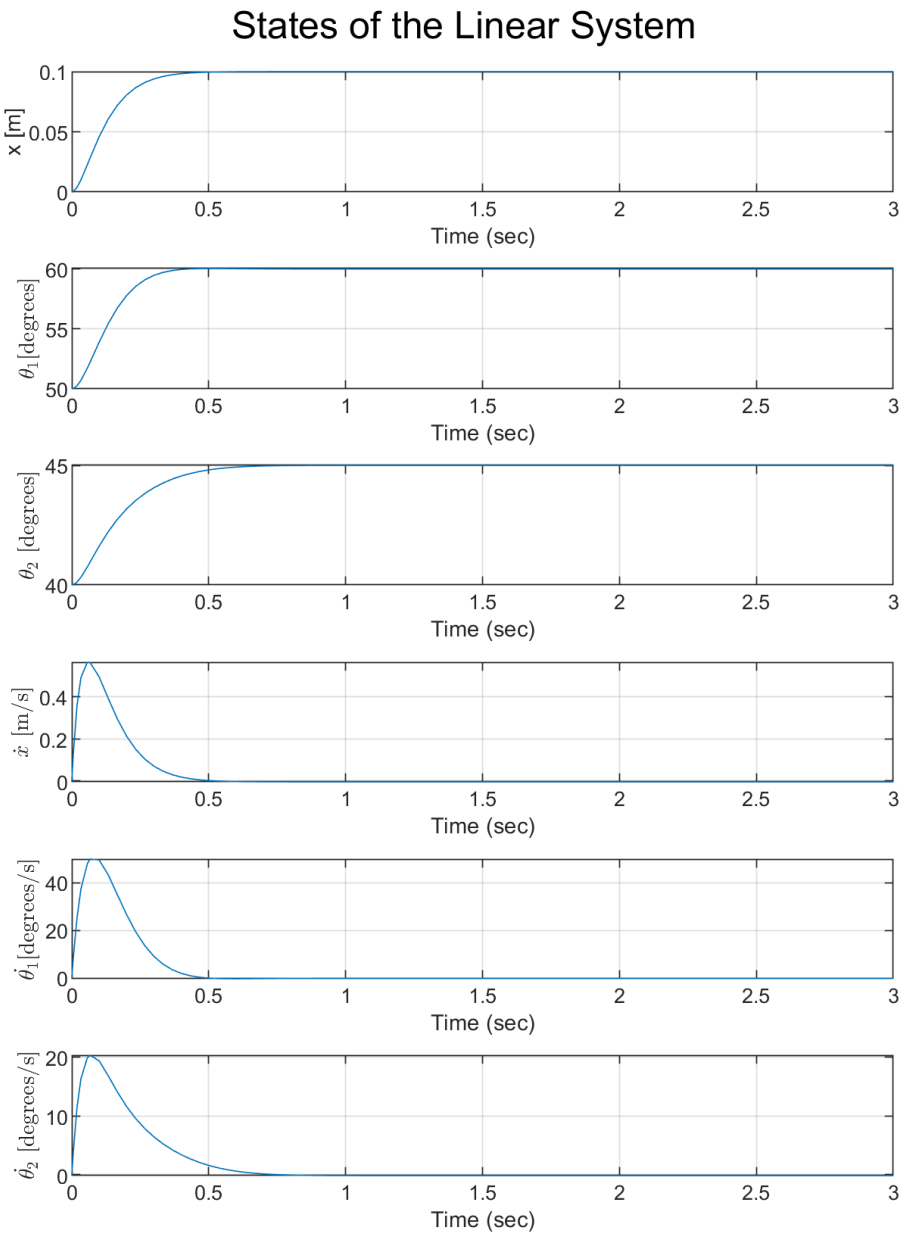
expand $\delta u = -K\delta y$ for the code, we get $u = -K\delta y + u_e$ and $\delta y = C(x - x_e)$.

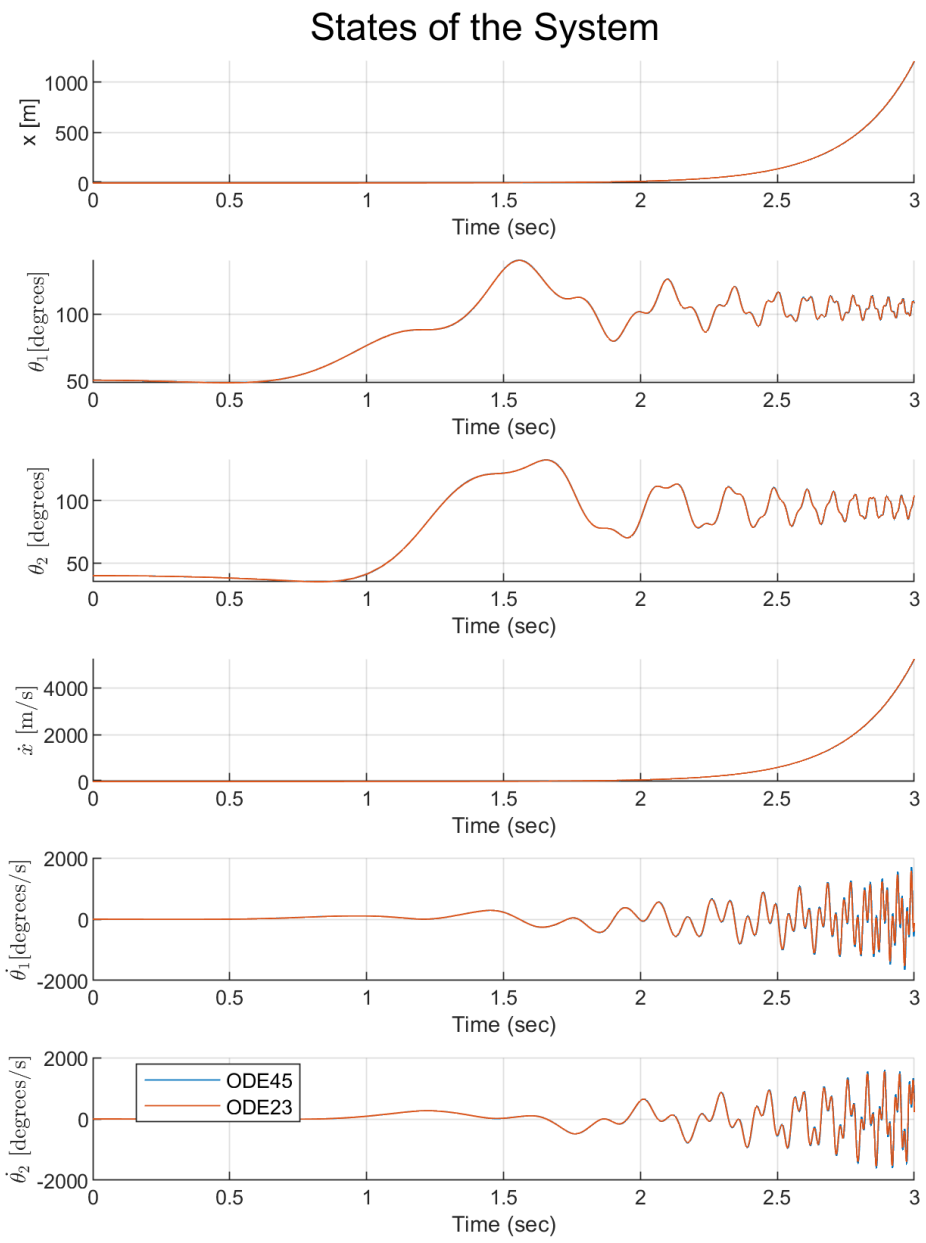


I did try to experiment and see what would happen if I added an αP term to the LMI equation. The next page contains the graphs. The linear model, yet again, performed similarly to the full state feedback controller. The response here is desirable. However, the nonlinear response is extremely undesirable. None of the responses get better- in fact they get worse. They go from oscillations with big periods to oscillations with an extremely short period. The system starts off relatively stable, and goes into into instability. Adding the α term just pronounced how quickly it goes to instability. Therefore, I left the α term out of this controller design.

Out of curiosity, I tested the eigenvalues of the closed-loop A without αP added, and found that the eigenvalues had non-asymptotically stable eigenvalues. Meaning the real parts were all 0 with just imaginary parts. When I added the αP term, two of the eigenvalues had their real parts

go positive (unstable), two of the eigenvalues had their real parts go negative (asymptotically stable), and then the other two's real parts stayed at zero. Making the overall system unstable.





The code:

```
clear y1 y2 t1 t2                                   % Delete old states and time
x0 = [0;-50*pi/180;-40*pi/180;0;0;0];               % Create the x state

% Compare odes
[t1,y1] = ode45(@(t1,y1) nonlinear_output(y1,x_e,u_e,K,C),span,x0);  % ODE45
[t2,y2] = ode23(@(t2,y2) nonlinear_output(y2,x_e,u_e,K,C),span,x0);  % ODE23
```

```
7
8  % Plot graphs
9  figure
10 hold on
11 sgtitle('States of the System')
12 subplot(6,1,1)
13 hold on
14 plot(t1,y1(:,1))
15 plot(t2,y2(:,1))
16 hold off
17 xlabel('Time (sec)')
18 ylabel('x [m]')
19 grid
20 subplot(6,1,2)
21 hold on
22 plot(t1,-y1(:,2)*180/pi)
23 plot(t2,-y2(:,2)*180/pi)
24 hold off
25 xlabel('Time (sec)')
26 ylabel('$\theta_1$[degrees]','Interpreter','latex')
27 grid
28 subplot(6,1,3)
29 hold on
30 plot(t1,-y1(:,3)*180/pi)
31 plot(t2,-y2(:,3)*180/pi)
32 hold off
33 xlabel('Time (sec)')
34 ylabel('$\theta_2$ [degrees]','Interpreter','latex')
35 grid
36 subplot(6,1,4)
37 hold on
38 plot(t1,y1(:,4))
39 plot(t2,y2(:,4))
40 hold off
41 xlabel('Time (sec)')
42 ylabel('$\dot{x}$ [m/s]','Interpreter','latex')
43 grid
44 subplot(6,1,5)
45 hold on
46 plot(t1,-y1(:,5)*180/pi)
47 plot(t2,-y2(:,5)*180/pi)
48 hold off
49 xlabel('Time (sec)')
50 ylabel('$\dot{\theta_1}$[degrees/s]','Interpreter','latex')
51 grid
52 subplot(6,1,6)
53 hold on
```

```
54 plot(t1,-y1(:,6)*180/pi)
55 plot(t2,-y2(:,6)*180/pi)
56 hold off
57 xlabel('Time (sec)')
58 ylabel('$\dot{\theta_2}$ [degrees/s]','Interpreter','latex')
59 legend('ODE45','ODE23')
60 grid
61 hold off
62
63 %% ============ Define Local Functions ===============
64 function[x_dot]= nonlinear_output(x,x_e,u_e,K,C)
65         delta_y = C * (x-x_e);
66         u = -K * delta_y + u_e;
67         x2 = x(2);
68         x3 = x(3);
69         x4 = x(4);
70         x5 = x(5);
71         x6 = x(6);
72         u1 = u(1);
73         u2 = u(2);
74         u3 = u(3);
75     % Insert Equations
76     x_dot = double([
77                         x4;
78                         x5;
79                         x6;
80                         -(420*u1 - (2943*sin(2*x2))/2 + 150*x5^2*sin(x2) ↩
                              + (135*x6^2*sin(x3))/2 - 180*u1*cos(2*x2 - 2*↩
                              x3) + 400*u3*cos(2*x2 - x3) - 840*u2*cos(x2) +↩
                               840*u3*cos(x2) - 400*u3*cos(x3) + (135*x6^2*↩
                              sin(2*x2 - x3))/2 + 360*u2*cos(x2 - 2*x3) - ↩
                              360*u3*cos(x2 - 2*x3))/(30*(5*cos(2*x2) + 9*↩
                              cos(2*x2 - 2*x3) - 26));
81                         (22800*u3 - 22800*u2 - 26487*sin(x2 - 2*x3) - ↩
                              91233*sin(x2) + 3600*u2*cos(2*x3) - 3600*u3*↩
                              cos(2*x3) + 4725*x6^2*sin(x2 - x3) + 750*x5^2*↩
                              sin(2*x2) - 4000*u3*cos(x2 + x3) + 4200*u1*cos↩
                              (x2) + 1350*x5^2*sin(2*x2 - 2*x3) - 1800*u1*↩
                              cos(x2 - 2*x3) + 13600*u3*cos(x2 - x3) + 675*↩
                              x6^2*sin(x2 + x3))/(150*(5*cos(2*x2) + 9*cos↩
                              (2*x2 - 2*x3) - 26));
82                         -(2700*sin(x2 - x3)*x5^2 + 1215*sin(2*x2 - 2*x3)*↩
                              x6^2 + 13600*u3 - 26487*sin(2*x2 - x3) + ↩
                              26487*sin(x3) - 4000*u3*cos(2*x2) + 1800*u1*↩
                              cos(2*x2 - x3) + 3600*u2*cos(x2 + x3) - 3600*↩
                              u3*cos(x2 + x3) - 1800*u1*cos(x3) - 12240*u2*↩
                              cos(x2 - x3) + 12240*u3*cos(x2 - x3))/(135*(5*↩
```

```
                        cos(2*x2) + 9*cos(2*x2 - 2*x3) - 26))
83                  ]);
84  end
```

Controller-Observer Compensator

We will start with the observer equation when setting up the problem so an LMI solver can solve it. The observer equation:

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

$$\dot{\tilde{x}} = (A - LC)\tilde{x} + Bu + Ly$$

We will use $(A - LC)$ as the closed-loop state matrix. Plug into the Lyapunov equation:

$$A^T P + PA \prec 0$$

$$P \succ 0$$

$$(A - LC)^T P + P(A - LC) \prec 0$$

Expand:

$$A^T P + PA - C^T L^T P - PLC \prec 0$$

This equation is a BMI in P and L. We can choose $Y = PL$ and rewrite:

$$A^T P + PA - C^T Y^T - YC \prec 0$$

With this equation we add an $\#\alpha P$ term for robustness- where $\alpha > 0$ and is fixed. We want the observer poles to be 2 to 6 times further back on the complex plane than the controller poles. After some experimentation, I made the number 5. Therefore, the final equations are:

$$A^T P + PA - C^T Y^T - YC + 5\alpha P \preceq 0$$

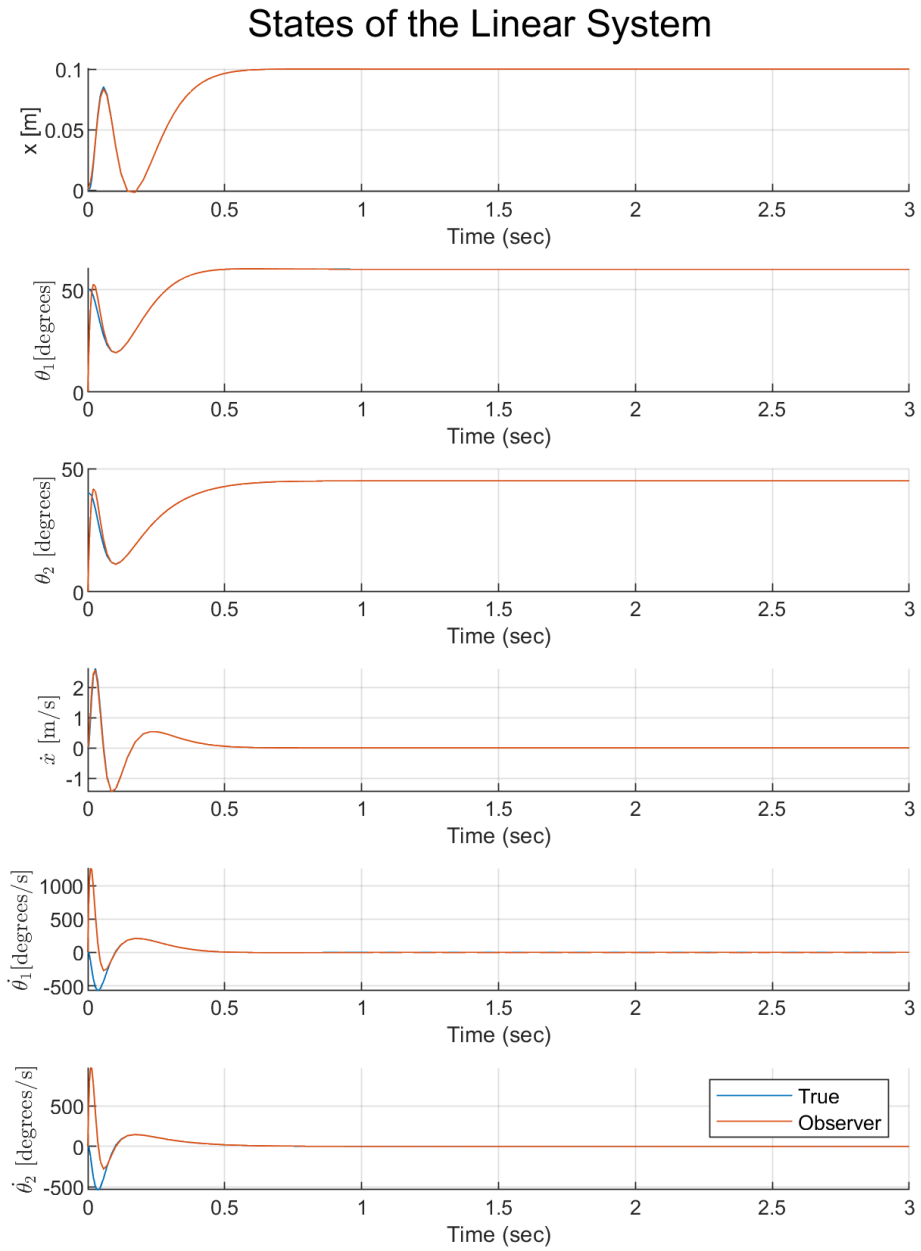
$$P \succ 0$$

We end up with the following L matrix:

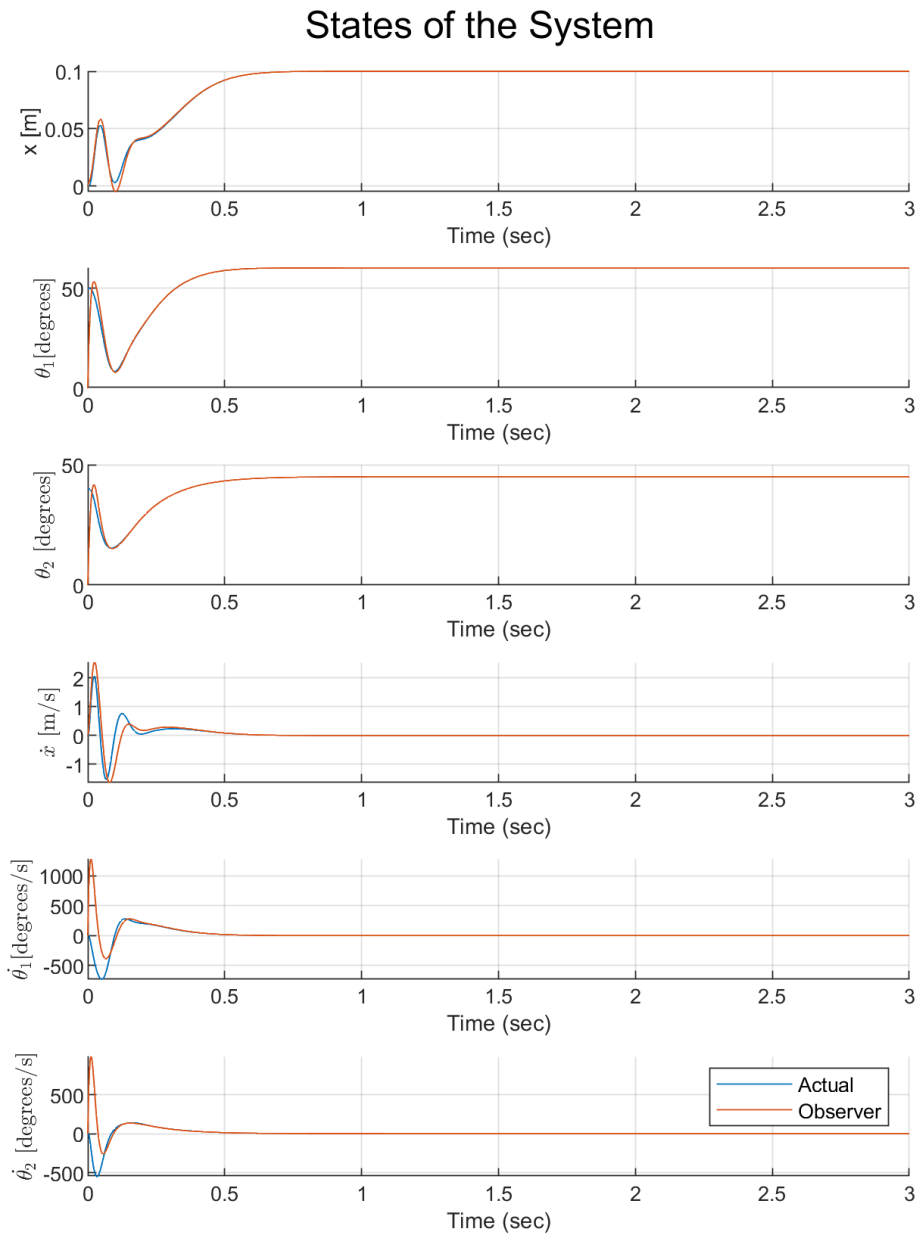
$$\begin{bmatrix} 117.002 & -0.095 & -1.384 \\ 0.873 & 167.383 & -47.015 \\ -0.455 & -45.544 & 163.545 \\ 4786.495 & -7.162 & -84.951 \\ 20.114 & 6947.733 & -1748.818 \\ -28.874 & -1574.659 & 6764.363 \end{bmatrix}$$

We will test this gain matrix with the linear model. The graphs will be on the next page. Looking at the graphs, we see that both the actual and observer states converge rapidly. The actual states mostly converge around 0.5 seconds. The observer states converged with the actual state around 0.1 seconds. Overall, the linear performance is desirable.

Linear model:



Nonlinear model:



Next, we look at the nonlinear model, and talk about how well the controller-observer compensator performs. This controller performs a little worse when compared to the full-state feedback controller, but that is to be expected. The compensator is working with estimates that are converging. Despite not performing as well, they still converged around the 0.5 second mark (just a little past it). Convergence of the estimation of the states take maybe 0.25 second with this design. Overall, I am very happy with this design.

Going onto the changes in the code. The observer is derived from the linear model, and therefore deals with perturbations. Therefore the observer should really be written as:

$$\frac{d}{dt}\delta\tilde{x} = A\delta\tilde{x} + B\delta u + L(\delta y - C\delta\tilde{x})$$

$$\delta\tilde{x} = \tilde{x} - x_e$$

$$\delta y = C(x - x_e)$$

$$\delta u = u - u_e$$

Where the nonlinear model only takes $\tilde{x}$ and u and not δu and $\delta\tilde{x}$. Therefore the input is $u = -K(\tilde{x} - x_e) + u_e$. These changes were noted when going to code the local function. I initialized the observer estimated values to start at:

$$\tilde{x} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

While the actual values started at:

$$x = \begin{bmatrix} 0 & 50 & 40 & 0 & 0 & 0 \end{bmatrix}^T$$

To use the ode solver for the observer, I had to stack the states together when passing them in, and then break them apart inside the local function. When I was done performing the code within the function, I would stitch the states back together before being output from the local function. The code is below. Linear Simulink show at the end printout since not necessary for this problem.

```
cvx_begin sdp

%Variable Definitions
variable P(n,n) symmetric
variable Y(n,p)

% LMIs
alpha = 12.3326;
P*A + A'*P - Y*C - C'*Y' + 5* alpha * P <= 0
P >= eps*eye(n)

cvx_end
L = P^-1*Y                              % Solve for L

clear x0
x0.x = 0;
x0.theta1 = -50*pi/180;
x0.theta2 = -40*pi/180;
x0.x_dot = 0;
x0.theta1_dot = 0;
x0.theta2_dot = 0;
sim('Linear_Control_ObserverLMI.slx',3);
open_system('Linear_Control_ObserverLMI.slx')


% Pull out Data
```

```
data = ans.logsout{1}.Values.Data;
data2 = ans.logsout{3}.Values.Data;
time = ans.logsout{1}.Values.Time';

% Plot State vs Time
figure
hold on
sgtitle('States of the Linear System')
subplot(6,1,1)
hold on
plot(time,data(:,1)')
plot(time,data2(:,1)')
hold off
xlabel('Time (sec)')
ylabel('x [m]')
grid
subplot(6,1,2)
hold on
plot(time,-data(:,2)'*180/pi)
plot(time,-data2(:,2)'*180/pi)
hold off
xlabel('Time (sec)')
ylabel('$\theta_1$[degrees]','Interpreter','latex')
grid
subplot(6,1,3)
hold on
plot(time,-data(:,3)'*180/pi)
plot(time,-data2(:,3)'*180/pi)
hold off
xlabel('Time (sec)')
ylabel('$\theta_2$ [degrees]','Interpreter','latex')
grid
subplot(6,1,4)
hold on
plot(time,data(:,4)')
plot(time,data2(:,4)')
hold off
xlabel('Time (sec)')
ylabel('$\dot{x}$ [m/s]','Interpreter','latex')
grid
subplot(6,1,5)
hold on
plot(time,-data(:,5)'*180/pi)
plot(time,-data2(:,5)'*180/pi)
hold off
xlabel('Time (sec)')
ylabel('$\dot{\theta_1}$[degrees/s]','Interpreter','latex')
```

```
74 grid
75 subplot(6,1,6)
76 hold on
77 plot(time,-data(:,6)'*180/pi)
78 plot(time,-data2(:,6)'*180/pi)
79 hold off
80 xlabel('Time (sec)')
81 ylabel('$\dot{\theta_2}$ [degrees/s]','Interpreter','latex')
82 grid
83 legend('True','Observer')
84 hold off
85
86 % Save Matrices for Easy Access
87 save('Values_Rev3.mat','A','B','C','D','f','K','x_e','u_e','L')
88
89 x0 = [0;-50*pi/180;-40*pi/180;0;0;0];                    % Create ↩
      the x state
90 obs0 = [0;0;0;0;0;0];
91 initial = [x0;obs0];
92
93 % Compare odes
94 [t1,y1] = ode45(@(t1,y1) nonlinear_cont_obsv(y1,x_e,u_e,K,L,A,C,B),↩
      span,initial);  % ODE45
95 [t2,y2] = ode23(@(t2,y2) nonlinear_cont_obsv(y2,x_e,u_e,K,L,A,C,B),↩
      span,initial);  % ODE23
96
97 % Plot graphs
98 figure
99 hold on
100 sgtitle('States of the System')
101 subplot(6,1,1)
102 hold on
103 plot(t1,y1(:,1))
104 plot(t1,y1(:,7))
105 hold off
106 xlabel('Time (sec)')
107 ylabel('x [m]')
108 grid
109 subplot(6,1,2)
110 hold on
111 plot(t1,-y1(:,2)*180/pi)
112 plot(t1,-y1(:,8)*180/pi)
113 hold off
114 xlabel('Time (sec)')
115 ylabel('$\theta_1$[degrees]','Interpreter','latex')
116 grid
117 subplot(6,1,3)
```

```
hold on
plot(t1,-y1(:,3)*180/pi)
plot(t1,-y1(:,9)*180/pi)
hold off
xlabel('Time (sec)')
ylabel('$\theta_2$ [degrees]','Interpreter','latex')
grid
subplot(6,1,4)
hold on
plot(t1,y1(:,4))
plot(t1,y1(:,10))
hold off
xlabel('Time (sec)')
ylabel('$\dot{x}$ [m/s]','Interpreter','latex')
grid
subplot(6,1,5)
hold on
plot(t1,-y1(:,5)*180/pi)
plot(t1,-y1(:,11)*180/pi)
hold off
xlabel('Time (sec)')
ylabel('$\dot{\theta_1}$[degrees/s]','Interpreter','latex')
grid
subplot(6,1,6)
hold on
plot(t1,-y1(:,6)*180/pi)
plot(t1,-y1(:,12)*180/pi)
hold off
xlabel('Time (sec)')
ylabel('$\dot{\theta_2}$ [degrees/s]','Interpreter','latex')
legend('Actual','Observer')
grid
hold off

%% ============ Define Local Functions ===============
function[y_dot]= nonlinear_cont_obsv(y,x_e,u_e,K,L,A,C,B)
        % Set-Up
        x = y(1:6);
        x_tilda = y(7:12);
        u_delta = -K * (x_tilda-x_e);
        x_delta = x - x_e;
        y_delta = C * x_delta;

        % Linear Part - Observer
        x_tilda_dot = (A - L*C)*(x_tilda-x_e) + B*u_delta + L*y_delta;

        % Define to move forward with Nonlinear Parts
```

```
165         u = -K * (x_tilda-x_e) + u_e;
166         x2 = x(2);
167         x3 = x(3);
168         x4 = x(4);
169         x5 = x(5);
170         x6 = x(6);
171         u1 = u(1);
172         u2 = u(2);
173         u3 = u(3);
174
175     % Insert Nonlinear Equations
176     x_dot = double([
177                         x4;
178                         x5;
179                         x6;
180                         -(420*u1 - (2943*sin(2*x2))/2 + 150*x5^2*sin(x2) ↩
                               + (135*x6^2*sin(x3))/2 - 180*u1*cos(2*x2 - 2*↩
                               x3) + 400*u3*cos(2*x2 - x3) - 840*u2*cos(x2) +↩
                                840*u3*cos(x2) - 400*u3*cos(x3) + (135*x6^2*↩
                               sin(2*x2 - x3))/2 + 360*u2*cos(x2 - 2*x3) - ↩
                               360*u3*cos(x2 - 2*x3))/(30*(5*cos(2*x2) + 9*↩
                               cos(2*x2 - 2*x3) - 26));
181                         (22800*u3 - 22800*u2 - 26487*sin(x2 - 2*x3) - ↩
                               91233*sin(x2) + 3600*u2*cos(2*x3) - 3600*u3*↩
                               cos(2*x3) + 4725*x6^2*sin(x2 - x3) + 750*x5^2*↩
                               sin(2*x2) - 4000*u3*cos(x2 + x3) + 4200*u1*cos↩
                               (x2) + 1350*x5^2*sin(2*x2 - 2*x3) - 1800*u1*↩
                               cos(x2 - 2*x3) + 13600*u3*cos(x2 - x3) + 675*↩
                               x6^2*sin(x2 + x3))/(150*(5*cos(2*x2) + 9*cos↩
                               (2*x2 - 2*x3) - 26));
182                         -(2700*sin(x2 - x3)*x5^2 + 1215*sin(2*x2 - 2*x3)*↩
                               x6^2 + 13600*u3 - 26487*sin(2*x2 - x3) + ↩
                               26487*sin(x3) - 4000*u3*cos(2*x2) + 1800*u1*↩
                               cos(2*x2 - x3) + 3600*u2*cos(x2 + x3) - 3600*↩
                               u3*cos(x2 + x3) - 1800*u1*cos(x3) - 12240*u2*↩
                               cos(x2 - x3) + 12240*u3*cos(x2 - x3))/(135*(5*↩
                               cos(2*x2) + 9*cos(2*x2 - 2*x3) - 26))
183                 ]);
184
185     % Put it Together Again
186     y_dot = [x_dot;x_tilda_dot];
187 end
```

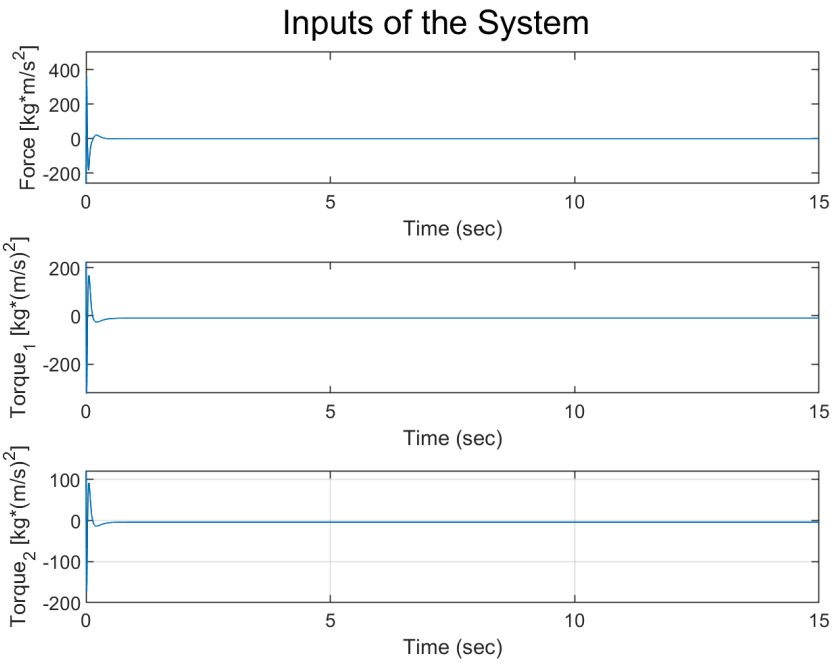
Animate the Controller-Observer

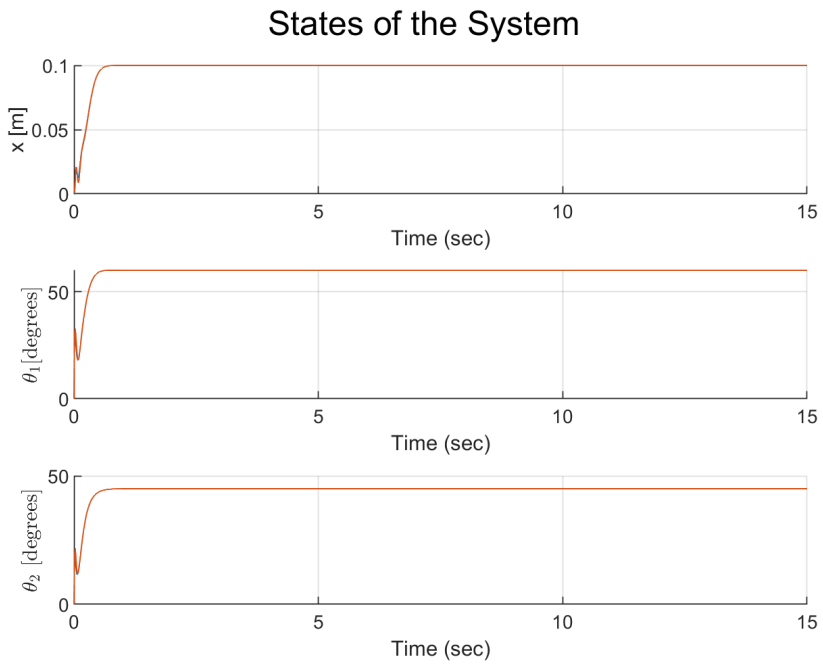
Video of the controller-observer compensator:

`https://youtu.be/GY-KvOjKK8E`

As we can see from the video and graphs, the controller ends up settling very quickly despite starting further away from the equilibrium initial condition. I do want to make a comment of the inputs into the system that make the DIPC to converge so quickly. If you look at the inputs, you will see that they exerting themselves beyond what is realistic. However, the point of this Funwork is not to make an admissible controller. I used the following initial condition:

$$x_0 = \begin{bmatrix} 0 & 30 & 20 & 0 & 0 & 0 \end{bmatrix}^T$$



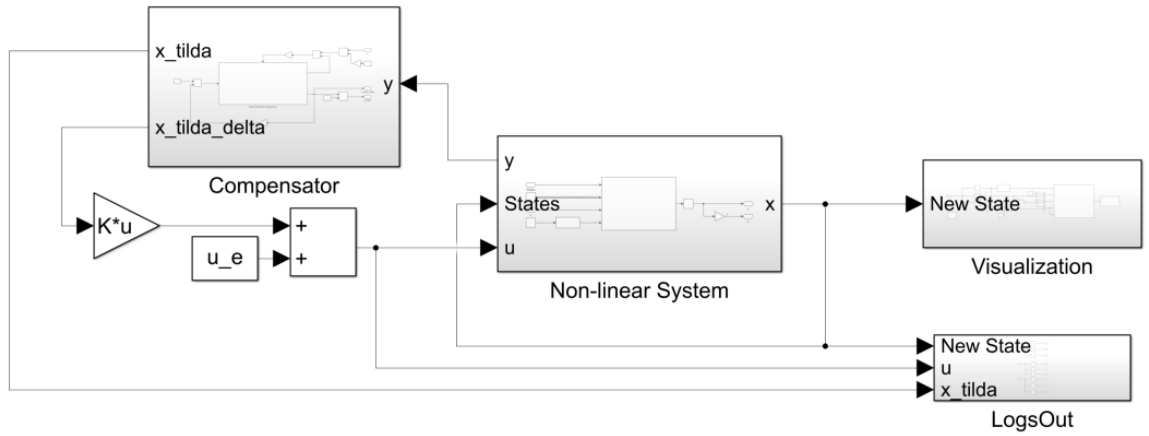


We talked about the Bus Architecture in the controller section of this report. The interface script:

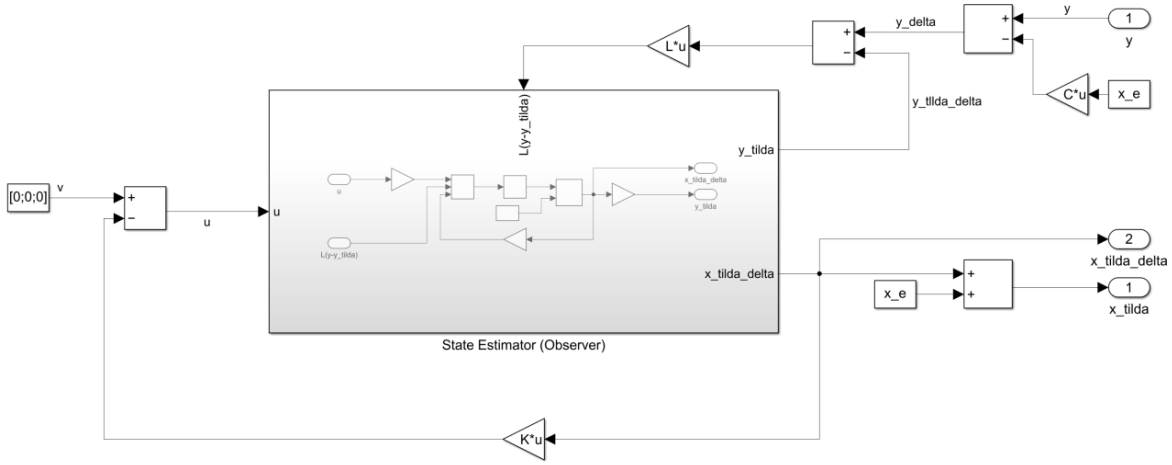
```
% Initial States
x0.x = 0;
x0.theta1 = -30*pi/180;
x0.theta2 = -20*pi/180;
x0.x_dot = 0;
x0.theta1_dot = 0;
x0.theta2_dot = 0;

% Reference Input
v.one = 0;
v.two = 0;
v.three = 0;
% Controller-Observer
sim('Three_Input_Model_CO.slx',15)
open_system('Three_Input_Model_CO.slx')
logsout = ans.logsout;

% Plot state vs time
figure(1)
hold on
sgtitle('States of the System')
subplot(3,1,1)
hold on
plot(logsout{1}.Values.Time',logsout{1}.Values.Data)
plot(logsout{1}.Values.Time',logsout{6}.Values.Data)
hold off
```

```
xlabel('Time (sec)')
ylabel('x [m]')
grid
subplot(3,1,2)
hold on
plot(logsout{2}.Values.Time',-logsout{2}.Values.Data*180/pi)
plot(logsout{2}.Values.Time',-logsout{7}.Values.Data*180/pi)
hold off
xlabel('Time (sec)')
ylabel('$\theta_1$[degrees]','Interpreter','latex')
grid
subplot(3,1,3)
hold on
plot(logsout{3}.Values.Time',-logsout{3}.Values.Data*180/pi)
plot(logsout{3}.Values.Time',-logsout{8}.Values.Data*180/pi)
hold off
xlabel('Time (sec)')
ylabel('$\theta_2$ [degrees]','Interpreter','latex')
grid
hold off

% Plot u vs time
figure(2)
hold on
sgtitle('Inputs of the System')
subplot(3,1,1)
plot(logsout{4}.Values.Time',logsout{4}.Values.Data)
xlabel('Time (sec)')
ylabel('Force [kg*m/s^2]')
subplot(3,1,2)
plot(logsout{5}.Values.Time',-logsout{5}.Values.Data)
xlabel('Time (sec)')
ylabel('Torque_1 [kg*(m/s)^2]')
subplot(3,1,3)
plot(logsout{6}.Values.Time',-logsout{9}.Values.Data)
xlabel('Time (sec)')
ylabel('Torque_2 [kg*(m/s)^2]')
grid
hold off
```

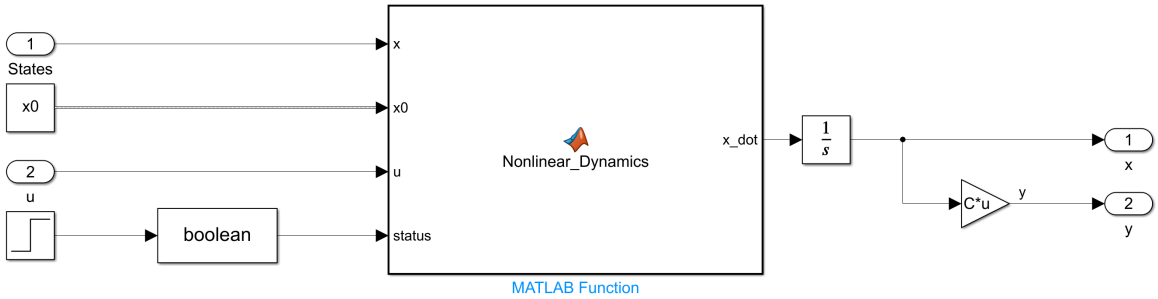
Some shots and explanation of the Simulink model follow. Below is the top level of the model. I have x feed into the visualization block just so I don't have to change the block from previous iterations. This way the visualization block will not be covered below for changes.



The observer is shown below:



The dynamics block has a few changes made to it from the previous iteration. I got rid of the workspace specific block just to make it a little easier for me to debug. Then since we do not have full-state feedback, I added the output for y to this block.



Then adding the estimated states to the outputs: