# ROCO219 - Inverted Pendulum Coursework 2020

Student No. 10618407

Wednesday 1st July, 2020

# Contents

# Chapter 1

# Theory and Design

## 1.1 State space model of the inverted pendulum

A state space model is a useful basis for demonstrating the control of a system such as an inverted pendulum. Therefore, we implement observation based state feedback control using the augmented positional state of the cart and integral control (these are added in later). Eq.(1.1) below shows the non-linear differential equation for the inverted pendulum kinematic model.

$$(I + ml^2)\frac{d^2\theta}{dt^2} + \mu\frac{d\theta}{dt} = mglsin(\theta) + ml\frac{d^2x_{\mathrm{p}}}{dt^2}cos(\theta) \tag{1.1}$$

This equation can be linearised about the pendulum's unstable equilibrium point. This gives Eq.(1.2), which describes an inverted pendulum's kinematic model.

$$(I + ml^2)\frac{d^2\theta}{dt^2} + \mu\frac{d\theta}{dt} = mgl\theta + ml\frac{d^2x_{\mathrm{p}}}{dt^2} \tag{1.2}$$

After many derivations we can write the state space model for the inverted pendulum as shown in Eq.(1.3). This takes into account the use of velocity control on the cart to stabilise the pendulum.

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix} = \begin{bmatrix}0 & 1\\ -a_2 & -a_1\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} + \begin{bmatrix}b_0\\ -a_1b_0\end{bmatrix}v_{\mathrm{c}} \tag{1.3}$$

Where:

$$a_1 = \frac{\mu}{(I + ml^2)} \tag{1.4}$$

$$a_2 = \frac{-mgl}{(I + ml^2)} \tag{1.5}$$

$$b_0 = \frac{ml}{(I + ml^2)} \tag{1.6}$$

For the inverted pendulum system, we were given the required constants to create a state space model. These are as follows:

- Gravity (g) = 9.81
- Length of pendulum rod (l) = 0.64m
- Mass of pendulum rod (M) = 0.314kg
- Coefficient of viscous friction ($\mu$) = 0.005
- Moment of inertia about the center of the rod (I) = $\frac{1}{12} * M * l^2 = 0.107 kgm^2$

For ease of calculation, the moment of inertia is considered to act about the middle of the pendulum. These constants are put into Eq.(1.4), Eq.(1.5), and Eq.(1.5). These are used to calculate $a_1$, $a_2$, and $b_0$. These values are used in Eq.(1.7), Eq.(1.8), and Eq.(1.9) to calculate the values of $A$, $B$, and $C$ respectively. These can be seen in the Matlab code below:

$$A = \begin{bmatrix} 0 & 1 \\ -a_2 & -a_1 \end{bmatrix} \tag{1.7}$$

$$B = \begin{bmatrix} b_0 \\ -a_1 b_0 \end{bmatrix} \tag{1.8}$$

$$C = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{1.9}$$

```
function [A,B,C] = stateSpaceModel ()

mu = 0.05;
g = 9.81;
l = 0.64;
M = 0.314;
I = (1/12)*M*l^2;

a1 = (mu)/(I + M*l^2);
a2 = -(M*g*l)/(I + M*l^2);
b0 = (M*l)/(I + M*l^2);

A = [0 1; -a2 -a1];
B = [b0; -a1*b0];
C = [1 0];

end
```

Figure 1.1: Code for calculating A, B, and C in Matlab

```
>> [A,B,C] = stateSpaceModel ()

A =

         0    1.0000
   14.1490   -0.3589


B =

    1.4423
   -0.5176


C =

     1     0
```

Figure 1.2: Outputs A, B, and C in Matlab

## 1.2 Design a Luenberger observer to estimate state

A Luenberger observer is used to estimate the system state of the inverted pendulum. This is done by using the state space model of the plant. To create a Luenberger observer in Matlab we use the place command along with $A$, $C'$ ($C$ transposed), and $P$. $A$ and $C$ are the matrices shown in Eq.(1.7) and Eq.(1.9) respectively. The values of $P$ are the poles of the system. $C'$ is used to

swap the orientation of $C$ from a 1x2 matrix to a 2x1. This allows the place command to work as shown below:

```
function [L] = stateSpacePlace (A,C)

P = 20 * [-1; -1.1];

L = place(A,C',P);

end
```

Figure 1.3: Code for creating a state space place

The output $L$ must be chosen such that the equation $A - LC$ yields eigenvalues with negative real parts. This causes the oscillations of the system to decay with time, making it stable. This is done by changing the poles of the system used in the place command. The output $L$ is shown below:

```
>> [L] = stateSpacePlace (A,C)

L =

   41.6411   31.0414
```

Figure 1.4: Output of L in Matlab

## 1.3 Augment positional state into the state space model

Although it is possible to design a controller using just the feedback control from the 2-dimensional state derived model, in practice we want to be able to control the position of the cart. Adding an augmented positional state into the state space model is used to control the velocity of the cart. This is done to bring the cart back to a known position (usually the center of the track). This prevents the cart from moving indefinitely. The positional state stops the cart from running out of track, which would cause the pendulum to become unbalanced and fall over. To do this, a third state, $x_3$, is added. This is used to represent the cart position. The differential of $x_3$

is equal to the velocity signal as shown in Eq.(1.10):

$$\frac{d}{dt}(x_3) = v_c \tag{1.10}$$

The new state space model for the inverted pendulum, including the augmented positional state, is shown in Eq.(1.11). The output of the system is shown in the Eq.(1.12).

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -a_2 & -a_1 & 0 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_0 \\ -a_1 b_0 \\ 1 \end{bmatrix} v_c \tag{1.11}$$

$$y = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{1.12}$$

## 1.4 Add integral action to the state space model

Integral control is used to reduce the state space controller's steady state error. To reduce this error, another term is added into the state space controller. The added term is the integral action. This compounds the positional errors forming a larger error over time. This helps the cart return to the absolute position set using the augmented positional state. To do this, a forth state, $x_4$, is added. The differential of $x_4$ is equal to $x_3 - r$, as shown in Eq.(1.13).

$$\frac{d}{dt}(x_4) = x_3 - r \tag{1.13}$$

For the inverted pendulum the value of $r$ is zero. This make the differential of $x_4$ equal to the value $x_3$, as shown in Eq.(1.14).

$$\frac{d}{dt}(x_4) = x_3 \tag{1.14}$$

The new state space model, for the inverted pendulum including the integral action, is shown

in Eq.(1.15). The output of the system is shown in the Eq.(1.16).

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -a_2 & -a_1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b_0 \\ -a_1 b_0 \\ 1 \\ 0 \end{bmatrix} v_c \tag{1.15}$$

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{1.16}$$

## 1.5 Design a state feedback controller

Once both augmented positional state and integral action have been added to the state space model, we can begin to design the complete state feedback controller as shown in the figures below:

```
function [A,B,C] = fullStateSpaceModel()

mu = 0.05;
g = 9.81;
l = 0.64;
M = 0.314;
I = (1/12)*M*l^2;

a1 = (mu)/(I + M*l^2);
a2 = -(M*g*l)/(I + M*l^2);
b0 = (M*l)/(I + M*l^2);

A = [0 1 0 0; -a2 -a1 0 0; 0 0 0 0; 0 0 1 0];
B = [b0; -a1*b0; 1; 0];
C = [1 0 0 0];

end
```

Figure 1.5: Code for creating a full state space model

```
>> [A,B,C] = fullStateSpaceModel()

A =

         0    1.0000         0         0
   14.1490   -0.3589         0         0
         0         0         0         0
         0         0    1.0000         0


B =

    1.4423
   -0.5176
    1.0000
         0


C =

     1     0     0     0
```

Figure 1.6: Output for A, B, and C in Matlab

To implement the full feedback state controller, we need to design the feedback gain, $K$. We use the values $\hat{x}_1$ and $\hat{x}_2$ from the Luenberger observer. To find the eigenvalues, $\lambda$, for the state feedback control system. We use its characteristic equation, as shown in Eq.(1.17).

$$|(A - BK - \lambda I)| = 0 \tag{1.17}$$

Using this equation, manipulating the location of the eigenvalues is straight forward. It is done by changing the value of $K$. This is done using the place command in Matlab as shown below:

```
function [K] = fullStateSpacePlace (A,B)

P = 4 * [-1 -1.1 -1.2 -1.3];

K = place(A,B,P);

end
```

Figure 1.7: Code for generating a full state space place

```
>> [K] = fullStateSpacePlace (A,B)

K =

   35.1501    8.9096  -28.0447  -31.0478
```

Figure 1.8: Output for the full state space place

# Chapter 2

# Implementation and Simulation

## 2.1 Implement the controller system using Euler integration

The implementation of the controller system, using Euler integration, uses three main sections of code to control the pendulum. The main control program, the state space integrator, and VCPendDotCB. These are all outlined below.

### 2.1.1 Main Control Program

The main control program runs through three main stages. The first stage is variable initialisation. The second stage is the main loop used to calculate the control values using the state space integrator. The third stage is used to plot the values calculated in the loop to use as an output animation for the pendulum.

#### Variable Initialisation

The variable initialisation starts by closing all of the open figures, clearing all of the values stored in the program and clearing the command window. This is done primarily for reassurance that the code is self contained and doesn't run any variables that may have been previously stored. This ensures a clean work space every time the program is run. The time points dictate how long each of the runs of the pendulum will be.

The variables for the simulation are setup. These values are the same as previously used to calculate the Lunberger observer and the state feedback gain. The values of $a_1$, $a_2$, and $b_0$ are

calculated. The transmission matrix, $D$, is initialised as a 4x1 matrix with all zeros. Although this could have been set up using a single value, 0, setting $D$ as a matrix allows you to change the values quickly. $L$ is then calculated, using a basic state space model of the inverted pendulum as in figures 1.3 and 1.4. $K$ is calculated using a full state feedback controller, incorporating augmented positional state and integral action, as in fig.(1.7) and fig.(1.8). The arrays xData, yData, tData, and kickFlag are also initialised.

```
close all
clear
clc

%Setup the time points
dt = 0.02; %increment value of the simulation
tFinal = 3; % final time value of the simulation
t = 0 : dt : tFinal;

%set up variables, [need to this twice, once for the lunenberger observer and
%once for the integral action]
mu = 0.05;
g = 9.81;
l = 0.64;
M = 0.314;
I = (1/12) * M * l^2;

a1 = (mu)/(I + M*l^2);
a2 = -(M*g*l)/(I + M*l^2);
b0 = (M*l)/(I + M*l^2);

D = [0 0 0 0];

%setting up the Lunenberger observer, L [2x2 matrices]
A = [0 1; -a2 -a1];
C = [1 0];
P = 10 * [-1; -1.1];
L = place(A, C', P);

%setting up the integral action gain, K [4x4 matrices]
A = [0 1 0 0; -a2 -a1 0 0; 0 0 0 0; 0 0 1 0];
B = [b0; -a1*b0; 1; 0];
C = [1 0 0 0];
P = 4 *[-1 -1.1 -1.2 -1.3];
K = place(A, B, P);

%Creates the title for the graphs, Title and Student No.
titleMessage = 'Controlled Inverted Pendulum: 10618407';
disp(titleMessage)

%initialise the arrays
xData=[];
yData=[];
tData=[];
kickFlag=[];
```

Figure 2.1: Initialisation of the parameters for the inverted pendulum

Setting the poles is important in controlling the motion of the pendulum. When using the poles shown in the initialisation code the pendulum is under control. The pendulum returns to its

equilibrium position within one oscillation. This is known as critical damping and is shown in the graph below:

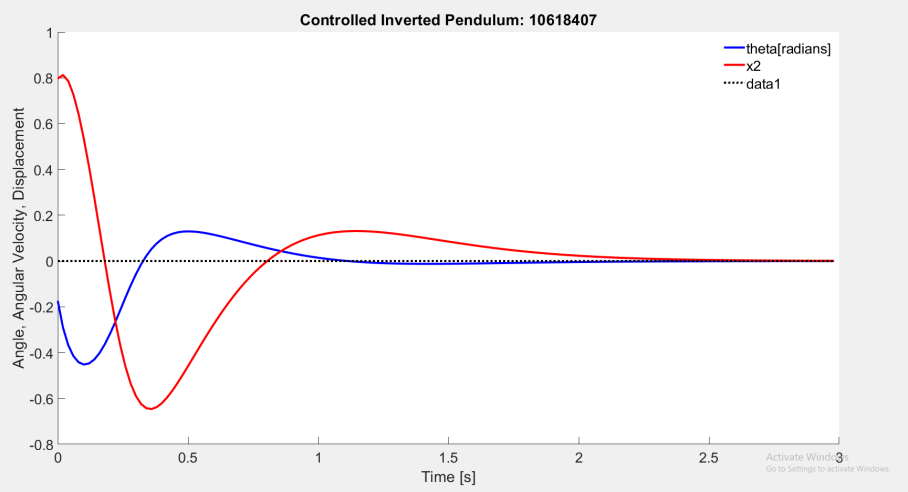


Figure 2.2: Single kick pendulum run with controlled poles

If the poles of the system are too small, as shown in fig.(2.4) below, the system will return back to its original position very slowly. This would slow down the response of the cart when the pendulum is falling. This is known as overdamping and is shown in the graph below:

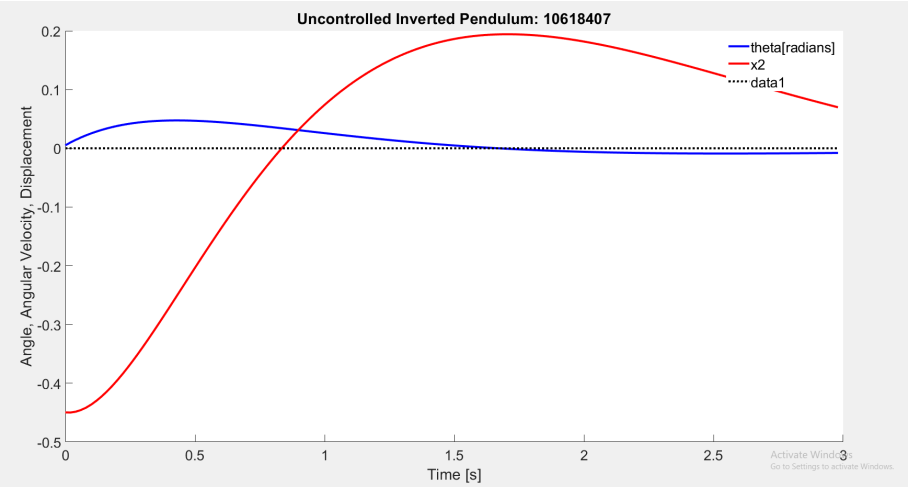


Figure 2.3: Initialisation code with poles that are too small

The poles to over damp the system were calculated using the code below:

```
%setting up the Lunenberger observer, L [2x2 matrices]
A = [0 1; -a2 -a1];
C = [1 0];
P = 1 * [-1; -1.1];
L = place(A, C', P);

%setting up the integral action gain, K [4x4 matrices]
A = [0 1 0 0; -a2 -a1 0 0; 0 0 0 0; 0 0 1 0];
B = [b0; -a1*b0; 1; 0];
C = [1 0 0 0];
P = 1 *[-1 -1.1 -1.2 -1.3];
K = place(A, B, P);
```

Figure 2.4: Single kick pendulum run with poles that are too small

If the poles of the system are too large, as shown in fig(2.6) below, the system will attempt to return to its original position too quickly. This is known as underdamping and will cause oscillation within the system. If these oscillations are not damped correctly, the system will become out of control because of the inherited instability of the controller with these pole values. In simulation, the pendulum and cart will travel to a distance nearing infinity. In reality, the cart will reach the end of the track and the pendulum will topple over. This is shown in the graph below:

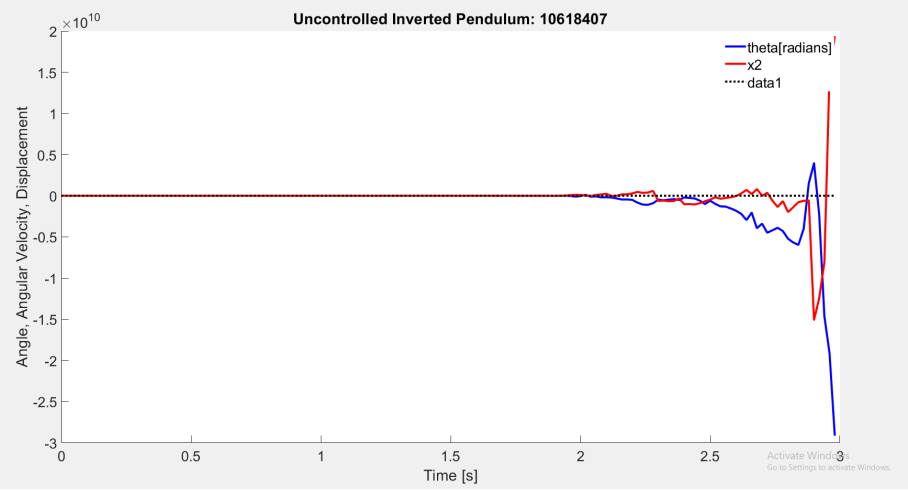


Figure 2.5: Initialisation code with poles that are too large

The poles to under damp the system were calculated using the code below:

```
%setting up the Lunenberger observer, L [2x2 matrices]
A = [0 1; -a2 -a1];
C = [1 0];
P = 15 * [-1; -1.1];
L = place(A, C', P);

%setting up the integral action gain, K [4x4 matrices]
A = [0 1 0 0; -a2 -a1 0 0; 0 0 0 0; 0 0 1 0];
B = [b0; -a1*b0; 1; 0];
C = [1 0 0 0];
P = 8 *[-1 -1.1 -1.2 -1.3];
K = place(A, B, P);
```

Figure 2.6: Single kick pendulum run with poles that are too large

### Main Loop

The variable 'runs', at the top of the loop, determines how many times the code will run (how many times the pendulum is 'kicked'). A pseudo-random set of initial conditions is generated for simulation purposes. If there were no initial conditions (all zero) the pendulum would remain perfectly upright, due to the noiseless nature of the simulation. The main loop is where the state space integrator function is called and run. The kick arrow is shown only at the start of each run so it does not obscure the view of the pendulum. The new data for each of the runs is concatenated with the old run. This is the data used for the animation, as shown in the code below:

```
%for the number of sub loop runs
runs = 3; %Pendulum does 3 runs
for kick = 1 : runs

    %for each run randomly generate the initial conditions
    x0 = [0; 3 * (rand - 0.5); 0; 0];

    %run the Euler integration
    [y, t, x] = myStateSpaceIntergrator (@VCPEndDotCB, a1, a2, b0, C, D, K, L, t, x0);

    %get the time
    newTime = (kick - 1) * t(end) + t;

    %show the kick only at the start of the run
    frames = length(t);
    kickFlagK = zeros(1,frames);
    if(x0(2) > 0)
        %scale arrow to size of kick
        kickFlagK(1: floor(frames/4)) = -abs(x0(2));
    else
        %scale arrow to size of kick
        kickFlagK(1: floor(frames/4)) = abs(x0(2));
    end

    %concatenate data between the runs
    xData = [xData x];
    yData = [yData y];
    tData = [tData newTime];
    kickFlag = [kickFlag kickFlagK];
end
```

Figure 2.7: Main loop for the pendulum code

### Plotting Values

The plotting variables code is used to output all of the data points calculated in the loops into the various plot functions required to run the pendulum. This includes the title message for all the figures:

```
%Plot all of the state variables
plotStateVariable(xData, tData, titleMessage);

%for all of the points, animate the results
figure
range = 1;

%sets where the cart is along the track
distance = yData(1,:);

%use the animate function
animatePendulumCart((yData + pi),  distance, 0.6, tData, range, kickFlag, titleMessage);
```

Figure 2.8: Plotted values for the main pendulum code

The plot functions animatePendulumCart and plotStateVariable were given to us previously and have been unchanged for this project.

### 2.1.2 State Space Integrator

The state space integrator is used to calculate the values for the state space model and the outputs $y$, $tout$, and $xout$. It is written in C-style, allowing the controller to be more easily implemented onto an Arduino or similar style microcontroller. This is useful for the real world implementation of the inverted pendulum. The loop inside the state space integrator is used to calculate the elements of the state space model for all of the frames of the animation. The state space integrator is shown in the figure below:

```
function [y, tout, xout] = myStateSpaceIntergrator (VCPEndDotCB, a1, a2, b0, C, D, K, L, t, x0)

% get signal length
len = length(t) -1;

% init output
y = zeros(1,len);
xout = zeros(4,len); %this has been updated from 2 to 4

% record the initial state
xout(:, 1, :, :) = x0;
x = x0;
xHat = x0;

%set the control for the system
u = - (K * xHat(1)) - (K(2) * xHat(2)) - (K(3) * x(3)) - (K(4) * x(4));

% for all remaining data points, simulate state-space model using C-language
%compatible formulation
for idx = 1:len

    % record time
    tout(idx) = t(idx);

    % get the duration between updates
    h = t(idx+1) - t(idx);

    %the observer estimated state feedback to compute U = -KX. This uses
    %the sfc gain, K, and xHat(1), xHat(2), x3, and x4
    u = - (K(1) * xHat(1)) - (K(2) * xHat(2)) - (K(3) * x(3)) - (K(4) * x(4));

    %calculate the real output using c compataile coding techniques
    y(1) = C(1) * x(1) + D(1) * u;
    y(2) = C(2) * x(2) + D(2) * u;
    y(3) = C(3) * x(3) + D(3) * u;
    y(4) = C(4) * x(4) + D(4) * u;

    % calculate state derivative from non-linear pendulum equations
    xDot = VCPEndDotCB(a1, a2, b0, x, u);

    %use the control velocity from the input
    xHatDot = VCPEndDotCB(a1, a2, b0, x, u);

    % update the state using Euler integration using c compataile coding techniques
    x(1) = x(1) + h * xDot(1);
    x(2) = x(2) + h * xDot(2);
    x(3) = x(3) + h * xDot(3);
    x(4) = x(4) + h * xDot(4);


    %calculating the observer correction term using c compataile coding
    %techniques
    ycorr(1) = L(1) * (y(1) - C(1) * xHat(1));
    ycorr(2) = L(2) * (y(2) - C(2) * xHat(2));

    %update the observer state xHat using Euler intergration using c
    %compataile coding techniques
    xHat(1) = xHat(1) + h * (xHatDot(1) + ycorr(1));
    xHat(2) = xHat(2) + h * (xHatDot(2) + ycorr(2));

    %record the state
    xout(:, idx) = x;

    % calculate output from theta and thetaDot state
    y(idx)= C(1) * xHat(1) + C(2) * xHat(2) + C(3) * xHat(3) + C(4) * xHat(4) + D(1) * u;
end

end
```

Figure 2.9: State space integrator code for the inverted pendulum

### 2.1.3 VCPendDotCB

VCPendDotCB is called inside of the state space integrator code. It is used to calculate the values of $\dot{x}_1$, $\dot{x}_2$, $\dot{x}_3$, and $\dot{x}_4$. This is shown in the figure below:

```
function [xDot] = VCPEndDotCB(a1, a2, b0, x, u)

%computing xDot(1)
xDot(1) = x(2) + b0 * cos(x(1)) * u;

%computing xDot(2)
xDot(2) = -a2 * sin(x(1)) - a1 * x(2) + (b0 * sin(x(1)) - a1 * b0 * cos(x(1))) * u;

%computing xDot(3)
xDot(3) = cos(x(1)) * u;

%computing xDot(4)
xDot(4) = x(3);

end
```

Figure 2.10: Output for the full state space place

The output of VCPendDotCB is the matrix $\dot{x}$ which is passed back into the state space integrator function.

## 2.2 Run the Matlab simulation

The simulation runs through all the frames for the pendulum. It displays a window that pops up onto the screen. This is the animation of the running pendulum, as shown in the figure below:

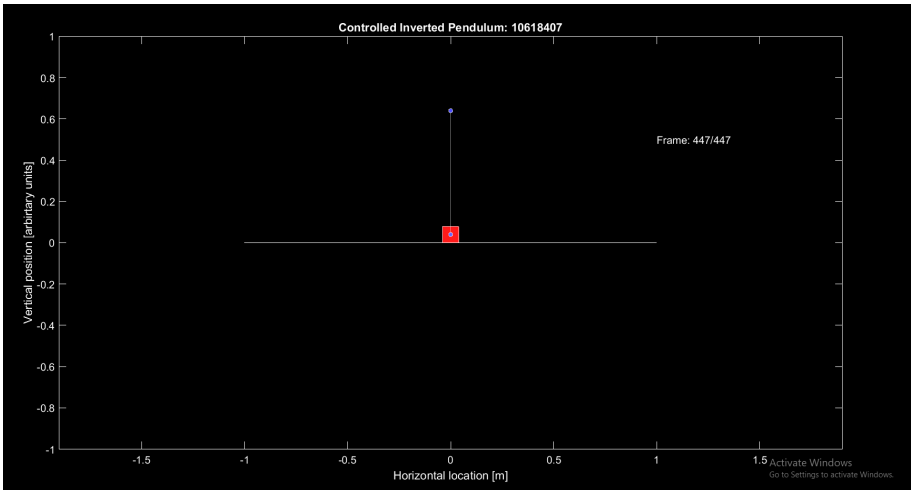


Figure 2.11: Final resting place of the inverted pendulum animation

To see the animation of the inverted pendulum use the following link: https://youtu.be/HhHBLydiXPE

The figure below shows the relative motions of the pendulum in accordance to the angle, angular velocity, and displacement of the cart and pendulum arm. It is demonstrated that the oscillations of the inverted pendulum are damped heavily and the pendulum returns to its upright equilibrium

state within two oscillations. The figure below shows the pendulum being controlled by the controller with effective critical damping:

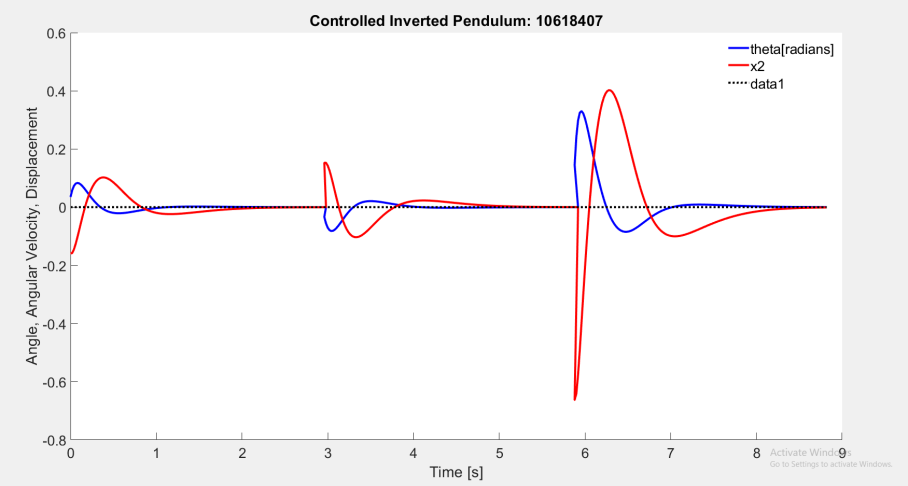


Figure 2.12: The graph for the velocity of the inverted pendulum