ROCO219 CONTROL ENGINEERING
LABORATORY REPORTS FOR COURSEWORK 2023
STUDENT NUMBER: XXXXXXXX

1. Uncontrolled pendulum simulation

From the uncontrolled pendulum simulation, we can see that the cart doesn't make any compensation for the arm moving. There is no feedback or target for the system to meet so the system does nothing.

2. Theory and design

In order to balance the pendulum, we are going to use 3 main things, feedback, Euler integration of a 4x4 model and an estimate calculation of what we are expecting the pendulum to do. The estimation, if done correctly, will help reduce how much the system must do. As the simulation is real-time the system can use the feedback sooner helping to correct the error sooner; if this system didn't have an estimate the system would react slightly slower therefore the system will have to work harder to correct the error because of the delay between readings.

For the real pendulum, we are going to simulate the first 2 states as we do not have access to the feedback needed for these; state 3 and 4 will be using the feedback from the real system as we have access to this data.

<u>The 4 states used</u>

X1 is the angle of the pendulum, this has feedback in the real system. This will use both the simulated (observer) gain and the real gain in the system. This will compare the 2 to get an error in the output.

X2 is the angular velocity of the system. This also has real feedback in the system; therefore, we will be using real gain for this as well as simulated.

X3 is used for the cart position, in the real system this will be estimated as we don't have feedback for this.

X4 is the integral of positional error. This is simulated as we don't have feedback for the system for this.

3. Parameter calculation in MATLAB

To make the code readable we split the parameters into separate files. We also shared values across the code so that we had consistency. This meant that there are no rounding issues later on in the code as all values are derived from the parameters.

The first thing we must do to get the pendulum to work is input the parameters for the pendulum. We need to ensure that the data is accurate as this will affect how similar the real and simulated pendulum is as well as the Leuenberger observer.

```
% put in approriate values here
params.m = 0.314;
params.lh = 0.32; %half langth
params.l = params.lh*2; %full length
params.g = -9.81;
params.mu = 0.05;
params.I = (1/3)*(params.m)*(params.l^2);    %inertia
```

Figure 1 (the parameter used)

It was later found that the formula I used for inertia was incorrect, this led to my simulation looking unnatural as well as the system now working when I applied it to the real-world pendulum. This shows how an inaccurate model of a system can lead to the system not working correctly or at all in my case.

4. Inverted pendulum Implementation and simulation

The simulation allowed us to see how we had made progress and weather we need to make any changes to the gains. While the estimate is fairly accurate, it is not the same as the real thing and therefore we will need to change certain things about the system in order for it to work with the real pendulum.

When we run the simulation, we can see that the pendulum balances as well as the cart position tending to 0 which us what we want.

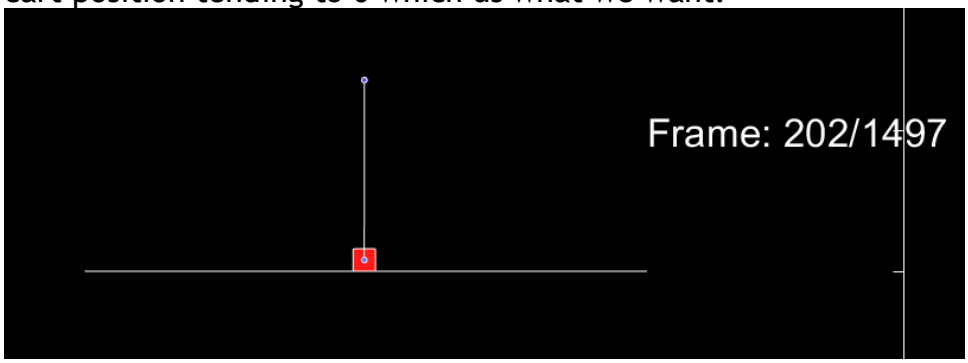


Figure 2 the system stabilising

So we can accurately gauge how the system works we graph the angle of the pendulum and the cart position.

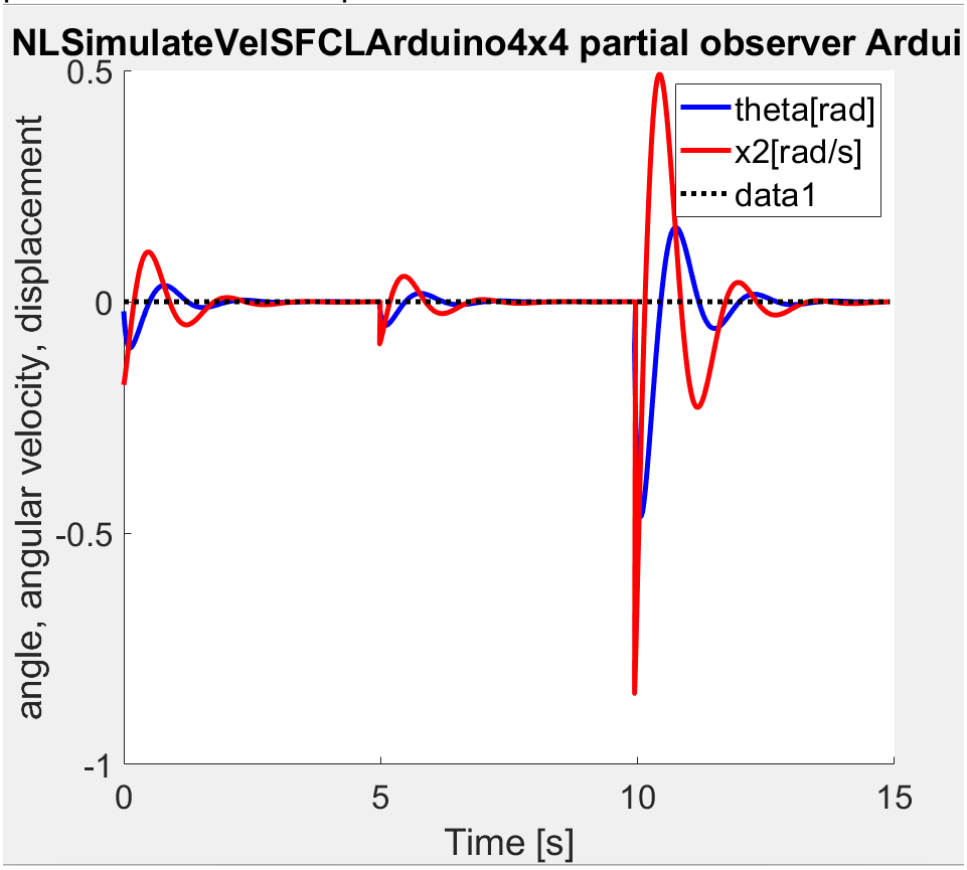


Figure 3

As we can see in figure 3, the system is underdamped as there is an oscillating delay before the system stabilises. To get the system to be critically damp we would have to play with the gain. We could also use a statistical method known as Kalman filtering, this would give us a more accurate gain for the system and would allow the system to stabilise quicker.

When playing with the poles for the system, it was found that if we make the poles for the first state too low the cart will become unstable and hit the sides of the rail.

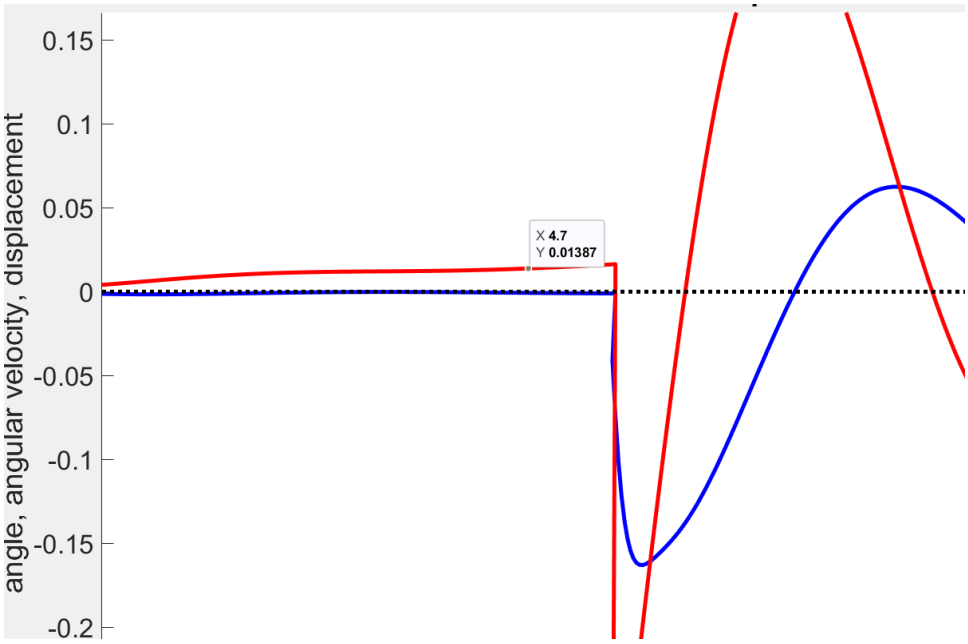


Figure 4

As we can see figure 4, when we change the pole for the cart position, the cart no longer tends to 0. the same is true for the second state.

As we decrease the state of the 3rd state, we see that the system stabilises quicker, but this does not take the cart position into account. This breaks the simulation if we have the poles too low. We need to take the constraints of the system into consideration when we are placing the poles as this will be the limitation of the real system; these constraints are the max speed of the cart and the max distance that the cart can travel.

When we decrease the pole for the 4th state, we can see that the system either doesn't settle or takes a long time to settle, this is not desirable in the real system

as we want the pendulum to balance.

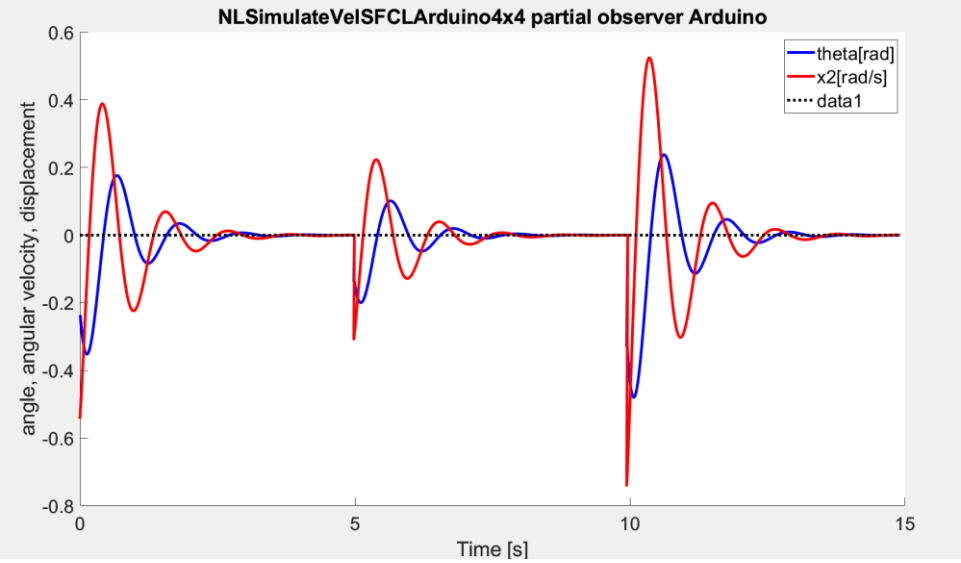


Figure 5

As we can see in figure 5 the cart takes a long time to reach a stable position.

When I was choosing my pole I ensured that the kicks I was getting were the same as this allowed me to better analyse the graph of the system. This was done by changing the initial conditions of the system (X0).

```
% add real task pendulum integration DFC loop here
%this add some kind of force to try and push the pendulum over as well
x0 = [0; 1* ( rand -1); 0; 0];
```

Figure 6 initial values

In figure 6 we can see the second state is the angular velocity, the term we have input is a random kick to the system to see if it stabilises. If we remove this and set it to something like -0.2, this will add a repeatable kick to the system that we

can use to gauge how well the system is stabilising

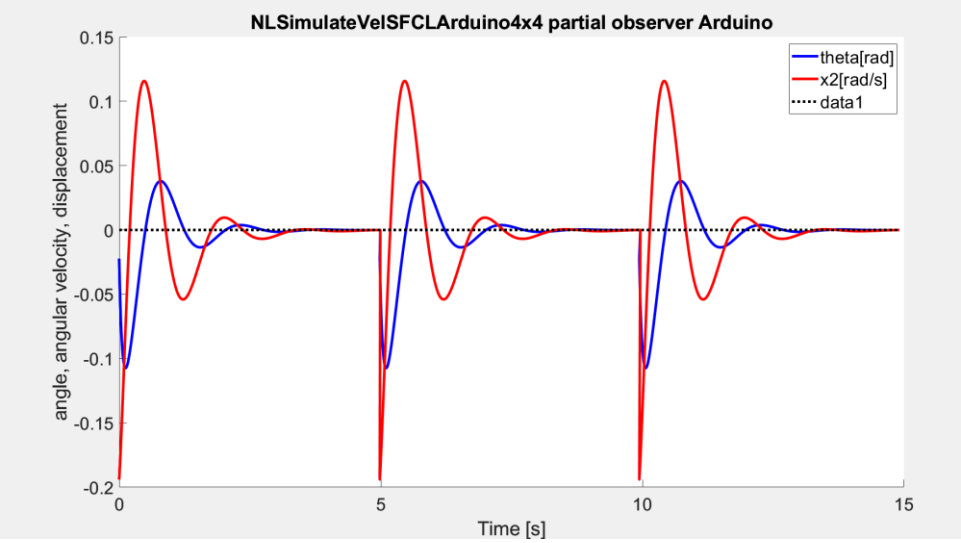


Figure 7

5. Implementation on real pendulum

To get the system working on the real pendulum we must convert the real system and the simulated system into Arduino. When we apply the system to the real world, we may find that some of the gains need changing as we have only designed the gains for the simulated system so far. We also need to remember that we are inputting pole in MATLAB rather than gain

```
% put your code for calculating L here
PX = [-6, -7];
L = place(ssm.A, ssm.C', PX);
disp(' ')
disp('L')
disp(L)

% for state space model with thetaDot, theta and  position of cart
% calculate SFC gain K here

 KX = [-0.8, -2, -3, - 4];
 K = place(ssmP.A,ssmP.B, KX);
 disp(' ')
 disp('K')
 disp(K)
```

Figure 8

```
L
   12.7545   -7.0299


K
   -1.2651   -9.2824    9.0541    3.9666
```

Figure 9

```
double K[4] = {-1.2651,-9.2824, 9.0541, 3.9666};
//these need to be + as it smothes out the response


// ENTER YOUR VALUES FOR THE OBSERVER GAINS HERE
// observer gain just for theta and thetadot
double L[2] = {12.7545, -7.0299};
```

Figure 10

To obtain the gains we need to either input the gain variable into the command window or get MATLAB to print the gain when the code runs.

When converting the MATLAB code into C++ we also need to remember that C++ does not understand matrix multiplication. We will therefore need to do each multiplication individually using an array. An example of this is the y correction term

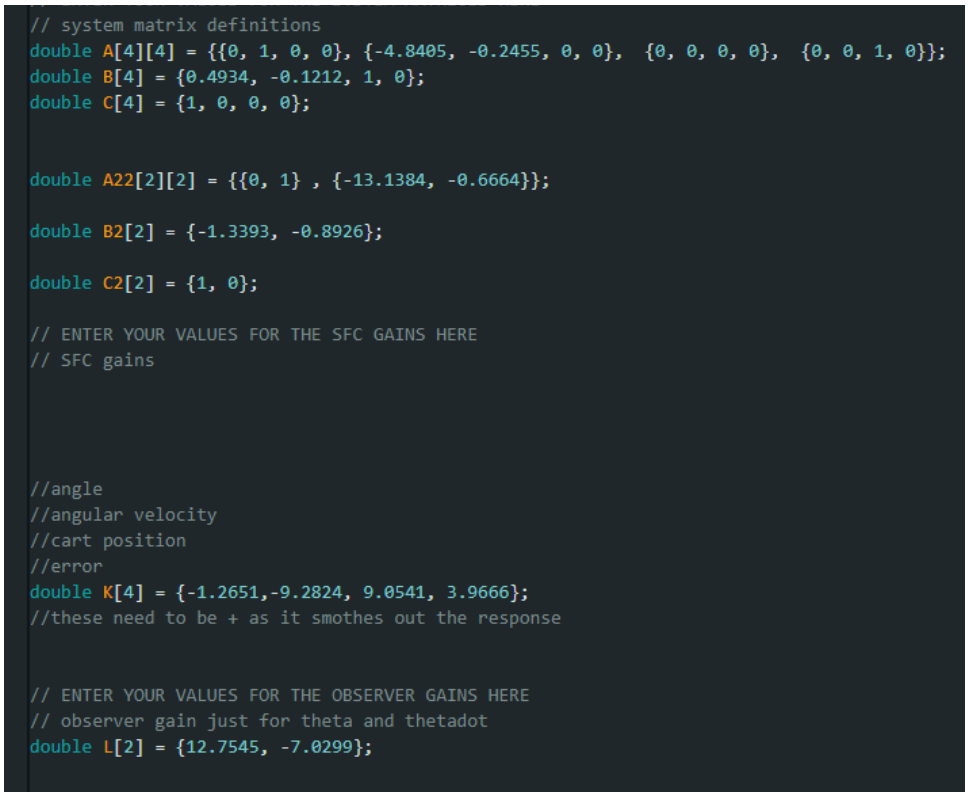
```
// calculate observer correction term
yCorr = (y - setPointAngle) - C[0] * xhat[0] - C[1] * xhat[1];
```

Figure 11

As we can see in the C++ implementation, we have multiplied each term individually rather than doing the whole matrix. As well as that we need to take the output of the real system so that the real output can be using in the feedback.

```
%estimated
yCorr = L'*(y- ssm.C * xhat);
```

Figure 12

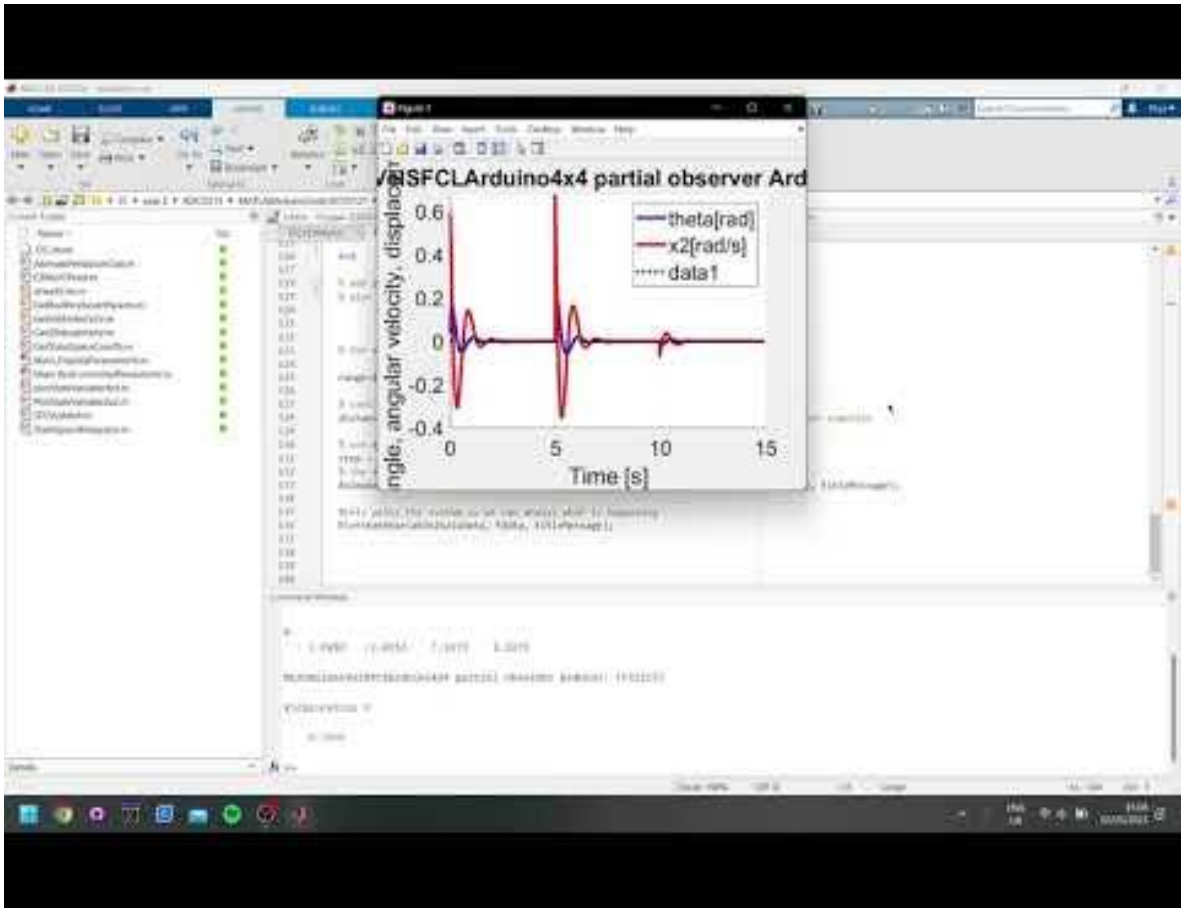


```
// system matrix definitions
double A[4][4] = {{0, 1, 0, 0}, {-4.8405, -0.2455, 0, 0},  {0, 0, 0, 0},  {0, 0, 1, 0}};
double B[4] = {0.4934, -0.1212, 1, 0};
double C[4] = {1, 0, 0, 0};


double A22[2][2] = {{0, 1} , {-13.1384, -0.6664}};

double B2[2] = {-1.3393, -0.8926};

double C2[2] = {1, 0};

// ENTER YOUR VALUES FOR THE SFC GAINS HERE
// SFC gains




//angle
//angular velocity
//cart position
//error
double K[4] = {-1.2651,-9.2824, 9.0541, 3.9666};
//these need to be + as it smothes out the response


// ENTER YOUR VALUES FOR THE OBSERVER GAINS HERE
// observer gain just for theta and thetadot
double L[2] = {12.7545, -7.0299};
```

Figure 13

When entering the parameters for the matrixes I found it easier to do all the math in MATLAB and copy that data into the arrays in C++. As these parameters won't be perfect this is okay as the system won't be perfect in the real world. This also means that there won't be any issues with a miscalculation or incorrect entry of the data.

Video of the real system working: https://youtube.com/shorts/iXPnF4RmJrI
Video of simulation: simulation



As we can see in the video the pendulum wants to return to its starting position even when we add weight and knock it, this is a result of Euler integration. The integration wants the system to tend back to the rod starting position (0).

If we were to do this project again, we would use Kalman filtering as this would give me better gain to help get my system closer to being critically damped. Critically damping the system would mean the system would be able to react to changes sooner allowing the system to react to more disturbances.