# CMPE243 Final Project

Joseph Adamson

June 12 2019

## Introduction

The project discussed in this document aims to identify an unknown system using the techniques learned during CMPE243. The system consists of a propeller mounted to a servo mechanism, with the whole system being free to rotate about a pivot point. The propeller thrust is constant, with the controller able to adjust the angle at which the propeller directs the thrust. Several modelling options are explored, and the results tested with a PID controller design.

## 1 Servo ID

### 1.1 Background

The first step of the project is to identify the discrete transfer function for the servo mechanism moving the propeller. This subsystem contains its own dynamics that we wish to model.

The arx model is used for this identification. A first-order model is used, since the continuous-time version (given in the assignment as $\frac{b}{s+a}$) is also first-order. This assumption was made based upon discussions in lecture, however we could multiply by 1/s, perform a partial fraction decomposition, and apply the inverse laplace- and forward z-transforms to see that the model works out to be of this form. The model is described as follows:

$$y(t) = -a_1 y(t-1) + b_1 u(t-1)$$

Where t is a discrete integer. Noting that this expression can be written as a matrix equation, the system ID procedure for this model consists of taking a large amount of data and using it to populate:

- $y$: vector of collected output data
- $\theta$ : unknown vector of model parameters
- $S$ : Sensitivity matrix containing past input/output data points. Contains more columns if system is higher order

Then, least-squares matrix inversion $S^\dagger = (S^T S)^{-1} S^T$ can be used to estimate the parameters contained in $\theta$, and plugged back into the above equation:

$$\theta = S^\dagger y$$

## 1.2 Application to servo

To identify the servo, the provided v-rep scene was used to take 60 seconds of data. The most recent 1000 points were taken and used to form the matrices discussed above. Note that these vectors begin at $y_2$ since we need to 'look back' one sample in the first-order system:

$$\mathbf{Y} = \begin{bmatrix} y_2 \\ ... \\ y_{1000} \end{bmatrix}$$

$$\mathbf{S} = \begin{bmatrix} -y_1, u_1 \\ ... \\ -y_{k-1}, u_{k-1} \end{bmatrix}$$

Using a script in MATLAB to apply the least-squares math to this setup results in the following difference equation describing the servo angle $\alpha$:

$$\boxed{\alpha(t) = -.3670_1 y(t-1) + .6327 u(t-1)}$$

This recurrence relation was used to generate a discrete transfer function in order to perform simulations. For comparison, a system model was also generated using the built-in MATLAB function arx() using first-order parameters. The results of these simulations are compared with the raw data below:

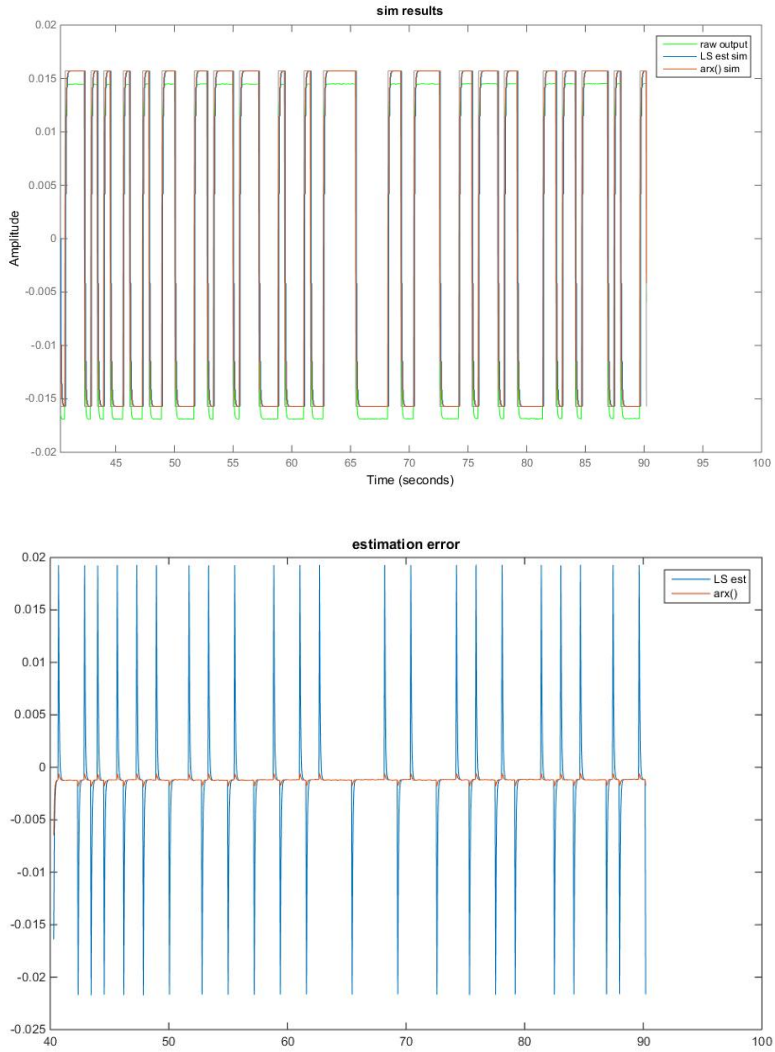



Figure 1: Estimated system response and error

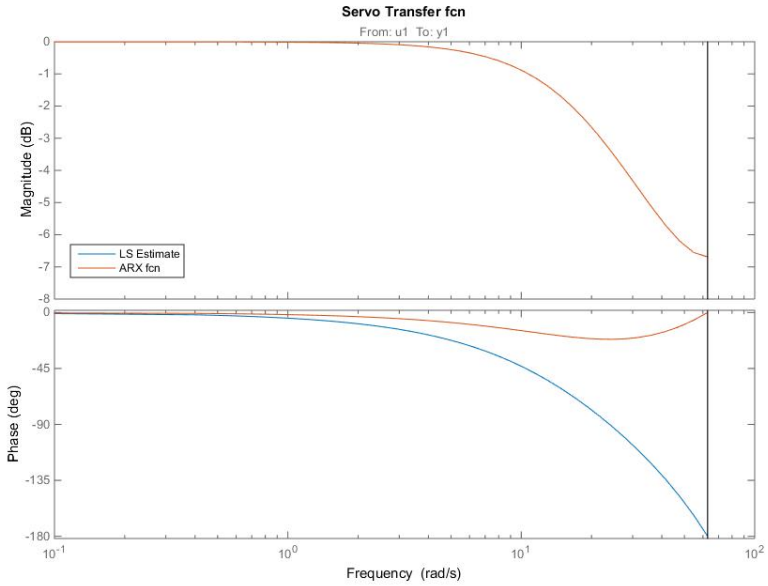



Figure 2: Bode plot of estimated systems

Even though the arx function produces smaller error, we will proceed with the LS model. This is because its phase characteristics match much more closely with the theoretical models of single-integrator systems we have studied in many previous classes. The gain characteristics are exactly the same.

It is important to discuss the fact that the data collected for alpha has a non-zero-mean, which is reflected in the offset present in figure 1 (shown in green)

I don't fully understand the source of this, and I tried running my script both with and without removing the mean from alpha before performing the estimations. I have reached the following conclusions:

- When the mean is removed from the collected data, the offset in the estimated model is not present and the single-step prediction error appears *very nearly white* indicating the model has 'squeezed out' all the information
- When the mean is not removed from the collected data, the estimation error spikes at every change in u, indicating that the model does not capture certain dynamics, but was estimated from non-altered data.

I will proceed with the script which does NOT remove the mean, since I don't want to just alter the data without understanding why it makes the prediction error white. If this model proves to be bad (doesn't work with controller design

in part 4) I will revisit this and subtract the mean. For comparison, the two versions of the one-step prediction error are provided below:

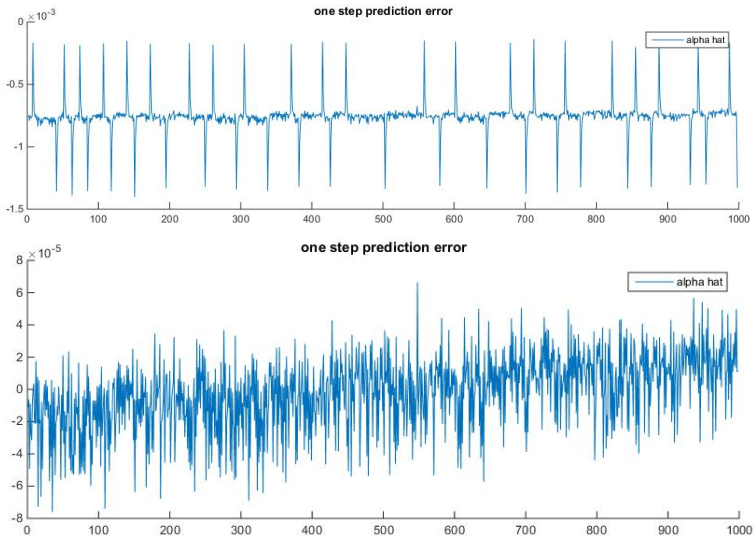



Figure 3: One-Step Prediction with mean (top) and mean removed (bottom)

Finally, using the matlab function d2c() we obtain the final continuous transfer function:

$$\boxed{G_{servo}(s) = \frac{20.04}{s + 20.05}}$$

# 2 Part 2

## 2.1 Background and Model Derivation

The full system consists of the servo motor and resultant dynamics of the beam, which is given as double-integration of the transfer function found above. To move this model into the discrete domain, we perform the ZOH discretization:

$$\frac{\theta(q)}{u(q)} = G(q) = \left(1 - z^{-1}\right) \mathcal{Z}\left\{\frac{1}{s}\frac{c}{s^2}G_s(s)\right\}$$

=

$$\frac{\theta(q)}{u(q)} = G(q) = \left(1 - z^{-1}\right) \mathcal{Z}\left\{\frac{c}{s^3}\frac{b}{s+a}\right\}$$

where b and a were estimated in part 1. To simplify the following derivation, a and b will be left symbolic until the end.

The total continuous transfer function, including the ZOH 1/s, is:

$$G_{total} = \frac{bc}{(s^3)(s+a)}$$

To perform the z-transformation, a partial fraction decomposition was performed on this expression (on scratch paper) and found to be:

$$G_{total}\{s\} = \left(\frac{bc}{a^3}\right)\left(\frac{1}{s}\right) - \left(\frac{bc}{a^2}\right)\left(\frac{1}{s^2}\right) + \left(\frac{bc}{a}\right)\left(\frac{1}{s^3}\right) - \left(\frac{bc}{a^3}\right)\left(\frac{1}{s+a}\right)$$

Applying the forward z transform by consulting a table, and applying the ZOH by multiplying by $\frac{z-1}{z}$:

$$G_{ZOH}\{z\} = \frac{z-1}{z}\left[\left(\frac{bc}{a^3}\right)\left(\frac{z}{z-1}\right) - \left(\frac{bc}{a^2}\right)\left(\frac{T_s z}{(z-1)^2}\right) + \left(\frac{bc}{2a}\right)\left(\frac{T_s^2(z)(z+1)}{(z-1)^3}\right) - \left(\frac{bc}{a^3}\right)\left(\frac{z}{z-e^{-aT_s}}\right)\right]$$

Simplifying, a lot of terms cancel out with the ZOH. Additionally, since the inverse laplace and forward z transforms are both linear operations, the bc parameter can simply be brought out front:

$$G_{ZOH}\{z\} = bc\left[\frac{1}{a^3} - \left(\frac{1}{a^2}\right)\left(\frac{T_s}{z-1}\right) + \left(\frac{1}{2a}\right)\left(\frac{(T_s^2)(z+1)}{z-1}\right) - \left(\frac{1}{a^3}\right)\left(\frac{z-1}{z-e^{-aT_s}}\right)\right]$$

To obtain the combined transfer function $G_{ZOH}$ equation, each of these portions of the discrete model were created in MATLAB as separate transfer functions, and then added together, taking advantage of its ability to add transfer functions in the z-domain. b and a were substituted for the values obtained above. I'd like to think of this as ingenuity rather than algebraic laziness. The resultant function $G\{z\}$, is:

$$G\{z\} = \frac{y\{z\}}{u\{z\}} = bc\frac{(1.651e-05)z^{-1} + (3.59e-05)z^{-2} - (4.239e-05)z^{-3} - (1.002e-05)z^{-4}}{1 - 3.367z^{-1} + 4.101z^{-2} - 2.101z^{-3} + 0.367z^{-4}}$$

where y is the position of the beam.

Multiplying through and noting that $z^{-1}$ is the delay operator, we can finally arrive at the difference equation:

$$y_k - 3.367y_{k-1} + 4.101y_{k-2} - 2.101y_{k-3} + 0.367y_{k-4} =$$
$$[(1.651e{-}05)u_{k-1} + (3.59e{-}05)uk - 2 - (4.239e{-}05)uk - 3 - (1.002e{-}05)u_{k-4}]bc$$

One can already see that this can be formulated as a least squares problem! Since we will estimate the combined parameter bc, we can then calculate c using the estimated value of b.

## 2.2 Application to System

With the model derived above, we can perform the same style of least-squares estimation used in part 1 to estimate the final parameter c. Since the other parameters are already determined, and have been 'absorbed' into the difference equation, the resultant S matrix is a single column that we can use to estimate c. Since this model is now fourth order, we must start at the fifth data point. The formulation, then, for N data points, is as follows:

$$\mathbf{Y} = \begin{bmatrix} y_5 - 3.367y_4 + 4.101y_3 - 2.101y_2 + 0.367y_1 \\ ... \\ y_N - 3.367y_{N-1} + 4.101y_{N-2} - 2.101y_{N-3} + 0.367y_{N-4} \end{bmatrix}$$

$$\mathbf{S} = \begin{bmatrix} (1.651e-05)u_4 + (3.59e-05)u3 - (4.239e-05)u2 - (1.002e-05)u1 \\ ... \\ (1.651e-05)u_{N-1} + (3.59e-05)uN-2 - (4.239e-05)uN-3 - (1.002e-05)u_{N-4} \end{bmatrix}$$

And therefore the least-squares estimate:

$$bc = S^{\dagger}y$$

Finally, running this math in matlab yields the value **bc = 32.463**, therefore **c = 1.6199** and thus the final discrete transfer function:

$$\boxed{G\{z\} = \frac{y\{z\}}{u\{z\}} = \frac{0.0005358z^{-1} + 0.001165z^{-2} - 0.001376z^{-3} - 0.0003253z^{-4}}{1 - 3.367z^{-1} + 4.101z^{-2} - 2.101z^{-3} + 0.367z^{-4}}}$$

The final results of parts 1 and 2 are summarized in the table below:

| **param** | **val** |
|---|---|
| a | 20.05 |
| b | 20.04 |
| c | 1.6199 |

With these results, we can form the final continuous transfer function if we want:

$$G_{total} = \frac{32.463}{(s^2)(s + 20.05)}$$

The discrete model boxed above is the result of a single data run, which is highly limited due to the unstable nature of the system. The bode plot indeed looks a bit odd (though not odd enough to make me mad):

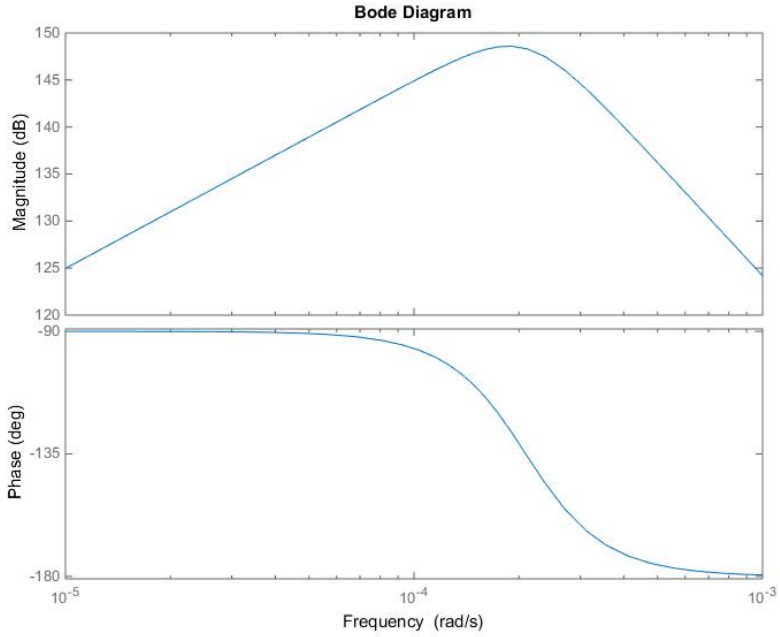



Figure 4: Full system frequency response

However, looking at the single-step prediction error tells a different story:

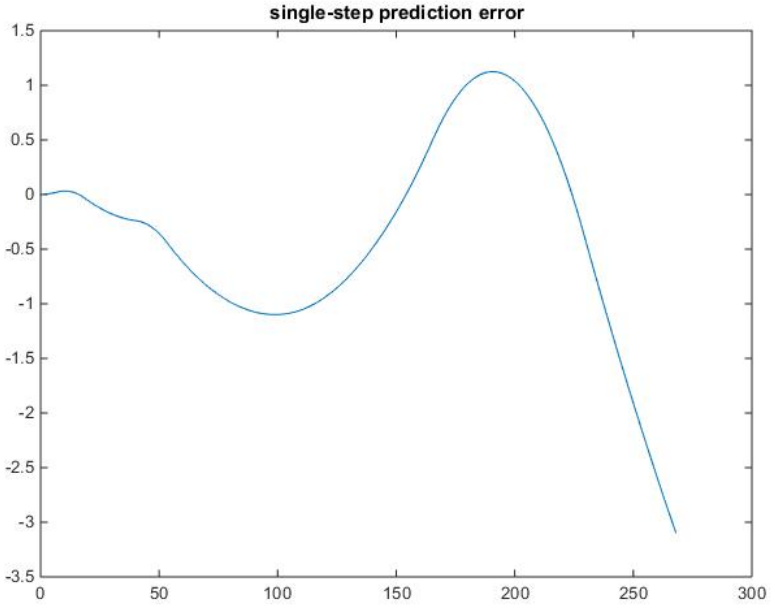



Figure 5: single-step prediction error of full system

We can see that the model has sadly failed severely to reduce prediction error down to a noise sequence. This makes sense given that the data for this section was difficult to take for more than 300 samples, as the system quickly began wrapping around as the propeller went unstable.

# 3 Part 3

The recursive least-squares algorithm is a method for applying least-squares to a sequence of data which comes in real time. The algorithm recursively considers each new data point. The derivation is beyond the scope of this paper, but the following lines (taken from lecture notes) show how the algorithm works:

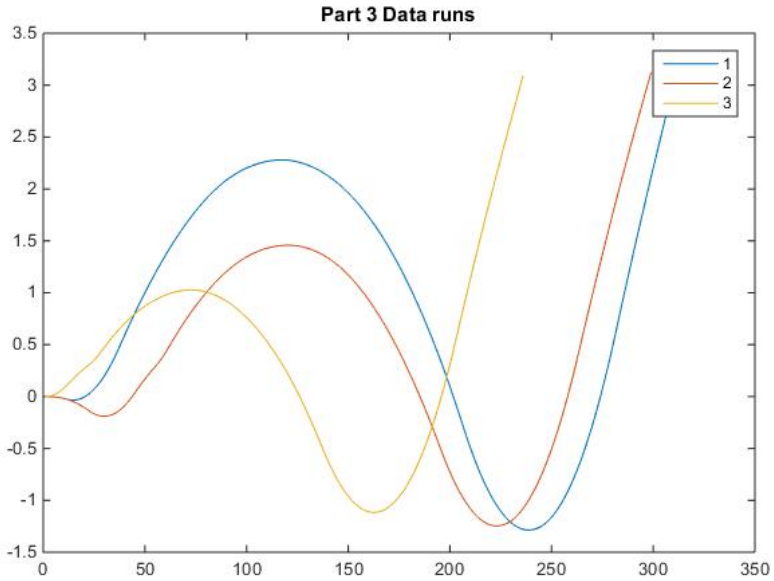
$$\hat{\theta}_t = \hat{\theta}_{t-1} + \overline{R}_{(t)}\varphi^{-1}(t)\varphi(t)\left(y(t) - \varphi^T(t)\hat{\theta}_{t-1}\right)$$
$$\overline{R}(t) = \lambda(t)\overline{R}(t-1) + \varphi(t)\varphi^T(t)$$

With each new data point (arriving at discrete time t) the estimate is updated with a scaled version of the error with regards to last timestep, which is then run through that time-step's sensitivity vector. This vector can be seen as a single row taken from the sensitivity matrix S described earlier. The scaling is done with a recursive scaling factor R which is calculated with both the previous

factors (modified by lambda) combined with the sum of the sensitivty elements squared.

To get a better recursive estimate of the c parameter, we will take the phi columns from several data runs and run them through the algorithm.

3 data runs were taken in addition to the inital run for part2. Each run was cut off as soon as the data wrapped around in the graph. The raw data is as follows:




The RLS Equations given above were implemented in matlab. A value of 1 was used for lambda, which as I recall was discussed in lecture as a way to give equal weighting to each incoming data point.

The end result, however, was utter failure to produce results; I don't think the calculations were able to get much 'grip' since each phi value (a scalar in this case) was very small (on the order of $10^3$) which, when squared, didn't contribute to the recursion enough to see changes in the iterations.

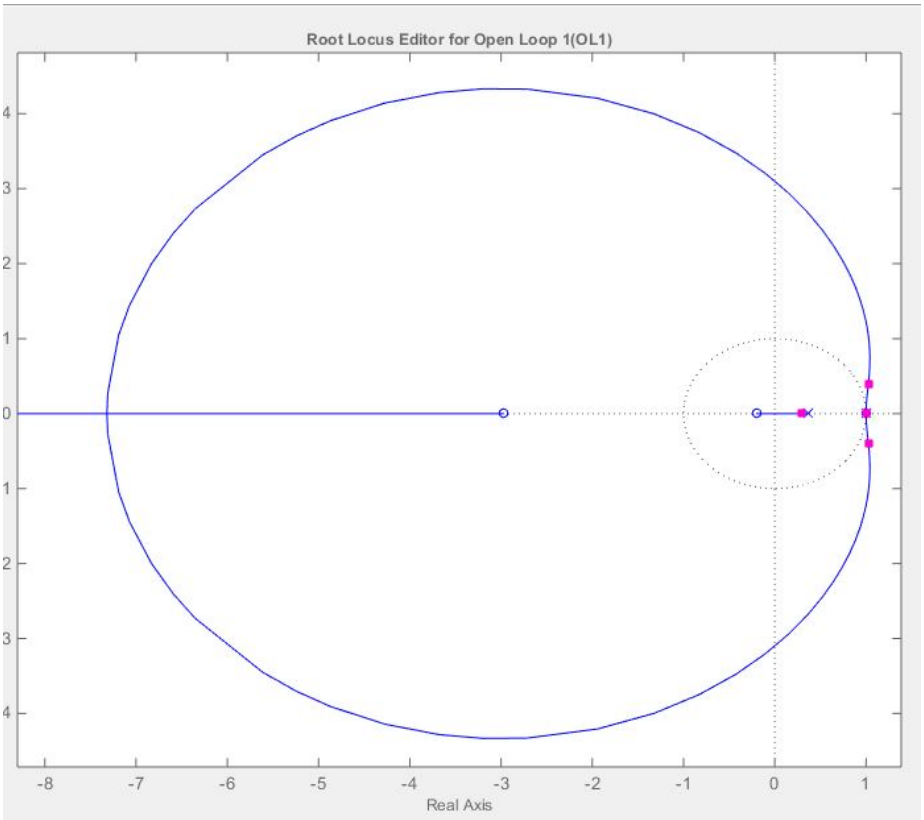
In the interest of producing something for part 4, I am leaving this to be completed if there's time at the end, but happy with my attempts and the research/understanding that went into them.

# 4 Part 4

In order to test the model that we have reached in the above sections, a controller will be implemented. The best way to verify the dynamics we've estimated is to see if the system behaves correctly when the controller is designed for those

dynamics.

The root locus diagram for the identified system is:




we can see that the system is unstable, which we expected. The purpose of the controller is to move these poles around in order to keep them within the unit circle. The PID loop is a ubiquitous method for altering the pole locations. However, following discussions in lecture, I have chosen to test my model by creating a Full-State-Feedback controller with LQR. While it would be very interesting to follow the provided hints for the PID development, I am interested in the practice for generating this type of controller, and if I understood correctly, this method is fine for testing the model. Also, I really really like LQR.

Following the provided hints, the state-space matrices were constructed:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & bc \\ 0 & 0 & -a \end{bmatrix}$$

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

Looking at the math, it appears this is corresponding to the state vector:

$$x = \begin{bmatrix} \theta \\ \omega \\ \alpha \end{bmatrix}$$

and the C matrix:

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

since we only want to know if we can control theta.

Control theory tells us that if the rank of the controllability matrix is full, then we can get the system where we want it. **Using ctrb() in matlab with the above matrices verifies that the identified system is controllable**. that the system should be controllable! So our parameters can't be THAT bad. Still, I am unsure about the final result for c.

LQR methodology, which is beyond the scope of this paper, was used to generate 3 feedback gains. This is for the full-state feedback equation $u = -k(x - x_d es)$. MATLAB's LQR function was used, with Q being the 3x3 identity matrix, and R being the scalar 1. This is to give even weighting across the states; future work might involve tweaking these matrices.

The resultant gains, which I will label according to the state variable they modify, are as follows:

| **param** | **val** |
|---|---|
| $k_{theta}$ | 1 |
| $k_{omega}$ | 1.04 |
| $k_{alpha}$ | 37.5338 |

These gains are finally applied using the full-state feedback equation to generate the control signal u:

$$u = -kx = -k_{theta} * \theta - k_{omega} * \omega - k_{alpha} * \alpha$$

The results can be found in the provided v-rep scene. IT doesn't work very well.