SF3 Interim Report

Chengyuan Li
cl854
Queens' College

May 2023

Introduction

In this report, we will be presenting the initial results obtained from simulating a system consisting of a pendulum attached to a cart. We will also investigate the usefulness of a linear model in predicting the behaviour of the system. The report will be divided into sections as per the tasks in the project handout.

Task 1.1

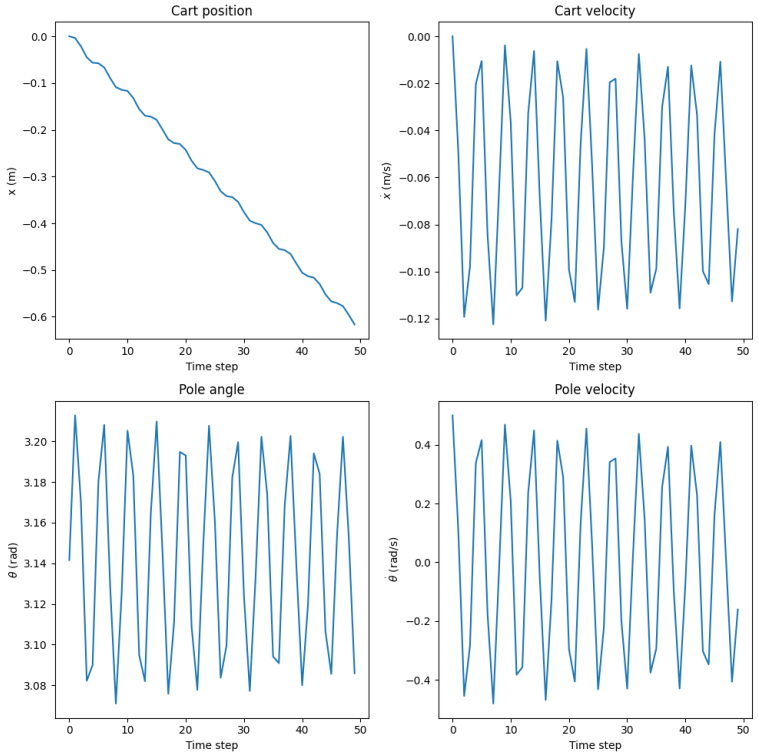

Figure 1: Time evolution of the system when it is set to oscillate around the stable equilibrium

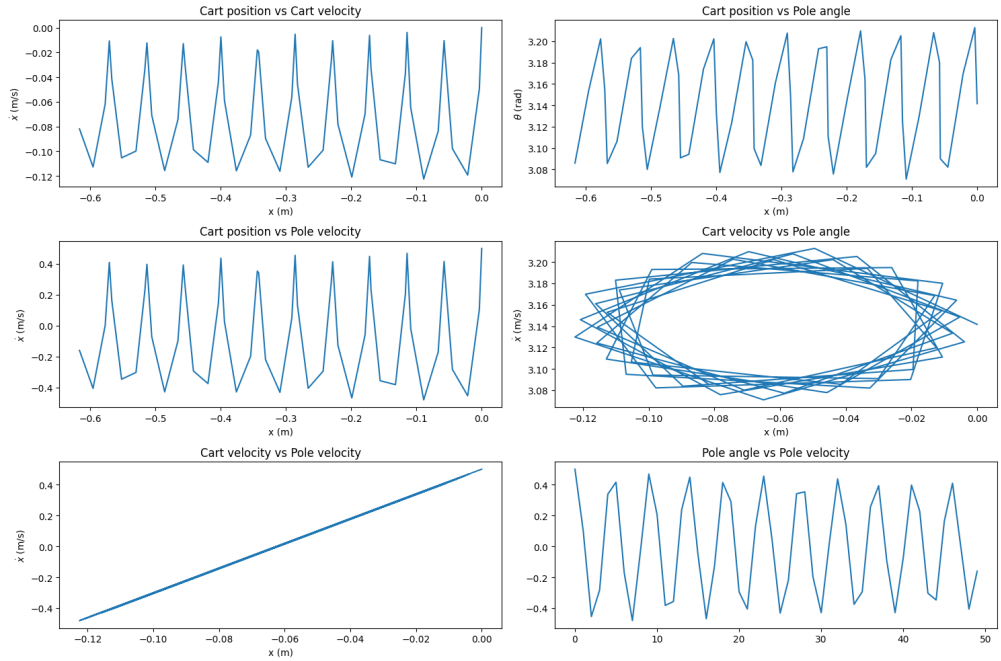

Figure 2: Different state variables of the system plotted against each other, when it is set to oscillate around the stable equilibrium

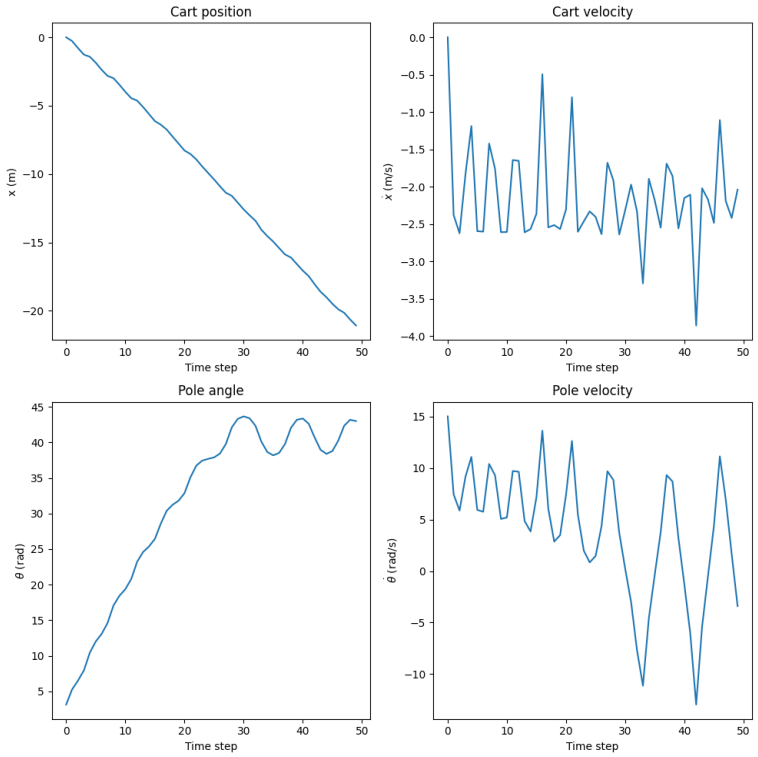

Figure 3: Time evolution of the system when the pendulum rotates completely

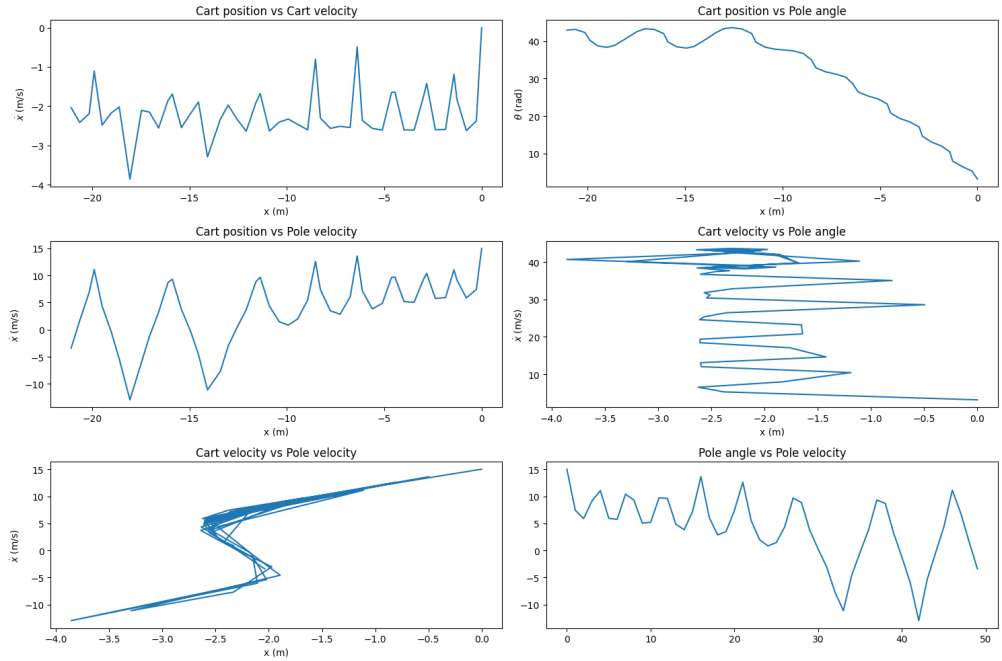

Figure 4: Different state variables of the system plotted against each other, when the pendulum rotates completely

The state of the system can be characterised by the vector $[x, \dot{x}, \theta, \dot{\theta}]$, where x is the position of the cart and θ is the angle that the pendulum makes with respect to the upright position. By specifying the initial conditions of the system, we can make the pendulum undergo different types of behaviours. The two behaviours we are interested in here are oscillation around the stable equilibrium (initial state = [0,0,π,0.5]), and complete rotation of the pendulum (initial state = [0,0,π,15]).

The top left graph on figures 1 and 3 shows that even though there is no initial cart velocity, the movement of the pendulum causes a net force on the system, causing the cart to move with net momentum one direction as well (average velocity equal to the offset of the cart velocity plot). In the absence of external forces, all three of the other state variables oscillates about some equilibrium.

Task 1.2

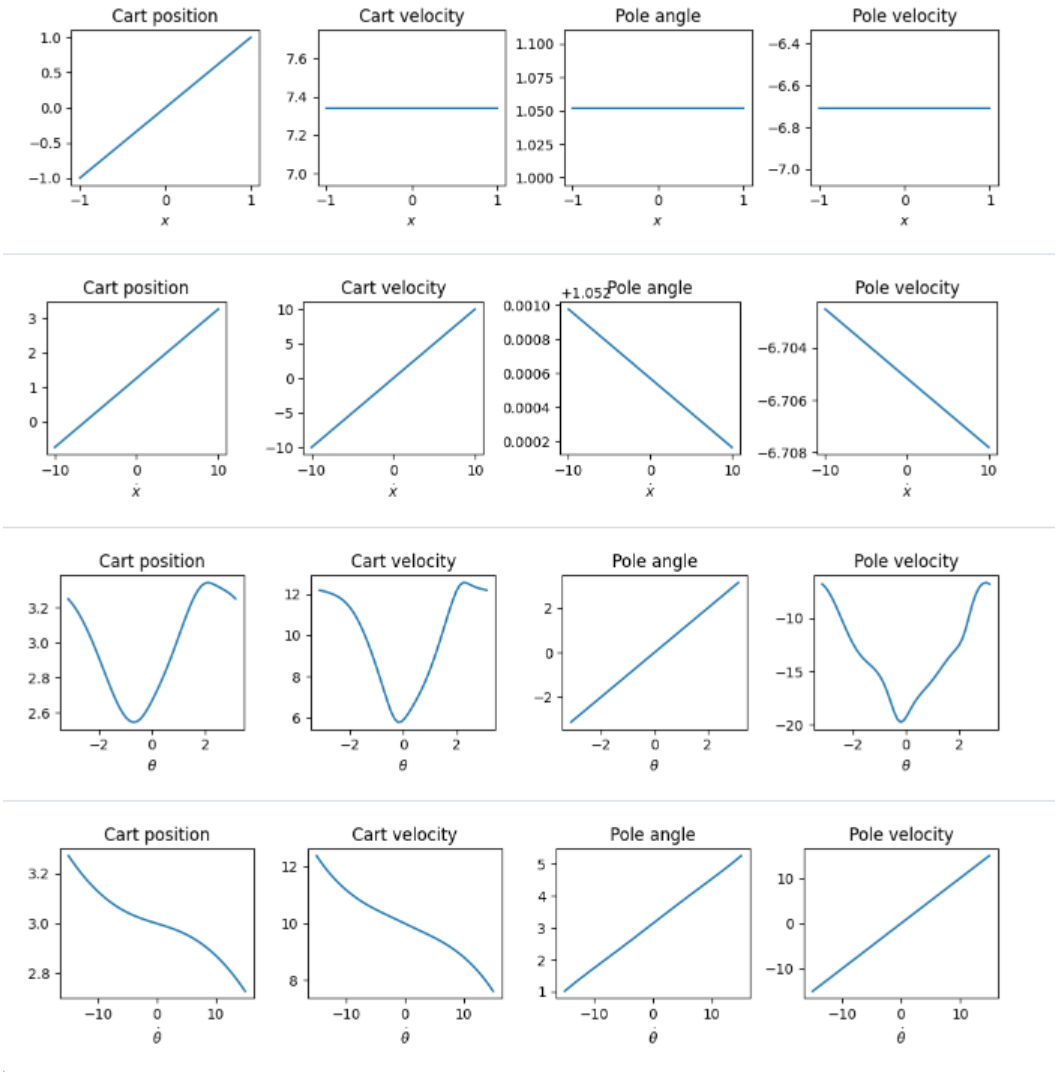

Figure 5: Sweep across a suitable range for each of the 4 state variables, plotting the value in one variable after a single call to performAction(), keeping the other variables constant

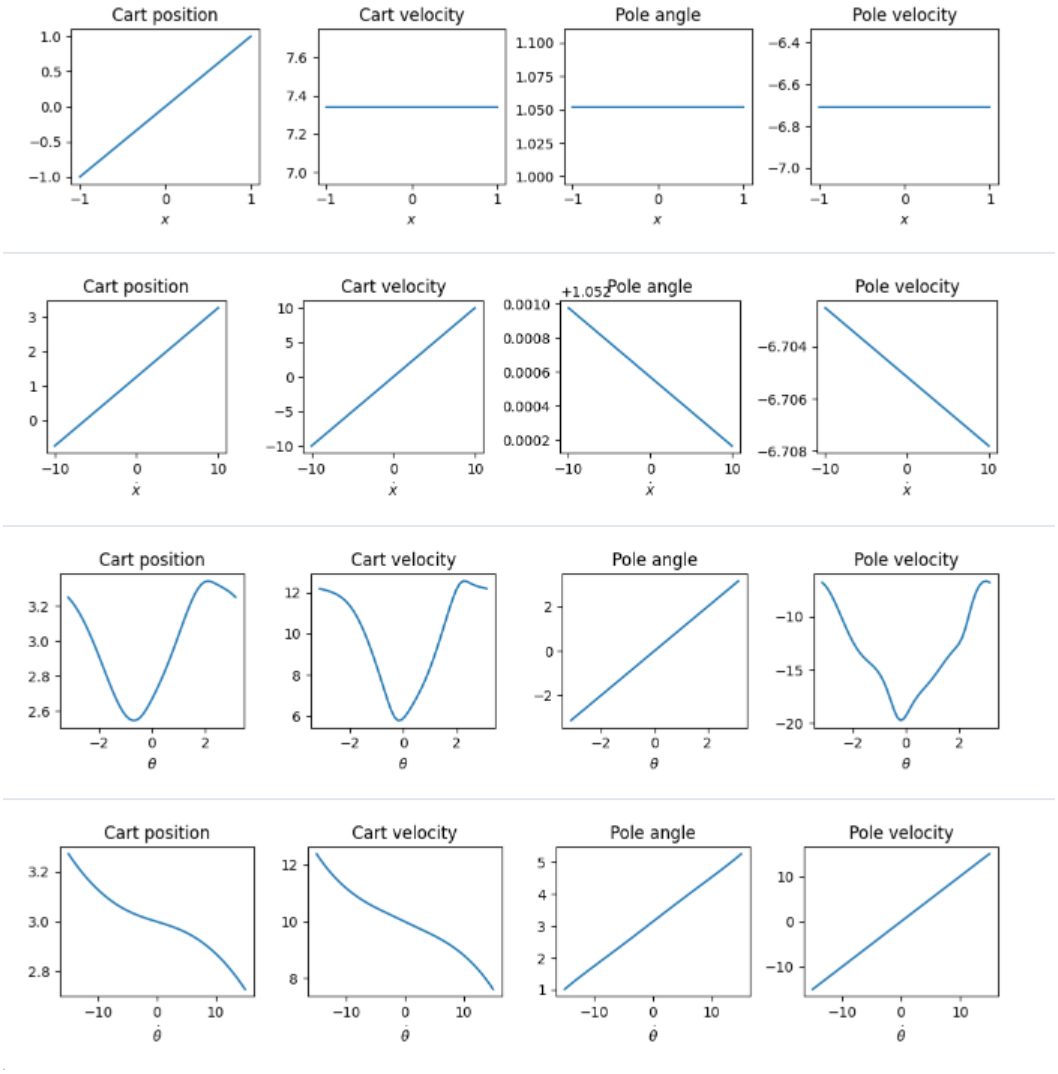

Figure 6: Sweep across a suitable range for each of the 4 state variables, plotting the change in one state variable after a single call to performAction(), while keeping the other variables constant

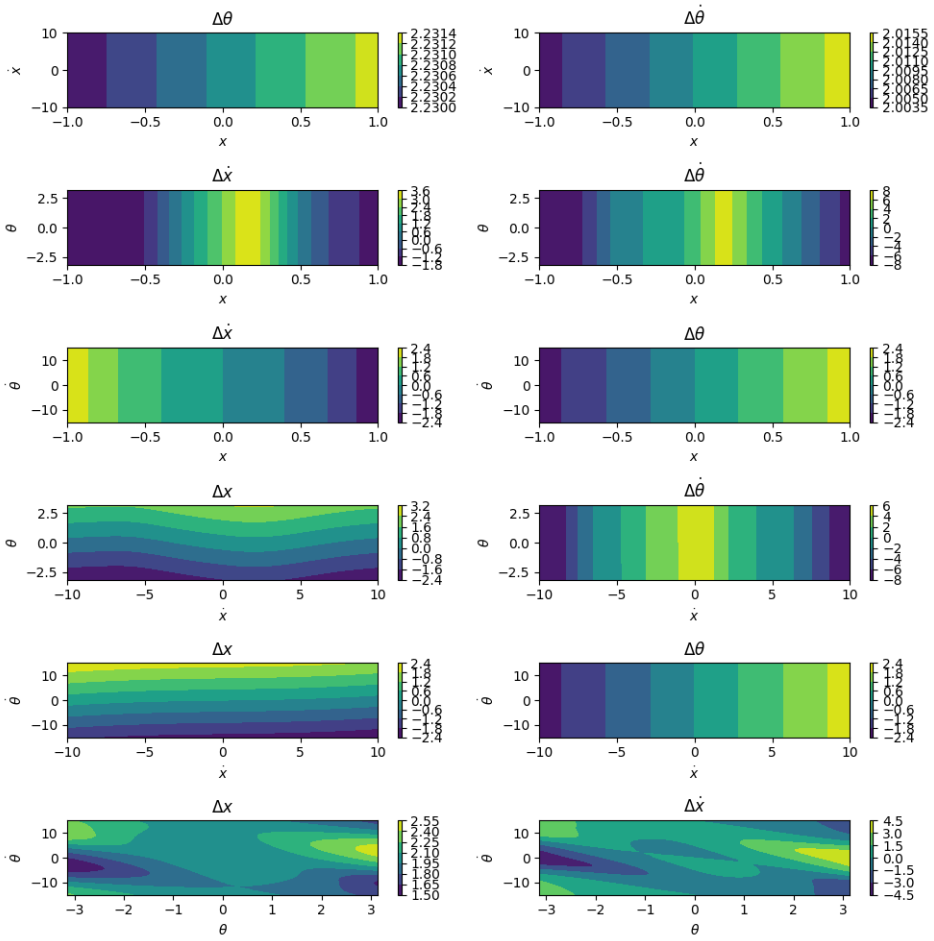

Figure 7: A series of contour plots, generated by keeping two of the state variables constant, while measuring the change in another.

The series of plots in figure 5 give the value of each of the state variables after sweeping a single call to the update function `performAction()` - the function which models the evolution of the system in time. The data is collected by sweeping each state variable across a corresponding suitable range, while keeping all other variables constant. As expected, the relationship of each variable with itself after evolving in time is linear. Looking at the different rows, we can clearly see, that the relationship between the cart location have no effect on the other state variables, and all 4 variables share a linear relationship with cart velocity. When the pole angle and the pole angular velocity are varied, however, the plots begin to exhibit varying degrees of non-linearity.

Figure 6 mirrored the approach of the first, but with a critical difference in the data being collected. Instead of focusing on the final state of each variable, we shifted our attention to the changes each state variable underwent after the invocation of the `performAction()` function. This resulted in another 16 plots illustrating the dynamics of the system. We can see that the shape of the plots are the same as those of figure 6, but the scale is reduced, as can be seen by the values on the y-axis.

Figure 7 shows a series of contour plots, the results of which further reinforces the linearity of the relationships between the variables shown previously. In each plot, two of the variables are being varied, while third variable gets evaluated while the final variable remains constant. The contour formed by the pole angle and angular velocity shows the highest degree of irregularity - showing multiple local maximums and minimums in the given range, while all other plots show a consistent

form, in the sense that there are easily identifiable trends in the values of the measured variable on the contour.

Task 1.3

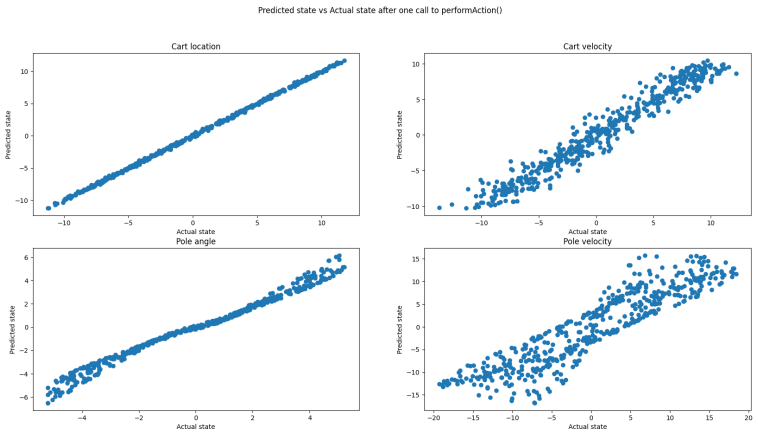

Figure 8: The actual values plotted against the predicted values using the linear model

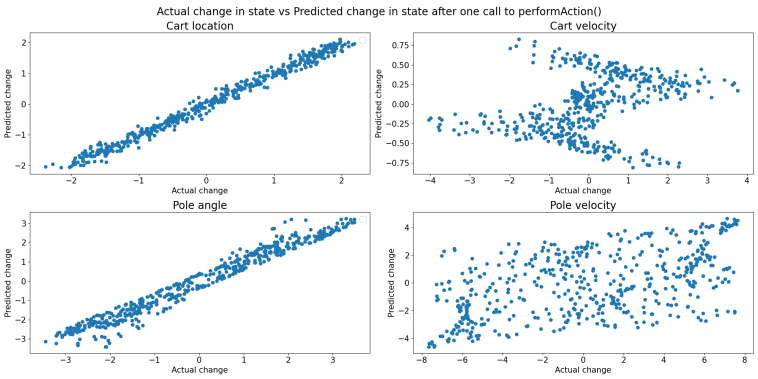

Figure 9: The actual changes in values plotted against the predicted values using the linear model

In this section, we investigated the performance of the linear model for modelling the state of the system over time. The target Y is assumed to be a linear function of the current state X: $f(X) = CX$, where C is a 4×4 matrix of coefficients.

First we generated an array of 500 random states, denoted with the vector X, and computed the target vector Y, which is equal to the input after running `performAction()` once. Next, using the `numpy.linalg.lstsq` function, we were able to calculate the optimal coefficient matrix, which we then used to compute predictions for a large number of different X values within the suitable range. The actual Y for these X variables are also computed and stored in an array. The actual Y of the four state variables are then plotted against their corresponding predictions using the linear model, and shown in figure 8. The better the prediction, the more straight the line. We can see that the linear model best predicts cart location, and do a decent job for the pole angle. But the plots for the cart velocity and and pole velocities are a lot noisier.

Next, we change the target variable Y to equal the change in the state vector after calling `performAction()`. The resulting plot is given in figure 9. We see that as before, the model predicts positions much better than velocities, but it is noisier for all four state variables, especially the velocities, which has hardly any semblance of a linear relationship.

We draw the conclusion that the linear model is better suited for linear models, which is a zeroth order derivative of time. The higher the order of the derivative, the more non-linear the relationship, which makes the linear model a poor fit. This is also the reason why the model performed worse across the board for the difference in state variables (figure 9), as taking the difference between time steps has a similar effect to taking the derivative. The model is also less suitable for the state of the pole, as there are more factors which govern its motion such as gravity, complicating the relationship.

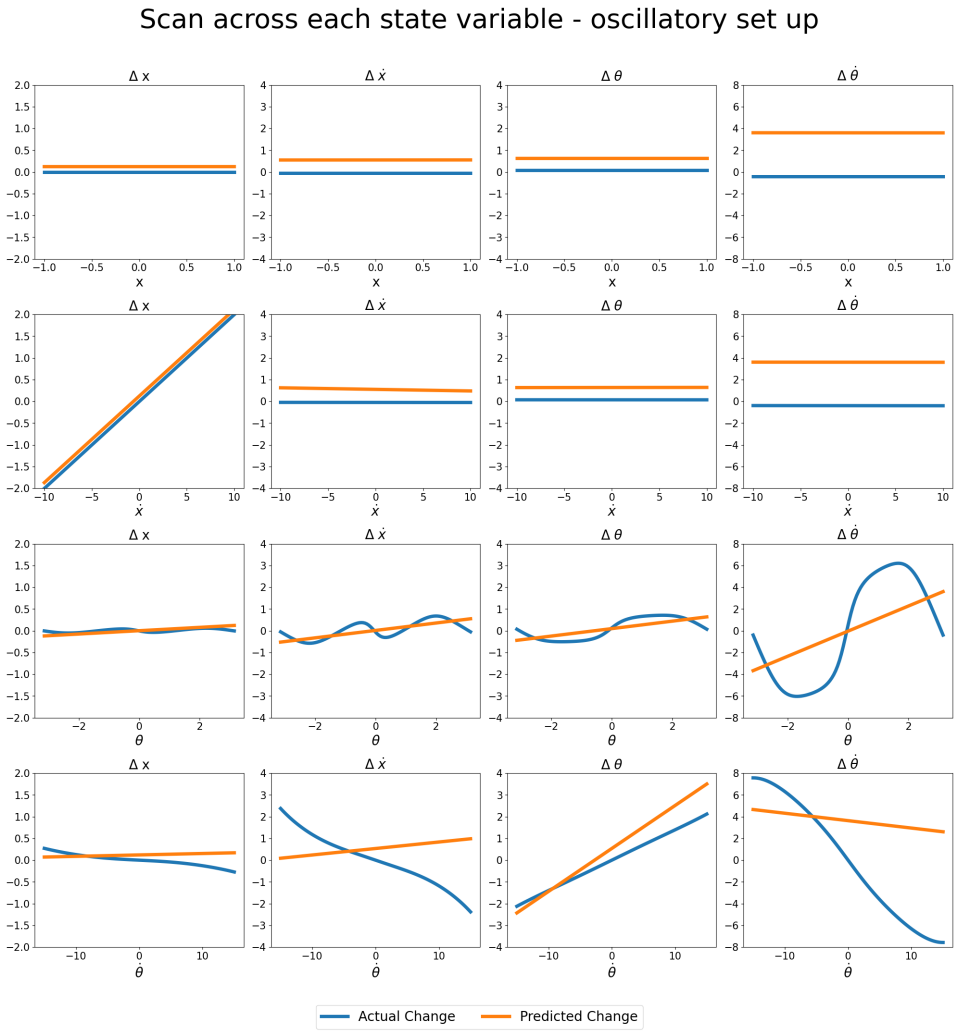

Figure 10: Repeat of the one variable sweep in section 1.2, with the system oscillating about its stable equilibrium

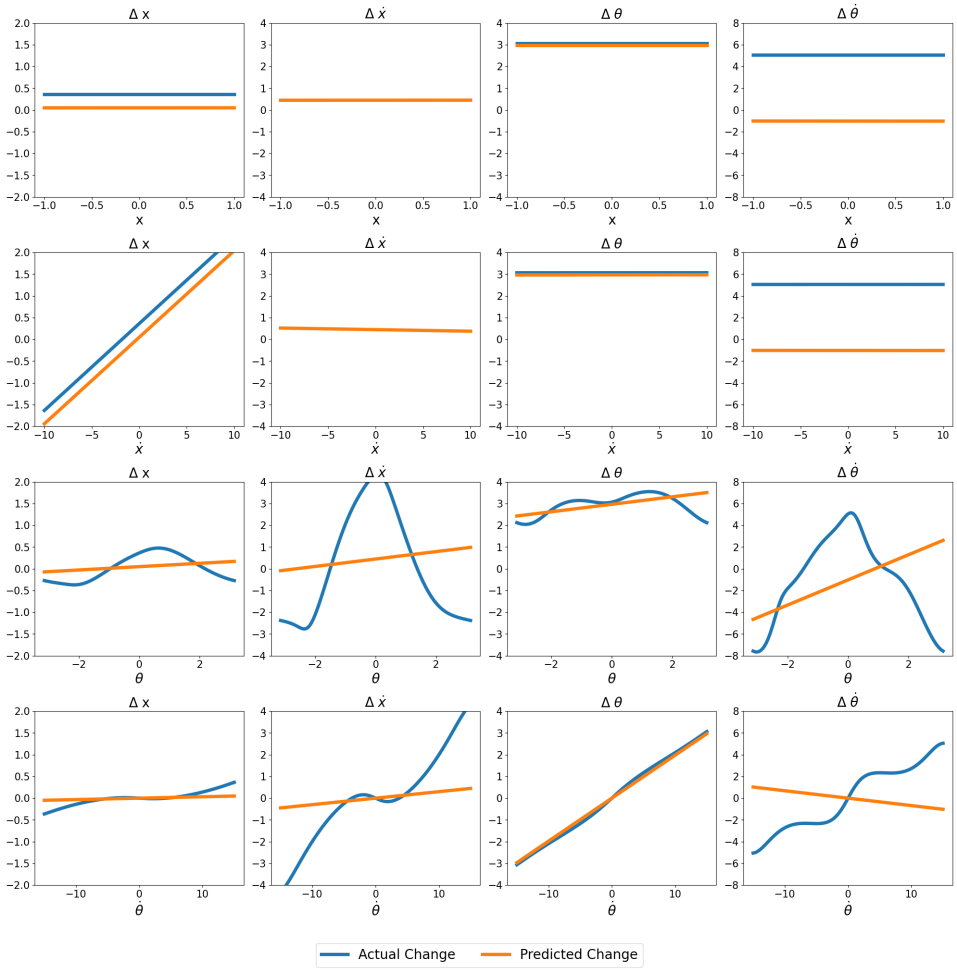

Figure 11: Repeat of the variable sweep in section 1.2, with the pendulum undergoing complete rotations

In figure 10 and 11, we repeated the single variable scan which we conducted in Task 1.2, producing two more grids of 16 plots. Figure 10 uses initial conditions that lead to the system oscillating, whereas figure 11 uses initial conditions that lead to the pendulum undergoing complete rotations. All plots have the prediction of the linear model overlayed on top of it. We can see that for scans of cart position and velocity, which takes a linear form, the model is generally a good fit, though still prone to error as is the case when $\dot{\theta}$ or $\Delta\dot{\theta}$ is being measured. For the plots where θ or $\dot{\theta}$ is across the horizontal axis, the plots are generally curved, and the linear model proves inadequate.

Task 1.4

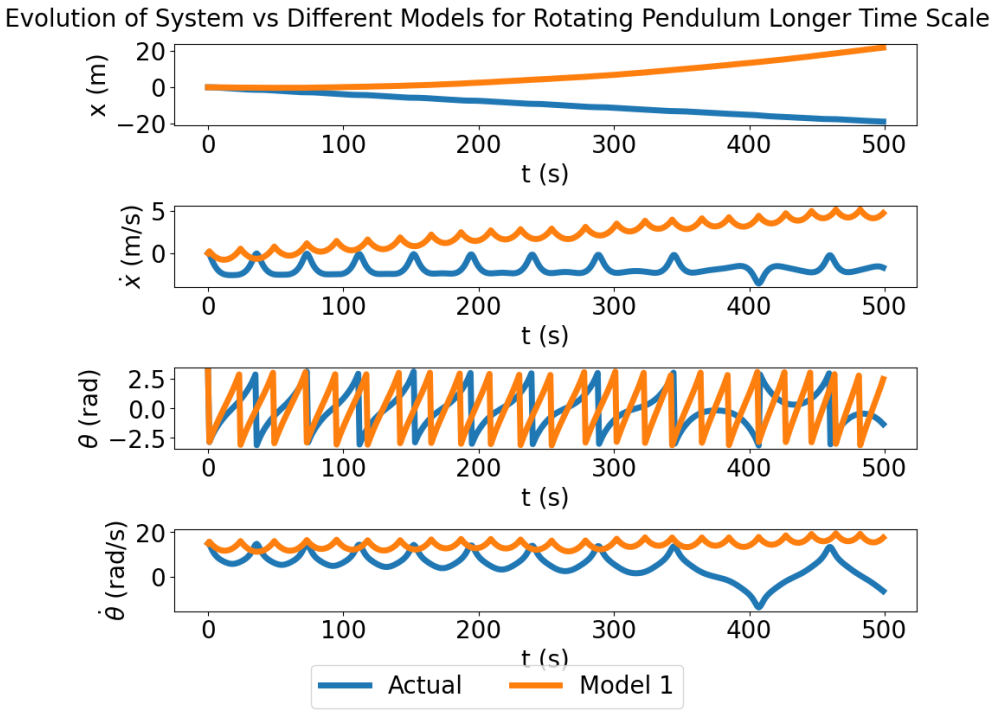

Figure 12: Prediction of time evolution of the system state vs prediction of time evolution using the linear model

Here, we test the model's ability at performing its true task - how it performs in predicting the time evolution of the system, rather than how well it matches with data already collected. The time series is mapped out over 2500 different data-points, and is shown in figure 12 for each state variable. Note that the linear model is applied at each time step, hence the overall evolution of the system is not constrained to a mere straight line.

We note that as time goes on, the time prediction for cart position and velocity diverges from the data collected from the actual simulation. For pole angle, and model is able to track the the roughly periodic trajectory of the collected data, though being noticeably out of sync. The performance for predicting pole velocity was poor, with the period of the model being different to that of the actual simulation.

Task 2.1

As observed from the results in task 1, we can see that the linear model leaves more to be desired in terms of prediction accuracy - even the optimal solution for the coefficient matrix C performs poorly when used for prediction in the time evolution. To obtain better results, we look to a kernel based nonlinear model instead.

For N pairs of X and Y vectors as the data set, if we choose M of the X vectors as centres for our kernel function, then the nonlinear model can be encapsulated in the following equation:

$$\alpha_M^{(j)} = \left(K_{MN}K_{NM} + \lambda K_{MM}\right)^{-1} K_{MN}Y_N^{(j)} \tag{1}$$

where $K(X, X') = e^{-\sum_j \frac{\left(X^{(j)} - X'^{(j)}\right)^2}{2\sigma_j^2}}$ is the the Gaussian kernel function.

Firstly, we randomly generate data that is to be used to train the model, comprising of 1000 $X - Y$ vector pairs. We visualise the data set in figure 13 below - showing that they are spread out nicely over the appropriate ranges.

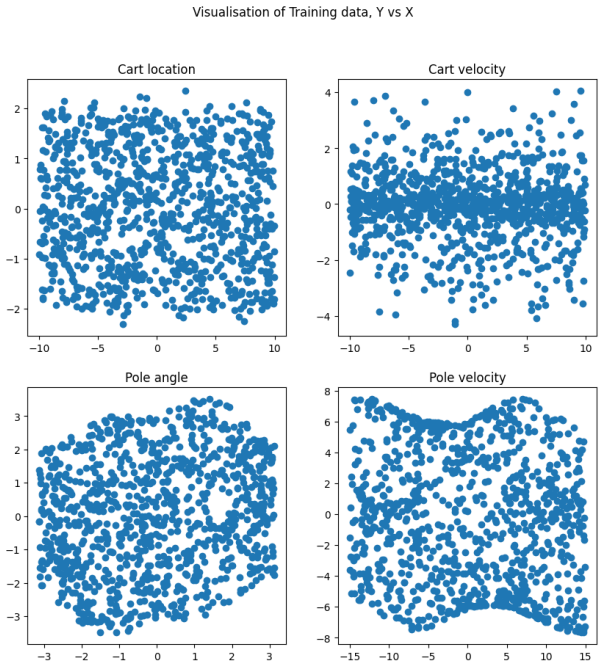

Figure 13: Plots showing the spread of the 1000 data points of the training data set

Next we trained the nonlinear model. The parameters of the kernel function - in particular σ_j - the length scale factors of the kernels, are paramount to the performance of the model. Before we optimise them, we ought to obtain a good fit using an appropriate initial guess, so that we can verify the validity of the model. For that, we used the standard deviations of the state variables in the training set. The scatter plots of the predictions of the model against the actual values of the target function (the change in state after running `performAction()`, $Y_k = f(X_{k+1}) - f(X_k)$) are shown in figure 14:

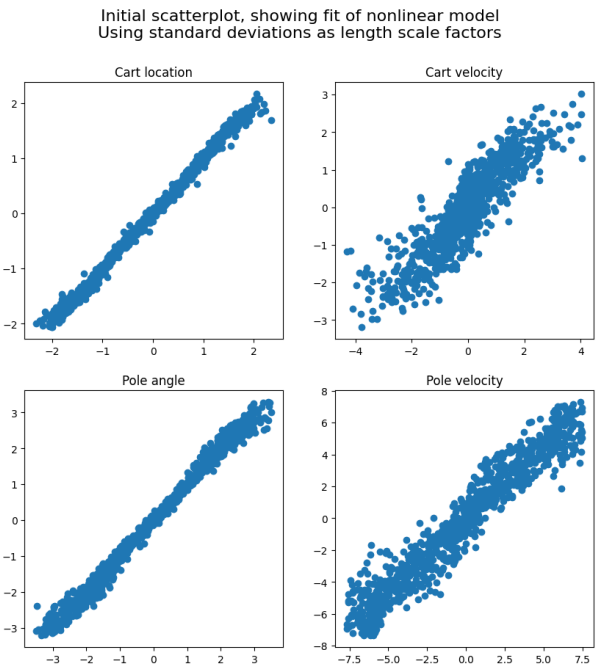

Figure 14: Scatter plots, using standard deviations as σ_j

We proceeded to run a grid search over the 4 σ values. The process involved retraining the model for each combination, and hence would take an inordinate amount of time to run. I used a list of 5 geometrically spaced values for each σ, centred around the standard deviation and spanning a factor of 10 each way, so that it would run in a reasonable amount of time. Despite the rough graduations, the optimal output was still much better than merely the standard deviations - as can be shown by the scatter plots in figure 15. The points are pretty much perfectly aligned in a straight line - indicative of high accuracy:

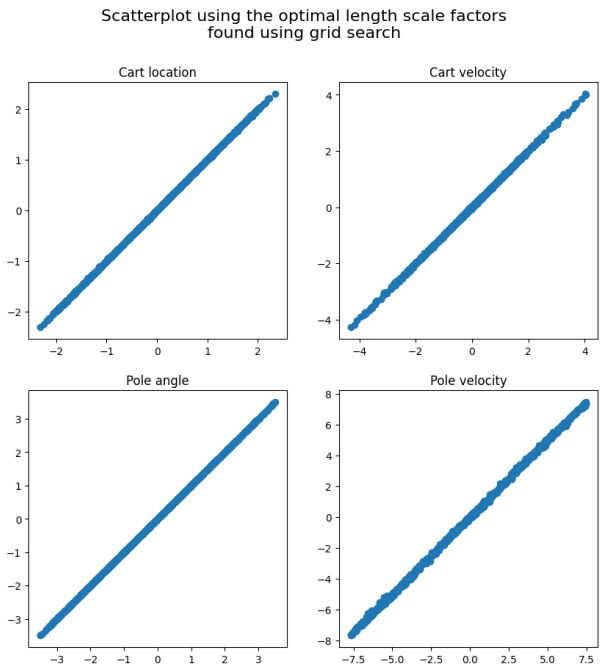

Figure 15: Using the optimal parameters found from grid search

Next we investigate the relationship between the error and the number of basis functions, as well as the number of data points. We plotted the MSE of the model prediction, varying M and N respectively. The results are shown in figures 15:

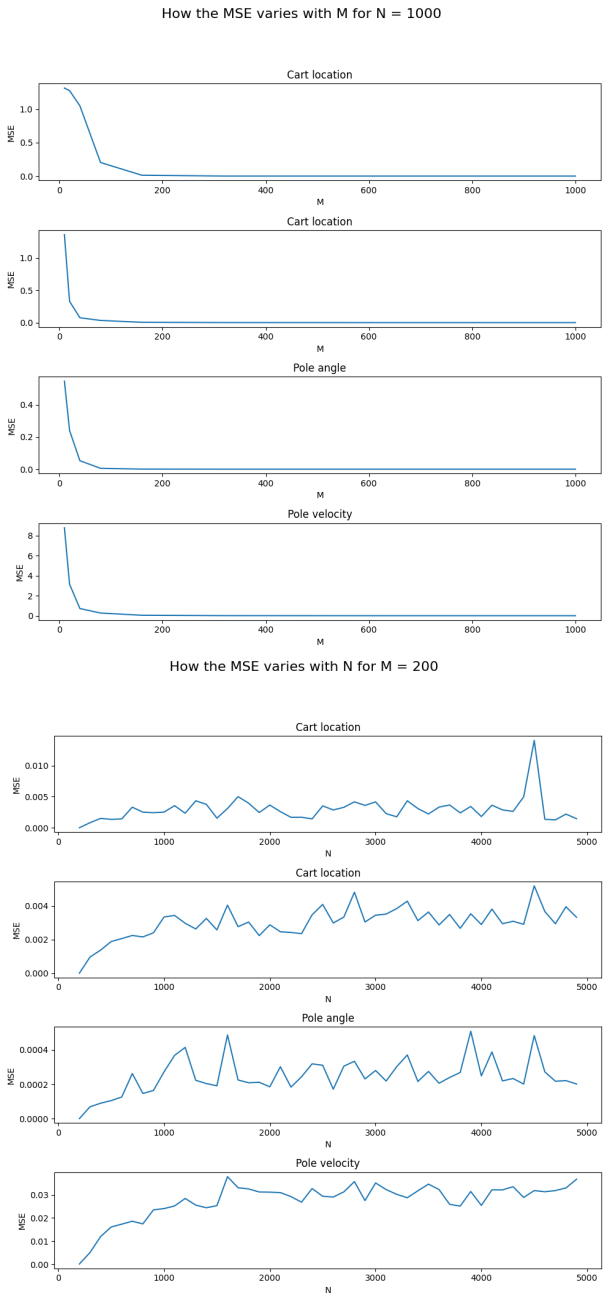

Figure 16: How MSE varies with N and M

We can see that the error sharply decreases as M increases when M is small. However past around $M = 200$, for $N = 1000$, the MSE does not decrease much further for more basis functions. Therefore I fixed M at 200, and then varied N from 200 to 5000. The relationship between the error and N is very unexpected - it is much more irregular, but generally increase rather than decrease as N gets larger. This suggests that it is how close M is to N that determines the accuracy of the model, more so than the size of the data set.

For the remainder of this task, we fix $N = 1000$ and $M = 500$.

To test the accuracy of the non-linear model, we take 2D slices across the suitable ranges of each pair of state variables, and compare how our model's predictions compare to the actual cart pole simulator. We can clearly see that the two plots are identical as far as the eye can see, illustrating the high accuracy of the nonlinear model.

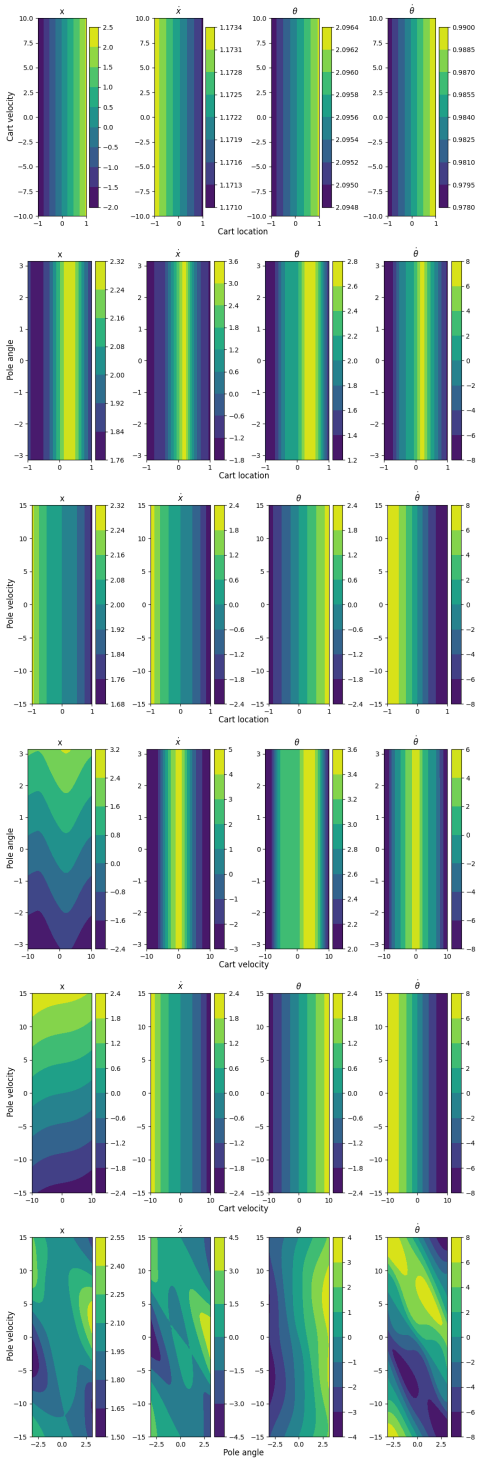

Figure 17: Set of 2D slices using actual training data

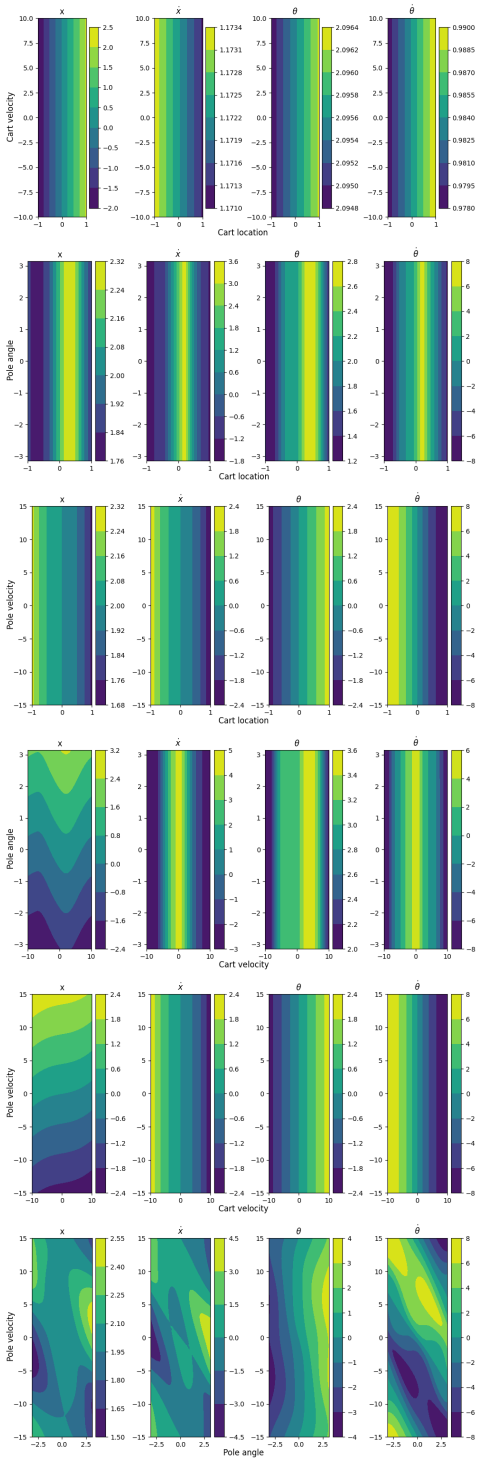

Figure 18: Set of 2D slices using model predictions

We then move onto testing how the model performs when predicting future motion in a time evolution. We initialise the model at two starting conditions - one set up so that the pendulum oscillates around its stable equilibrium, the other leading it onto completing an entire rotation:

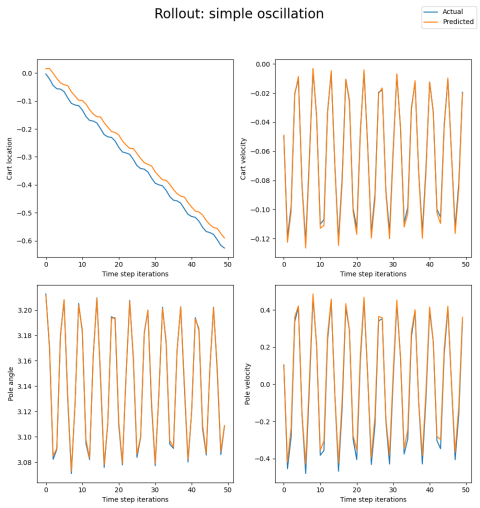

Figure 19: Initial conditions - simple oscillation

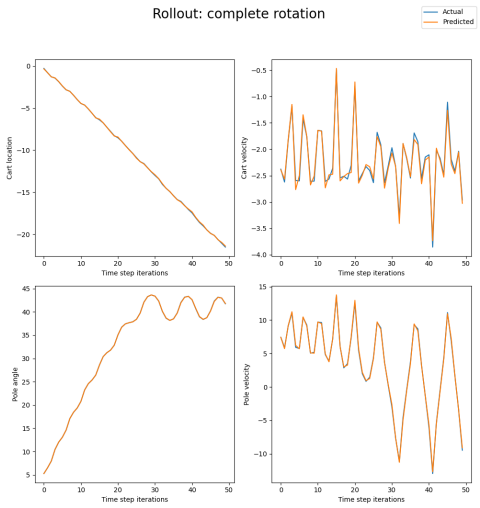

Figure 20: Initial conditions - complete rotation

In both cases, the model performs brilliantly, matching the actual motion almost exactly for both sets of initial conditions. This shows a marked improvement over the time evolution predictions of the linear model, which drifts apart from the actual motion after a few iterations.

Task 2.2

Up until this point, we have assumed no external force acts upon the system. In this task we expanded the input vector to the system, from 4 up to 5 state variables, one of which is the input force ('action'). In the following task we investigated the performance of both the linear model and the nonlinear model with the addition of a new input variable.

To start of with, we generate a new training data set, where X now includes the action, and Y remains the same four element vector as before.

The figure below shows the 1D sweep for the linear model, alongside the rollout time evolution using the same two initial conditions as before - however this time with a constant input force:

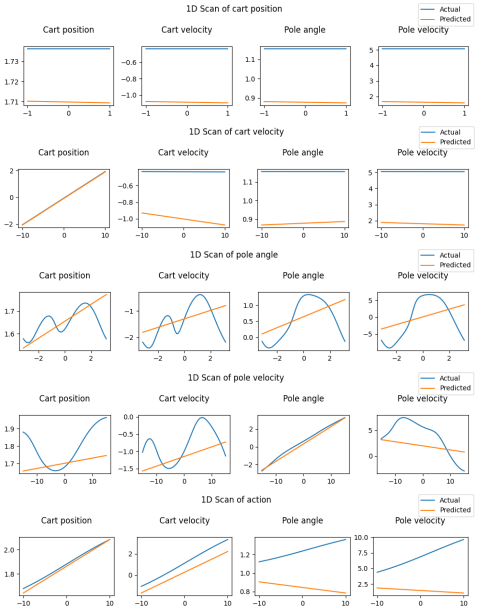

Figure 21:

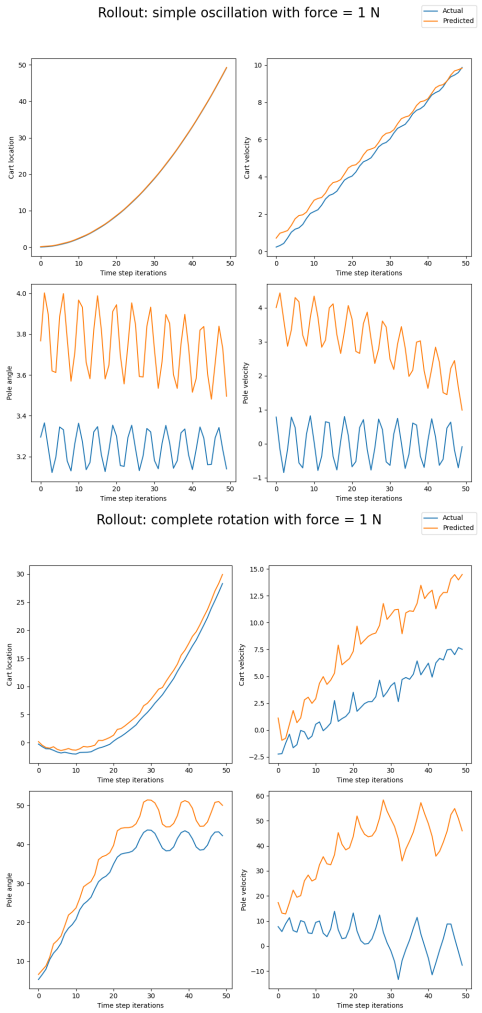

Figure 22: Rollouts for simple oscillation and complete rotation, using the linear model

The linear model works exactly as we would expect in the 1D sweep across the four input variables (note that action is not a dependent variable, hence there are only four plots per row). We see that the addition of action did not hinder its performance, and the resulting plots are the same as what we got in Task 1.

Now repeat the same thing with the nonlinear model - including action:

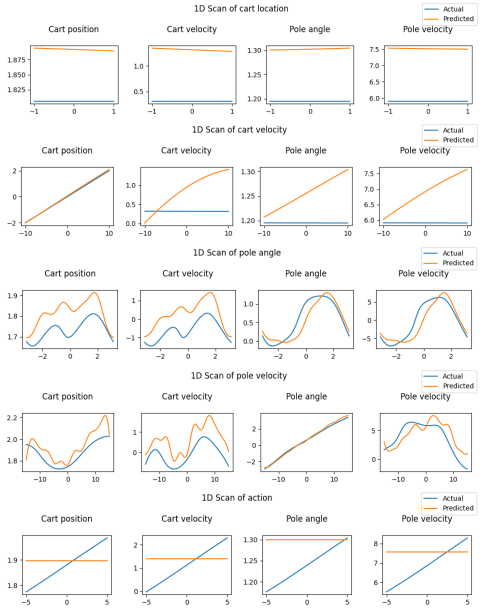

Figure 23: 1D sweep using the nonlinear model

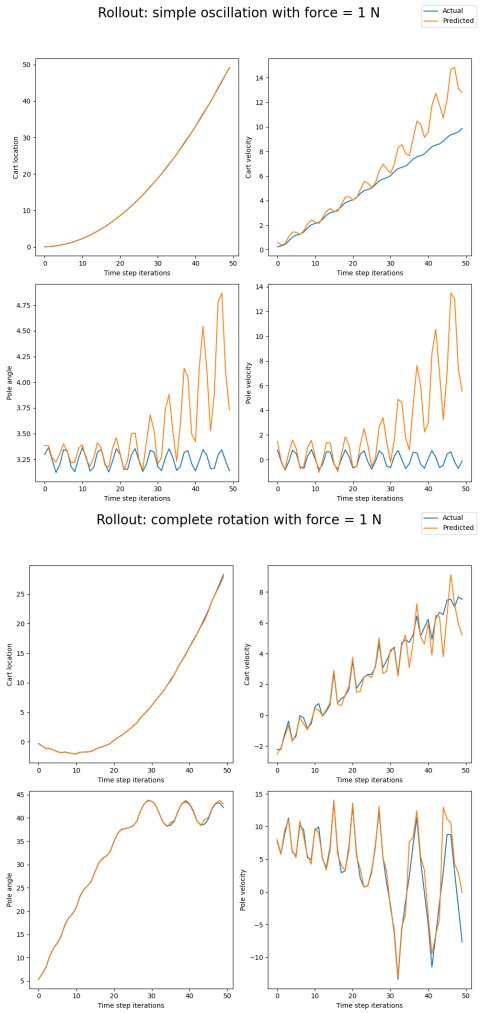

Figure 24: Rollouts for simple oscillation and complete rotation, using the nonlinear model

The scans of the nonlinear model fit the curves of the actual system much better, but there were several cases of it mispredicting linear relationships. It also failed to wrap the curve around when action is being scanned across its suitable range, showing that the model is limited when it comes to making predictions regarding varying action force.

The rollouts did not have the offset that we see in the rollout predictions for the linear model. However it did drift and lose accuracy after circa. 10 iterations - once again showing the introduction of action throws it off its peak performance that we saw when no action was taken into consideration.

Task 2.3

In this section, we attempt to find an optimal linear control policy, in the form $\mathbf{p}\dot{\mathbf{X}}$. To do so, we define a loss function, and then another function which computes the accumulated losses throughout a short time evolution, and then optimise said function using the Nelder-Mead method (available from `cipy.optimize.minimize()`). Up until this point, we have always ran the rollouts for 50 iterative steps. We make it 10 this time, as after 10 steps the nonlinear model, which we will be using in the next task, loses its predictive accuracy. The policy is a function which outputs an action - that has

the aim of stabilising the system in its desired configuration. In our case, that would be when all state variables are zero, and the system is stationary with the pendulum being upright vertically. THis is achieved by minimising the loss function, defined as $L = \sum_{i=1}^{N} l(X_i)$ with $l(X) = 1 - e^{-X^2/2\sigma_l^2}$.

To find the optimal policy, we initialised the vector $\mathbf{p}$ at random different initial points where each element of $\mathbf{p}$ ranges from -10 to 10, and runs the Nelder-Mead algorithm with the objective function being the cumulative loss at every time step. This outputs a local minimum and a cumulative loss. The reason we run it many times it so that we can maximise our chances of the algorithm landing on the global minimum, since the 4D state space has a multitude of different local minima. The figure below shows the cumulative loss plotted against 1D sweeps for each element of $\mathbf{p}$:

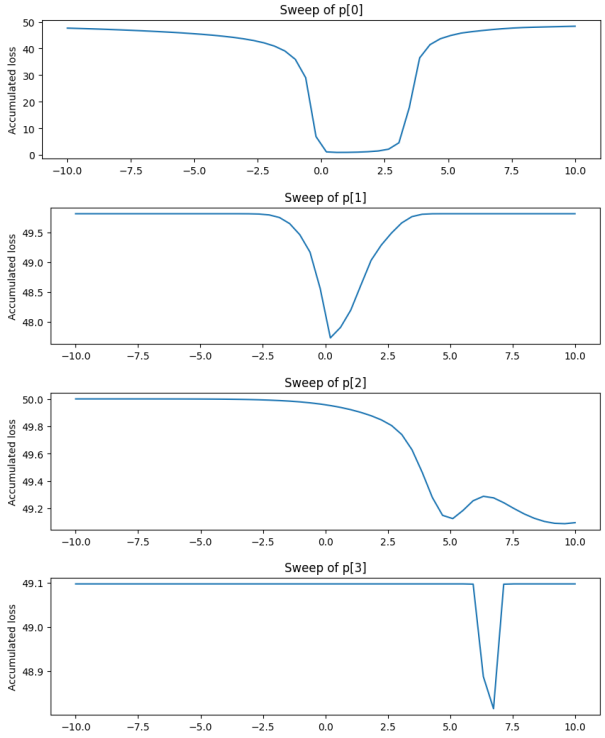

Figure 25: 1D sweep of each element of $\mathbf{p}$

Now we put the optimal policy we found to the test. In the following 4 plots we can see how the loss for each state variable quickly converges to 0 in time, when the policy is implemented. Note that this only works if the initial conditions are already close to the desired state.

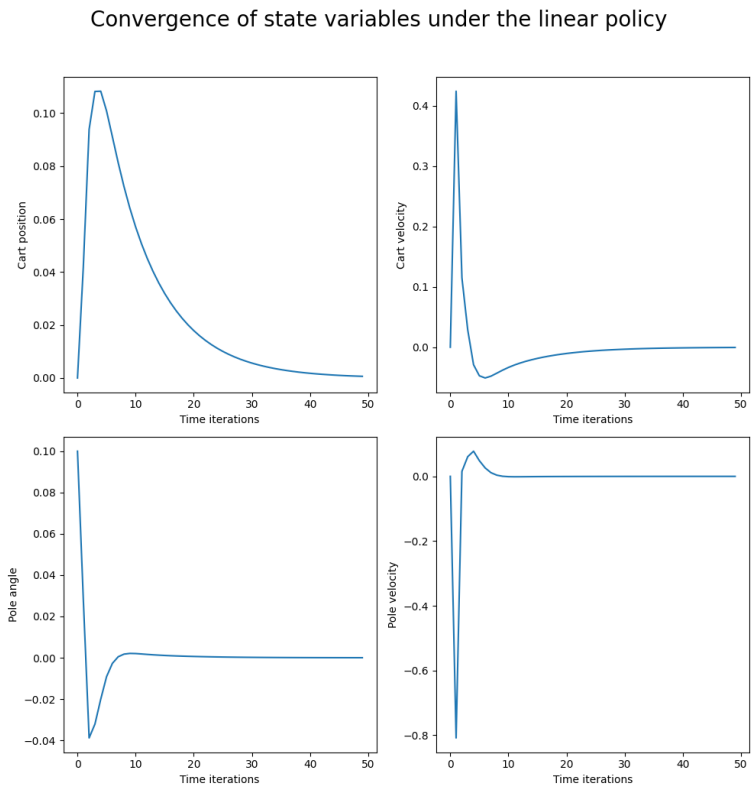

Figure 26: Convergence under the linear policy found with the actual system simulator

Task 2.4

This task is a repeat of Task 2.3, except that rather than using the loss function of the actual cart pole simulator, we are using the nonlinear model we have previous trained to perform the same optimisation procedure. The reasoning behind why one might want to be able to do this, is that it is often much cheaper to gather data by running simulations of models, than to collect data from the physical system that the model is trying to emulate.

The chart below shows how the losses for the variables converge to zero, using the policy found with the nonlinear model:

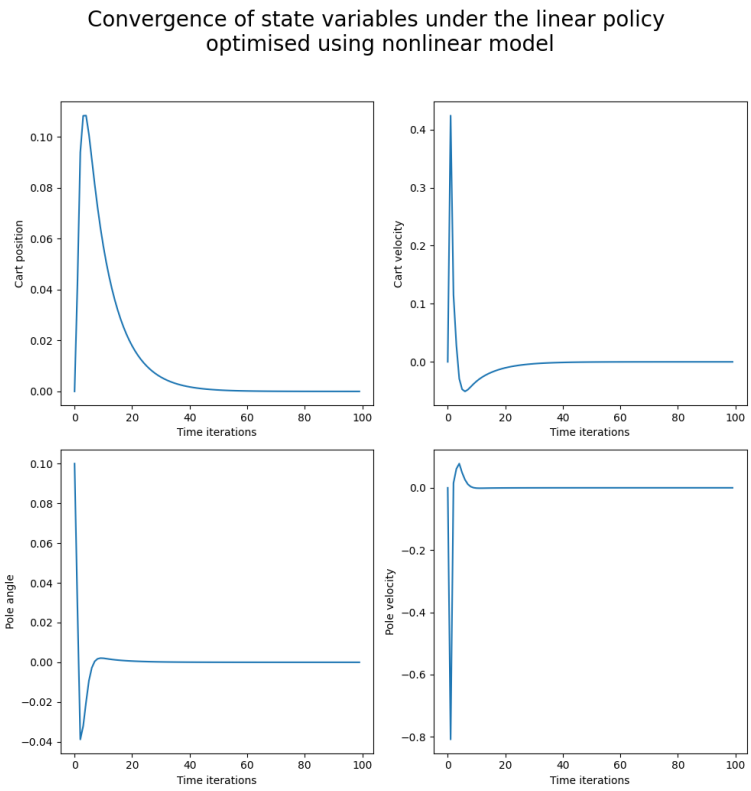

Figure 27: Convergence under the linear policy found using our kernel based model

It is important to note, that throughout tasks 2.3 and 2.4, we used the initial state vector `[0, 0, 0.1, 0]`, which is only a small perturbation away from the desired configuration. More complicated initial states lead to the policy failing to converge. For a more robust control policy, see Task 4.

Task 3.1

In this task, we introduce noise to the observed state of the system, and evaluate how the prediction accuracy of the linear and nonlinear models would be impacted. We generate noisy training data, by adding random Gaussian vectors with a specified mean and variance, where the noise corresponding to each state variable is scaled by a factor proportional to the range of that state variable. We then trained the linear and non-linear models as before, using the noisy training dataset. We can investigate the impact that adding noise to the training dataset has on the resulting model performance, by comparing the original scatter plots of the models to those trained with noise:

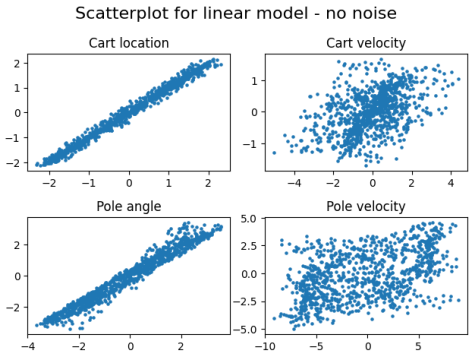

Figure 28: Scatter plot for linear model trained on noise-free data

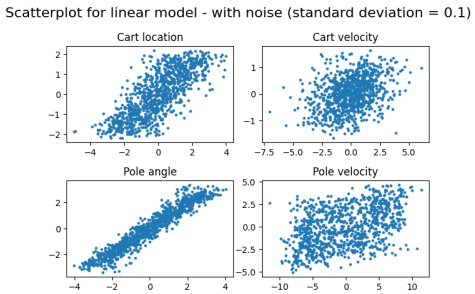

Figure 29: Scatter plot for linear model trained on noisy data

The original linear model was already not ideal, with the data points barely having any semblance of a positive linear correlation in the case of the velocities. It is still fairly easy to tell, however, that the model trained on the noisy data set has a larger variance for each of the state variables.

Now we do the same for the nonlinear model:

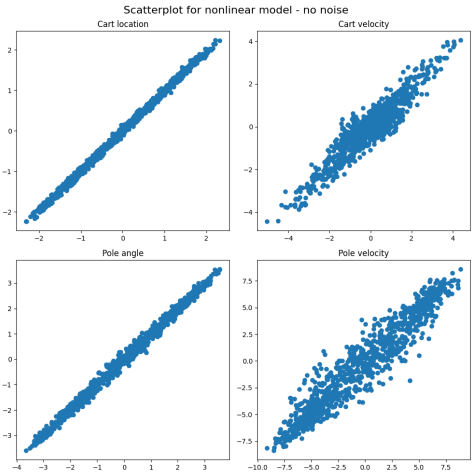

Figure 30: Scatter plot for nonlinear model trained on noise-free data

Figure 31: Scatter plot for nonlinear model trained on noisy data

Once again there is a very noticeable increase in variance between the two cases. The random noise vectors added to all of the above cases had the same standard deviation, and so to compare how the addition of noise into the training data set affects the linear and nonlinear models respectively, we calculate the factor by which the MSE decreases between the noisy and noise free versions.

	Linear model	Nonlinear model
Noise free	`[ 0.0168283  1.55228327  0.14114946 15.88518575]`	`[0.00251737 0.19608424 0.01389892 1.19`
Noisy	`[ 0.98057665  2.69896374  0.25103913 18.79899035]`	`[0.48919268 0.68232456 0.06254616 2.52`

Table 1: MSE for different models, with and without observed noise

Finding the factor of difference for each state variable, we get that: for the linear model, using noisy data increased MSE of the linear model by factors of [58.37 1.74 1.78 1.1], whereas for the nonlinear

model it increases by factors of [195.68 3.48 4.50 2.11]. We can draw the inference that the nonlinear model is more sensitive to noise than the linear model. This makes sense intuitively, as for a linear function, the noise above and below the diagonal straight line on the plots cancel out more or less. The nonlinear model is more sensitive to the noise, and is able to fit to it to a certain degree, which degrades its performance in making predictions for the true dynamics.

The next figure shows the time evolution of the state variables, if we use the optimal noise vector found using the noisy data set. In this task, the key assumption is that the noise only occurs in the observed state - not the actual state.

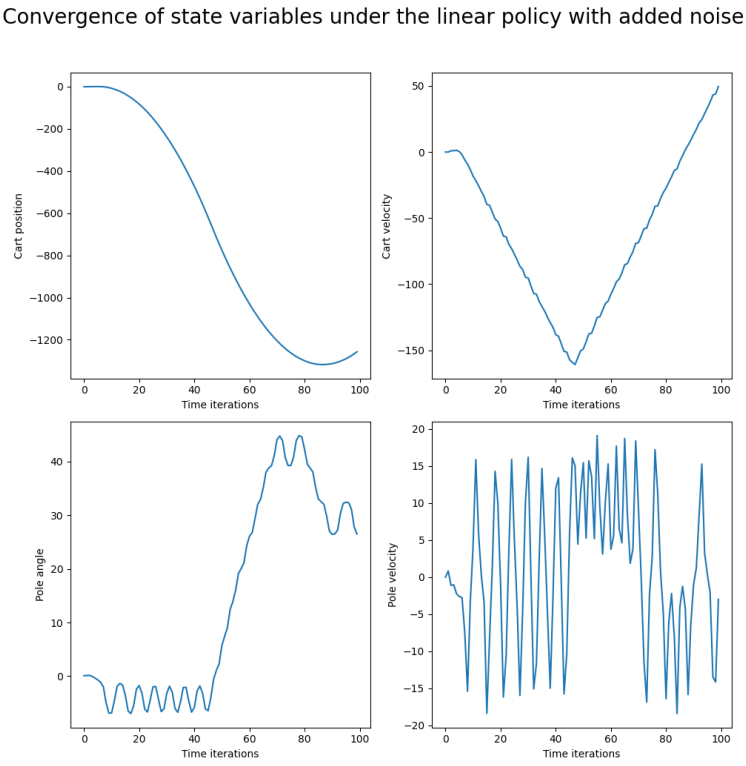

Figure 32: Time evolution of the state variables, using the optimal noise vector found using the noisy data set

We see that, in stark contrast to the equivalent plots in Task 2, the optimal `p` we found does not converge to zero. Both the cart location and velocity end up drifting away from zero. The pole variables fluctuates violently around zero, but show no signs of stabilising. This is obtained by running the Nelder-Mead algorithm 100 times with randomly initialised starting guesses, which apparently was not able to minimise the loss function. It's possible that given more time, the Nelder-Mead algorithm would stumble upon an initial guess for $\mathbf{p}$ which happen to lead to a much lower minimum. However, due to time constraints we will leave this as a potential improvement. The number of runs it takes before finding a new initial guess that leads to a smaller minimum only gets bigger. It is fair to assume that after 100 optimisation attempts with a different initial guess each time, over a small range - it will be difficult to stumble upon another one that optimises to a smaller state.

Task 3.2

This task is largely a repeat of the previous. The only difference is, now we are introducing the noise to the actual state of the system at each iterative step, as opposed to merely the observed value, where

we can simply add it in at the end. We optimised the Below is the time evolution, with noise actually affecting my performance.

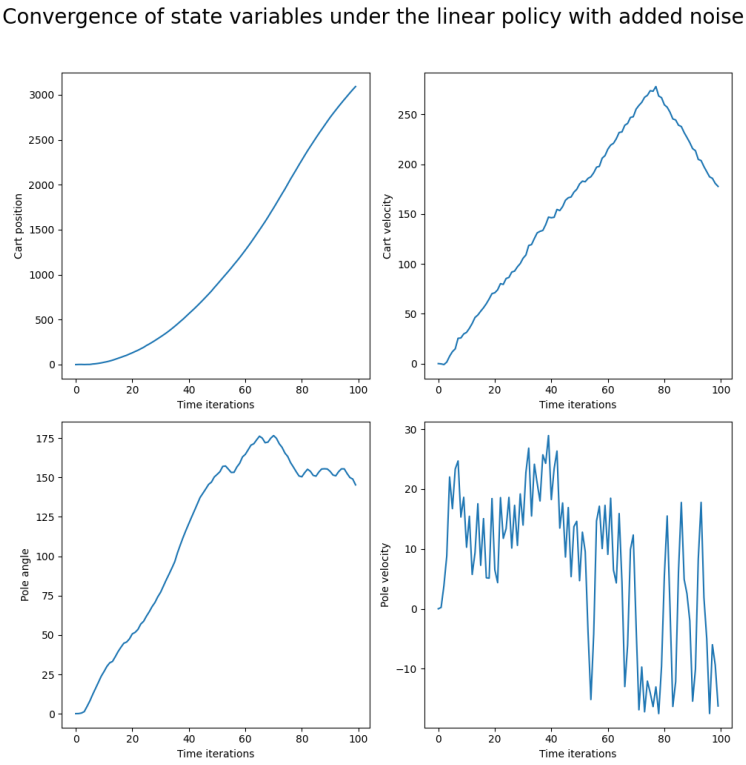

Figure 33: Time evolution of the state variables with noise introduced to the actual system

Task 4

Having seen the limitations of the linear control policy, we determine that in order to obtain robust control over a wider range of initial conditions, in the presence of noise, we need to implement a nonlinear control policy. Similar to how the nonlinear model yielded excellent performance when compared to the linear model, we can expect the nonlinear policy to provide better control than the linear policy. The policy function we use for this task is

$$p(X) = \sum_i w_i e^{-0.5(X-X_i)^T W (X-X_i)} \tag{2}$$

and our objective is to find the optimum combination of parameters, much like what we did in task 3.

The challenge is, the large number of hyper-parameters this policy would involve. The recommended number of basis functions to be is between 5 and 20, meaning there will be in the range between 41 and 116 arguments to pass into the objective functions. Using any minimisation algorithm provided by Scipy would be infeasible with the compute resources available. A grid search would also be extremely impractical.

It is deemed that the only suitable method that will run within a reasonable time frame would be trying a large number of randomly generated hyperparamters. If the number of combinations tried is large, then there is a good chance that we will stumble upon a policy which is close to the optimum and yields a smaller MSE over a set number of time iterations.

In order to investigate the practicality of the proposed method, I wrote code that randomly generated a large number of different combinations of the hyperparameter values. I chose to use only 5 basis functions, to reduce a computational expense, since the principle is the same regardless of the number of basis functions. I varied two different factors in this approach: the number of different combinations generated and the range of values over which they were generated. Below I showed the plots of the minimum MSE found using this method, as I varied the number of combinations and range for the values (keeping the other fixed).

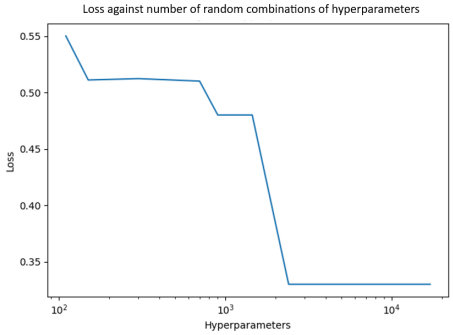

Figure 34: Minimum MSE against the number of configurations generated

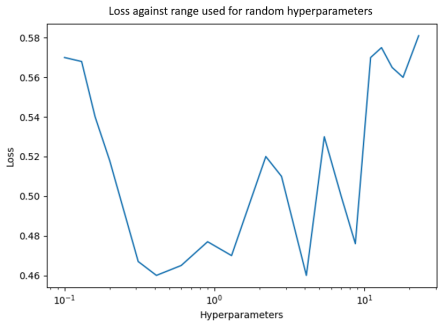

Figure 35: Minimum MSE against the range from which the hyperparameters are generated

We see that there appear to be an optimum when the range is around 0.5. This makes intuitive sense, as the smaller the range, the greater the density of the random configurations, and hence the more likely an smaller optimal loss is found. However, if the range is too small then it becomes restrictive - hence the reduction in performance when the range is ¡0.5. As we would expect, generally, the greater the number of configurations generated, the smaller the optimal loss. However, it is interesting to note how it plateaus after when the number of guesses surpasses 2000, indicative of the limitations of this method. Of course, since the configurations are generated pseudo-randomly, so long as we do not give it the same seed, running the procedure again may well give different results, but the key ideas would remain the same.

Conclusion

To conclude, let us summarised what we have achieved and learnt in this project. Initially, we successfully used least squares regression to find the optimal linear model, and used it to predict the evolution of the cart pole system in time, from different starting conditions, and found sub optimal results. We then implemented a kernel based nonlinear model that uses Gaussian basis functions, optimising its parameters using grid search, and used it to achieve much better time series predictions of the state of the system.

A key benefit of an accurate nonlinear model, is that we can use it to find the optimal control policy, without having to operate the actual system (though it is merely a simulation in the case of this project, in real life it may be expensive to run the physical system). We were able to do so successfully - finding an optimal control policy using the Nelder-Mead algorithm using the nonlinear model, which successfully stabilises the system. However, we saw how the introduction of noise disrupted the policy. In order to find a more robust policy that can work from a greater range of starting points and more resistant to noise, we introduced a nonlinear policy function, and had an attempt at finding the optimal hyperparameters for it using random number generation. The project was an overall success, and I was able to much deeper insight into the principles of reinforcement learning.

Appendix

```
import numpy as np
import matplotlib.pyplot as plt
from CartPole import CartPole

cp = CartPole()

n = 50

initial_state_simple_osc = [0, 0, np.pi, 0.5]
initial_state_complete_rot = [0, 0, np.pi, 15]

# Oscillation around the equilibrium

def rollout(initial_state):
    cp.setState(initial_state_simple_osc)
    pos = []
    velo = []
    pole_pos = []
    pole_velo = []

    for _ in range(n):
        pos.append(cp.cart_location)
        velo.append(cp.cart_velocity)
        pole_pos.append(cp.pole_angle)
        pole_velo.append(cp.pole_velocity)
        cp.performAction()

    times = np.arange(0,len(pos),1)

    plt.figure(figsize=(10,10))
    plt.subplot(2,2,1)
    plt.plot(times, pos)
    plt.xlabel('Time step')
    plt.ylabel('x (m)')
    plt.title("Cart position")
    plt.subplot(2,2,2)
    plt.plot(times, velo)
    plt.xlabel('Time step')
    plt.ylabel(r'$\dot{x}$ (m/s)')
    plt.title("Cart velocity")
    plt.subplot(2,2,3)
    plt.plot(times, pole_pos)
    plt.xlabel('Time step')
    plt.ylabel(r'$\theta$ (rad)')
    plt.title("Pole angle")
    plt.subplot(2,2,4)
    plt.plot(times, pole_velo)
    plt.xlabel('Time step')
    plt.ylabel(r'$\dot{\theta}$ (rad/s)')
    plt.title("Pole velocity")
    plt.tight_layout()
    plt.show()
```

```
    plt.figure(figsize=(15,10))
    plt.subplot(3,2,1)
    plt.plot(pos, velo)
    plt.title("Cart position vs Cart velocity")
    plt.xlabel("x (m)")
    plt.ylabel(r"$\dot{x}$ (m/s)")
    plt.subplot(3,2,2)
    plt.plot(pos, pole_pos)
    plt.title("Cart position vs Pole angle")
    plt.xlabel("x (m)")
    plt.ylabel(r"$\theta$ (rad)")
    plt.subplot(3,2,3)
    plt.plot(pos, pole_velo)
    plt.title("Cart position vs Pole velocity")
    plt.xlabel("x (m)")
    plt.ylabel(r"$\dot{x}$ (m/s)")
    plt.subplot(3,2,4)
    plt.plot(velo, pole_pos)
    plt.title("Cart velocity vs Pole angle")
    plt.xlabel("x (m)")
    plt.ylabel(r"$\dot{x}$ (m/s)")
    plt.subplot(3,2,5)
    plt.plot(velo, pole_velo)
    plt.title("Cart velocity vs Pole velocity")
    plt.xlabel("x (m)")
    plt.ylabel(r"$\dot{x}$ (m/s)")
    plt.subplot(3,2,6)
    plt.plot(times, pole_velo)
    plt.title("Pole angle vs Pole velocity")
    plt.tight_layout()
    plt.show()


rollout(initial_state_simple_osc)
rollout(initial_state_complete_rot)
```

Task 1.2 code

```
    import random

# Y defined here as the output of PerformAction with X as the input

# Using [-10, 10] as suitable range for cart position

cp = CartPole()
n = 50

c_location = random.uniform(-10,10)
c_velocity = random.uniform(-10,10)
p_angle = random.uniform(-np.pi, np.pi)
p_velocity = random.uniform(-15,15)
state = [c_location, c_velocity, p_angle, p_velocity]
```

```
def sweep1(var):
    l_list = []
    v_list = []
    pa_list = []
    pv_list = []
    cp.setState(state)
    # sweep across cart position

    label = 0

    if var == 1:
        x_range = np.linspace(-1,1,n)
        for x in x_range:
            state[0] = x
            cp.setState(state)
            cp.performAction()
            l_list.append(state[0])
            v_list.append(cp.cart_velocity)
            pa_list.append(cp.pole_angle)
            pv_list.append(cp.pole_velocity)
            label = r'$x$'
    # sweep across cart velocity
    if var == 2:
        x_range = np.linspace(-10,10,n)
        for x in x_range:
            state[1] = x
            cp.setState(state)
            cp.performAction()
            l_list.append(cp.cart_location)
            v_list.append(state[1])
            pa_list.append(cp.pole_angle)
            pv_list.append(cp.pole_velocity)
            label = r'$\dot{x}$'
    # sweep across pole angle
    if var == 3:
        x_range = np.linspace(-np.pi,np.pi,n)
        for x in x_range:
            state[2] = x
            cp.setState(state)
            cp.performAction()
            l_list.append(cp.cart_location)
            v_list.append(cp.cart_velocity)
            pa_list.append(state[2])
            pv_list.append(cp.pole_velocity)
            label = r'$\theta$'
    # sweep across pole velocity
    if var == 4:
        x_range = np.linspace(-15,15,n)
        for x in x_range:
            state[3] = x
            cp.setState(state)
            cp.performAction()
```

```
            l_list.append(cp.cart_location)
            v_list.append(cp.cart_velocity)
            pa_list.append(cp.pole_angle)
            pv_list.append(state[3])
            label = r'$\dot{\theta}$'
    plt.figure(figsize=(10,2.5))
    plt.subplot(1,4,1)
    plt.plot(x_range, l_list)
    plt.xlabel(label)
    plt.title("Cart position")
    plt.subplot(1,4,2)
    plt.plot(x_range,v_list)
    plt.xlabel(label)
    plt.title("Cart velocity")
    plt.subplot(1,4,3)
    plt.plot(x_range,pa_list)
    plt.xlabel(label)
    plt.title("Pole angle")
    plt.subplot(1,4,4)
    plt.plot(x_range, pv_list)
    plt.xlabel(label)
    plt.title("Pole velocity")
    plt.tight_layout()
    plt.show()


sweep1(1)
sweep1(2)
sweep1(3)
sweep1(4)

c_location = random.uniform(-10,10)
c_velocity = random.uniform(-10,10)
p_angle = random.uniform(-np.pi, np.pi)
p_velocity = random.uniform(-15,15)
state = [c_location, c_velocity, p_angle, p_velocity]

def sweep2(var):
    l_list = []
    v_list = []
    pa_list = []
    pv_list = []
    cp.setState(state)
    # sweep across cart position
    if var == 1:
        x_range = np.linspace(-1,1,n)
        for x in x_range:
            state[0] = x
            cp.setState(state)
            prev_cl = state[0]
            prev_cv = state[1]
            prev_pl = state[2]
            prev_pv = state[3]
            cp.performAction()
```

```
        l_list.append(cp.cart_location - x)
        v_list.append(cp.cart_velocity - prev_cv)
        pa_list.append(cp.pole_angle - prev_pl)
        pv_list.append(cp.pole_velocity - prev_pv)
# sweep across cart velocity
if var == 2:
    x_range = np.linspace(-10,10,n)
    for x in x_range:
        state[1] = x
        cp.setState(state)
        prev_cl = state[0]
        prev_cv = state[1]
        prev_pl = state[2]
        prev_pv = state[3]
        cp.performAction()
        l_list.append(cp.cart_location - prev_cl)
        v_list.append(cp.cart_velocity - x)
        pa_list.append(cp.pole_angle - prev_pl)
        pv_list.append(cp.pole_velocity - prev_pv)
# sweep across pole angle
if var == 3:
    x_range = np.linspace(-np.pi,np.pi,n)
    for x in x_range:
        state[2] = x
        cp.setState(state)
        prev_cl = state[0]
        prev_cv = state[1]
        prev_pl = state[2]
        prev_pv = state[3]
        cp.performAction()
        l_list.append(cp.cart_location - prev_cl)
        v_list.append(cp.cart_velocity - prev_cv)
        pa_list.append(cp.pole_angle - x)
        pv_list.append(cp.pole_velocity - prev_pv)
# sweep across pole velocity
if var == 4:
    x_range = np.linspace(-15,15,n)
    for x in x_range:
        state[3] = x
        cp.setState(state)
        prev_cl = state[0]
        prev_cv = state[1]
        prev_pl = state[2]
        prev_pv = state[3]
        cp.performAction()
        l_list.append(cp.cart_location - prev_cl)
        v_list.append(cp.cart_velocity - prev_cv)
        pa_list.append(cp.pole_angle - prev_pl)
        pv_list.append(cp.pole_velocity - x)

plt.figure(figsize=(10,2.5))
plt.subplot(1,4,1)
plt.plot(x_range, l_list)
plt.title("Cart position", pad=20)
```

```
    plt.subplot(1,4,2)
    plt.plot(x_range,v_list)
    plt.title("Cart velocity",pad=20)
    plt.subplot(1,4,3)
    plt.plot(x_range,pa_list)
    plt.title("Pole angle", pad=20)
    plt.subplot(1,4,4)
    plt.plot(x_range, pv_list)
    plt.title("Pole velocity", pad=20)
    plt.tight_layout()
    plt.show()

sweep2(1)
sweep2(2)
sweep2(3)
sweep2(4)

import matplotlib.tri as tri

# List of pairs of meshgrids:
# cl - cv
# cl - pa
# cl - pv
# cv - pa
# cv - pv
# pa - pv

cp = CartPole()
n = 50

c_location = random.uniform(-10,10)
c_velocity = random.uniform(-10,10)
p_angle = random.uniform(-np.pi, np.pi)
p_velocity = random.uniform(-15,15)
state = [c_location, c_velocity, p_angle, p_velocity]


cl_range = np.linspace(-1,1,50)
cv_range = np.linspace(-10,10,50)
pa_range = np.linspace(-np.pi,np.pi,50)
pv_range = np.linspace(-15,15,50)


Y_cl_range = np.zeros((50,50))
Y_cv_range = np.zeros((50,50))
Y_pa_range = np.zeros((50,50))
Y_pv_range = np.zeros((50,50))

plt.figure(figsize=(10,10))


# Pair 1: cl - cv

for i, cl in enumerate(cl_range):
```

```
    for j, cv in enumerate(cv_range):
        state[0] = cl
        state[1] = cv
        cp.setState(state)
        cp.performAction()

        new_state = cp.getState()
        Y_cl_range[i][j] = new_state[0] - state[0]
        Y_cv_range[i][j] = new_state[1] - state[1]
        Y_pa_range[i][j] = new_state[2] - state[2]
        Y_pv_range[i][j] = new_state[3] - state[3]



meshed_cl, meshed_cv = np.meshgrid(cl_range,cv_range)
meshed_cl = meshed_cl.flatten()
meshed_cv = meshed_cv.flatten()
Y_cl_range = Y_cl_range.flatten()
Y_cv_range = Y_cv_range.flatten()
Y_pa_range = Y_pa_range.flatten()
Y_pv_range = Y_pv_range.flatten()
triang_1 = tri.Triangulation(meshed_cl, meshed_cv)


plt.subplot(6,2,1)
plt.tricontourf(triang_1, Y_pa_range)
plt.xlabel(r'$x$')
plt.ylabel(r'$\dot{x}$')
plt.title(r'$\Delta\theta$')
plt.colorbar()

plt.subplot(6,2,2)
plt.tricontourf(triang_1, Y_pv_range)
plt.xlabel(r'$x$')
plt.ylabel(r'$\dot{x}$')
plt.title(r'$\Delta \dot{\theta}$')
plt.colorbar()


# Pair 2: cl - pa

Y_cl_range = np.zeros((50,50))
Y_cv_range = np.zeros((50,50))
Y_pa_range = np.zeros((50,50))
Y_pv_range = np.zeros((50,50))
cl_range = np.linspace(-1,1,50)
cv_range = np.linspace(-10,10,50)
pa_range = np.linspace(-np.pi,np.pi,50)
pv_range = np.linspace(-15,15,50)

for i, cl in enumerate(cl_range):
    for j, pa in enumerate(pa_range):
        state[0] = cl
        state[2] = pa
```

```
        cp.setState(state)
        cp.performAction()

        new_state = cp.getState()
        Y_cl_range[i][j] = new_state[0] - state[0]
        Y_cv_range[i][j] = new_state[1] - state[1]
        Y_pa_range[i][j] = new_state[2] - state[2]
        Y_pv_range[i][j] = new_state[3] - state[3]


meshed_cl, meshed_pa = np.meshgrid(cl_range,pa_range)
meshed_cl = meshed_cl.flatten()
meshed_pa = meshed_pa.flatten()
Y_cl_range = Y_cl_range.flatten()
Y_cv_range = Y_cv_range.flatten()
Y_pa_range = Y_pa_range.flatten()
Y_pv_range = Y_pv_range.flatten()
triang_2 = tri.Triangulation(meshed_cl, meshed_pa)

plt.subplot(6,2,3)
plt.tricontourf(triang_2, Y_cv_range)
plt.xlabel(r'$x$')
plt.ylabel(r'$\theta$')
plt.title(r'$\Delta \dot{x}$')
plt.colorbar()

plt.subplot(6,2,4)
plt.tricontourf(triang_2, Y_pv_range)
plt.xlabel(r'$x$')
plt.ylabel(r'$\theta$')
plt.title(r'$\Delta \dot{\theta}$')
plt.colorbar()

# Pair 3: cl - pv

Y_cl_range = np.zeros((50,50))
Y_cv_range = np.zeros((50,50))
Y_pa_range = np.zeros((50,50))
Y_pv_range = np.zeros((50,50))
cl_range = np.linspace(-1,1,50)
cv_range = np.linspace(-10,10,50)
pa_range = np.linspace(-np.pi,np.pi,50)
pv_range = np.linspace(-15,15,50)

for i, cl in enumerate(cl_range):
    for j, pv in enumerate(pv_range):
        state[0] = cl
        state[3] = pv
        cp.setState(state)
        cp.performAction()

        new_state = cp.getState()
        Y_cl_range[i][j] = new_state[0] - state[0]
        Y_cv_range[i][j] = new_state[1] - state[1]
```

```
        Y_pa_range[i][j] = new_state[2] - state[2]
        Y_pv_range[i][j] = new_state[3] - state[3]


meshed_cl, meshed_pv = np.meshgrid(cl_range,pv_range)
meshed_cl = meshed_cl.flatten()
meshed_pv = meshed_pv.flatten()
Y_cl_range = Y_cl_range.flatten()
Y_cv_range = Y_cv_range.flatten()
Y_pa_range = Y_pa_range.flatten()
Y_pv_range = Y_pv_range.flatten()
triang_3 = tri.Triangulation(meshed_cl, meshed_pv)

plt.subplot(6,2,5)
plt.tricontourf(triang_3, Y_cv_range)
plt.xlabel(r'$x$')
plt.ylabel(r'$\dot{\theta}$')
plt.title(r'$\Delta \dot{x}$')
plt.colorbar()

plt.subplot(6,2,6)
plt.tricontourf(triang_3, Y_pa_range)
plt.xlabel(r'$x$')
plt.ylabel(r'$\dot{\theta}$')
plt.title(r'$\Delta \theta$')
plt.colorbar()

# Pair 4: cv - pa

Y_cl_range = np.zeros((50,50))
Y_cv_range = np.zeros((50,50))
Y_pa_range = np.zeros((50,50))
Y_pv_range = np.zeros((50,50))
cl_range = np.linspace(-1,1,50)
cv_range = np.linspace(-10,10,50)
pa_range = np.linspace(-np.pi,np.pi,50)
pv_range = np.linspace(-15,15,50)

for i, cv in enumerate(cv_range):
    for j, pa in enumerate(pa_range):
        state[1] = cv
        state[2] = pa
        cp.setState(state)
        cp.performAction()

        new_state = cp.getState()
        Y_cl_range[i][j] = new_state[0] - state[0]
        Y_cv_range[i][j] = new_state[1] - state[1]
        Y_pa_range[i][j] = new_state[2] - state[2]
        Y_pv_range[i][j] = new_state[3] - state[3]


meshed_cv, meshed_pa = np.meshgrid(cv_range,pa_range)
meshed_cv = meshed_cv.flatten()
```

```
meshed_pa = meshed_pa.flatten()
Y_cl_range = Y_cl_range.flatten()
Y_cv_range = Y_cv_range.flatten()
Y_pa_range = Y_pa_range.flatten()
Y_pv_range = Y_pv_range.flatten()
triang_4 = tri.Triangulation(meshed_cv, meshed_pa)

plt.subplot(6,2,7)
plt.tricontourf(triang_4, Y_cl_range)
plt.xlabel(r'$\dot{x}$')
plt.ylabel(r'$\theta$')
plt.title(r'$\Delta x$')
plt.colorbar()

plt.subplot(6,2,8)
plt.tricontourf(triang_4, Y_pv_range)
plt.xlabel(r'$\dot{x}$')
plt.ylabel(r'$\theta$')
plt.title(r'$\Delta \dot{\theta}$')
plt.colorbar()

# Pair 5: cv - pv

Y_cl_range = np.zeros((50,50))
Y_cv_range = np.zeros((50,50))
Y_pa_range = np.zeros((50,50))
Y_pv_range = np.zeros((50,50))
cl_range = np.linspace(-1,1,50)
cv_range = np.linspace(-10,10,50)
pa_range = np.linspace(-np.pi,np.pi,50)
pv_range = np.linspace(-15,15,50)

for i, cv in enumerate(cv_range):
    for j, pv in enumerate(pv_range):
        state[1] = cv
        state[3] = pv
        cp.setState(state)
        cp.performAction()

        new_state = cp.getState()
        Y_cl_range[i][j] = new_state[0] - state[0]
        Y_cv_range[i][j] = new_state[1] - state[1]
        Y_pa_range[i][j] = new_state[2] - state[2]
        Y_pv_range[i][j] = new_state[3] - state[3]


meshed_cv, meshed_pv = np.meshgrid(cv_range,pv_range)
meshed_cv = meshed_cv.flatten()
meshed_pv = meshed_pv.flatten()
Y_cl_range = Y_cl_range.flatten()
Y_cv_range = Y_cv_range.flatten()
Y_pa_range = Y_pa_range.flatten()
Y_pv_range = Y_pv_range.flatten()
triang_5 = tri.Triangulation(meshed_cv, meshed_pv)
```

```
plt.subplot(6,2,9)
plt.tricontourf(triang_5, Y_cl_range)
plt.xlabel(r'$\dot{x}$')
plt.ylabel(r'$\dot{\theta}$')
plt.title(r'$\Delta x$')
plt.colorbar()

plt.subplot(6,2,10)
plt.tricontourf(triang_5, Y_pa_range)
plt.xlabel(r'$\dot{x}$')
plt.ylabel(r'$\dot{\theta}$')
plt.title(r'$\Delta \theta$')
plt.colorbar()

# Pair 6: pa - pv

Y_cl_range = np.zeros((50,50))
Y_cv_range = np.zeros((50,50))
Y_pa_range = np.zeros((50,50))
Y_pv_range = np.zeros((50,50))
cl_range = np.linspace(-1,1,50)
cv_range = np.linspace(-10,10,50)
pa_range = np.linspace(-np.pi,np.pi,50)
pv_range = np.linspace(-15,15,50)

for i, pa in enumerate(pa_range):
    for j, pv in enumerate(pv_range):
        state[2] = pa
        state[3] = pv
        cp.setState(state)
        cp.performAction()

        new_state = cp.getState()
        Y_cl_range[i][j] = new_state[0] - state[0]
        Y_cv_range[i][j] = new_state[1] - state[1]
        Y_pa_range[i][j] = new_state[2] - state[2]
        Y_pv_range[i][j] = new_state[3] - state[3]


meshed_pa, meshed_pv = np.meshgrid(pa_range,pv_range)
meshed_pa = meshed_pa.flatten()
meshed_pv = meshed_pv.flatten()
Y_cl_range = Y_cl_range.flatten()
Y_cv_range = Y_cv_range.flatten()
Y_pa_range = Y_pa_range.flatten()
Y_pv_range = Y_pv_range.flatten()
triang_6 = tri.Triangulation(meshed_pa, meshed_pv)

plt.subplot(6,2,11)
plt.tricontourf(triang_6, Y_cl_range)
plt.xlabel(r'$\theta$')
plt.ylabel(r'$\dot{\theta}$')
plt.title(r'$\Delta x$')
```

```
plt.colorbar()

plt.subplot(6,2,12)
plt.tricontourf(triang_6, Y_cv_range)
plt.xlabel(r'$\theta$')
plt.ylabel(r'$\dot{\theta}$')
plt.title(r'$\Delta\dot{x}$')
plt.colorbar()


plt.tight_layout()
plt.show()
```

Task 1.3 code

```
import numpy as np
import matplotlib.pyplot as plt
from cartpole import CartPole

# Define all functions here
def generate_random_state(x_range, x_dot_range, theta_range,
    theta_dot_range):
     return np.array([np.random.uniform(*x_range), np.random.uniform(*
         x_dot_range),
                      np.random.uniform(*theta_range), np.random.
                         uniform(*theta_dot_range)])

def generate_data(number_of_points, delta_time, sim_steps,
    generate_next_state_randomly, start_state,
                  x_range, x_dot_range, theta_range, theta_dot_range,
                     remap_angle):

     past_state = np.zeros((number_of_points, 4))
     future_state = np.zeros((number_of_points, 4))
     cartpole = CartPole()
     cartpole.delta_time = delta_time
     cartpole.sim_steps = sim_steps

     if generate_next_state_randomly:
         np.random.seed(0)
         for i in range(number_of_points):
             cartpole.setState(generate_random_state(x_range,
                x_dot_range, theta_range, theta_dot_range))
             current_state = cartpole.getState()
             if remap_angle:
                 current_state[2] = remap_angle(current_state[2])
             past_state[i] = current_state
             cartpole.performAction(0)
             new_state = cartpole.getState()
             if remap_angle:
                 new_state[2] = remap_angle(new_state[2])
             future_state[i] = new_state
     else:
```

```
        for i in range(number_of_points):
            current_state = start_state if i == 0 else future_state[i
                -1]
            if remap_angle:
                current_state[2] = remap_angle(current_state[2])
            past_state[i] = current_state
            cartpole.setState(past_state[i])
            cartpole.performAction(0)
            new_state = cartpole.getState()
            if remap_angle:
                new_state[2] = remap_angle(new_state[2])
            future_state[i] = new_state

    change = future_state - past_state
    return past_state, change

def get_model(past_state, change):
    return np.linalg.lstsq(past_state, change, rcond=None)[0]

def predict_change(state, model):
    return state @ model

def predict_state(state, model, remap_theta=True):
    change = predict_change(state, model)
    new_state = state + change
    if remap_theta:
        new_state[2] = remap_angle(new_state[2])
    return new_state

# Main code starts here
number_of_points = 500
delta_time = 0.2
sim_steps = 50
generate_next_state_randomly = True
x_range = (-10, 10)
x_dot_range = (-10, 10)
theta_range = (-np.pi, np.pi)
theta_dot_range = (-15, 15)
remap_angle = False
start_state = None

past_state, change = generate_data(number_of_points, delta_time,
   sim_steps, generate_next_state_randomly, start_state,
                x_range, x_dot_range, theta_range, theta_dot_range,
                   remap_angle)

small_time_step_model = get_model(past_state, change)
print(small_time_step_model)

# Continue from previous part

future = past_state + change
model_predictions = predict_change(past_state, small_time_step_model)
predictions = model_predictions.T
```

```
future_predictions = past_state + model_predictions

def generate_plot(ax, future, future_predictions, label, xlabel,
   ylabel, title):
    ax.scatter(future, future_predictions, label=label)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.title.set_text(title)

fig, ax = plt.subplots(2, 2, figsize=(20, 10))

generate_plot(ax[0, 0], future[:, 0], future_predictions[:, 0], 'x', '
   Actual state', 'Predicted state', 'Cart location')
generate_plot(ax[0, 1], future[:, 1], future_predictions[:, 1], 'x_dot
   ', 'Actual state', 'Predicted state', 'Cart velocity')
generate_plot(ax[1, 0], future[:, 2], future_predictions[:, 2], 'theta
   ', 'Actual state', 'Predicted state', 'Pole angle')
generate_plot(ax[1, 1], future[:, 3], future_predictions[:, 3], '
   theta_dot', 'Actual state', 'Predicted state', 'Pole velocity')

fig.suptitle('Predicted state vs Actual state after one call to
   performAction()')
plt.show()

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
fig.suptitle("Actual change in state vs Predicted change in state
   after one call to performAction()", fontsize=20)

generate_plot(ax1, change[:, 0], predictions[0], None, 'Actual change
   ', 'Predicted change', 'Cart location')
generate_plot(ax2, change[:, 1], predictions[1], None, 'Actual change
   ', 'Predicted change', 'Cart velocity')
generate_plot(ax3, change[:, 2], predictions[2], None, 'Actual change
   ', 'Predicted change', 'Pole angle')
generate_plot(ax4, change[:, 3], predictions[3], None, 'Actual change
   ', 'Predicted change', 'Pole velocity')

for ax in (ax1, ax2, ax3, ax4):
    ax.tick_params(axis='both', which='major', labelsize=15)
    ax.set_xlabel("Actual change", fontsize=15)
    ax.set_ylabel("Predicted change", fontsize=15)

plt.tight_layout()
plt.show()

def initialize_system(number_of_search_points, state_constant, model,
                      state_range=((-1, 1), (-10, 10), (-np.pi, np.pi)
                          , (-15, 15)), delta_time=0.2, sim_steps=50):

    initial_state_case_1 = np.full((4, number_of_search_points),
       state_constant)
    initial_state = np.stack([initial_state_case_1] * 4, axis=0)

    for idx in range(4):
```

```
        initial_state[idx, idx, :] = np.linspace(state_range[idx][0],
            state_range[idx][1], number_of_search_points)

    change_in_state = np.zeros_like(initial_state)
    model_change_in_state = np.zeros_like(initial_state)

    return initial_state, change_in_state, model_change_in_state


def compute_change(number_of_search_points, model, initial_state,
    change_in_state, model_change_in_state, delta_time, sim_steps):
    cartpole = CartPole()
    cartpole.delta_time = delta_time
    cartpole.sim_steps = sim_steps

    for i in range(4):
        for j in range(number_of_search_points):
            cartpole.setState(initial_state[i, :, j])
            cartpole.performAction(0)
            change_in_state[i, :, j] = cartpole.getState() -
                initial_state[i, :, j]
            model_change_in_state[i, :, j] = model.predict_change(
                initial_state[i, :, j])

    return change_in_state, model_change_in_state


def single_variable_scan(number_of_search_points, state_constant,
    model,
                         state_range=((-1, 1), (-10, 10), (-np.pi, np.
                             pi), (-15, 15)), delta_time=0.2, sim_steps
                             =50):
    initial_state, change_in_state, model_change_in_state =
        initialize_system(number_of_search_points, state_constant,
        model, state_range, delta_time, sim_steps)
    change_in_state, model_change_in_state = compute_change(
        number_of_search_points, model, initial_state, change_in_state,
         model_change_in_state, delta_time, sim_steps)

    return initial_state, change_in_state, model_change_in_state

# Plot functions
def plot_change(initial_state, change_in_state, model_change_in_state,
    fig=None, axs=None, case="", linestyle1="solid", linestyle2="solid
    "):
    if axs is None:
        fig, axs = plt.subplots(4, 4)
    fig.suptitle('Change in state variables when one state variable is
        varied', fontsize=30, y = 1.5)
    state_variables = ["x", "$\\dot{x}$", "$\\theta$", "$\\dot{\\theta
        }$"]
    for i in range(4):
        x_axis = initial_state[i, i, :]
        for j in range(4):
```

```
            y_actual = change_in_state[i, j, :]
            y_predicted = model_change_in_state[i, j, :]
            axs[i, j].plot(x_axis, y_actual, label="Actual Change",
               linewidth=5, linestyle=linestyle1)
            axs[i, j].plot(x_axis, y_predicted, label="Predicted
               Change", linewidth=5, linestyle=linestyle2)
            axs[i, j].set_title(f"$\\Delta$ {state_variables[j]}",
               fontsize=20)
            axs[i, j].set_xlabel(state_variables[i], fontsize=20)
            axs[i, j].set_ylabel(None, fontsize=20)
            axs[i, j].tick_params(axis='both', which='major',
               labelsize=15)
    return axs

def plot_difference_in_change(initial_state, change_in_state,
   model_change_in_state, fig=None, axs=None, case="", linestyle="
   solid"):
    if axs is None:
        fig, axs = plt.subplots(4, 4)
    fig.suptitle('Difference between Actual Change and Pricted Change
       ', fontsize=30)
    state_variables = ["x", "$\\dot{x}$", "$\\theta$", "$\\dot{\\theta
       }$"]
    for i in range(4):
        x_axis = initial_state[i, i, :]
        for j in range(4):
            y_actual = change_in_state[i, j, :]
            y_predicted = model_change_in_state[i, j, :]
            difference = y_actual - y_predicted
            axs[i, j].plot(x_axis, difference, label="Difference" +
               case, linewidth=5, linestyle=linestyle)
            axs[i, j].set_title(f"$\\Delta$ {state_variables[j]}",
               fontsize=20)
            axs[i, j].set_xlabel(state_variables[i], fontsize=20)
            axs[i, j].set_ylabel(None, fontsize=20)
            axs[i, j].tick_params(axis='both', which='major',
               labelsize=15)
    return axs

number_of_train_points = 1000
number_of_points = 100
state_constant = (0, 0, np.pi, 0)
delta_time = 0.2
sim_steps = 50

model = LinearModel(number_of_points=number_of_train_points,
   delta_time=delta_time, sim_steps=sim_steps)

linear_initial_state, linear_change_in_state,
   linear_model_change_in_state = single_variable_scan(
    number_of_points, state_constant, model, delta_time=delta_time,
       sim_steps=sim_steps)

oscillatory_initial_state, oscillatory_change_in_state,
```

```
    oscillatory_model_change_in_state = single_variable_scan(
     number_of_points, state_constant, model, delta_time=delta_time,
        sim_steps=sim_steps)

rotation_initial_state, rotation_change_in_state,
    rotation_model_change_in_state = single_variable_scan(
     number_of_points, state_constant, model, delta_time=delta_time,
        sim_steps=sim_steps)

fig, axs = plt.subplots(4, 4, figsize=(20, 20), layout="constrained")
plot_change(oscillatory_initial_state, oscillatory_change_in_state,
    oscillatory_model_change_in_state, fig, axs, " - Case: 2",
    linestyle1="solid", linestyle2="solid")

# Have to choose the range of ticks for the y axis to be the same for
    each column
for ax in axs:
     ax[0].set_ylim([-2, 2])
     ax[1].set_ylim([-4, 4])
     ax[2].set_ylim([-4, 4])
     ax[3].set_ylim([-8, 8])

handles, labels = plt.gca().get_legend_handles_labels()
fig.legend(handles, labels, loc='lower center', bbox_to_anchor=(0.5,
    -0.05), fontsize=20, ncol=3)
fig.suptitle('Scan across each state variable - oscillation about
    stable equilibirum', fontsize=40,y=1.02)
plt.tight_layout()
plt.show()
fig, axs = plt.subplots(4, 4, figsize=(20, 20), layout="constrained")
plot_change(rotation_initial_state, rotation_change_in_state,
    rotation_model_change_in_state, fig, axs, linestyle1="solid",
    linestyle2="solid")

for ax in axs:
     ax[0].set_ylim([-2, 2])
     ax[1].set_ylim([-4, 4])
     ax[2].set_ylim([-4, 4])
     ax[3].set_ylim([-8, 8])

handles, labels = plt.gca().get_legend_handles_labels()
fig.legend(handles, labels, loc='lower center', bbox_to_anchor=(0.5,
    -0.05), fontsize=20, ncol=3)
fig.suptitle('Scan across each state variable - complete rotation of
    pendulum', fontsize=40,y=1.02)
plt.tight_layout()
plt.show()

     class CartPoleControl:
     def __init__(self, initial_state, force, time_steps, remap_angle=
        False, delta_time=0.2, sim_steps=50):
         self.initial_state = initial_state
         self.force = force
         self.time_steps = time_steps
```

```
        self.remap_angle = remap_angle
        self.t = np.arange(0, time_steps, 1)
        self.state = np.zeros((time_steps, 4))
        self.state[0] = initial_state
        self.time_multiplier = delta_time * sim_steps
        self.remapped = np.zeros(time_steps)
        self.delta_time = delta_time
        self.sim_steps = sim_steps

    # The rest of the class remains unchanged

# Initialize
number_of_train_points = 2500
delta_time = 0.02
sim_steps = 50
oscillatory_cartpole_initial_state = np.array([0, 0, np.pi, 15])
number_of_points= 300

def zero_force(state):
    return 0

# Create model
random_model = LinearModel()

# Train model
random_model.train_model(train_initial_state, train_change_in_state)

# Create control and do simulation
oscillating_cartpole_pole_control = CartPoleControl(initial_state=
   oscillatory_cartpole_initial_state,
                                                    force=zero_force,
                                                    remap_angle=True,
                                                    time_steps=
                                                       number_of_points
                                                       ,
                                                    delta_time=
                                                       delta_time,
                                                    sim_steps=
                                                       sim_steps)
oscillating_cartpole_pole_control.do_simulation()

# Perform prediction
past_state, change = get_data_iteratively(
   oscillatory_cartpole_initial_state, random_model, number_of_points,
    remap_theta=True)

# Duplicate past state and add noise
repeated_past_state = np.repeat(past_state, 100, axis=0)
noise_mean = 0
noise_std = 0.001
noise = np.random.normal(noise_mean, noise_std, repeated_past_state.
   shape)
new_past_state = repeated_past_state + noise
new_change = random_model.predict_change(new_past_state, add_noise=
```

```
    True, noise_mean=noise_mean, noise_std=noise_std)

# Create and train new model
new_model = LinearModel()
new_model.train_model(new_past_state, new_change)

# Predict states
oscillating_cartpole_states = predict_states(random_model,
    oscillatory_cartpole_initial_state, number_of_points, remap_theta=
    True)
oscillating_cartpole_iterative_control = predict_states(
    oscillatory_cartpole_iterative_model,
    oscillatory_cartpole_initial_state, number_of_points, remap_theta=
    True)
new_control = predict_states(new_model,
    oscillatory_cartpole_initial_state, number_of_points, remap_theta=
    True)

# Predict changes
predicted_change = random_model.predict_change(past_state)
new_predicted_change = new_model.predict_change(past_state)

# Plot
fig, axs = plt.subplots(4, 1, figsize=(10, 8))
oscillating_cartpole_pole_control.plot_four_quantities(axs, label='
    Actual')
time = oscillating_cartpole_pole_control.t *
    oscillating_cartpole_pole_control.time_multiplier
for i in range(4):
    axs[i].plot(time, oscillating_cartpole_states[:, i], label='Model
        1', linewidth=5, linestyle='solid')
    handles, labels = axs[i].get_legend_handles_labels()
fig.legend(handles, labels, loc='lower center', bbox_to_anchor=(0.5,
    -0.05),  ncol=4, fontsize=20)
fig.suptitle("Linear Model for predicting", fontsize=20)
plt.tight_layout()
```

Task 2.1

```
import numpy as np
import matplotlib.pyplot as plt

from CartPole import CartPole

cp = CartPole()
# Generating data points for training the nonlinear model


def XY_data_generation(n, seeds):
    """
    Function for generating (X, Y) datapoints, used for model training
        .

    Target function Y is chosen to be the change in state after
        performAction()
```

```
    """
    X = []
    low = [-10,-10,-np.pi,-15]
    high = [10,10,np.pi,15]
    for i in range(4):
        np.random.seed(seeds[i])

        X.append(np.random.uniform(low[i], high[i], n))
    X = np.array(X).T

    Y = []

    for i in range(n):
        x = X[i]
        cp.setState(x)
        cp.performAction()
        Y.append(cp.getState() - x)

    return np.array(X), np.array(Y)

    # Functions involved in computing the nonlinear model

def kernel_compute(X, Xp, sig):
    """
    Returns a matrix K of dimension N x M, where
    N = no. of datapoints
    M = no. of basis functions
    """
    K = np.zeros((len(X), len(Xp)))
    for i in range(len(X)):
        for j in range(len(Xp)):
            exp_cl = (X[i][0] - Xp[j][0])**2 / (2*sig[0]**2)
            exp_cv = (X[i][1] - Xp[j][1])**2 / (2*sig[1]**2)
            exp_pa = np.sin((X[i][2] - Xp[j][2])/2)**2 / (2*sig[2]**2)
            exp_pv = (X[i][3] - Xp[j][3])**2 / (2*sig[3]**2)
            exponent = exp_cl + exp_cv + exp_pa + exp_pv
            K[i][j] = np.exp(-1*exponent)
    return K


def nonlinear_model(X, Y, M, sig, lmda):
    """
    Non-linear model: Does the matrix operations involving nonlinear
       model
    Returns the M sized vector alpha_M, used to compute the
       predictions
    """
    Xp_indices = np.random.choice(len(X), size=M, replace=False, p=
       None)
    Xp = X[Xp_indices]
    K_NM = kernel_compute(X, Xp, sig)
    K_MN = K_NM.T
    K_MM = kernel_compute(Xp, Xp, sig)
    a = K_MN @ K_NM + lmda * K_MM
```

```
    b = K_MN @ Y

    alpha_M = np.linalg.lstsq(a, b, rcond=None)[0]

    return alpha_M, Xp

# Generate data:

seeds = [0,1,2,3] # Seeds for each state variable
N_train = 1000    # We start with 1000 data points to train our
   nonlinear model

X_train, Y_train = XY_data_generation(N_train, seeds)


# Plot training data (can be used later to visualise distribution
# of M randomly selected datapoints)

fig, ax = plt.subplots(2,2,figsize=(10,10))
ax[0,0].scatter(X_train.T[0], Y_train.T[0])
ax[0,0].set_title('Cart location')
ax[0,1].scatter(X_train.T[1], Y_train.T[1])
ax[0,1].set_title('Cart velocity')
ax[1,0].scatter(X_train.T[2], Y_train.T[2])
ax[1,0].set_title('Pole angle')
ax[1,1].scatter(X_train.T[3], Y_train.T[3])
ax[1,1].set_title('Pole velocity')

fig.suptitle('Visualisation of Training data, Y vs X')
plt.show()

# Next we optimise the sigma values to get a good fit
# First we need to find the standard deviations of the X values, and
   use the standard
# deviations as a good initial estimate for the length scale
   hyperparameters:

std_list = []
for row in X_train.T:
    sq_mean = np.mean(np.square(row))
    mean = np.mean(row)
    variance = sq_mean - mean**2
    std_list.append(np.sqrt(variance))

alpha_M, Xp_train = nonlinear_model(X_train, Y_train, M, std_list,
   lmda)

K_NM = kernel_compute(X_train, Xp_train, std_list)

Y_pred = K_NM @ alpha_M

fig, ax = plt.subplots(2,2,figsize=(10,10))
ax[0,0].scatter(Y_train.T[0], Y_pred.T[0])
ax[0,0].set_title('Cart location')
```

```
ax[0,1].scatter(Y_train.T[1], Y_pred.T[1])
ax[0,1].set_title('Cart velocity')

ax[1,0].scatter(Y_train.T[2], Y_pred.T[2])
ax[1,0].set_title('Pole angle')

ax[1,1].scatter(Y_train.T[3], Y_pred.T[3])
ax[1,1].set_title('Pole velocity')

fig.suptitle('Initial scatterplot, showing fit of nonlinear model\
   nUsing standard deviations as length scale factors', fontsize=16)
plt.show()

# We start by trying a grid search, to narrow down the optimal value
   for
# the sigmas

# List of 5 scale factors to multiply each standard deviation by
scale_factors = np.geomspace(0.1, 10, 5)


sigma_range_0 = std_list[0] * scale_factors
sigma_range_1 = std_list[1] * scale_factors
sigma_range_2 = std_list[2] * scale_factors
sigma_range_3 = std_list[3] * scale_factors

# Manual grid search to find the optimal combination of sigmas
# May take an inordinate amount of time to run

opt_mse = np.inf
opt_sig = []
count = 1
for s0 in sigma_range_0:
    for s1 in sigma_range_1:
        for s2 in sigma_range_2:
            for s3 in sigma_range_3:
                sig = [s0, s1, s2, s3]
                mse = np.sum(find_mse(X_train, Y_train, M, sig, lmda))
                if mse < opt_mse:
                    opt_mse = mse
                    opt_sig = sig
                print(f"{count}/625", mse)
                count += 1

print(opt_mse, opt_sig)

# Plot to visualise the fit of the optimised model:

lmda_opt = 0.001 # Set lambda to a very small value to minimise noise
M = 500 # Set M to a large value
alpha_M, Xp_train = nonlinear_model(X_train, Y_train, M,
   sigma_opt_grid_search, lmda_opt)
```

```
K_NM = kernel_compute(X_train, Xp_train, sigma_opt_grid_search)

Y_pred = K_NM @ alpha_M

fig, ax = plt.subplots(2,2,figsize=(10,10))
ax[0,0].scatter(Y_train.T[0], Y_pred.T[0])
ax[0,0].set_title('Cart location')

ax[0,1].scatter(Y_train.T[1], Y_pred.T[1])
ax[0,1].set_title('Cart velocity')

ax[1,0].scatter(Y_train.T[2], Y_pred.T[2])
ax[1,0].set_title('Pole angle')

ax[1,1].scatter(Y_train.T[3], Y_pred.T[3])
ax[1,1].set_title('Pole velocity')

fig.suptitle('Scatterplot using the optimal length scale factors\
   nfound using grid search', fontsize=16)
plt.show()

# Now we investigate how the MSE correlates with M, for N = 1000
   datapoints

M_list = [10, 20, 40, 80, 160, 320, 640, 1000]
mse_list = []
for M in M_list:
    mse_list.append(find_mse(X_train, Y_train, M,
       sigma_opt_grid_search, lmda))

mse_list = np.array(mse_list).T

fig, ax = plt.subplots(4,1,figsize=(10,10))
ax[0].plot(M_list, mse_list[0])
ax[0].set_ylabel('MSE')
ax[0].set_title('Cart location')
ax[0].set_xlabel('M')

ax[1].plot(M_list, mse_list[1])
ax[1].set_ylabel('MSE')
ax[1].set_title('Cart location')
ax[1].set_xlabel('M')

ax[2].plot(M_list, mse_list[2])
ax[2].set_ylabel('MSE')
ax[2].set_title('Pole angle')
ax[2].set_xlabel('M')

ax[3].plot(M_list, mse_list[3])
ax[3].set_ylabel('MSE')
ax[3].set_title('Pole velocity')
ax[3].set_xlabel('M')

fig.suptitle('How the MSE varies with M for N = 1000', fontsize=16, y
```

```
    =1.05)
plt.tight_layout()
plt.show()

# Now we investigate how the MSE correlates with the number of
    datapoints, for M = 100 basis functions:

seeds = [0,1,2,3]
N_list = np.arange(200,5000,100)
mse_list = []
M = 200

for N in N_list:
    X, Y = XY_data_generation(N, seeds)
    mse = find_mse(X, Y, M, sigma_opt_grid_search, lmda)
    mse_list.append(mse)

mse_list = np.array(mse_list).T
fig, ax = plt.subplots(4,1,figsize=(10,10))
ax[0].plot(N_list, mse_list[0])
ax[0].set_ylabel('MSE')
ax[0].set_title('Cart location')
ax[0].set_xlabel('N')

ax[1].plot(N_list, mse_list[1])
ax[1].set_ylabel('MSE')
ax[1].set_title('Cart location')
ax[1].set_xlabel('N')

ax[2].plot(N_list, mse_list[2])
ax[2].set_ylabel('MSE')
ax[2].set_title('Pole angle')
ax[2].set_xlabel('N')

ax[3].plot(N_list, mse_list[3])
ax[3].set_ylabel('MSE')
ax[3].set_title('Pole velocity')
ax[3].set_xlabel('N')

fig.suptitle('How the MSE varies with N for M = 200', fontsize=16, y
    =1.05)
plt.tight_layout()
plt.show()

# Now we create 2D plots of the target function (change after
# one iteration) against the other state variables

import matplotlib.tri as tri

# List of 6 pairs of meshgrids:
# cl - cv
# cl - pa
# cl - pv
# cv - pa
```

```
# cv - pv
# pa - pv

np.random.seed(100) # To ensure consistent starting state
c_location = np.random.uniform(-10,10)
c_velocity = np.random.uniform(-10,10)
p_angle = np.random.uniform(-np.pi, np.pi)
p_velocity = np.random.uniform(-15,15)
state = [c_location, c_velocity, p_angle, p_velocity]


cl_range = np.linspace(-1,1,50)
cv_range = np.linspace(-10,10,50)
pa_range = np.linspace(-np.pi,np.pi,50)
pv_range = np.linspace(-15,15,50)


def two_d_plots(range_1, range_2):
    Y_cl_range = np.zeros((50,50))
    Y_cv_range = np.zeros((50,50))
    Y_pa_range = np.zeros((50,50))
    Y_pv_range = np.zeros((50,50))

    if np.array_equal(range_1, cl_range):
        x_label = 'Cart location'
        x_idx = 0
    elif np.array_equal(range_1, cv_range):
        x_label = 'Cart velocity'
        x_idx = 1
    elif np.array_equal(range_1, pa_range):
        x_label = 'Pole angle'
        x_idx = 2

    if np.array_equal(range_2, cv_range):
        y_label = 'Cart velocity'
        y_idx = 1
    elif np.array_equal(range_2, pa_range):
        y_label = 'Pole angle'
        y_idx = 2
    elif np.array_equal(range_2, pv_range):
        y_label = 'Pole velocity'
        y_idx = 3

    for i, x in enumerate(range_1):
        for j, y in enumerate(range_2):
            state[x_idx] = x
            state[y_idx] = y
            cp.setState(state)
            cp.performAction()

            new_state = cp.getState()
            Y_cl_range[i][j] = new_state[0] - state[0]
            Y_cv_range[i][j] = new_state[1] - state[1]
            Y_pa_range[i][j] = new_state[2] - state[2]
```

```
            Y_pv_range[i][j] = new_state[3] - state[3]

    mesh_1, mesh_2 = np.meshgrid(range_1, range_2)
    mesh_1 = mesh_1.flatten()
    mesh_2 = mesh_2.flatten()
    Y_cl_range = Y_cl_range.flatten()
    Y_cv_range = Y_cv_range.flatten()
    Y_pa_range = Y_pa_range.flatten()
    Y_pv_range = Y_pv_range.flatten()

    triang = tri.Triangulation(mesh_1, mesh_2)

    fig, ax = plt.subplots(1,4,figsize=(10,5))
    plot0 = ax[0].tricontourf(triang, Y_cl_range)
    ax[0].set_title(r'x')
    fig.colorbar(plot0, ax=ax[0])
    plot1 = ax[1].tricontourf(triang, Y_cv_range)
    ax[1].set_title(r'$\dot{x}$')
    fig.colorbar(plot1, ax=ax[1])
    plot2 = ax[2].tricontourf(triang, Y_pa_range)
    ax[2].set_title(r'$\theta$')
    fig.colorbar(plot2, ax=ax[2])
    plot3 = ax[3].tricontourf(triang, Y_pv_range)
    ax[3].set_title(r'$\dot{\theta}$')
    fig.colorbar(plot3, ax=ax[3])
    fig.text(0.5, 0.01, x_label, ha='center', va='center', fontsize
       =12)
    fig.text(0.01, 0.5, y_label, ha='center', va='center', rotation='
       vertical', fontsize=12)
    plt.tight_layout()
    plt.show()


two_d_plots(cl_range, cv_range)
two_d_plots(cl_range, pa_range)
two_d_plots(cl_range, pv_range)
two_d_plots(cv_range, pa_range)
two_d_plots(cv_range, pv_range)
two_d_plots(pa_range, pv_range)

np.random.seed(100) # To ensure consistent starting state
c_location = np.random.uniform(-10,10)
c_velocity = np.random.uniform(-10,10)
p_angle = np.random.uniform(-np.pi, np.pi)
p_velocity = np.random.uniform(-15,15)
state = [c_location, c_velocity, p_angle, p_velocity]


def two_d_plots_pred(range_1, range_2):
    Y_cl_range = np.zeros((50,50))
    Y_cv_range = np.zeros((50,50))
    Y_pa_range = np.zeros((50,50))
    Y_pv_range = np.zeros((50,50))
```

```
if np.array_equal(range_1, cl_range):
    x_label = 'Cart location'
    x_idx = 0
elif np.array_equal(range_1, cv_range):
    x_label = 'Cart velocity'
    x_idx = 1
elif np.array_equal(range_1, pa_range):
    x_label = 'Pole angle'
    x_idx = 2

if np.array_equal(range_2, cv_range):
    y_label = 'Cart velocity'
    y_idx = 1
elif np.array_equal(range_2, pa_range):
    y_label = 'Pole angle'
    y_idx = 2
elif np.array_equal(range_2, pv_range):
    y_label = 'Pole velocity'
    y_idx = 3

for i, x in enumerate(range_1):
    for j, y in enumerate(range_2):
        state[x_idx] = x
        state[y_idx] = y

        K_NM = kernel_compute(np.array([state]), Xp_train,
           sigma_opt_grid_search)
        state_pred = K_NM @ alpha_M  # alpha_M already declared
           globally after training
        Y_cl_range[i][j] = state_pred[0]
        Y_cv_range[i][j] = state_pred[1]
        Y_pa_range[i][j] = state_pred[2]
        Y_pv_range[i][j] = state_pred[3]

mesh_1, mesh_2 = np.meshgrid(range_1, range_2)
mesh_1 = mesh_1.flatten()
mesh_2 = mesh_2.flatten()
Y_cl_range = Y_cl_range.flatten()
Y_cv_range = Y_cv_range.flatten()
Y_pa_range = Y_pa_range.flatten()
Y_pv_range = Y_pv_range.flatten()

triang = tri.Triangulation(mesh_1, mesh_2)

# print(type(triang))

fig, ax = plt.subplots(1,4,figsize=(10,5))
plot0 = ax[0].tricontourf(triang, Y_cl_range)
ax[0].set_title(r'x')
fig.colorbar(plot0, ax=ax[0])
plot1 = ax[1].tricontourf(triang, Y_cv_range)
ax[1].set_title(r'$\dot{x}$')
fig.colorbar(plot1, ax=ax[1])
plot2 = ax[2].tricontourf(triang, Y_pa_range)
```

```
    ax[2].set_title(r'$\theta$')
    fig.colorbar(plot2, ax=ax[2])
    plot3 = ax[3].tricontourf(triang, Y_pv_range)
    ax[3].set_title(r'$\dot{\theta}$')
    fig.colorbar(plot3, ax=ax[3])
    fig.text(0.5, 0.01, x_label, ha='center', va='center', fontsize
       =12)
    fig.text(0.01, 0.5, y_label, ha='center', va='center', rotation='
       vertical', fontsize=12)
    plt.tight_layout()
    plt.show()


two_d_plots(cl_range, cv_range)
two_d_plots(cl_range, pa_range)
two_d_plots(cl_range, pv_range)
two_d_plots(cv_range, pa_range)
two_d_plots(cv_range, pv_range)
two_d_plots(pa_range, pv_range)

# Now we use the nonlinear model to plot the state variable
# for rollouts, for different initial conditions:

# 3 different initial states:
#initial_state_linear = [0, 5, np.pi, 0]
initial_state_simple_osc = [0, 0, np.pi, 0.5]
initial_state_complete_rot = [0, 0, np.pi, 15]


def rollouts(no_iters,state, remap):
    # Train the model:
    alpha_M, Xp_train = nonlinear_model(X_train, Y_train, M,
       sigma_opt_grid_search, lmda_opt)
    if state == initial_state_simple_osc:
        title = 'Rollout: simple oscillation'
    if state == initial_state_complete_rot:
        title = 'Rollout: complete rotation'
    cp.setState(state)
    state_list = []
    pred_list = []

    # for _ in range(no_iters):
    #     cp.performAction()
    #     if remap == True:
    #         cp.remap_angle()
    #     state = cp.getState()
    #     state_list.append(state)

    #     K_NM = kernel_compute(np.array([state]), Xp_train,
       sigma_opt_grid_search)
    #     y_pred = K_NM @ alpha_M
    #     pred_list.append(y_pred.squeeze() + state)

    for _ in range(no_iters):
```

```
            K_NM = kernel_compute(np.array([state]), Xp_train,
               sigma_opt_grid_search)
            y_pred = K_NM @ alpha_M
            pred_list.append(y_pred.squeeze() + state)
            cp.performAction()
            if remap == True:
                cp.remap_angle()
            state = cp.getState()
            state_list.append(state)

    times = np.arange(0,no_iters,1)
    pred_list = np.array(pred_list).T
    state_list = np.array(state_list).T

    fig, ax = plt.subplots(2,2,figsize=(10,10))
    actual, = ax[0,0].plot(times, state_list[0], label='Actual')
    predicted, = ax[0,0].plot(times, pred_list[0], label='Predicted -
       nonlinear model')
    ax[0,0].set_xlabel('Time step iterations')
    ax[0,0].set_ylabel('Cart location')
    ax[0,1].plot(times, state_list[1], label='Actual')
    ax[0,1].plot(times, pred_list[1], label='Predicted')
    ax[0,1].set_xlabel('Time step iterations')
    ax[0,1].set_ylabel('Cart velocity')
    ax[1,0].plot(times, state_list[2], label='Actual')
    ax[1,0].plot(times, pred_list[2], label='Predicted')
    ax[1,0].set_xlabel('Time step iterations')
    ax[1,0].set_ylabel('Pole angle')
    ax[1,1].plot(times, state_list[3], label='Actual')
    ax[1,1].plot(times, pred_list[3], label='Predicted')
    ax[1,1].set_xlabel('Time step iterations')
    ax[1,1].set_ylabel('Pole velocity')
    fig.suptitle(title,y=1.05, fontsize=20)
    fig.legend([actual, predicted],['Actual','Predicted'])
    plt.tight_layout()
    plt.show()


# rollouts(50, initial_state_linear, remap=False)
rollouts(50, initial_state_simple_osc, remap=False)
rollouts(50, initial_state_complete_rot, remap=False)
```

Task 2.2

```
    # Training the linear model:

def find_C(X, Y):
    return np.linalg.lstsq(X, Y, rcond=None)[0]

C_opt = find_C(X_train, Y_train)

Y_pred_list = np.matmul(X_train, C_opt)

# Visualising the fit of the linear model using scatterplots
```

```
dot_size = 5
fig, ax = plt.subplots(2,2,figsize=(10,10))
ax[0,0].scatter(Y_train.T[0], Y_pred_list.T[0])
ax[0,0].set_title('Cart location')

ax[0,1].scatter(Y_train.T[1], Y_pred_list.T[1])
ax[0,1].set_title('Cart velocity')

ax[1,0].scatter(Y_train.T[2], Y_pred_list.T[2])
ax[1,0].set_title('Pole angle')

ax[1,1].scatter(Y_train.T[3], Y_pred_list.T[3])
ax[1,1].set_title('Pole velocity')

fig.suptitle('Scatterplot for linear model with action', fontsize=16)
plt.show()

# 1D scan for the linear model


def sweep_linear(var, n, seed):
    """
    Sweep across the ranges of each state variable one by one
    """
    np.random.seed(seed)
    c_location = np.random.uniform(-10,10)
    c_velocity = np.random.uniform(-10,10)
    p_angle = np.random.uniform(-np.pi, np.pi)
    p_velocity = np.random.uniform(-15,15)
    action = np.random.uniform(-10,10)
    state = np.array([c_location, c_velocity, p_angle, p_velocity,
       action])

    #print("STATE:", state)

    X = []
    Y_list = []

    # sweep across cart position
    if var == 1:
        x_range = np.linspace(-1,1,n)
        for i in range(n):
            state[0] = x_range[i]
            X.append(state.copy())
            cp.setState(state)
            cp.performAction(state[4])
            new_state = np.array([cp.cart_location, cp.cart_velocity,
               cp.pole_angle, cp.pole_velocity])
            Y_list.append(new_state - state[:4])
        X = np.array(X)
        Y_pred_list = np.matmul(X, C_opt)
        sweep_var = 'cart position'

    # sweep across cart velocity
```

```
if var == 2:
    x_range = np.linspace(-10,10,n)
    for i in range(n):
        state[1] = x_range[i]
        X.append(state.copy())
        cp.setState(state)
        cp.performAction(state[4])
        new_state = np.array([cp.cart_location, cp.cart_velocity,
            cp.pole_angle, cp.pole_velocity])
        Y_list.append(new_state - state[:4])
    X = np.array(X)
    Y_pred_list = np.matmul(X, C_opt)
    sweep_var = 'cart velocity'

# sweep across pole angle
if var == 3:
    x_range = np.linspace(-np.pi,np.pi,n)
    for i in range(n):
        state[2] = x_range[i]
        X.append(state.copy())
        cp.setState(state)
        cp.performAction(state[4])
        new_state = np.array([cp.cart_location, cp.cart_velocity,
            cp.pole_angle, cp.pole_velocity])
        Y_list.append(new_state - state[:4])
    X = np.array(X)
    Y_pred_list = np.matmul(X, C_opt)
    sweep_var = 'pole angle'

# sweep across pole velocity
if var == 4:
    x_range = np.linspace(-15,15,n)
    for i in range(n):
        state[3] = x_range[i]
        X.append(state.copy())
        cp.setState(state)
        cp.performAction(state[4])
        new_state = np.array([cp.cart_location, cp.cart_velocity,
            cp.pole_angle, cp.pole_velocity])
        Y_list.append(new_state - state[:4])
    X = np.array(X)
    Y_pred_list = np.matmul(X, C_opt)
    sweep_var = 'pole velocity'

# sweep across action
if var == 5:
    x_range = np.linspace(-10, 10, n)
    for i in range(n):
        state[4] = x_range[i]
        X.append(state.copy())
        cp.setState(state)
        cp.performAction(state[4])
        new_state = np.array([cp.cart_location, cp.cart_velocity,
            cp.pole_angle, cp.pole_velocity])
```

```
            Y_list.append(new_state - state[:4])
        X = np.array(X)
        Y_pred_list = np.matmul(X, C_opt)
        sweep_var = 'action'

    Y_list = np.array(Y_list).T

    plt.figure(figsize=(10,2.5))
    plt.subplot(1,4,1)
    actual, = plt.plot(x_range, Y_list[0])
    predicted, = plt.plot(x_range, Y_pred_list.T[0])
    plt.title("Cart position", pad=20)


    plt.subplot(1,4,2)
    plt.plot(x_range, Y_list[1])
    plt.plot(x_range, Y_pred_list.T[1])
    plt.title("Cart velocity",pad=20)

    plt.subplot(1,4,3)
    plt.plot(x_range, Y_list[2])
    plt.plot(x_range, Y_pred_list.T[2])
    plt.title("Pole angle", pad=20)

    plt.subplot(1,4,4)
    plt.plot(x_range, Y_list[3])
    plt.plot(x_range, Y_pred_list.T[3])
    plt.title("Pole velocity", pad=20)

    plt.suptitle(f'1D Scan of {sweep_var}')

    plt.tight_layout()
    plt.figlegend([actual, predicted], ['Actual','Predicted'])
    plt.show()

seed = 42
no_iters = 50
sweep_linear(1, no_iters, seed)
sweep_linear(2, no_iters, seed)
sweep_linear(3, no_iters, seed)
sweep_linear(4, no_iters, seed)
sweep_linear(5, no_iters, seed)

# Plotting the rollouts for the linear model

def rollouts_linear(no_iters, state, remap):
    if state == initial_state_simple_osc_action:
        title = 'Rollout: simple oscillation with force = 1 N'
    if state == initial_state_complete_rot_action:
        title = 'Rollout: complete rotation with force = 1 N'

    # Train the model:
    alpha_M, Xp_train = nonlinear_model(X_train, Y_train, M,
       sigma_opt_grid_search, lmda_opt)
```

```
    cp.setState(state)
    state_list = []
    pred_list = []

    for _ in range(no_iters):
        y_pred = np.matmul(state, C_opt) + state[:4]
        pred_list.append(y_pred.squeeze())
        cp.performAction(state[4])
        if remap == True:
            cp.remap_angle()
        state = np.append(cp.getState(), state[4])
        state_list.append(state)



    times = np.arange(0,no_iters,1)
    pred_list = np.array(pred_list).T
    state_list = np.array(state_list).T

    fig, ax = plt.subplots(2,2,figsize=(10,10))
    actual, = ax[0,0].plot(times, state_list[0], label='Actual')
    predicted, = ax[0,0].plot(times, pred_list[0], label='Predicted -
       linear model')
    ax[0,0].set_xlabel('Time step iterations')
    ax[0,0].set_ylabel('Cart location')
    ax[0,1].plot(times, state_list[1], label='Actual')
    ax[0,1].plot(times, pred_list[1], label='Predicted')
    ax[0,1].set_xlabel('Time step iterations')
    ax[0,1].set_ylabel('Cart velocity')
    ax[1,0].plot(times, state_list[2], label='Actual')
    ax[1,0].plot(times, pred_list[2], label='Predicted')
    ax[1,0].set_xlabel('Time step iterations')
    ax[1,0].set_ylabel('Pole angle')
    ax[1,1].plot(times, state_list[3], label='Actual')
    ax[1,1].plot(times, pred_list[3], label='Predicted')
    ax[1,1].set_xlabel('Time step iterations')
    ax[1,1].set_ylabel('Pole velocity')
    fig.suptitle(title,y=1.05, fontsize=20)
    fig.legend([actual, predicted],['Actual','Predicted'])
    plt.tight_layout()
    plt.show()



#rollouts_linear(50, initial_state_linear_action, remap=False)
rollouts_linear(50, initial_state_simple_osc_action, remap=False)
rollouts_linear(50, initial_state_complete_rot_action, remap=False)

# Training the new nonlinear model


lmda_opt = 0.001 # Set lambda to a very small value to minimise noise
M = 500 # Set M to a large value
```

```
alpha_M, Xp_train = nonlinear_model(X_train, Y_train, M,
   sigma_opt_grid_search, lmda_opt)

K_NM = kernel_compute(X_train, Xp_train, sigma_opt_grid_search)

Y_pred = K_NM @ alpha_M

fig, ax = plt.subplots(2,2,figsize=(10,10))
ax[0,0].scatter(Y_train.T[0], Y_pred.T[0])
ax[0,0].set_title('Cart location')

ax[0,1].scatter(Y_train.T[1], Y_pred.T[1])
ax[0,1].set_title('Cart velocity')

ax[1,0].scatter(Y_train.T[2], Y_pred.T[2])
ax[1,0].set_title('Pole angle')

ax[1,1].scatter(Y_train.T[3], Y_pred.T[3])
ax[1,1].set_title('Pole velocity')

fig.suptitle('Scatterplot for nonlinear model with action', fontsize
   =16)
plt.show()

# 1D Scan for the nonlinear model:

def sweep_nonlinear(var, n, seed):
    np.random.seed(seed)
    c_location = np.random.uniform(-10,10)
    c_velocity = np.random.uniform(-10,10)
    p_angle = np.random.uniform(-np.pi, np.pi)
    p_velocity = np.random.uniform(-15,15)
    action = np.random.uniform(-5,5)
    state = np.array([c_location, c_velocity, p_angle, p_velocity,
       action])

    ##print("STATE:", state)

    X = []
    Y_list = []

    if var == 1:
        x_range = np.linspace(-1,1,n)
        sweep_var = 'cart location'
    elif var == 2:
        x_range = np.linspace(-10,10,n)
        sweep_var = 'cart velocity'
    elif var == 3:
        x_range = np.linspace(-np.pi,np.pi,n)
        sweep_var = 'pole angle'
    elif var == 4:
        x_range = np.linspace(-15,15,n)
        sweep_var = 'pole velocity'
    elif var == 5:
```

```
            x_range = np.linspace(-5, 5, n)
            sweep_var = 'action'

        for i in range(n):
            state[var-1] = x_range[i]
            X.append(state.copy())
            cp.setState(state[:4])
            cp.performAction(state[4])
            new_state = np.array([cp.cart_location, cp.cart_velocity, cp.
                pole_angle, cp.pole_velocity])
            Y_list.append(new_state - state[:4])

        X = np.array(X)
        K_NM = kernel_compute(X, Xp_train, sigma_opt_grid_search)
        Y_pred_list = K_NM @ alpha_M

        Y_list = np.array(Y_list).T

        plt.figure(figsize=(10,2.5))
        plt.subplot(1,4,1)
        actual, = plt.plot(x_range, Y_list[0])
        predicted, = plt.plot(x_range, Y_pred_list.T[0])
        plt.title("Cart position", pad=20)


        plt.subplot(1,4,2)
        plt.plot(x_range, Y_list[1])
        plt.plot(x_range, Y_pred_list.T[1])
        plt.title("Cart velocity",pad=20)

        plt.subplot(1,4,3)
        plt.plot(x_range, Y_list[2])
        plt.plot(x_range, Y_pred_list.T[2])
        plt.title("Pole angle", pad=20)

        plt.subplot(1,4,4)
        plt.plot(x_range, Y_list[3])
        plt.plot(x_range, Y_pred_list.T[3])
        plt.title("Pole velocity", pad=20)

        plt.suptitle(f'1D Scan of {sweep_var}')

        plt.tight_layout()
        plt.figlegend([actual, predicted], ['Actual','Predicted'])
        plt.show()

seed = 42
no_iters = 200
sweep_nonlinear(1, no_iters, seed)
sweep_nonlinear(2, no_iters, seed)
sweep_nonlinear(3, no_iters, seed)
sweep_nonlinear(4, no_iters, seed)
sweep_nonlinear(5, no_iters, seed)
```

```
def rollouts_nonlinear(no_iters,state, remap):
    # Train the model:
    alpha_M, Xp_train = nonlinear_model(X_train, Y_train, M,
       sigma_opt_grid_search, lmda_opt)
    if state == initial_state_simple_osc_action:
        title = 'Rollout: simple oscillation with force = 1 N'
    if state == initial_state_complete_rot_action:
        title = 'Rollout: complete rotation with force = 1 N'
    cp.setState(state)
    state_list = []
    pred_list = []

    for _ in range(no_iters):

        K_NM = kernel_compute(np.array([state]), Xp_train,
           sigma_opt_grid_search)
        y_pred = K_NM @ alpha_M + state[:4]
        pred_list.append(y_pred.squeeze())

        cp.performAction(state[4])
        if remap == True:
            cp.remap_angle()
        state = np.append(cp.getState(), state[4])
        state_list.append(state)



    times = np.arange(0,no_iters,1)
    pred_list = np.array(pred_list).T
    state_list = np.array(state_list).T

    fig, ax = plt.subplots(2,2,figsize=(10,10))
    actual, = ax[0,0].plot(times, state_list[0], label='Actual')
    predicted, = ax[0,0].plot(times, pred_list[0], label='Predicted -
       nonlinear model')
    ax[0,0].set_xlabel('Time step iterations')
    ax[0,0].set_ylabel('Cart location')
    ax[0,1].plot(times, state_list[1], label='Actual')
    ax[0,1].plot(times, pred_list[1], label='Predicted')
    ax[0,1].set_xlabel('Time step iterations')
    ax[0,1].set_ylabel('Cart velocity')
    ax[1,0].plot(times, state_list[2], label='Actual')
    ax[1,0].plot(times, pred_list[2], label='Predicted')
    ax[1,0].set_xlabel('Time step iterations')
    ax[1,0].set_ylabel('Pole angle')
    ax[1,1].plot(times, state_list[3], label='Actual')
    ax[1,1].plot(times, pred_list[3], label='Predicted')
    ax[1,1].set_xlabel('Time step iterations')
    ax[1,1].set_ylabel('Pole velocity')
    fig.suptitle(title,y=1.05, fontsize=20)
    fig.legend([actual, predicted],['Actual','Predicted'])
    plt.tight_layout()
    plt.show()
```

```
# rollouts_nonlinear(50, initial_state_linear_action, remap=False)
rollouts_nonlinear(50, initial_state_simple_osc_action, remap=False)
rollouts_nonlinear(50, initial_state_complete_rot_action, remap=False)
```

Task 2.3

```
def rollouts_nonlinear(no_iters,initial_state, remap):
    # Train the model:
    alpha_M, Xp_train = nonlinear_model(X_train, Y_train, M,
       sigma_opt_grid_search, lmda_opt)
    if initial_state == initial_state_simple_osc_action:
        title = 'Rollout: simple oscillation with force = 1 N'
    if initial_state == initial_state_complete_rot_action:
        title = 'Rollout: complete rotation with force = 1 N'
    cp.setState(initial_state)
    state_list = []
    pred_list = []
    residuals = []
    state = initial_state

    for i in range(no_iters):
        K_NM = kernel_compute(np.array([state]), Xp_train,
           sigma_opt_grid_search)
        y_pred = K_NM @ alpha_M + state[:4]
        pred_list.append(y_pred.squeeze())

        residuals.append(y_pred.squeeze() - state[:4])

        cp.performAction(state[4])
        if remap == True:
            cp.remap_angle()
        state = np.append(cp.getState(), state[4])
        state_list.append(state)

    times = np.arange(0,no_iters,1)
    pred_list = np.array(pred_list).T
    state_list = np.array(state_list).T

    fig, ax = plt.subplots(2,2,figsize=(10,10))
    actual, = ax[0,0].plot(times, state_list[0], label='Actual')
    predicted, = ax[0,0].plot(times, pred_list[0], label='Predicted -
       nonlinear model')
    ax[0,0].set_xlabel('Time step iterations')
    ax[0,0].set_ylabel('Cart location')
    ax[0,1].plot(times, state_list[1], label='Actual')
    ax[0,1].plot(times, pred_list[1], label='Predicted')
    ax[0,1].set_xlabel('Time step iterations')
    ax[0,1].set_ylabel('Cart velocity')
    ax[1,0].plot(times, state_list[2], label='Actual')
    ax[1,0].plot(times, pred_list[2], label='Predicted')
    ax[1,0].set_xlabel('Time step iterations')
    ax[1,0].set_ylabel('Pole angle')
```

```
    ax[1,1].plot(times, state_list[3], label='Actual')
    ax[1,1].plot(times, pred_list[3], label='Predicted')
    ax[1,1].set_xlabel('Time step iterations')
    ax[1,1].set_ylabel('Pole velocity')
    fig.suptitle(title,y=1.05, fontsize=20)
    fig.legend([actual, predicted],['Actual','Predicted'])
    plt.tight_layout()
    plt.show()

    # fig, ax = plt.subplots(1,4,figsize=(10,2.5))
    # ax[0].plot(times, residuals[0])
    # ax[1].plot(times, residuals[1])
    # ax[2].plot(times, residuals[2])
    # ax[3].plot(times, residuals[3])
    # plt.show()

#rollouts_nonlinear(50, initial_state_linear_action, remap=False)
rollouts_nonlinear(50, initial_state_simple_osc_action, remap=False)
rollouts_nonlinear(50, initial_state_complete_rot_action, remap=False)

# Visualise the loss function via 1D scans

def visualise_losses(var, n, p, initial_state, seed):
    state = np.pad(initial_state, (0,1))

    # print("STATE:", state)

    accum_losses_list = []

    p_range = np.linspace(-10,10,n)

    for i in range(n):
        p[var-1] = p_range[i]
        accum_losses_list.append(accumulated_loss(p))

    accum_losses_list = np.array(accum_losses_list).T

    plt.figure(figsize=(10,2.5))
    plt.plot(p_range, accum_losses_list)
    plt.ylabel('Accumulated loss')
    plt.title(f"Sweep of p[{var-1}]")

    plt.show()


seed = 42
no_iters = 50
visualise_losses(1, no_iters, p_opt, initial_state, seed)
visualise_losses(2, no_iters, p_opt, initial_state, seed)
visualise_losses(3, no_iters, p_opt, initial_state, seed)
visualise_losses(4, no_iters, p_opt, initial_state, seed)
# Rollout using control policy:

def linear_policy_rollout(n ,initial_state, p):
```

```
    state_list = []
    cp.setState(initial_state)
    times = np.arange(0,n,1)
    for i in range(n):
        state_list.append(cp.getState())
        cp.performAction(np.dot(p, cp.getState()))

    state_list = np.array(state_list).T

    fig, ax = plt.subplots(2,2,figsize=(10,10))
    ax[0,0].plot(times, state_list[0])
    ax[0,0].set_xlabel('Time iterations')
    ax[0,0].set_ylabel('Cart position')
    ax[0,1].plot(times, state_list[1])
    ax[0,1].set_xlabel('Time iterations')
    ax[0,1].set_ylabel('Cart velocity')
    ax[1,0].plot(times, state_list[2])
    ax[1,0].set_xlabel('Time iterations')
    ax[1,0].set_ylabel('Pole angle')
    ax[1,1].plot(times, state_list[3])
    ax[1,1].set_xlabel('Time iterations')
    ax[1,1].set_ylabel('Pole velocity')
    fig.suptitle('Convergence of state variables under the linear
       policy',y=1.03,fontsize=20)
    plt.tight_layout()
    plt.show()

initial_state = [0, 0, 0.1, 0]

no_iters = 50
print(p_opt)
linear_policy_rollout(no_iters, initial_state, p_opt)
```

Task 2.4

```
# Time horizon within which my model is still accurate: 8 steps (from
   above comparision)

# Set the initial state to something close to the desired state:
initial_state = [0, 0, 0.1, 0]
cp.setState(initial_state)
M = 500
no_iters = 8
alpha_M, Xp_train = nonlinear_model(X_train, Y_train, M,
   sigma_opt_grid_search, lmda_opt)


def cp_loss(state):
    sig = 0.5
    return 1 - np.exp(-np.dot(state,state.T)/(2.0*sig**2))


def accumulated_loss_model(p):
    cp.setState(initial_state)
    loss = 0
```

```
    for i in range(no_iters):
        cp.remap_angle()
        cp.performAction(np.dot(p, cp.getState()))
        K_NM = kernel_compute(np.array([cp.getState()]), Xp_train,
           sigma_opt_grid_search)
        pred_state = K_NM @ alpha_M + cp.getState()
        loss += cp_loss(pred_state)
    return loss


p_opt_model = None
min_loss_model = np.inf
for _ in range(30):
    p_initial = 10*(np.random.rand(4) - 0.5)
    result = minimize(accumulated_loss_model, p_initial, method="
       nelder-mead")
    loss = accumulated_loss(result['x'])
    if loss < min_loss_model:
        min_loss_model = loss
        p_opt_model = result['x']
    print(p_opt_model, min_loss_model)

    # Rollout using control policy:

def linear_policy_rollout_model(n ,initial_state, p):
    state_list = []
    cp.setState(initial_state)
    times = np.arange(0,n,1)
    for i in range(n):
        state_list.append(cp.getState())
        cp.performAction(np.dot(p, cp.getState()))

    state_list = np.array(state_list).T

    fig, ax = plt.subplots(2,2,figsize=(10,10))
    ax[0,0].plot(times, state_list[0])
    ax[0,0].set_xlabel('Time iterations')
    ax[0,0].set_ylabel('Cart position')
    ax[0,1].plot(times, state_list[1])
    ax[0,1].set_xlabel('Time iterations')
    ax[0,1].set_ylabel('Cart velocity')
    ax[1,0].plot(times, state_list[2])
    ax[1,0].set_xlabel('Time iterations')
    ax[1,0].set_ylabel('Pole angle')
    ax[1,1].plot(times, state_list[3])
    ax[1,1].set_xlabel('Time iterations')
    ax[1,1].set_ylabel('Pole velocity')
    fig.suptitle('Convergence of state variables under the linear
       policy\n optimised using nonlinear model',y=1.03,fontsize=20)
    plt.tight_layout()
    plt.show()


initial_state = [0, 0, 0.1, 0]
```

```
no_iters = 100
linear_policy_rollout_model(no_iters, initial_state, p_opt)
```

Task 3.1

```
# Generating data points for training the nonlinear model
def XY_data_generation(n, seeds):
    """
    Function for generating (X, Y) datapoints, used for model training
       .

    Target function Y is chosen to be the change in state after
       performAction()
    """
    X = []
    low = [-10,-10,-np.pi,-15]
    high = [10,10,np.pi,15]
    for i in range(4):
        np.random.seed(seeds[i])
        X.append(np.random.uniform(low[i], high[i], n))
    X = np.array(X).T

    Y = []

    for i in range(n):
        x = X[i]
        cp.setState(x)
        cp.performAction()
        Y.append(cp.getState() - x)

    return np.array(X), np.array(Y)

# Functions involved in computing the nonlinear model

def kernel_compute(X, Xp, sig):
    """
    Returns a matrix K of dimension N x M, where
    N = no. of datapoints
    M = no. of basis functions
    """
    K = np.zeros((len(X), len(Xp)))
    for i in range(len(X)):
        for j in range(len(Xp)):
            exp_cl = (X[i][0] - Xp[j][0])**2 / (2*sig[0]**2)
            exp_cv = (X[i][1] - Xp[j][1])**2 / (2*sig[1]**2)
            exp_pa = np.sin((X[i][2] - Xp[j][2])/2)**2 / (2*sig[2]**2)
            exp_pv = (X[i][3] - Xp[j][3])**2 / (2*sig[3]**2)
            exponent = exp_cl + exp_cv + exp_pa + exp_pv
            K[i][j] = np.exp(-1*exponent)
    return K


def nonlinear_model(X, Y, M, sig, lmda):
    """
```

```
    Non-linear model: Does the matrix operations involving nonlinear
       model
    Returns the M sized vector alpha_M, used to compute the
       predictions
    """
    Xp_indices = np.random.choice(len(X), size=M, replace=False, p=
       None)
    Xp = X[Xp_indices]
    K_NM = kernel_compute(X, Xp, sig)
    K_MN = K_NM.T
    K_MM = kernel_compute(Xp, Xp, sig)
    a = K_MN @ K_NM + lmda * K_MM
    b = K_MN @ Y

    alpha_M = np.linalg.lstsq(a, b, rcond=None)[0]

    return alpha_M, Xp


# Parameters for training and using the nonlinear model
N_train = 1000
M = 500
sigma_opt_grid_search = [58.11976831212202, 57.671269211037085,
   0.563954257408644, 2.6938726555306434]
lmda_opt = 0.01


# Initial conditions:
initial_state_simple_osc = [0, 0, np.pi, 0.5]
initial_state_complete_rot = [0, 0, np.pi, 15]

# Initial conditions with an action of 1 applied
initial_state_simple_osc_action = [0, 0, np.pi, 0.5, 1]
initial_state_complete_rot_action = [0, 0, np.pi, 15, 1]


# Generate Training Data
# We set the range of action force to go from -10 to 10:
def XY_data_generation_action(n, seeds):
    """
    Function for generating (X, Y) datapoints, used for model training
       .
    Target function Y is chosen to be the change in state after
       performAction()
    """
    X = []
    # Set the lower and upper bounds for the various state variables
    low = [-10,-10,-np.pi,-15,-5]
    high = [10,10,np.pi,15,5]
    for i in range(5):
        np.random.seed(seeds[i])
        X.append(np.random.uniform(low[i], high[i], n))
    X = np.array(X).T
```

```
    Y = []

    for i in range(n):
        x = X[i]
        cp.setState(x)
        cp.performAction(x[4]) # Including the force
        Y.append(cp.getState() - x[:4])

    return np.array(X), np.array(Y)


def noisy_XY_data_generation_action(n, seeds,std):
    """
    Function for generating (X, Y) datapoints with scaled noise added
       to Y
    """
    X = []
    # Set the lower and upper bounds for the various state variables
    low = [-10,-10,-np.pi,-15,-5]
    high = [10,10,np.pi,15,5]
    for i in range(5):
        np.random.seed(seeds[i])
        X.append(np.random.uniform(low[i], high[i], n))
    X = np.array(X).T

    Y = []

    for i in range(n):
        x = X[i]
        noise = np.random.normal(0,std,4)
        noise[0] *= 10
        noise[1] *= 10
        noise[2] *= np.pi
        noise[3] *= 15
        cp.setState(x)
        cp.performAction(x[4]) # Including the force
        Y.append(cp.getState() - x[:4] + noise)

    return np.array(X), np.array(Y)

seeds = [0,1,2,3,4]
X_train, Y_train = XY_data_generation_action(N_train, seeds)

# Start with a standard deviation of 0.1
noisy_X_train, noisy_Y_train = noisy_XY_data_generation_action(N_train
   , seeds, 0.1)

# Train linear model:

C = np.linalg.lstsq(X_train, Y_train, rcond=None)[0]

Y_pred_linear = np.matmul(X_train, C)

fig, ax = plt.subplots(2,2)
```

```
dot_size = 5
ax[0,0].scatter(Y_train.T[0], Y_pred_linear.T[0], s=dot_size)
ax[0,0].set_title('Cart location')
ax[0,1].scatter(Y_train.T[1], Y_pred_linear.T[1], s=dot_size)
ax[0,1].set_title('Cart velocity')
ax[1,0].scatter(Y_train.T[2], Y_pred_linear.T[2], s=dot_size)
ax[1,0].set_title('Pole angle')
ax[1,1].scatter(Y_train.T[3], Y_pred_linear.T[3], s=dot_size)
ax[1,1].set_title('Pole velocity')
fig.suptitle('Scatterplot for linear model - no noise', fontsize=16)
plt.tight_layout()
plt.show()


noisy_C = np.linalg.lstsq(noisy_X_train, noisy_Y_train, rcond=None)[0]

noisy_Y_pred_linear = np.matmul(noisy_X_train, noisy_C)

fig, ax = plt.subplots(2,2)
dot_size = 5
ax[0,0].scatter(noisy_Y_train.T[0], noisy_Y_pred_linear.T[0], s=
    dot_size)
ax[0,0].set_title('Cart location')
ax[0,1].scatter(noisy_Y_train.T[1], noisy_Y_pred_linear.T[1], s=
    dot_size)
ax[0,1].set_title('Cart velocity')
ax[1,0].scatter(noisy_Y_train.T[2], noisy_Y_pred_linear.T[2], s=
    dot_size)
ax[1,0].set_title('Pole angle')
ax[1,1].scatter(noisy_Y_train.T[3], noisy_Y_pred_linear.T[3], s=
    dot_size)
ax[1,1].set_title('Pole velocity')
fig.suptitle('Scatterplot for linear model - with noise (standard
    deviation = 0.1)', fontsize=16)
plt.tight_layout()
plt.show()

alpha_M, Xp_train = nonlinear_model(X_train, Y_train, M,
    sigma_opt_grid_search, lmda_opt)

K_NM = kernel_compute(X_train, Xp_train, sigma_opt_grid_search)

Y_pred = K_NM @ alpha_M

fig, ax = plt.subplots(2,2,figsize=(10,10))
ax[0,0].scatter(Y_train.T[0], Y_pred.T[0])
ax[0,0].set_title('Cart location')
ax[0,1].scatter(Y_train.T[1], Y_pred.T[1])
ax[0,1].set_title('Cart velocity')
ax[1,0].scatter(Y_train.T[2], Y_pred.T[2])
ax[1,0].set_title('Pole angle')
ax[1,1].scatter(Y_train.T[3], Y_pred.T[3])
ax[1,1].set_title('Pole velocity')
fig.suptitle('Scatterplot for nonlinear model - no noise', fontsize
```

```
    =16)
plt.tight_layout()
plt.show()

noisy_alpha_M, noisy_Xp_train = nonlinear_model(noisy_X_train,
    noisy_Y_train, M, sigma_opt_grid_search, lmda_opt)

noisy_K_NM = kernel_compute(noisy_X_train, noisy_Xp_train,
    sigma_opt_grid_search)

noisy_Y_pred = noisy_K_NM @ noisy_alpha_M

fig, ax = plt.subplots(2,2,figsize=(10,10))
ax[0,0].scatter(noisy_Y_train.T[0], noisy_Y_pred.T[0])
ax[0,0].set_title('Cart location')
ax[0,1].scatter(noisy_Y_train.T[1], noisy_Y_pred.T[1])
ax[0,1].set_title('Cart velocity')
ax[1,0].scatter(noisy_Y_train.T[2], noisy_Y_pred.T[2])
ax[1,0].set_title('Pole angle')
ax[1,1].scatter(noisy_Y_train.T[3], noisy_Y_pred.T[3])
ax[1,1].set_title('Pole velocity')
fig.suptitle('Scatterplot for nonlinear model - with noise (standard
    deviation = 0.1)', fontsize=16)
plt.tight_layout()

plt.show()

# Characterise the degradation in prediction accuracy:

lmda = 0.0001


def nonlinear_find_mse(X, Y, M, sig, lmda):
    alpha_M, Xp = nonlinear_model(X, Y, M, sig, lmda)
    K_NM = kernel_compute(X, Xp, sig)
    Y_pred = K_NM @ alpha_M
    mse = np.square(Y_pred-Y).mean(axis=0)
    return mse


def find_mse(pred, acc):
    mse = np.square(pred - acc).mean(axis=0)
    return mse


print('For linear model:')
print("MSE of noise-free data:", find_mse(Y_pred_linear, Y_train))
print("MSE of noisy data:", find_mse(noisy_Y_pred_linear,
    noisy_Y_train))

print('For nonlinear model:')
print("MSE of noise-free data:",nonlinear_find_mse(X_train, Y_train, M
    , sigma_opt_grid_search, lmda))
print("MSE of noisy data:", nonlinear_find_mse(noisy_X_train,
```

```
    noisy_Y_train, M, sigma_opt_grid_search, lmda))

def accumulated_loss_noise_observed(p):
    cp.setState(initial_state)
    loss = 0
    for i in range(no_iters):
        noise = np.random.normal(0,std,4)
        noise[0] *= 10
        noise[1] *= 10
        noise[2] *= np.pi
        noise[3] *= 15
        cp.remap_angle()
        cp.performAction(np.dot(p, cp.getState()))
        no_noise_state = cp.getState()
        cp.setState(no_noise_state+noise)
        loss += cp.loss()
        cp.setState(no_noise_state)
    return loss


p_opt_noise = None
min_loss_noise = np.inf
for _ in range(100):
    p_initial = 10*(np.random.rand(4) - 0.5)
    result = minimize(accumulated_loss_noise_observed, p_initial,
       method="bfgs")
    loss = accumulated_loss_noise_observed(result['x'])
    if loss < min_loss_noise:
        min_loss_noise = loss
        p_opt_noise = result['x']
    print(p_opt_noise, min_loss_noise)

# Rollout using control policy:

def noisy_linear_policy_rollout(n ,initial_state, p):
    state_list = []
    cp.setState(initial_state)
    times = np.arange(0,n,1)
    for i in range(n):
        cp.remap_angle()
        state_list.append(cp.getState())
        cp.performAction(np.dot(p, cp.getState()))

    state_list = np.array(state_list).T

    fig, ax = plt.subplots(2,2,figsize=(10,10))
    ax[0,0].plot(times, state_list[0])
    ax[0,0].set_xlabel('Time iterations')
    ax[0,0].set_ylabel('Cart position')
    ax[0,1].plot(times, state_list[1])
    ax[0,1].set_xlabel('Time iterations')
    ax[0,1].set_ylabel('Cart velocity')
    ax[1,0].plot(times, state_list[2])
    ax[1,0].set_xlabel('Time iterations')
```

```
    ax[1,0].set_ylabel('Pole angle')
    ax[1,1].plot(times, state_list[3])
    ax[1,1].set_xlabel('Time iterations')
    ax[1,1].set_ylabel('Pole velocity')
    fig.suptitle('Convergence of state variables under the linear
       policy optimised using noisy training data',y=1.03,fontsize=20)
    plt.tight_layout()
    plt.show()


initial_state = [0, 0, 0.1, 0]

no_iters = 100
noisy_linear_policy_rollout(no_iters, initial_state, p_opt_noise)
```

Task 3.2

```
# Now introduce noise to the actual system

std = 0.01
no_iters = 10

def accumulated_loss_noise(p):
    cp.setState(initial_state)
    loss = 0
    for i in range(no_iters):
        noise = np.random.normal(0,std,4)
        noise[0] *= 10
        noise[1] *= 10
        noise[2] *= np.pi
        noise[3] *= 15
        cp.remap_angle()
        cp.performAction(1*np.dot(p, cp.getState()))
        cp.setState(cp.getState()+noise)
        loss += cp.loss()
    return loss


p_opt_noise = None
min_loss_noise = np.inf
for _ in range(100):
    p_initial = 10*(np.random.rand(4) - 0.5)
    result = minimize(accumulated_loss_noise, p_initial, method="bfgs
       ")
    loss = accumulated_loss_noise(result['x'])
    if loss < min_loss_noise:
        min_loss_noise = loss
        p_opt_noise = result['x']
    print(p_opt_noise, min_loss_noise)




def noisy_linear_policy_rollout(n ,initial_state, p):
    state_list = []
```

```
    cp.setState(initial_state)
    times = np.arange(0,n,1)
    for i in range(n):
        # noise = np.random.normal(0,std,4)
        # noise[0] *= 10
        # noise[1] *= 10
        # noise[2] *= np.pi
        # noise[3] *= 15
        state_list.append(cp.getState())
        cp.performAction(np.dot(p, cp.getState()))
        #cp.setState(cp.getState() + noise)

    state_list = np.array(state_list).T

    fig, ax = plt.subplots(2,2,figsize=(10,10))
    ax[0,0].plot(times, state_list[0])
    ax[0,0].set_xlabel('Time iterations')
    ax[0,0].set_ylabel('Cart position')
    ax[0,1].plot(times, state_list[1])
    ax[0,1].set_xlabel('Time iterations')
    ax[0,1].set_ylabel('Cart velocity')
    ax[1,0].plot(times, state_list[2])
    ax[1,0].set_xlabel('Time iterations')
    ax[1,0].set_ylabel('Pole angle')
    ax[1,1].plot(times, state_list[3])
    ax[1,1].set_xlabel('Time iterations')
    ax[1,1].set_ylabel('Pole velocity')
    fig.suptitle('Convergence of state variables under the linear
       policy with added noise',y=1.03,fontsize=20)
    plt.tight_layout()
    plt.show()


initial_state = [0, 0, 0.1, 0]

no_iters = 100
noisy_linear_policy_rollout(no_iters, initial_state, p_opt_noise)
```

Task 4

```
import numpy as np
import matplotlib.pyplot as plt
from cartpole import CartPole

def compute_loss(args):
    num_basis_funcs = 5
    weights, states, matrix_W = [], [], np.zeros((4, 4))

    for idx, arg in enumerate(args):
        if idx < num_basis_funcs: # Weights
            weights.append(arg)
        elif idx < 2 * num_basis_funcs: # States
            if (idx - num_basis_funcs + 1) % 4 == 0:
                states.append(np.array(args[idx-3:idx+1]))
        else: # Matrix
```

```
            m, n = divmod(idx - 2 * num_basis_funcs, 4)
            matrix_W[m, n] = arg

    cp = CartPole()
    cp.setState([0, 0, np.pi, -10])

    loss = cp.loss()
    for _ in range(9): # Iteration over policy action
        curr_state = cp.getState()
        action = policy(curr_state, np.array(weights), np.array(states
            ), matrix_W)
        cp.performAction(action)
        loss += cp.loss()

    return loss

def policy(X, weights, states, matrix_W):
    policy_val = 0
    for i in range(len(weights)):
        vec = X - states[i]
        policy_val += weights[i] * np.exp(-0.5 * vec.T @ matrix_W @
            vec)

    return policy_val

def find_optimal_coarse(num_tries, range_args=5):
    tries = np.random.rand(int(num_tries), 41) * range_args -
        range_args / 2
    losses = [compute_loss(t) for t in tries]
    min_loss_idx = np.argmin(losses)
    print(f"min loss index: {min_loss_idx}, min loss value: {losses[
        min_loss_idx]}")

    return tries[min_loss_idx], losses[min_loss_idx]

def find_optimal_fine(best_coarse_arg, num_tries):
    tries = np.random.rand(num_tries, 41) - 0.5 + best_coarse_arg
    losses = [compute_loss(t) for t in tries]
    min_loss_idx = np.argmin(losses)
    print(f"min loss index: {min_loss_idx}, min loss value: {losses[
        min_loss_idx]}")

    return tries[min_loss_idx]

def plot_loss_vs_range(ranges, num_combs):
    losses = [find_optimal_coarse(num_combs, r)[1] for r in ranges]
    plt.plot(ranges, losses)
    plt.xscale('log')
    plt.xlabel('Range of hyperparameters')
    plt.ylabel('Loss')
    plt.title("Loss vs Range of Hyperparameters")
    plt.show()

def plot_loss_vs_datapoints(data_points, range_args=2.5):
```

```
    losses = [find_optimal_coarse(d, range_args)[1] for d in
       data_points]
    plt.plot(data_points, losses)
    plt.xscale('log')
    plt.xlabel('Number of Hyperparameter Sequences')
    plt.ylabel('Loss')
    plt.title("Loss vs Number of Hyperparameter Sequences")
    plt.show()

ranges = np.logspace(np.log2(0.1), np.log2(60), num=20, base=2)
num_combs = 1000
plot_loss_vs_range(ranges, num_combs)

data_points = np.logspace(np.log2(100), np.log2(18000), num=15, base
   =2)
plot_loss_vs_datapoints(data_points, range_args=2.5)
```