SF3: Machine Learning Project IIA

Aditya Jain (aj563)

Christ’s College

18/05/2021

Final report

# Introduction

This project is centred around modelling and controlling a cartpole system. In Task 1, I develop a linear model for the simulated cartpole. I also experiment with changing the timestep in the simulation mechanics. In task 2, I develop a non-linear model by converting the dataset using basis functions and then performing linear regression. I also develop a linear controller using the cart simulation. In addition, I experiment with model predictive control – using my non-linear model to develop the linear policy. In Task 3, I add various types of noise to the measurements and the actual simulation mechanics and observe the effect on the predictions. In Task 4, I revert back to the no-noise case and develop a non-linear controller which, when combined with the linear controller, can keep the pole upright given specific initial conditions.

# Task 1

## Task 1.1

In this task, I wrote a function (start\_the\_cart) that plotted the 4 measures of cart dynamics over a specified number of steps. I was curious how the trajectories of the different cart dynamics (cart location, cart velocity, pole angle, pole velocity) will be affected by the initial conditions, so I incorporated that into the aforementioned function. The cart dynamics will be represented in square brackets in this order: [Cart location, Cart velocity, Pole angle, Pole velocity].

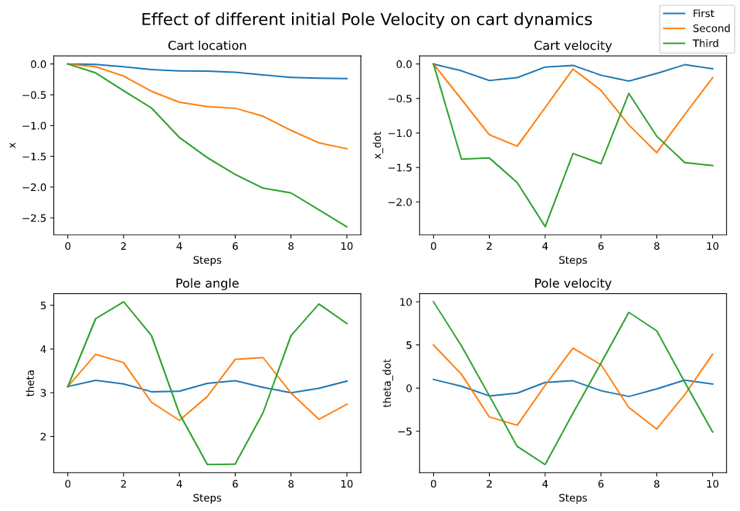


Figure 1a. Cart dynamics (discreet) time evolution from a stable equilibrium. The different lines are different initial pole velocities. Unedited dt.

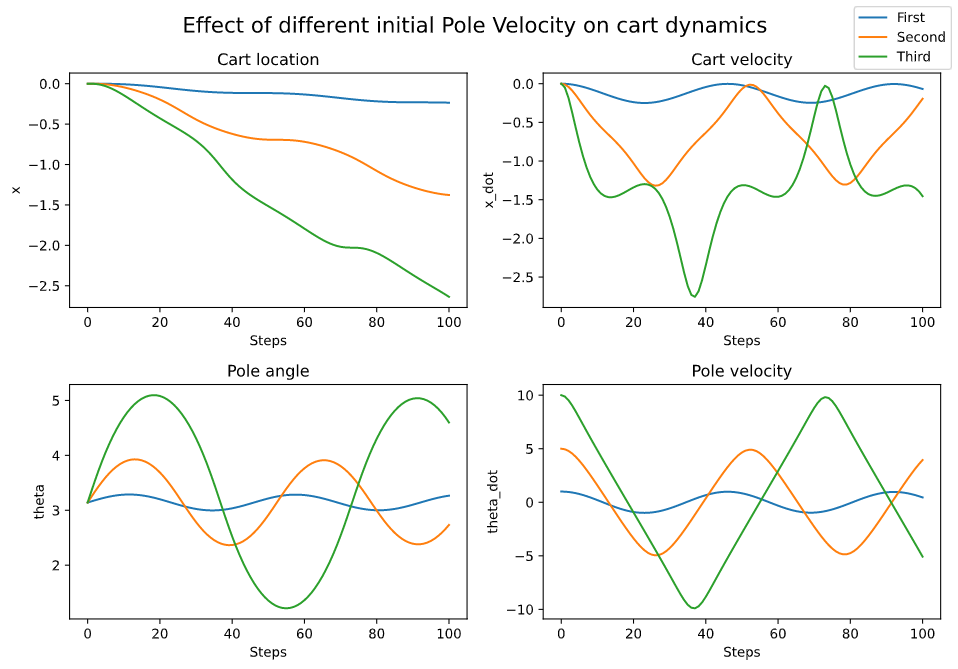


Figure 1b. Cart dynamics (discreet) time evolution from a stable equilibrium. The different lines are different initial pole velocities. Small dt.

Figure 1a shows how the cart behaves when at a stable equilibrium, [0,0,π,0], but the pole velocities are non-zero. The figure has the following initial conditions: First (blue) – [0,0,π,1]; Second (orange) - [0,0,π,5]; Third (green) - [0,0,π,10]. We see, as expected, the higher pole velocities lead to a larger change in cart location (supported by a higher cart velocity), and a large variation in pole angle. The function can also be called to vary a different variable. Note that the angle is not remapped. We see that in Figure 1a, the pole angle never crosses 2π or 0, so I increased the initial conditions to [0,0,π,15]. This produced Figure 2, where we clearly see that multiple full rotations have occurred (from pole angle). Note that unlike Figure 1a, the pole velocity is always positive as the pole never “falls back”.

Figure 1b is the result of experimentation of the timestep (dt) in the cartpole.py file. I reduced dt by a factor of 10 and increased the number of timesteps to 100. We see that Figures 1a and 1b are very similar, but Figure 1b is smoother. This is expected since the smaller timestep will cause our simulation to model the real continuous system better.

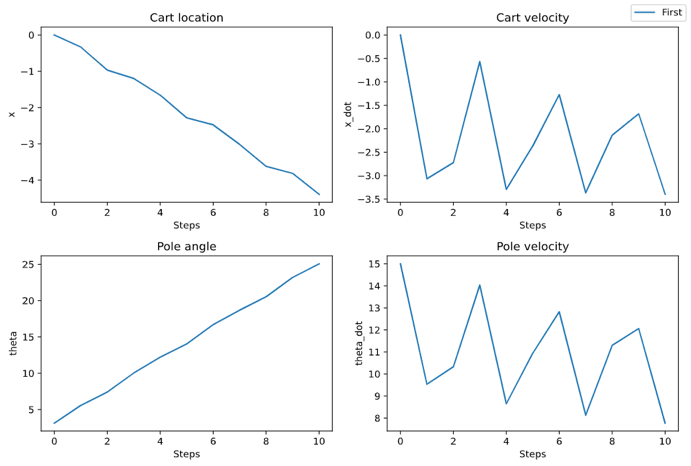


Figure 2. Cart dynamics after complete rotations of the pendulum.

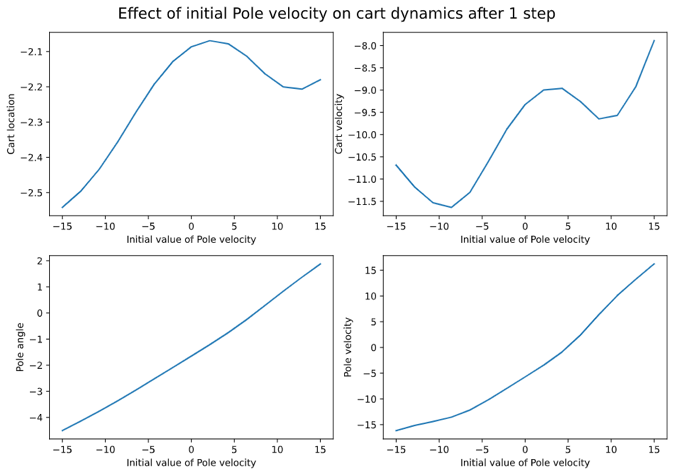


Figure 3. Cart dynamics after one step while varying initial pole velocity (and keeping other conditions constant).

## Task 1.2

I randomly initialized the starting conditions, [-0.24,-9.27,-1.07,9.09], and scanned over one variable at a time. I observed how the cart dynamics varied after one step with the (modified) initial conditions. Since one step didn’t cause a large change, the resulting plot was roughly linear. Figure 3 shows one of these plots, where I varied initial pole velocity. I used the original dt here.

We see the bottom-right graph in Figure 3 is roughly linear, as expected. To get a better idea of the effect of varying a single variable, we plot the *change* in cart dynamics after 1 step. Figure 4 shows a similar graph to Figure 3, but now the vertical axis displays the difference instead of the next value. We see that the dynamics depend non-linearly on the pole angle and velocity. We can also see that cart location doesn’t affect the next step (top left graph).

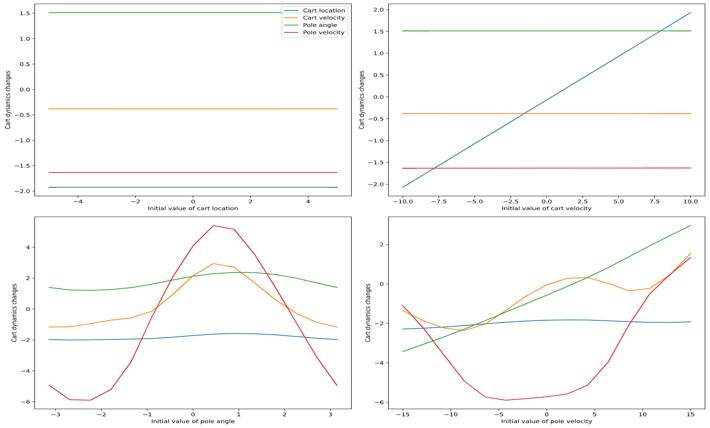


Figure 4. Change in cart dynamics after one step

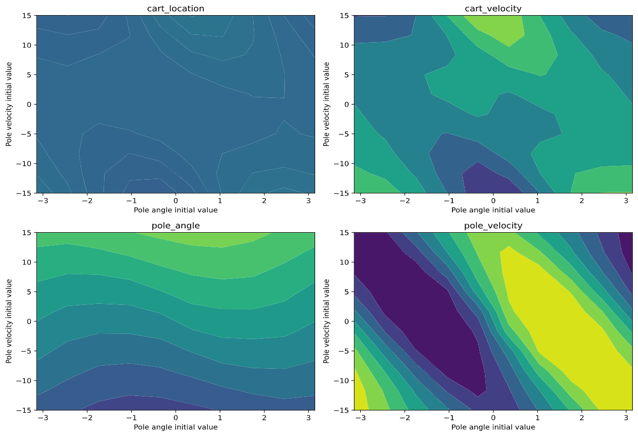


Figure 5. Contour plots displaying the change in cart dynamics with 2 changing initial variables – Pole angle and Pole velocity.

Another interesting way to visualise these results is to scan over 2 variables (keeping others fixed) and observe the change in cart dynamics after 1 step, shown in Figure 5.

In my code, I have plotted all combinations of the 4 variables and in each one it is clear that cart location doesn’t have an effect. I have included Figure 5 here to display the fact that pole angle and pole velocity have a non-linear effect on the change in cart dynamics after 1 step.

## Task 1.3

To perform linear regression, I generated 500 (X) data points randomly and got 1-step change (Y) points by running the perform\_action function once. I split these (X,Y) pairs into a train and a test dataset. To perform linear regression, I tried 2 ways. Firstly, using the numpy pseudoinverse function, I obtained the optimal weights W, where:

Secondly, out of interest, I used the “sklearn” package to perform linear regression as well. The results were nearly identical and can be found in my code. Throughout this report I will be discussing the results of the first method.

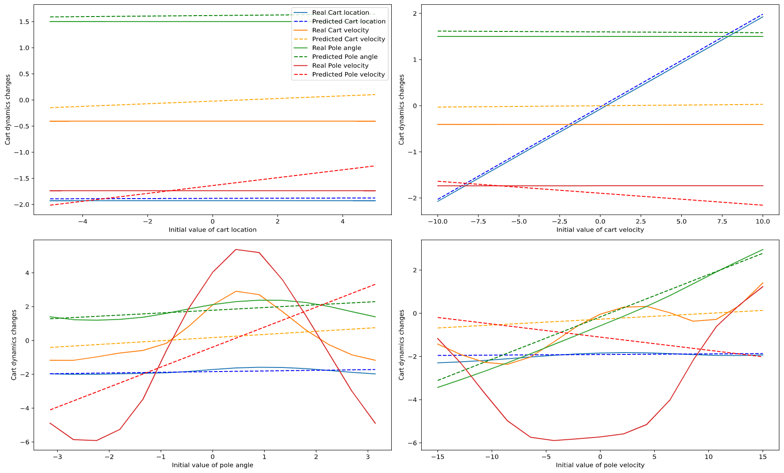


Figure 7. Predictions vs real cart dynamics after 1 step change, scanned over all variables.

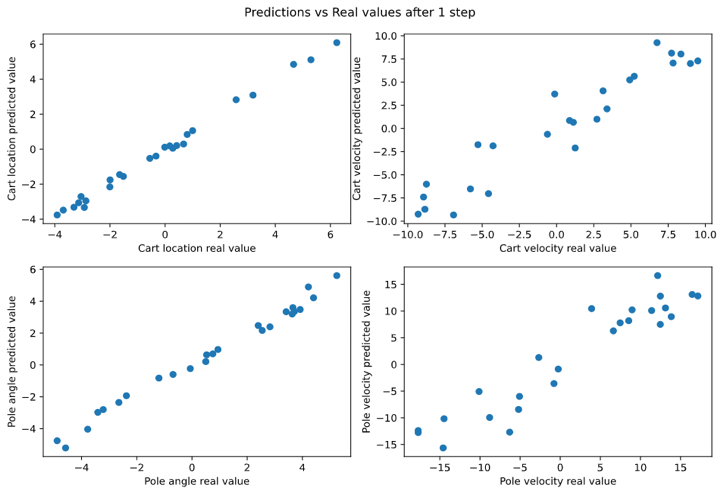


Figure 6. The next real value on the x-axis and the next predicted value on the y-axis for all variables.

In Figure 6 I have plotted the predictions after 1 step versus the real cart dynamics after 1 step. The perfect predictor would give a straight line of gradient 1. Visually, we see that the linear regression results work relatively well for cart location, pole angle, and to an extent, cart velocity too. However, scanning through the pole velocity did not produce predictions close to the real values. This is evident visually through the bottom charts of Figure 6, but I wanted to quantify that gap (since the axes have different scales). For this, I wrote the function “rmse\_calc” that calculates the root mean square error (RMSE) of each variable. The results were [0.106, 1.01, 0.282, 3.07]. These figures confirm my visual observations.

From Figure 7, we can see why cart location has such a small RMSE – as we vary any variable, the real cart velocity and pole angle remain roughly linear, thus can be predicted by a linear model. The other 2 variables vary non-linearly over the scans, so can’t be predicted accurately by a linear model. In the top 2 graphs, there is an offset with the predictions, even though the real scans are linear. This might occur due to the non-linearity in the scans of the other variables. The non-linear contours for cart velocity, pole angle, and pole velocity in Task 1.2 support the results shown in Figure 7.

## Task 1.4

To model how well the linear model predicts into the future, I decided to explore 2 paths. Firstly, I used real dynamics from the previous step to predict the next step - displayed in Figure 8. Secondly, I used the predicted value from the previous step to predict the next step, displayed in Figure 9.

We see from Figure 8 that the linear model performs well on cart location and pole angle given the previous real dynamics (this is what we saw in Task 3, that the model performs well for 1 step). The cart velocity and pole velocity are also predicted well, but an obvious delay is noticeable. This might be the model registering the previous value and adjusting its prediction for the next step.

The lines in Figure 9 are much smoother than those in Figure 8. This is expected since the model doesn’t receive any “correction” data, but just propagates the trend it started. In general, the predictions are worse after the first step, which is expected since our model was only trained on data after one step. More complex (non-linear) models could make the results better. Note that this model performs significantly worse in the case where the pendulum makes a full round.

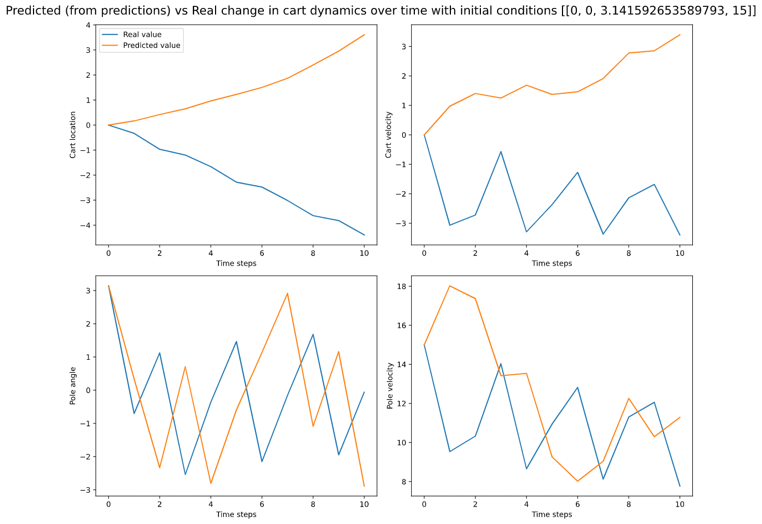
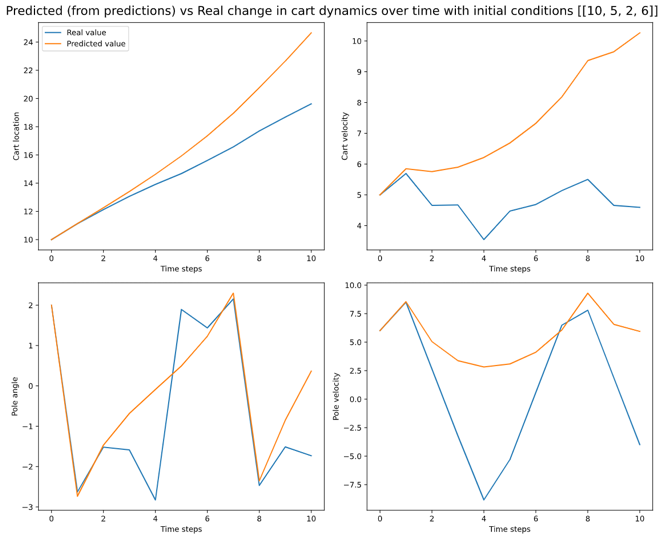


Figure 9a. Time forecasting using previous prediction. Initial conditions [10,5,2,6].

Figure 9b. Time forecasting using previous prediction. Initial conditions [0,0, π,15]. A full rotation of the pendulum.

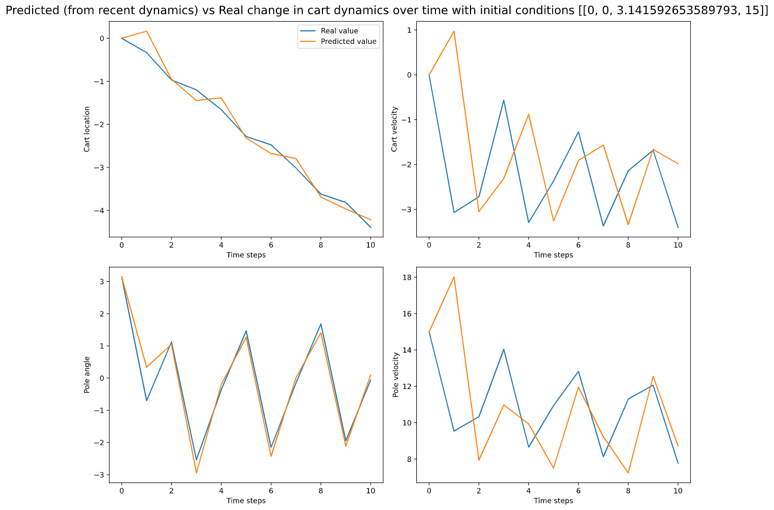
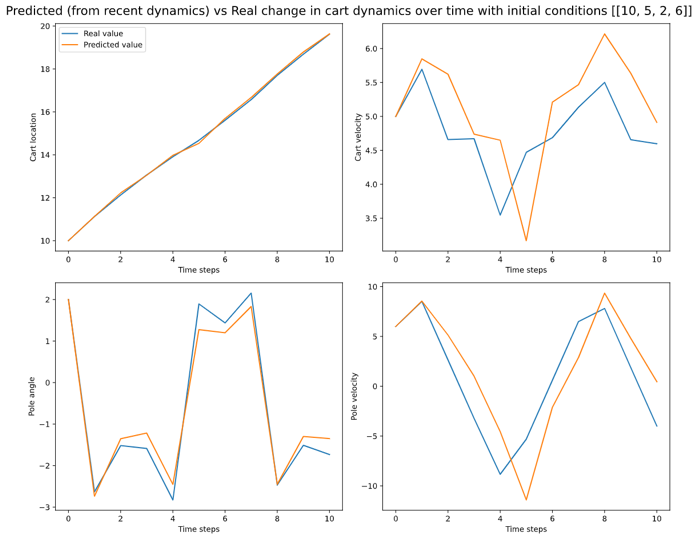


Figure 8a. Time forecasting using real change in dynamics. Initial conditions [10,5,2,6].

Figure 8b. Time forecasting using real change in dynamics. Initial conditions [0,0, π,15]. A full rotation of the pendulum.

If the angle was not remapped during the model rollout, the predictions diverge from the real dynamics. This is due to the fact that the equations of motion only contain θ in trigonometric functions. Hence, for the real dynamics, it won’t matter if the angle is remapped or not. For instance, a pole angle of 3π will result in the same next step as an angle of π. However, for a linear model, an angle of 3π is vastly different to π. Therefore, without remapping, the real dynamics and predictions will diverge; all the variables will diverge since pole angle affects all variables.

This raised a question in my head. Will the prediction diverge absolutely (ie. to infinity), or will it only diverge from the real dynamics but stay stable? To answer this, I created a model that is trained and deployed without the angle being remapped. The coefficient matrices produced by the remapped and the non-remapped models both have stable eigenvalues (within the unit circle). This can be seen in my code. Hence, we can conclude that although the prediction diverges from the real dynamics without remapping, it stays stable.

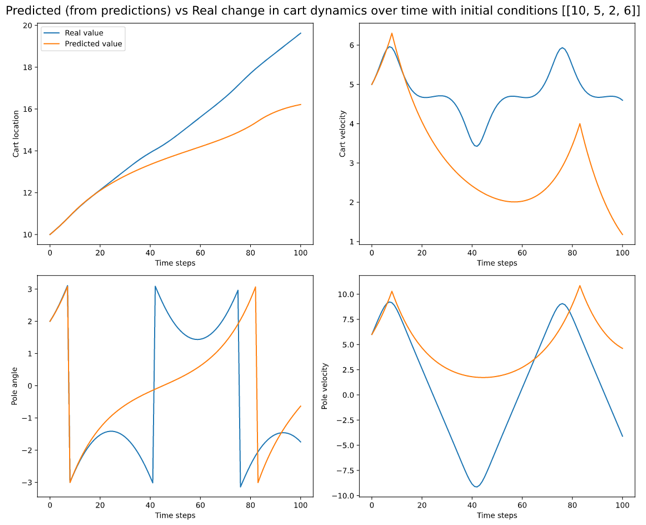
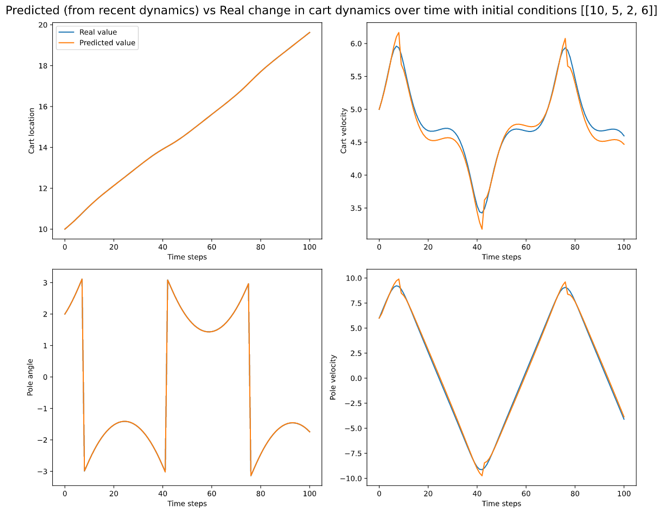


Figure 9c. Time forecasting using previous real dynamic. Initial conditions [10,5,2,6]. Small dt.

Figure 9d. Time forecasting using previous prediction. Initial conditions [0,0, π,15]. A full rotation of the pendulum. Small dt.

Figures 9c and 9d bring back the concept of reducing dt in cartpole.py. Note that the real dynamics (blue line) of Figures 9c and 9d are just smoother versions of Figures 8a and 9a. However, the predictions are much more accurate (especially when based off the previous real dynamic in Figure 9c). This is because with a small time-step, the cart dynamics change very little, and that change can be accurately approximated by a linear model. The question is, why not make dt infinitesimally small? Naturally, by doing that we increase computational costs. Furthermore, as demonstrated in Figure 9d, reducing dt doesn’t cause the model to predict well based off the previous prediction. A better solution for model-based predictive control would be to experiment with a non-linear model. In the rest of the report, for computational reasons, I will be using the original timestep.

# Task 2

## Task 2.1 and 2.2

In Task 2.1, I introduced a non-linear model in order to improve upon our linear predictions from Task 1. This new model made use of a Gaussian Kernel function to define the non-linear basis functions. Performing linear regression on these basis functions should lead to increased accuracy. The target function was, as before, the change in state after 1 step.

In Task 2.2, I expanded on the previous model by introducing a 5th state vector – the action. While an extra variable implies that our model’s accuracy should decrease, this was not largely evident through the results. Hence, instead of showing very similar graphs (from Task 2.1 and 2.2), in this section I will mainly analyse the results of Task 2.2, with some reference to Task 2.1.

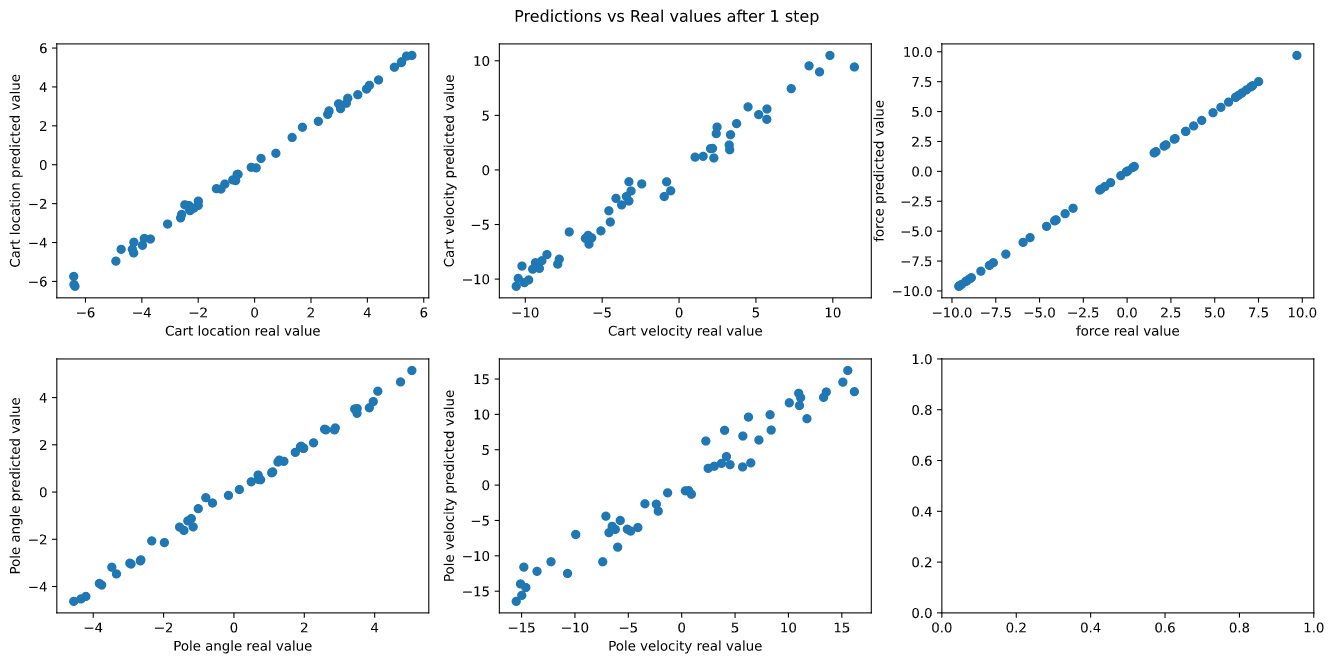


Figure 10. The next real value on the x-axis and the next predicted value on the y-axis for all variables, including force.

From Figure 10, we see lines of gradient approximately equal to 1. This indicates that the predicted value after 1 step (y-axis) is similar to the real next step (x-axis). The predictions of force are 100% accurate, since in this case the force is constant. In order to quantify the accuracy of these predictions, we look at the RMSE.

Figure 11 shows the RMSE evolution with regards to the number of data points (n) and number of basis centres (m) for each variable. Initially, the RMSE reduces dramatically for all variables with increasing m. As m increases, the rate of RMSE reduction slows down. For instance, the change in RMSE between m=10 and m=20 is much greater than between m=160 and m=320. Since increasing m leads to larger calculation times, there is a trade-off between accuracy and time. For the rest of this section, I will be using 320 basis centres. Looking at the y-axis values, we see that the RMSE for cart location and pole angle are relatively low, whereas pole velocity is high. This corresponds to Figure 10, where the pole velocity scatter plot does not follow the ideal gradient line as closely as the other variables.

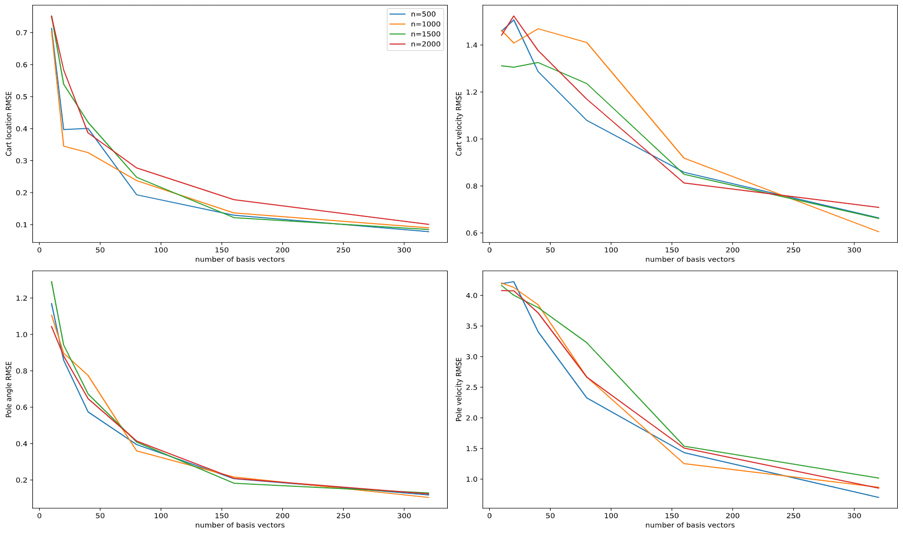


Figure 11. Convergence of RMSE with varying number of data points (n) and basis centres (m).

Interestingly, increasing the number of data points (n) does not decrease RMSE considerably. In fact, the RMSE generated by n=2000 is sometimes higher than the RMSE generated by lower values of n. This could be because proportionally, m represents a smaller subset of n when n is large. For instance, with m=320, that would mean 71% of training data is a basis centre for n=500 (assuming 95% training split). By comparison, only 17% of training data acts as a basis centre when n=2000. Using this logic, the model should perform similarly for the combinations (m=320, n=2000) and (m=80, n=500). However, Figure 11 clearly displays this is not the case. Hence, we conclude that increasing m reduces RMSE with diminishing returns and increasing n doesn’t affect it much. The small variation due to changing n might be attributed to random selection of basis vectors rather than n/m proportions.

Taking a step back, one important question that comes to mind is the hyperparameter selection. What values of sigma and lambda are best for our case? For sigma, I began with calculating the standard deviation of the variables in my generated data. This gave sigma = [2.89, 5.78, 1.85, 8.11, 5.78]. To optimize this, I imagined a 5-dimension grid, where we calculate the RMSE at each point, eventually settling on the sigma with the lowest RMSE. However, this would be computationally complex, especially considering that the optimum sigma could vary with state, so instead I decided to vary one component of sigma at a time while keeping the rest fixed. This is shown in Figures 12a and 12b.

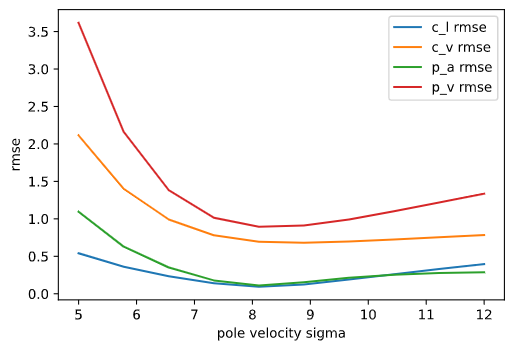
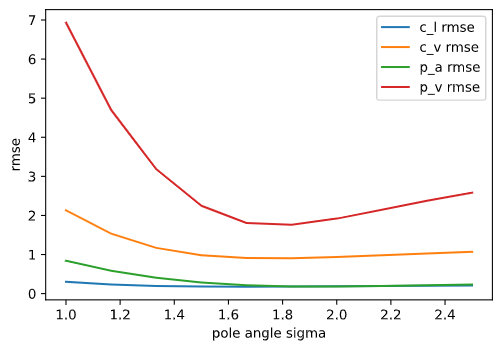


Figure 12a. Varying the sigma value corresponding to pole angle and observing effect on RMSE.

Figure 12b. Varying the sigma value corresponding to pole velocity and observing effect on RMSE.

From Figures 12a and 12b, we see that the minimum RMSE occurs very close to the original values obtained by standard deviation. This is assuming that the range provided is correct, so that there is no global minimum that lies out of range. Of course, this isn’t necessarily the minimum since we are only changing one variable at a time, but this cost-efficient method confirms that I can proceed with the original sigma. Scanning over the other 3 variables confirms this as well.

I plotted lambda on the x-axis using a logarithmic scale, and RMSE on the y-axis for all variables. From Figure 13a we see that the RMSE is low for all values of lambda lower than 10-2. This is illustrated in Figure 13b, which confirms that the predictions are inaccurate when we set lambda=0.1. Interpreting lambda as the data noise, this makes sense. If lambda is large, our model should perform worse. For the rest of this section, I will set lambda = 10-5.

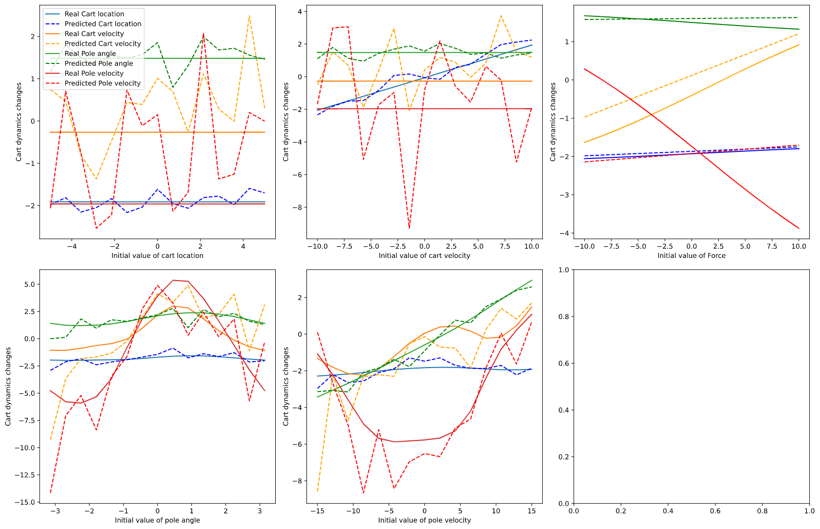
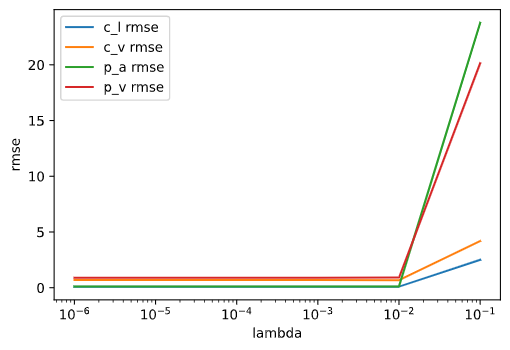


Figure 13a. RMSE with different values of lambda.

Figure 13b. Scans over variables with model trained with lambda=0.1.

Once the hyperparameters had been set (assuming that their optimum values don’t vary greatly with state), I plotted 1D scans over variables with initial conditions [-0.24,-9.27,-1.07,9.09,1], shown in Figure 14. We see that the pole angle and cart location are predicted accurately, and there is some variation in the cart velocity and the pole velocity. This is consistent with the higher RMSE of these 2 variables in Figure 11. Furthermore, the non-linear predictions in Figure 14 visually demonstrate that our non-linear model works (and is better than the linear model).

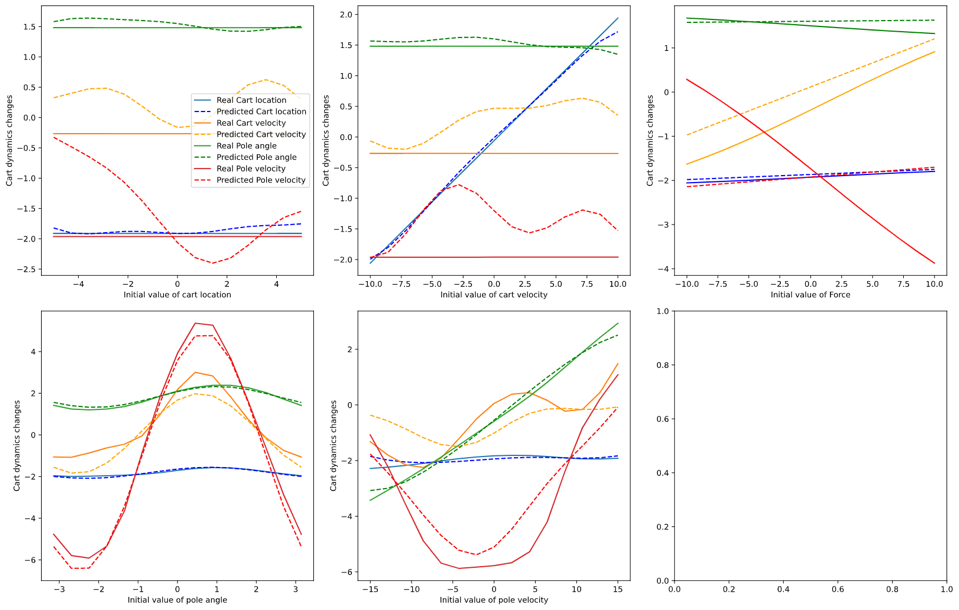


Figure 14. Scans over variables with real dynamics and predictions. Lambda = 10^-5.

Figures 15a and 16a show the predicted change in variables when we scan over 2 variables at a time. Figures 15b and 16b show the real change for those same variables. We see that the predicted contours are non-linear, and manage to capture the rough shape of the real contours. However, the scale bar shows some disparity between the values. The larger disparities occur in cart velocity and pole velocity; this is consistent with Figure 11 and Figure 14 because they show these variables having a larger RMSE. The difference in scale but similarity in shape can also suggest an offset between predictions and the real dynamics, which is also observed in Figure 14. In the 1D scans, we see the predictions of pole velocity and cart velocity roughly follow the real shape, but oftentimes with an offset.

Next, I investigated rollouts to see how closely the predictions matched the real dynamics with time. Figure 17 shows these rollouts with different initial conditions when the model took real data as its last input. Figure 18 shows these rollouts when the model took its previous prediction as the input.

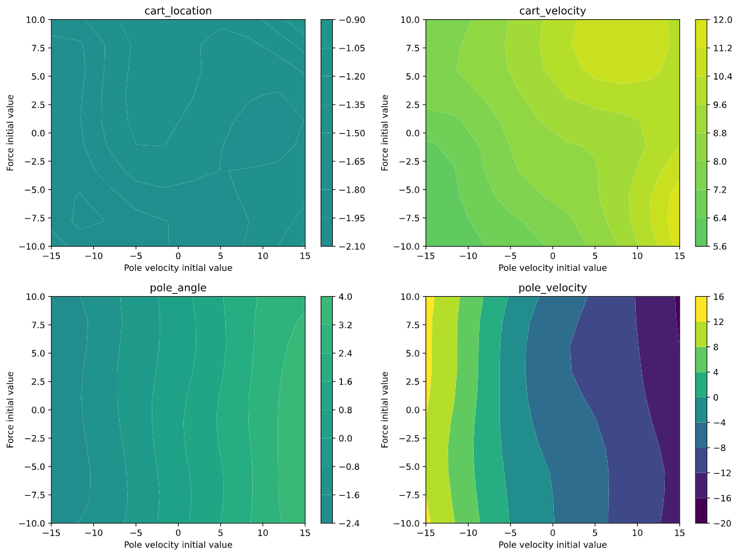
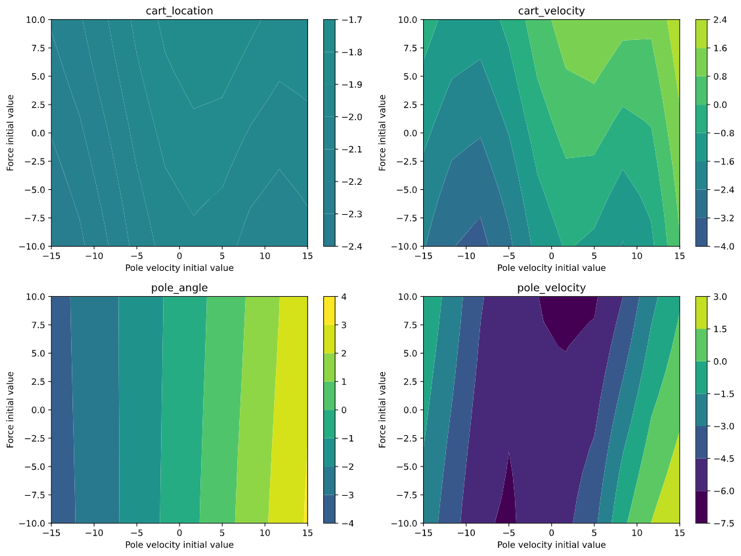


Figure 15a. 2D scan predictions while varying Pole velocity and Force.

Figure 15b. 2D scan real dynamics while varying Pole velocity and Force.

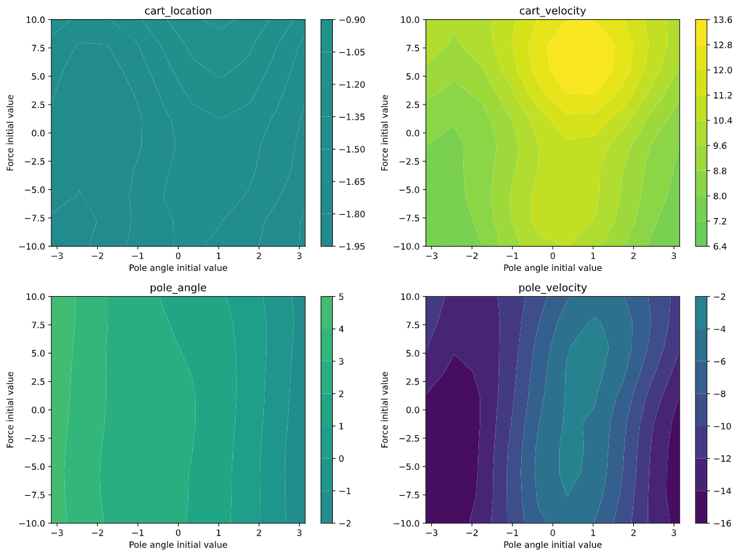
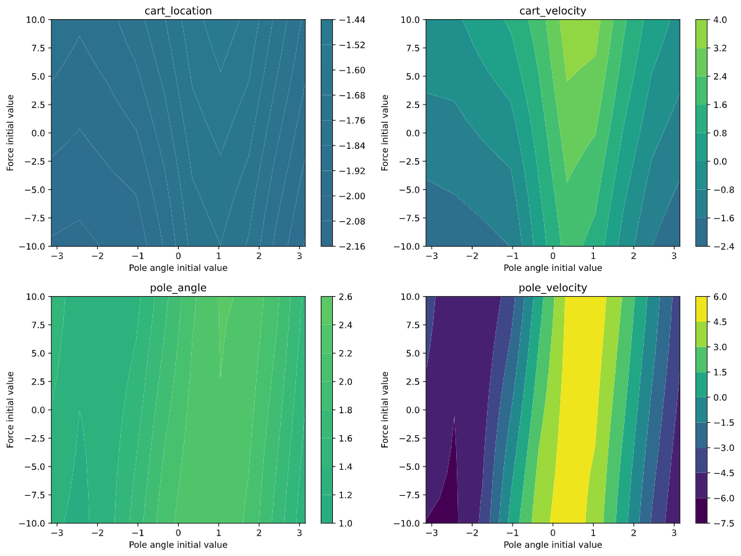


Figure 16a. 2D scan predictions while varying Pole angle and Force.

Figure 16b. 2D scan real dynamics while varying Pole angle and Force.

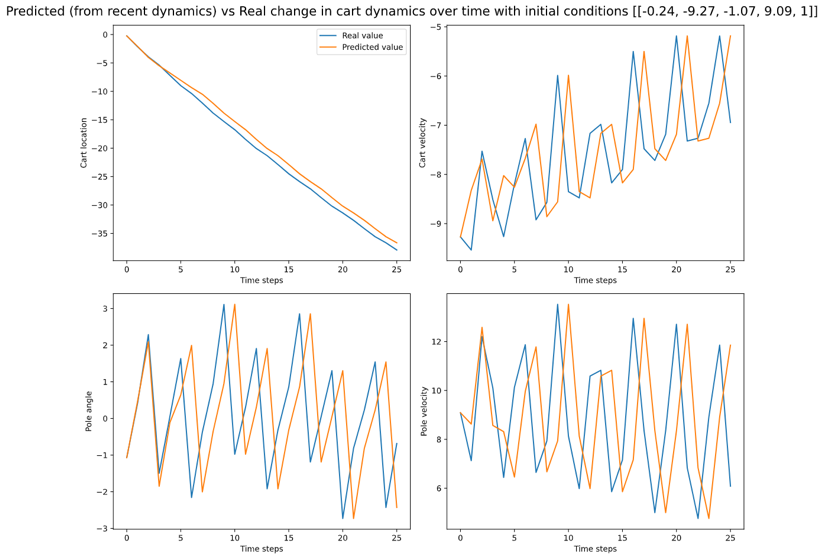
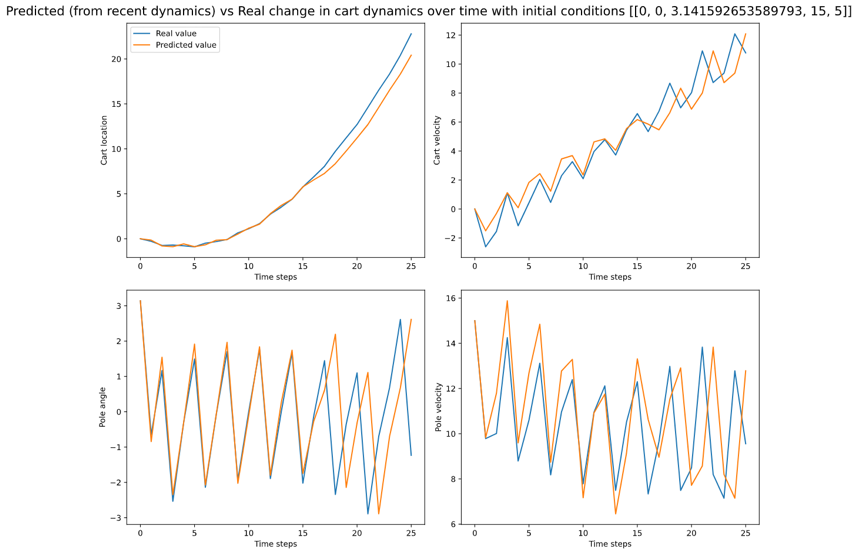


Figure 17a. Predictions vs Real dynamics using previous real dynamics. Initial condition [-0.24,-9.27,-1.07,9.09,1]

Figure 17b. Predictions vs Real dynamics using previous real dynamics. Initial condition [0,0,π,15,5]

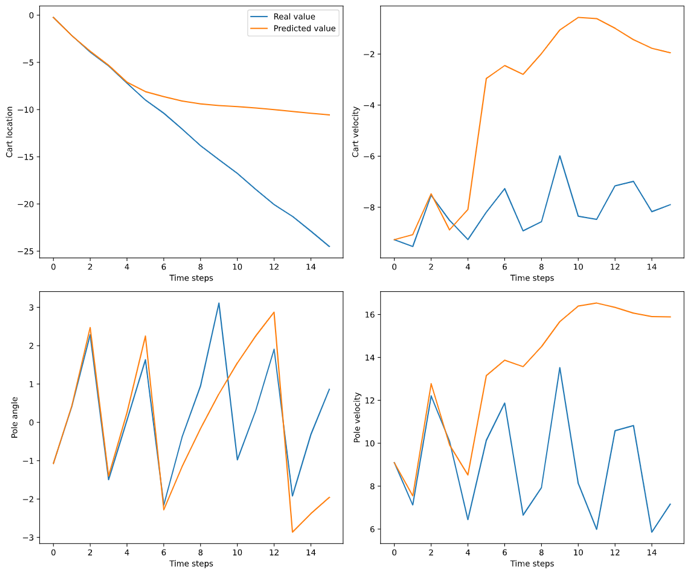
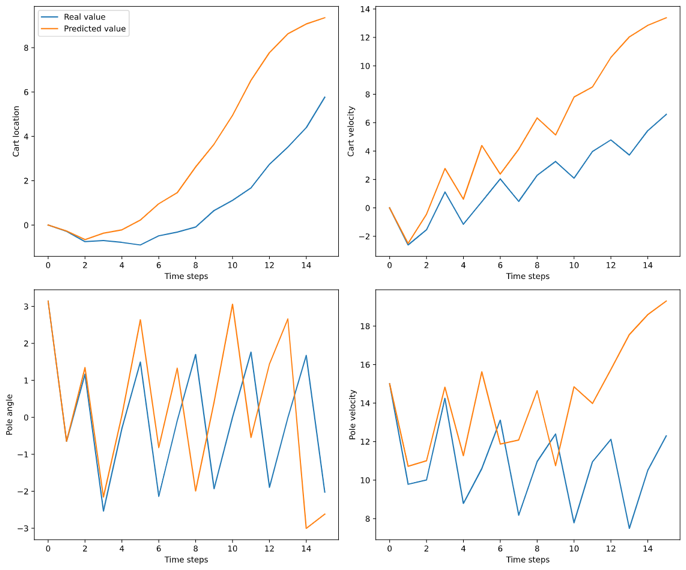


Figure 18a. Predictions vs Real dynamics using previous prediction. Initial condition [-0.24,-9.27,-1.07,9.09,1]

Figure 18b. Predictions vs Real dynamics using previous prediction. Initial condition [0,0,π,15,5]

Figure 17 shows that the model predicts the real dynamics extremely well for one initial condition (17b), and relatively less well for another starting condition (17a). With both starting conditions, the model predicts cart location and pole angle with greater accuracy than cart velocity and pole velocity; this is consistent with the RMSE values measured in the test data. The model predicts cart location and pole angle exactly for 4 time steps (roughly 1 oscillation), but stays close for the rest of the graph of 25 steps. In 17b, the model predicts nearly exactly for 15 time steps (roughly 5 oscillations). Perhaps this might be due to its pole angle starting from a stable position.

Figure 18 shows that if the model uses its own predictions to forecast the dynamics, the forecast will diverge from the real dynamics after 4 to 8 steps (2 to 3 oscillations).

## Task 2.3

Here we are introduced to linear policies – a function that determines the next action based on the current state in order to get the desired position. We define the desired state as [0,0,0,0]. We also introduce a loss function that is higher when the state is further away from our desired state. Currently, we will keep the same weightage for each variable when calculating loss, but this assumption will be relaxed later.

Figure 19 shows 1D and 2D scans of components of the p vector and their effect on loss. The p vector is used to determine the action for the next step (which is passed through a tanh function to get the final force). We see that there are certain clear optimums in our choice of p that reduce loss. Naturally, this will depend on our state vector as well.

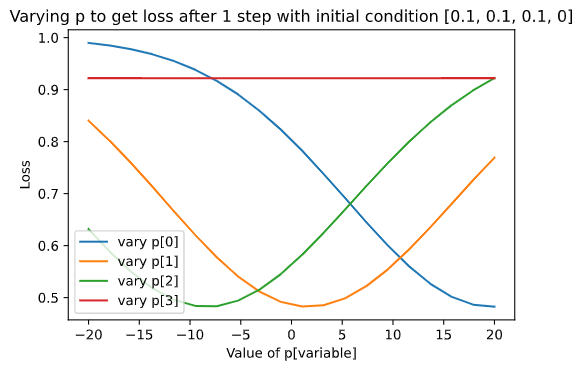
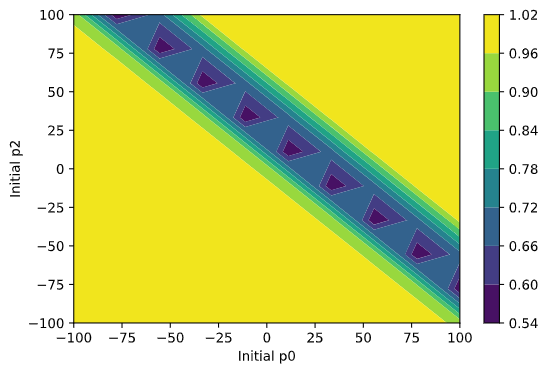


Figure 19a. Varying single components of p=[0.5,2,10,10] and measuring loss

Figure 19b. Varying 2 components of p=[0.5,2,10,10] and measuring loss

In order to get the best value of p for states close to our desired state, I used a rollout of 20 steps starting from [0,0,0.1,0]. Then I used scipy to optimize the p vector so that it generates the lowest cumulative loss over a real rollout.

Figure 20 shows the rollout of the cart dynamics using actions dictated by our policy. Figure 20a shows an initial condition that leads to the pole being upright even after 20 time steps. Figure 20b shows a rollout that diverges from our ideal state within 2 time steps – this might be because the initial conditions are too different from the initial condition the policy was optimized upon. I investigated further by training the linear policy on a rollout starting from [0,0.5,0.5,0.5], but the policy obtained from that still didn’t stabilise the system. This shows the limitation of a linear policy.

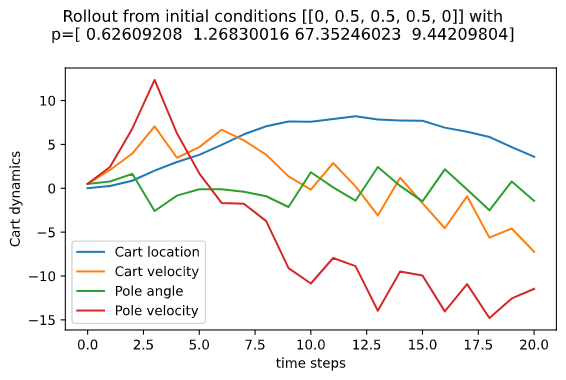
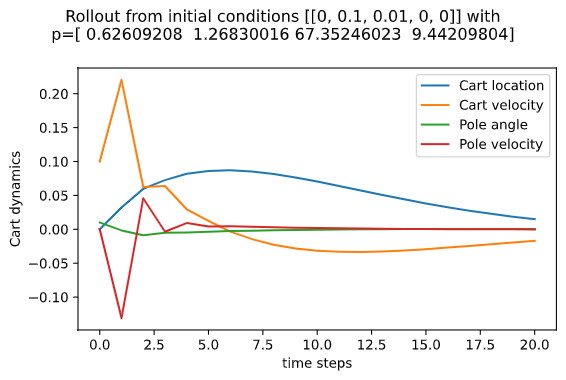


Figure 20a. Rollout from initial conditions in title.

Figure 20b. Rollout from initial conditions in title

## Task 2.4

In this task, I plotted rollouts using data from the non-linear model developed in Task 2.2. Due to the results in Figures 17 and 18, I set the number of time steps to 6, which is when the model might diverge from the real dynamics. Now, I obtained the optimum value of p by running my model, accumulating the loss at each timestep, and minimizing that.

Figure 21a shows the rollout of our non-linear model using the policy vector obtained by minimizing loss on the model rollout. Figure 21b shows the same policy used on the real dynamics. We see that the pole is kept approximately upright in 21a longer than in 21b. This is expected since the policy was trained on the model rollout. We see that the cart velocity is non-zero in both Figures (albeit closer to 0 in Figure 21a), which displays the inadequateness of a linear policy in maintaining our desired state.

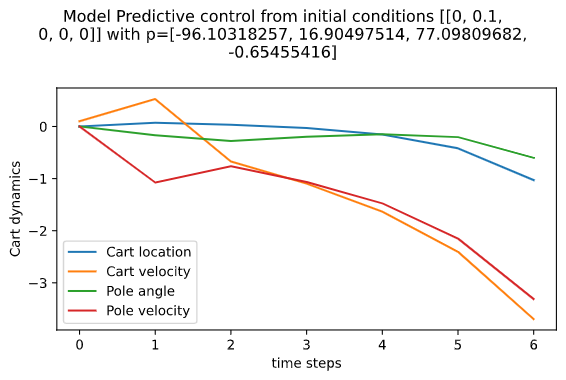


Figure 21a. Model rollout using p from model predictive control from initial conditions in title.

Figure 21b. Real rollout using p from model predictive control from initial conditions in title.

## Task 2 extra investigation

Coming back to the loss function, we want our cart to ideally be still and our pole to be upright. This means that we must place more emphasis on pole angle and cart velocity to be 0 than on cart location to be 0. I experimented by editing the loss function. The old loss sigma used above was [0.5,0.5,0.5,0.5]. I changed that to be [1,0.3,0.3,0.3]. This meant that the loss would be minimal if cart location was far from 0, but the loss would be great if the pole angle or cart velocity were far from 0.

Figure 22a shows the rollout from an initial state where cart location is high. We see that the policy doesn’t make the cart location go to 0 quickly, but rather focuses on keeping the cart velocity and pole angle close to 0. By comparison, Figure 22b shows a rollout of the same conditions but with the p value used in Section 2.3. We see that due to the high initial cart location, loss is large and the policy makes drastic actions, leading to instability. This new loss function seems promising.

To test this new value of p, I initialised the system with a large cart velocity, shown in Figure 22c. The system stabilised the pole angle, pole velocity, and cart location within 5 steps, and gradually reduced cart location. Figure 22d shows the same initial system but with the p value from Task 2.3. Here we see that cart velocity decreases quickly but overshoots the 0 mark and becomes negative. This might be the system trying to make cart location tend to 0 quickly. When both cart location and cart velocity are negative (around the 10th timestep), the system attempts to increase cart velocity. The pole velocity and pole angle are drastically affected as a result of this linear policy. Hence, we can conclude that in this case, this new value of loss sigma leads to better results.

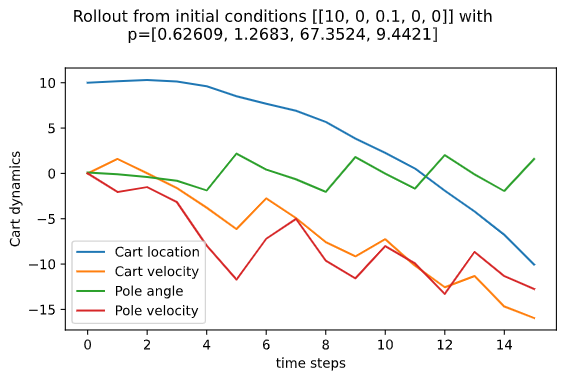
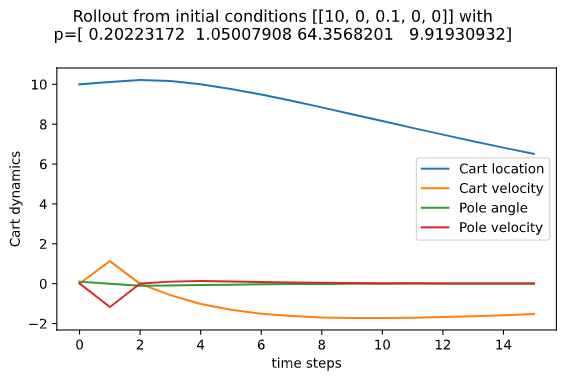


Figure 22a. Rollout after experimenting with loss sigma

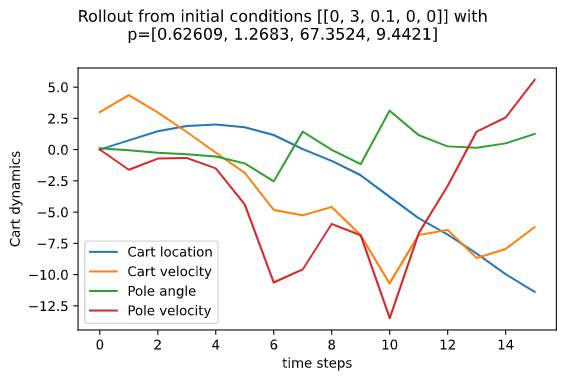
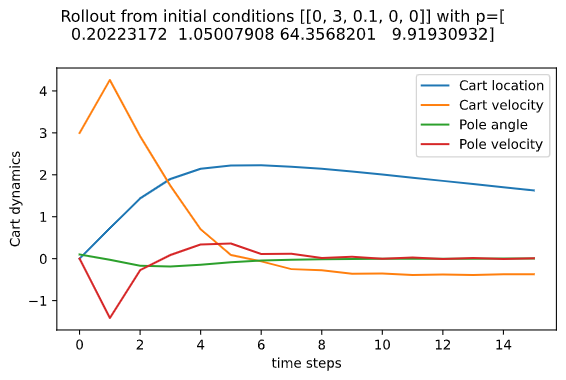


Figure 22b. Rollout after experimenting with loss sigma

Figure 22c. Rollout after experimenting with loss sigma

Figure 22d. Rollout after experimenting with loss sigma

# Task 3

## Task 3.1

In this task, I introduced noise to the observed dynamics. Noise was added to the training “x” data that I fed into the models, and into input during the model rollout. Initially I added random Gaussian noise to each variable, then experimented with adding bias.

### Random Gaussian Noise without bias

Figure 23 shows some plots from the linear model with unbiased gaussian noise with standard deviation [1,1,0.5,1,1]. The difference in RMSE after training, when compared to the no-noise case in Task 1 was [0.20,0.65,0.13,0.85]. This fits with the standard deviation and Figure 23a.

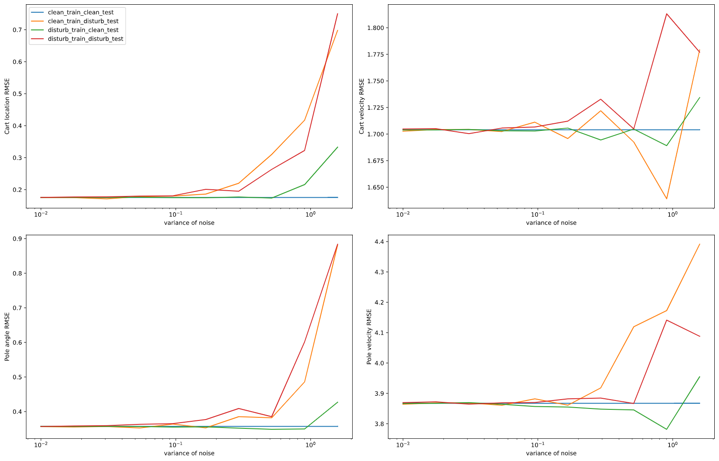
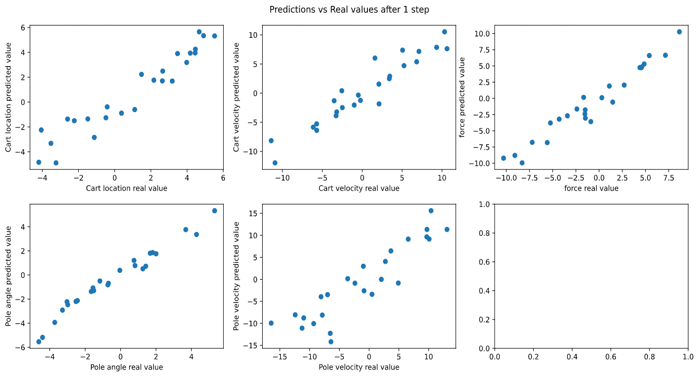


Figure 23a. Linear model predictions after 1 step versus real with noise.

Figure 23b. RMSE after 1 step for various test/train noise as standard deviation increases

Figure 23b aims to characterise the degradation in accuracy (or increase in RMSE) as standard deviation of gaussian noise increases. I experimented with 4 combinations created by adding noise to the train and/or test data. We see that the no-noise dataset for both train and test (blue) creates a horizontal line. This is because the noise deviation doesn’t affect it. As the noise increases, the RMSE increases for other cases, especially the case with noisy training and test (red). Occasionally we see drops in RMSE as noise increases; this can be attributed to the randomness of the noise.

Figure 24 shows the results of adding noise in the non-linear case. Figure 24a is similar to the accuracy degradation in the linear case (Figure 23b). The noiseless case remains at a constant RMSE while the cases with noise fluctuate.

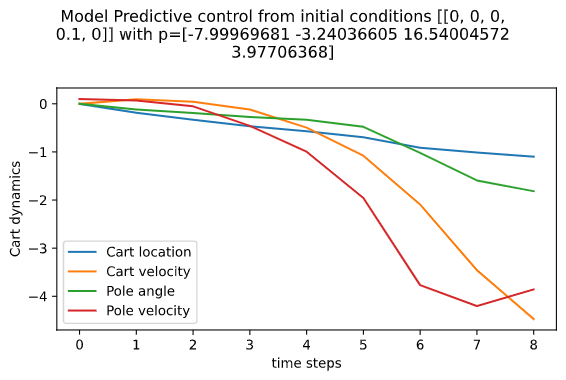
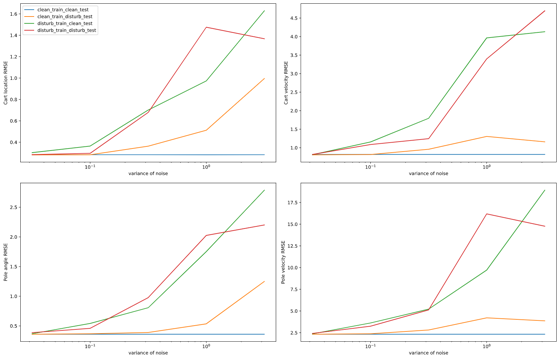


Figure 24a. RMSE after 1 step for various test/train noise as standard deviation increases

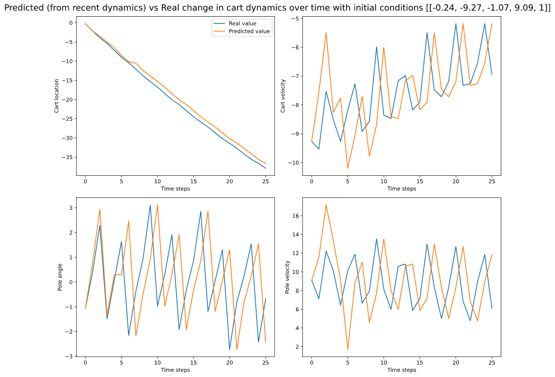


Figure 24b. Model rollout using p from model predictive control from initial conditions in title.

Figure 24c. Rollout after adding noise to measurements

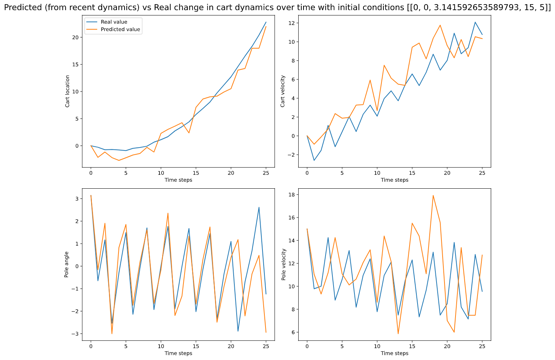


Figure 24d. Rollout after adding noise to measurements

Figure 24b shows the rollout of the model when a linear policy is active. We see that the pole angle is stable up until 5 steps, but then deteriorates. This is slightly worse than the no-noise case. Figures 24c and 24d can be compared to Figures 17a and 17b respectively. These figures clearly demonstrate the noise addition in the rollout (and previously in the training data) can lead to worse predictions.

### Random Gaussian Noise with bias

Introducing bias (mean) to the noise (during training and rollouts) tries to replicate a scenario in which the sensors not only have disturbance, but they also have an offset of [5,5,0,5,0]. Figure 25 shows some of the results this produced.

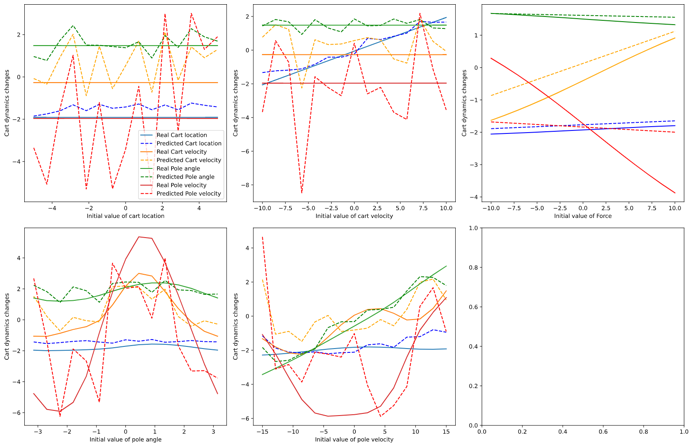
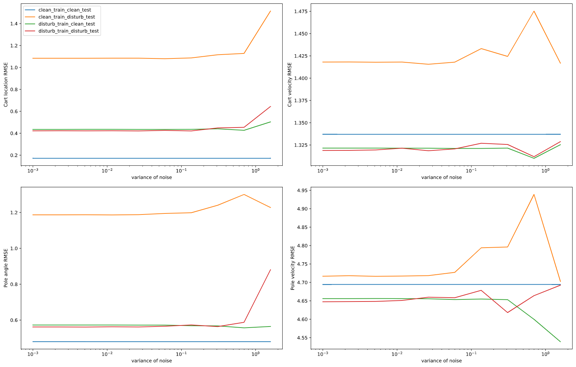


Figure 25a. RMSE after 1 step for various test/train noise as standard deviation increases (linear model)

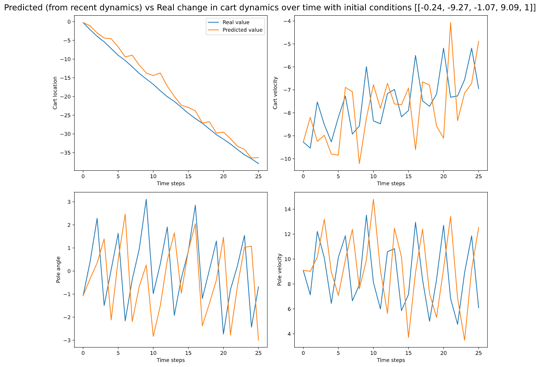


Figure 25b. 1D scans for non-linear model with noise.

Figure 25c. Rollout after adding unbiased noise to measurements (after training on biased)



Figure 25d. Rollout after adding biased noise to measurements (after training on biased)

Figure 25a shows the degradation of accuracy as variance is increased. When compared to the non-bias case (Figure 23b) for the linear model, we see that the clean training and noisy test data (orange) performs the worst. This large offset from the other scenarios can be explained by the large bias introduced in the test data, but not in the train. Figure 25b shows the non-linear scans over variables and their predictions after 1 step. We see that the predictions perform significantly worse than without noise (Figure 14). Figure 25c shows a rollout when the bias is added in the training data, but the next step is predicted using accurate data. By contrast, Figure 25d shows a rollout on the same conditions but the next step is predicted using data with biased noise added. We see a clear offset in predictions in Figure 25d, which suggests that an offset in measurements is detrimental to the model even though it is trained on such data. Note that pole angle predictions don’t get affected much, which is most likely due to the 0-bias added to it.

It is also interesting to compare Figures 24c and 25c, because they are identical except that 25c is trained on biased data. Figure 25c performs slightly worse than Figure 24c, suggesting that adding biased noise to the training data is detrimental. From this section, we can conclude that adding 0-mean noise to measurements reduces accuracy (proportional to the variance of noise), but adding biased noise reduces accuracy much more, even if the model is trained on it.

## Task 3.2

In this task I added random (unbiased) noise to the cart dynamics with standard deviation [1,1,1,1,1]. To place it in a real-world scenario, this would be analogous to the table being vibrated underneath the cart, whereas Task 3.1 would be bad sensors.

Figure 26a displays the real dynamics and non-linear predictions after adding noise to both. When comparing it with the measurement-noise case (Figure 25b), we see that Figure 26a has, as expected, a messy real dynamic (due to the noise), and predictions that fluctuate more but not very far from the real. This is due to Figure 25b having a bias in the noise. When comparing it to the noise-free case (Figure 14), we can see that adding noise to both measurement and simulation is extremely detrimental.

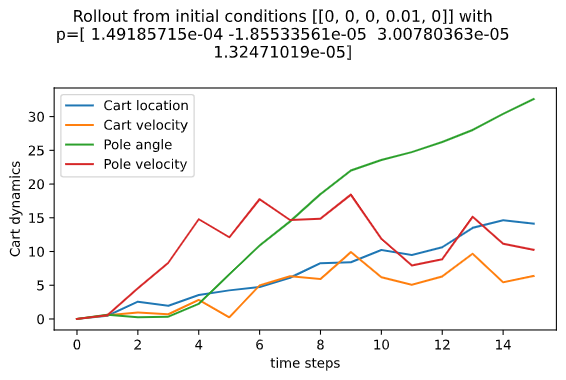
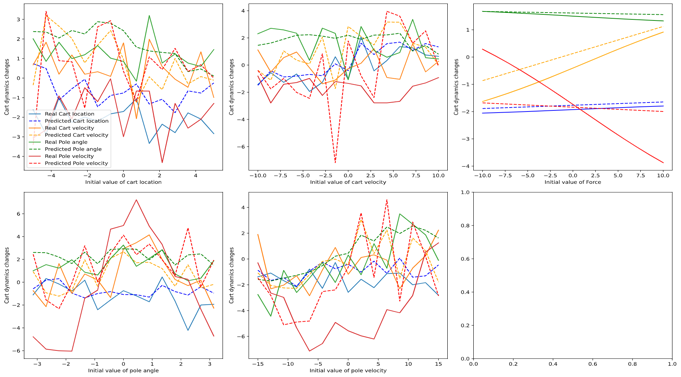


Figure 26a. Non-linear model predictions after 1 step versus real with noise.

Figure 26b. Rollout using policy trained with noisy dynamics and carried out with noisy dynamics.

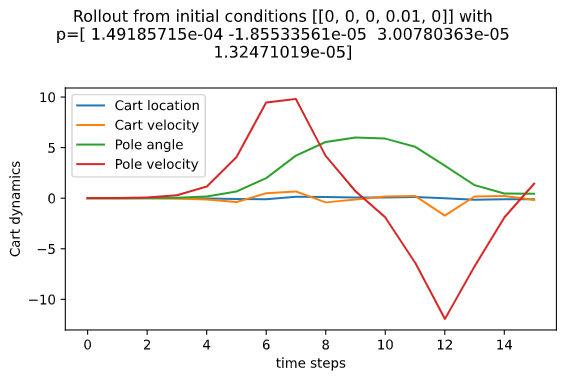


Figure 26c. Rollout using policy trained with noisy dynamics and carried out with noisy dynamics.

Figure 26b shows a rollout with a linear policy that has been trained on noisy dynamics and deployed with noisy dynamics. This would correspond to training a policy on a shaking table then trying it out on the shaking table. Figure 26c corresponds to training a policy on a shaking table, but deploying it on a noiseless table. We see that the linear policy is able to perform much better in 26c, even though it is trained on noisy data. It is important to note that both Figures 26b and 26c perform worse than the no-noise case in Figure 20a, as expected. It is interesting to note the order of magnitude of the p vector in both Figures 26b and 26c with noise is very low; this suggests that with large noise in dynamics and measurement, the linear policy doesn’t do much. It finds the optimum in letting the action tend to 0.

An interesting concept to explore is how much measurement noise is equally detrimental to accuracy as real dynamics noise. Posing it as a business case, if one had a limited budget to spend on good sensors or noiseless tables, where should they spend their money?

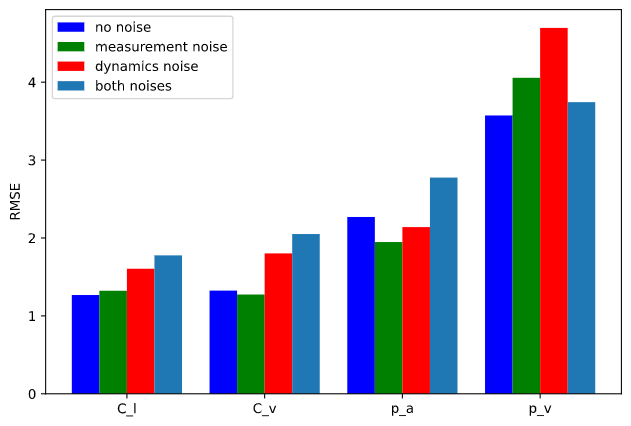
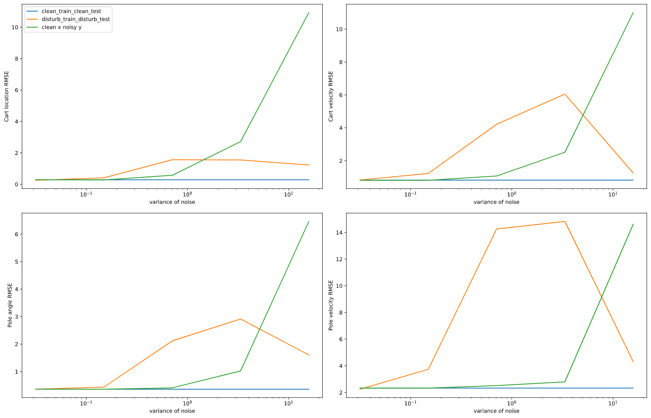


Figure 27a. RMSE after 1 step comparison of measurement (x) and dynamics (y) noise with increasing variance.

Figure 27b. RMSE comparison of measurement (x) and dynamics (y) noise over a rollout.

Figure 27a shows the evolution of RMSE after 1 step as we increase noise variance for 3 cases: no noise (blue), measurement noise (orange), and dynamics noise (green). We see that for all variables except pole angle, the dynamics noise introduces more error. However, we see that mainly occurs after a certain standard deviation of noise; this suggests that one should get better sensors if the expected table vibration is less than a certain value. However, if the table vibration std. deviation is large, it would be better to invest in a noise-free table. Figure 27b shows the average RMSE over a rollout of 25 steps when the model is trained and deployed using the 4 cases in the legend. Due to the nature of random noise addition, we see some fluctuation, however the general trend is as follows. The no-noise case has the lowest average RMSE, followed by the measurement noise, followed by dynamics noise, and the largest is both noises (as expected). I looked at the RMSE of a rollout because in a practical situation, we want to quantify (and reduce) the average error over time.

# Task 4

In this task I attempted to develop a non-linear policy that aims to stabilise the system with the pole upright. This was a hard task, and after several ideas, I managed to get a policy that stabilises the system with very specific initial conditions. In the table below, I will summarise all of my approaches (in chronological order) and then explore the interesting ones below that.

In response to the question posed in the handout, I ensured that the W matrix was symmetric by creating 10 parameters and optimising those in a 4x4 matrix. This can be found in my function titled “f” in Task 4. For instance, for a 5-basis optimisation, I introduced 35 parameters: 5 for the multipliers (w\_i), 10 for the W matrix, and 20 for the basis centres (X\_i).

|  |  |  |
| --- | --- | --- |
|  | Approach summary | Comments |
| 1 | Initiate all variables randomly using same dist. and run Scipy.optimize | I initialized all variables using various distributions (in successive experiments) such as Gaussian and Uniform. I also randomly generated the initial conditions. This worked, but the resulting parameters were so small that the policy could not do much. See Figure 28a. From here on, if I randomly initialize a variable, I will do so using a Gaussian distribution. |
| 2 | Initiate basis centres (X\_i) as states randomly picked from a rollout without an active policy | In this step, I experimented with changing X\_i and not including it in the optimisation loop. This resulted in one of 3 things. Firstly, the iteration limit or function evaluation number was crossed. Secondly, the loss did not move (I printed out the loss at each iteration). Thirdly, at times, the loss function became so large the algorithm declared it as “NaN” and couldn’t finish. The fluctuations might have been caused by random initialisations. |
| 3 | Initiate basis centres (X\_i) manually | Similar concept to approach 2 but I manually set the X\_i locations to be equally spaced around the circle. Same results as approach 2. |
| 4 | Initiate variables randomly using Gaussian dist. with different variances | I was beginning to think that my initialisations were the problem. This is because when the optimisation worked, it didn’t cause a large decrease in loss. Thus, I thought that my initial conditions were in a place where the loss function was flat, hence difficult to optimise. Hence, similarly to approach 1, I initialised all parameters randomly, but this time I experimented with variances. For instance, I initialised the W matrix using a distribution with mean 0 and variance 1, and the X\_i using a distribution with mean 0 and variance 5. This approach didn’t work and led to similar errors described in approach 2. |
| 5 | Changing optimisation method and epsilon | I experimented with different optimisation methods provided by the Scipy minimize package. Most notably, I tried the “BFGS” method, in which I could change the absolute step size ‘epsilon’. I figured that increasing step size could be a good way to leave the flat region of the loss function and obtain the global minimum. Changing methods often led to an error stating that the desired error was not achieved due to precision loss. This didn’t lead to any interesting developments. From here on I reverted back to the Nelder-Mead method. |
| 6 | Experiment with loss sigma values | As in the “Task 2 extra investigation”, I changed the loss function to have different sigma values. This did not lead to any different results. Then I experimented with loss sigma values that change with time. As is coded in my “loss3” function, in the beginning of a rollout, I set the loss sigma as [3,3,3,3]. As the rollout proceeds, I gradually reduce the sigma values to [1,0.3,0.3,0.3]. The logic behind this was that at the start of a rollout, the policy should be able to swing the pole and experiment without being penalized by the loss function. As time proceeds, the policy should bring the system closer to the desired state (hence the narrow loss sigma values). This led to larger changes in loss, signalling that it was working, but still gave similar results to approach 1. |
| 7 | Reduce timestep in cartpole.py | After trying all the above steps and not making any major progress, I decided to use my experiments with timestep (dt) in week 1. I reduced the timestep by a factor of 4 and tried approach 6 again. This time, the optimization algorithm was able to significantly reduce my time-dependent loss. Figure 28b shows a rollout with this policy. The system came close to having the pole upright but could not maintain it. Note that this increased the number of timesteps, so I increased max number of iterations and function evaluations to avoid those errors. |
| 8 | Experiment with number of basis functions and number of rollouts | After the progress made in approach 7, I thought a good policy could be developed by increasing the number of basis centres or by increasing input data. Figure 29a shows a rollout using a policy with 10 basis centres. The policy doesn’t seem to be doing much. I thought this might be because 1 rollout for training wouldn’t be enough to optimise 60 parameters, so I let it train on multiple rollouts. Figure 29b shows the result of that. Surprisingly, it didn’t work (neither did 15 basis centres), so I reverted back to 5 basis centres and 1 rollout for training. |
| 9 | Introduce linear policy when pole angle is close to 0. | Carrying on from approach 7, I noticed that the non-linear policy manages to bring the pole angle close to 0 but can’t fully stabilize it there. So, I changed the code so that the linear policy takes over when the pole has been under 0.6 radians for the last 5 timesteps. The result is shown in Figure 30a. This solution worked. |
| 10 | Incorporating force attenuation into non-linear policy. | By printing the action at each step in approach 9, I realised that the majority of actions performed by the non-linear policy were of magnitude 20 (the max force). This might be because I had coded in the tanh function too late, so the parameters being trained were resulting in large actions that were later morphed. I introduced the tanh function earlier in the optimization. This did not cause the policy to be successful in stabilizing the pole, but when combined with the linear policy, it stabilized the pole in fewer timesteps. See Figure 30b. |

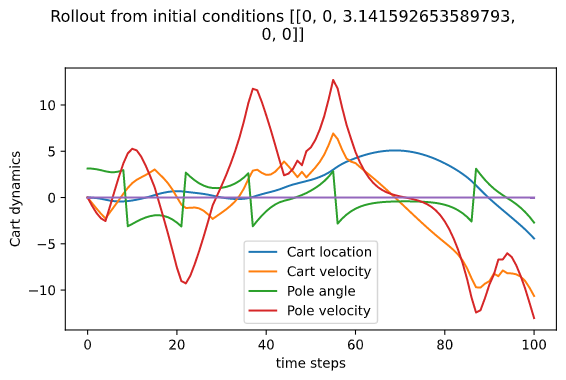
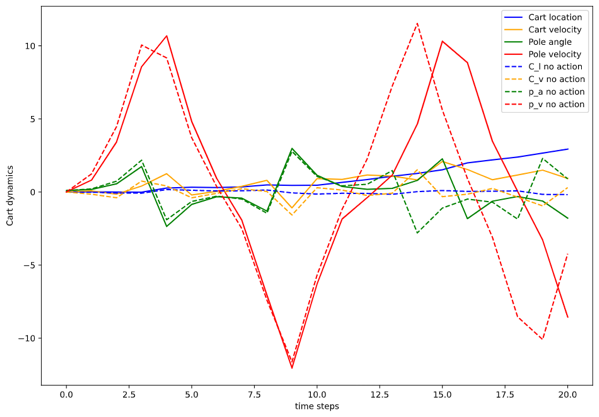


Figure 28a. Rollout on non-linear policy with random initial parameters. Dashed line shows no active policy. 5-basis centres.

Figure 28b. Rollout on non-linear policy with random initial parameters with reduced timestep. 5-basis centres.

From Figure 28a we can see that the non-linear policy developed in approach 1 did not do much. The rollout without any policy (dashed) is very similar to the rollout with policy (solid). Figure 28b shows the results of reducing the timesteps (approach 7). Note that around timestep 70, the pole comes very close to 0, but then drops back and continues doing loops. This is rectified with a linear policy later.

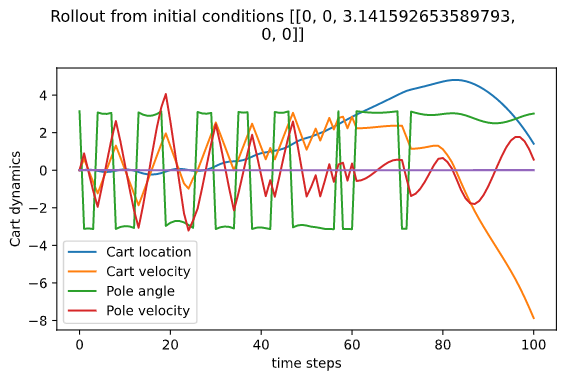


Figure 29a. Rollout with 10 basis centres. Policy trained on 1 rollout.

Figure 29b. Rollout with 10 basis centres. Policy trained on multiple rollouts.

Figure 29a displays the lack of any results caused by the policy with 10 basis functions. The pole is oscillating around the stable equilibrium while the cart location is continually rising. Figure 29a shows a policy trained on multiple rollouts. It does not perform any better.

Figure 30a shows the 5-basis centre policy coming very close to stabilizing the system, at which point the linear policy is activated. It is interesting to note that around timestep 55, the linear policy is activated but quickly loses control, at which point the non-linear policy takes control. The linear policy kicks in once again around timestep 120, then is successful. Figure 30b shows a similar trend, but this time the system stabilizes much quicker. This is the result of incorporating the force attenuation earlier in the optimization process.

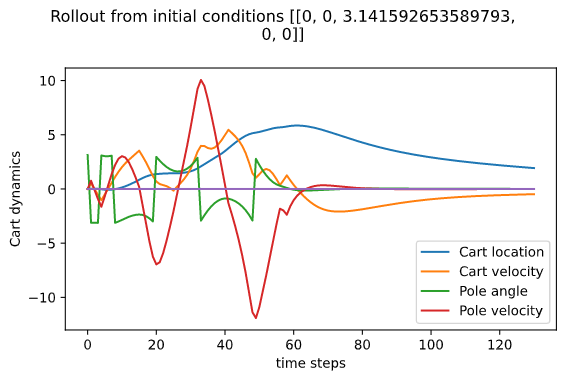
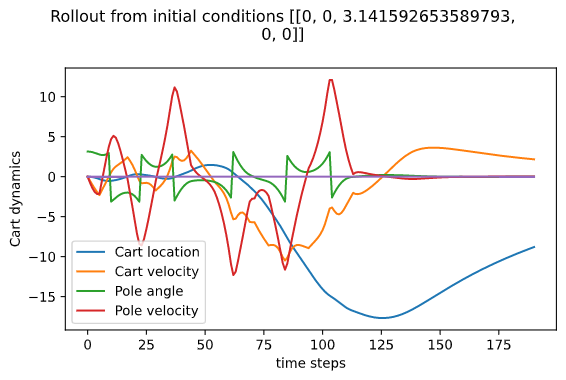


Figure 30a. Rollout with 5 basis centres and linear policy activated. This corresponds to approach 9.

Figure 30b. Rollout with 5 basis centres and linear policy activated. This corresponds to approach 10.

It is important to note that this is not a robust policy. Stability matters heavily on the initial condition. It works when the initial condition is at or near the stable equilibrium. For instance, it works for [0,0.2,π,0] but doesn’t work for [0,0.6,π,0]. This is because this solution relies on the linear policy. First of all, the non-linear policy must bring the pole angle close to 0 for several consecutive timesteps to activate the linear policy. Following that, the linear policy will only be able to stabilize the pole if the conditions are appropriate when it is activated. This means that this solution is precarious and only works with certain initial conditions.

To improve on this solution, I would try to eliminate the dependency on the linear policy. With more time, I would have explored the possibility of more basis functions and more training rollouts. It is surprising why approach 8 did not improve upon the results of approach 7.

# Conclusions

* Reducing the timestep (dt) in the Euler method leads to much better predictions for the linear predictor (Task 1).
* The non-linear model has a much lower RMSE than the linear model (Task 1 and 2).
* For the non-linear model, the number of basis centres matters much more than the number of data points. However, increasing basis centres increases accuracy with diminishing returns (Task 2).
* The linear policy developed from the real cart dynamics can stabilize the system well if initialised close to [0,0,0,0]. Using model predictive control, this precision in control is harder to achieve (Task 2).
* The loss sigma values can play a large part in reaching the desired state (Task 2).