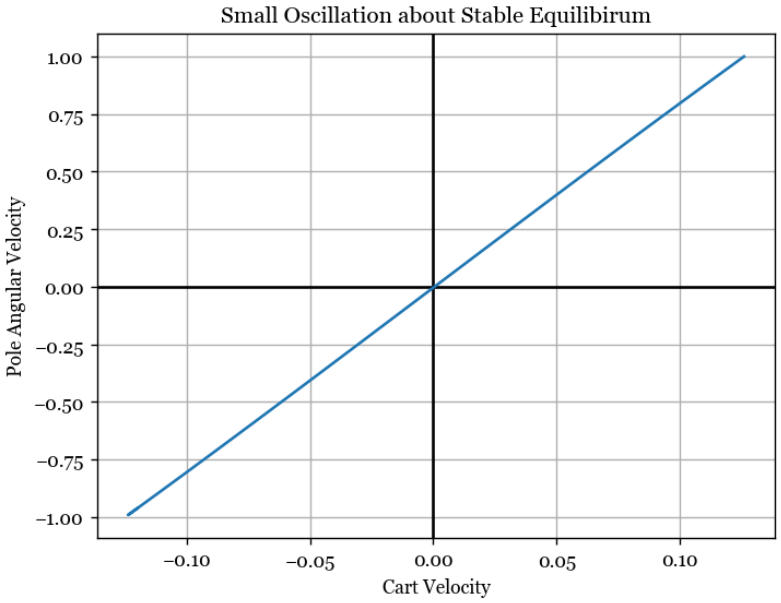
SF3: Machine Learning
Final Report

Oscar Saharoy

Abstract. Machine learning is a rapidly advancing field with applications across the world. I investigate the classic cartpole system and apply fundamental machine learning techniques to model and control the system.

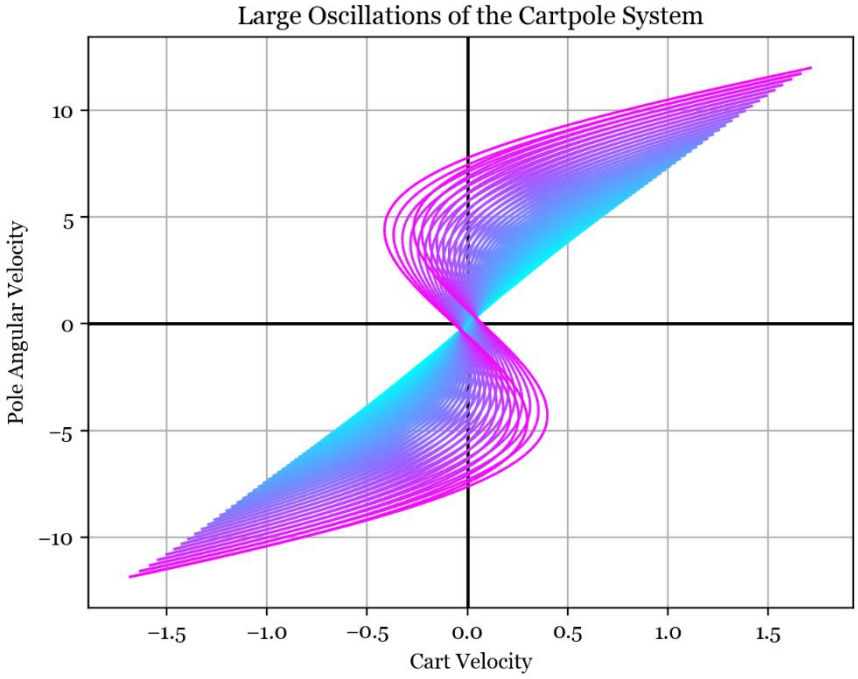
Task 1.1 - Simulation of Rollout

I established a few different initial conditions for the cartpole system and then plotted graphs to represent the resulting motions. I recorded the values of the state vector as the system evolved and then plotted some of the elements of the vector against each other at each timestep to form a curve.

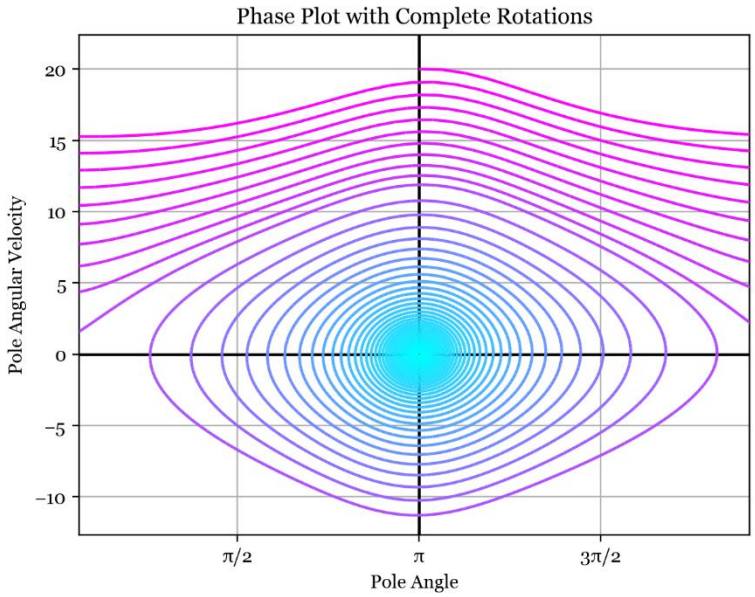


Small Oscillation about Stable Equilibrium. The plot on the left shows pole angular velocity plotted against cart velocity as time progresses through small oscillations - the system traverses up and down the blue line as the pole swings. The linear trend reflects the small angle assumption - the restoring force from the pendulum is a linear function of the deflection. From linear vibration theory we know the cart and pole vibrate at the same frequency so they "move together" and so their velocities are linearly correlated.

Large Oscillations. The plot to the right shows large amplitude oscillations, with the line turning from magenta to cyan as time progresses. The small angle approximation no longer holds, with the curve of the lines showing the tendency of the cart to change direction during the pole's swing. The plot shows how the amplitude and curvature of the line decreases as energy is dissipated and we return to the small oscillation regime.



Full Revolutions. This is the phase portrait of the pole angle and angular velocity, again starting from magenta and going to cyan. The upper region is where full rotations of the pole occur; then when enough energy is dissipated the system falls into the lower spiral towards the stable equilibrium at a pole angle of pi.

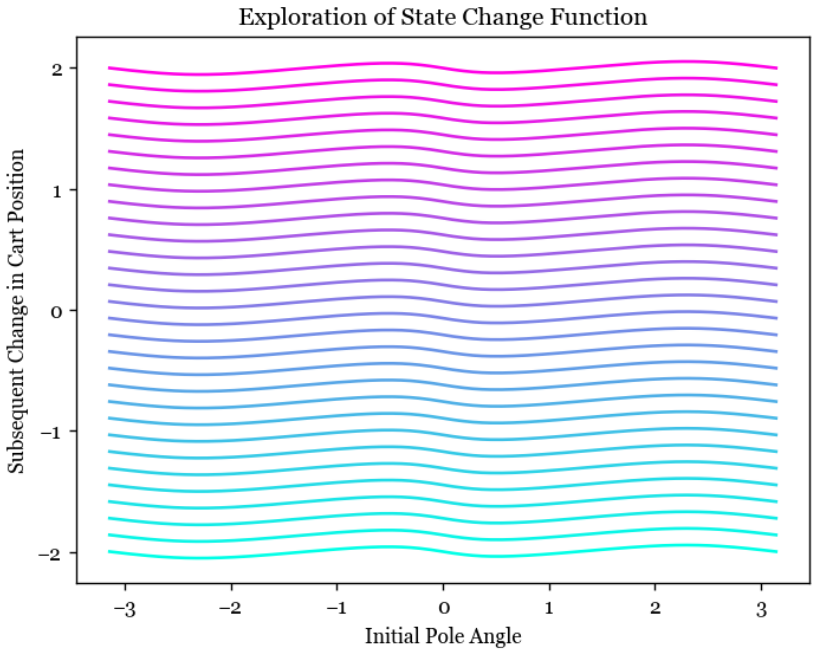


Task 1.2 - Changes in State

The change in state vector over the next time step $\boldsymbol{Y} = \boldsymbol{X}(\boldsymbol{T}) - \boldsymbol{X}(\boldsymbol{0})$ is related to the current state vector $\boldsymbol{X}(\boldsymbol{0}) = [\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{\theta}, \dot{\boldsymbol{\theta}}]$ by some complex function which I want to model. I made some plots showing how the change in state vector depends on the current state vector, noting that the change in state vector is not a function of the cart position due to the translational symmetry/invariance of the system.

It is more helpful to investigate the change in state as a function of the initial state rather than the subsequent state as a function of the current state. This is because the subsequent state will be quite similar (almost linearly related) to the initial state, assuming a small timestep. We can capture more meaning in our model by investigating the change in state, which subtracts off the information we already know (the initial state).

Varying Initial Cart Velocity and Pole Angle. In the plot to the right, I am varying initial pole angle along the lines in the x direction, and varying initial cart velocity between lines, from -10 in cyan to +10 in magenta. The y axis is the change in cart position. We can see that the lines are equally spaced, so the change in cart position seems to be a linear function of cart velocity which makes sense.



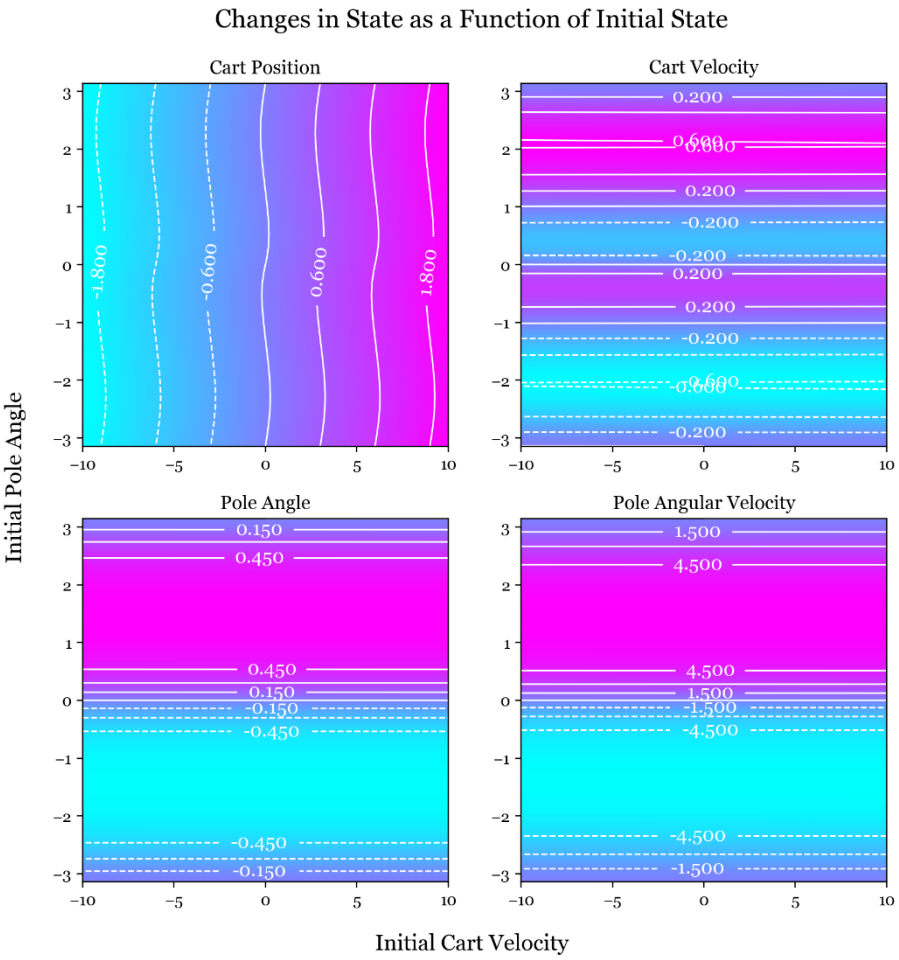
However, the change in cart position is a slightly nonlinear function of the pole angle since there is a wobble in the lines. Plotting the changes in other state variables in this way yields coincident lines which are better represented in the contour plot below.

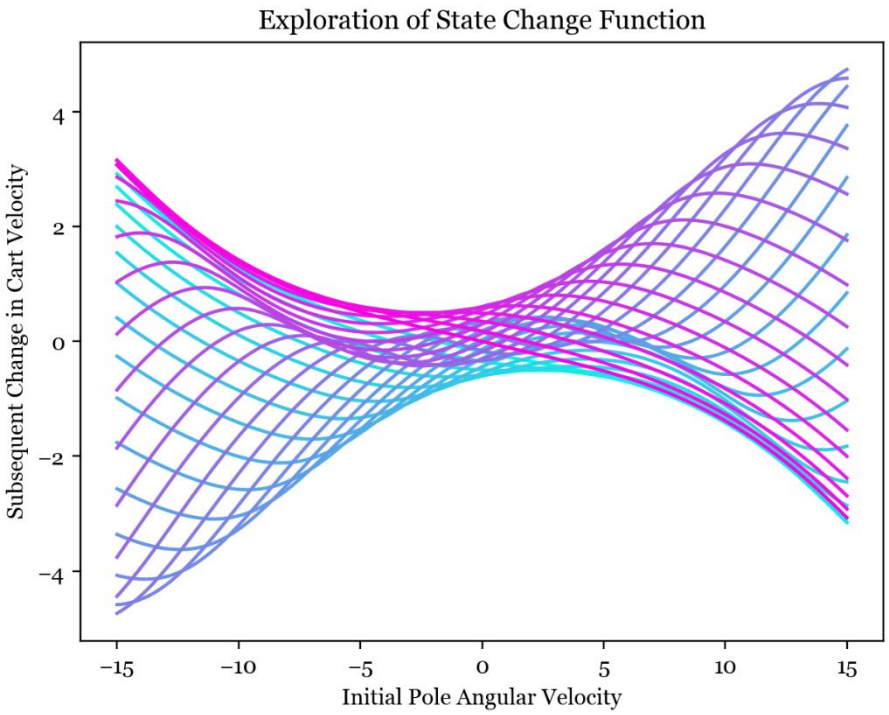
In this plot, the initial cart velocity varies in the x direction and the initial pole angle in the y direction. The subsequent changes in the state vector are shown as contour plots. The changes in cart velocity, pole angle and pole angular velocity are not functions of the initial cart velocity. This must be the case because any initial cart velocity can be achieved by assuming a different inertial reference frame, leaving the rest of the dynamics unchanged.

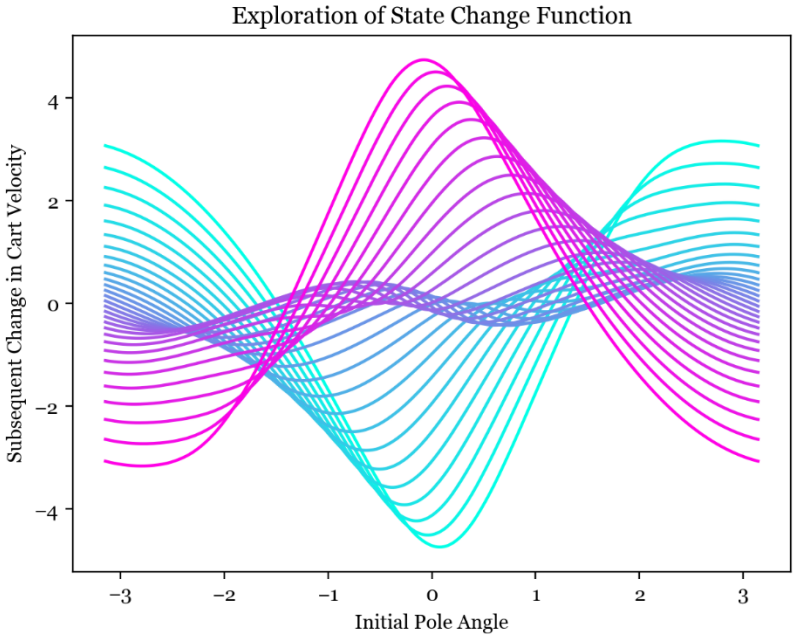
I also tried different initial conditions for all contours plots and found that the structure was similar but with slightly different values.

The same reasoning leads to the contour plots for varying cart velocity and pole angular velocity consisting of straight lines so I will move to the more interesting combination of varying pole angle and angular velocity.

Varying Initial Pole Angle and Angular Velocity. The plot to the right shows how the change in cart velocity at the next time step is a highly nonlinear function of the two variables that we are changing in the plot - the initial pole angle and angular velocity. The more cyan curves are an initial angle around $-\pi$ and the more magenta curves are an initial angle around π.

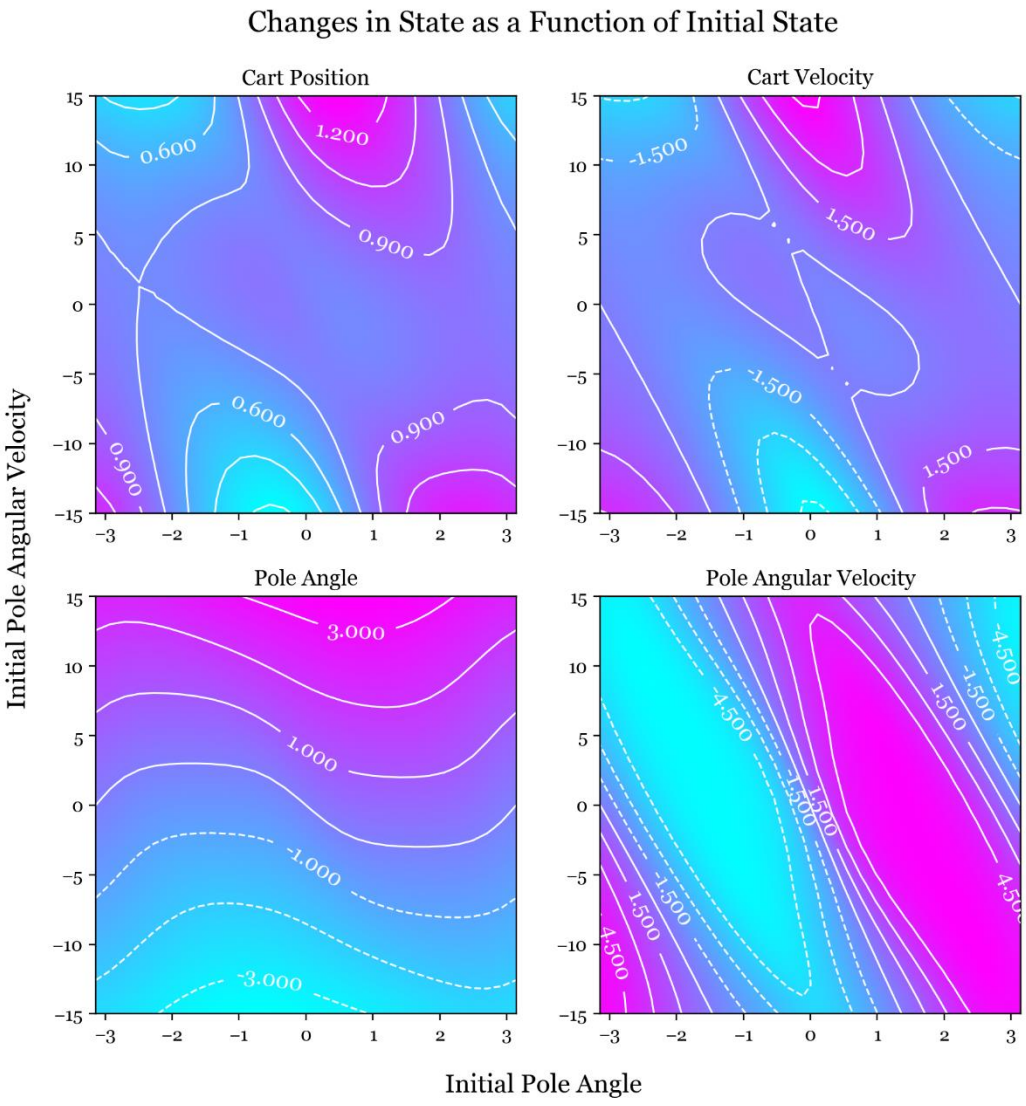






This is the same as the plot above but this time plotting change in cart velocity as a function of initial pole angle, varying the initial angular velocity from -15 in cyan to +15 in magenta. It looks like a linear estimator of Y from X will perform poorly, but looking at the central, dense part of the graph above (which represents a small initial angular velocity) we can see the curvature of the lines are much smaller. Keeping in mind the truncated Taylor series approximation to a function for small deviations, there is some scope to use linear estimation especially for small deviations about an equilibrium.

In these contour plots I vary initial pole angle in the x direction and initial pole angular velocity in the y direction and show the changes in the state variables in the contour plots. It is quite hard to find linearity within. However, it makes sense that the change in pole angle appears to be a mostly linear function of the pole angular velocity, with some dependence on the pole angle due to gravity acting on the pole.



Task 1.3 – Linear Model

Finding the Least-Squares Fit. I assume that the change in state is a linear function of the current state:

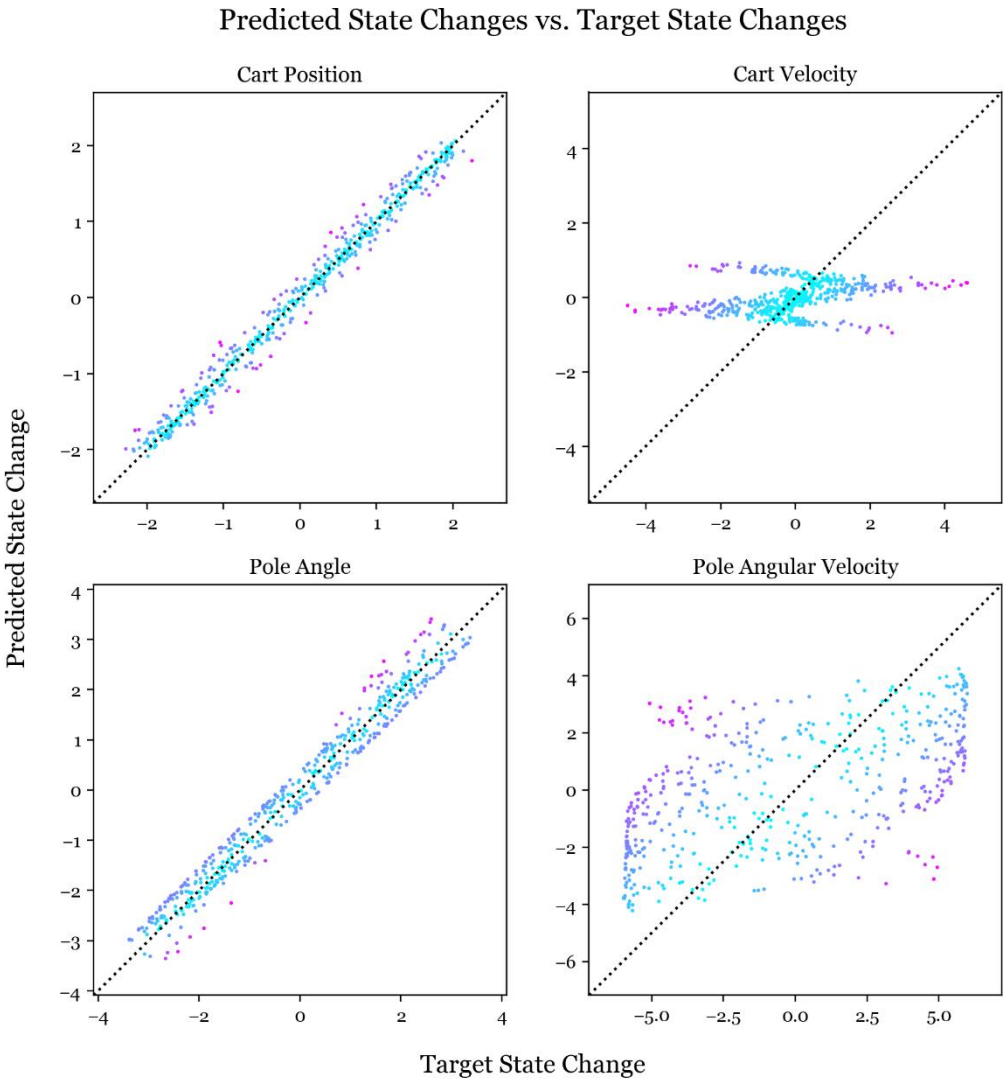
$$\boldsymbol{Y = f(X) = XC}$$

where $\boldsymbol{C}$ is a 4x4 matrix. First, I generated 500 input/output pairs by randomly generating initial states and then performing 1 step (one call to `perform_action`) to generate subsequent states, and then used the Moore-Penrose inverse of $\boldsymbol{X}$ to find the least-squares linear fit.

$$\boldsymbol{Y = XC}$$
$$\boldsymbol{X^{+}Y = X^{+}XC = C}$$

I then investigated the linear predications and compared them to the analytic target data.

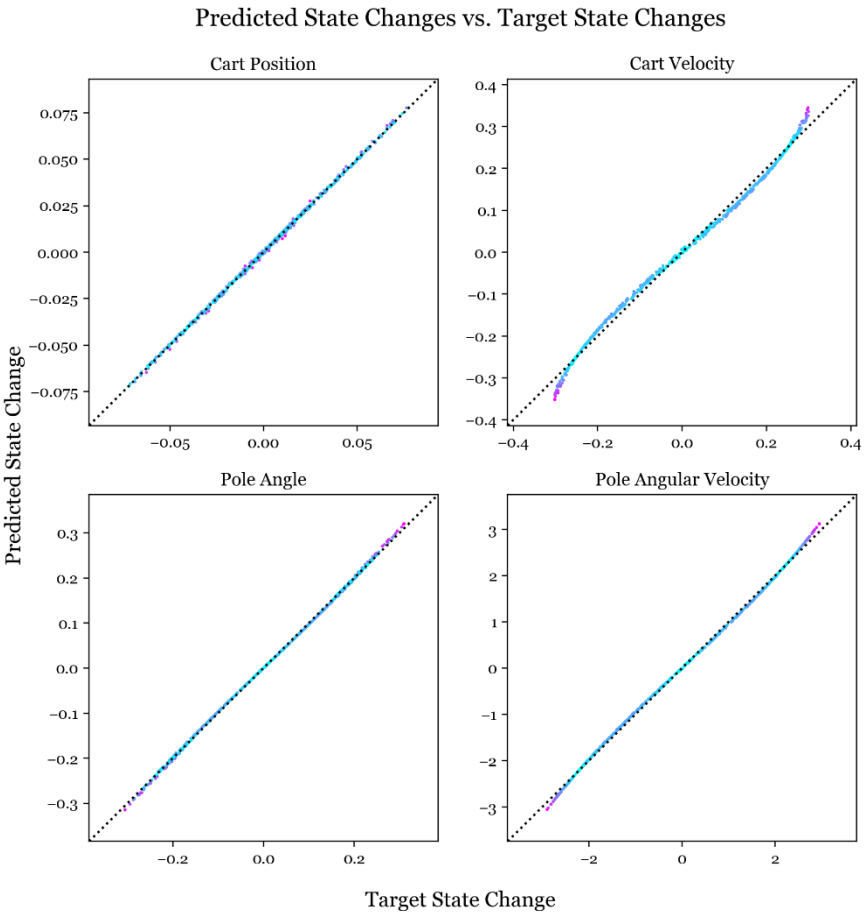
Predicted State Change vs. Target State Change. Here I have plotted one point per training data element on a graph for each element in the state change vector. For each point, the x coordinate is the target change in state as returned from `perform_action` and the y coordinate is the predicted change as returned by the linear estimator. So for perfect predictions these would be equal, and the points would all lie on the line $\boldsymbol{y = x}$ shown in dotted black.



The changes in cart position and pole angle are well predicted by the linear estimator, with the points lying close to the ideal line. The change in cart velocity as least has the central mass of points lying near the ideal line, but with many outliers at the edges, while the change in pole angular velocity is very poorly explained by the linear model with the points avoiding the ideal line.

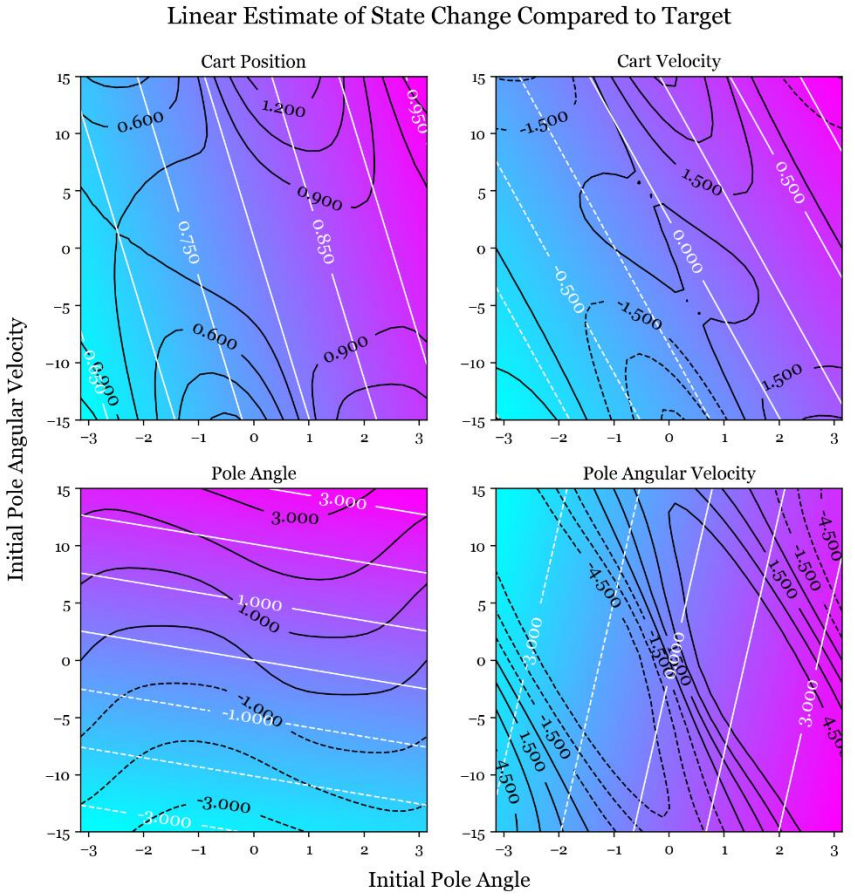
Linear Model for Small Deflections Only. I repeated the above procedure but limited the training data to small deflections from equilibrium (initial states between -0.25 and 0.25). Thinking back to the Taylor series approximation for a function I expected this scheme to yield better predictions near the equilibrium.

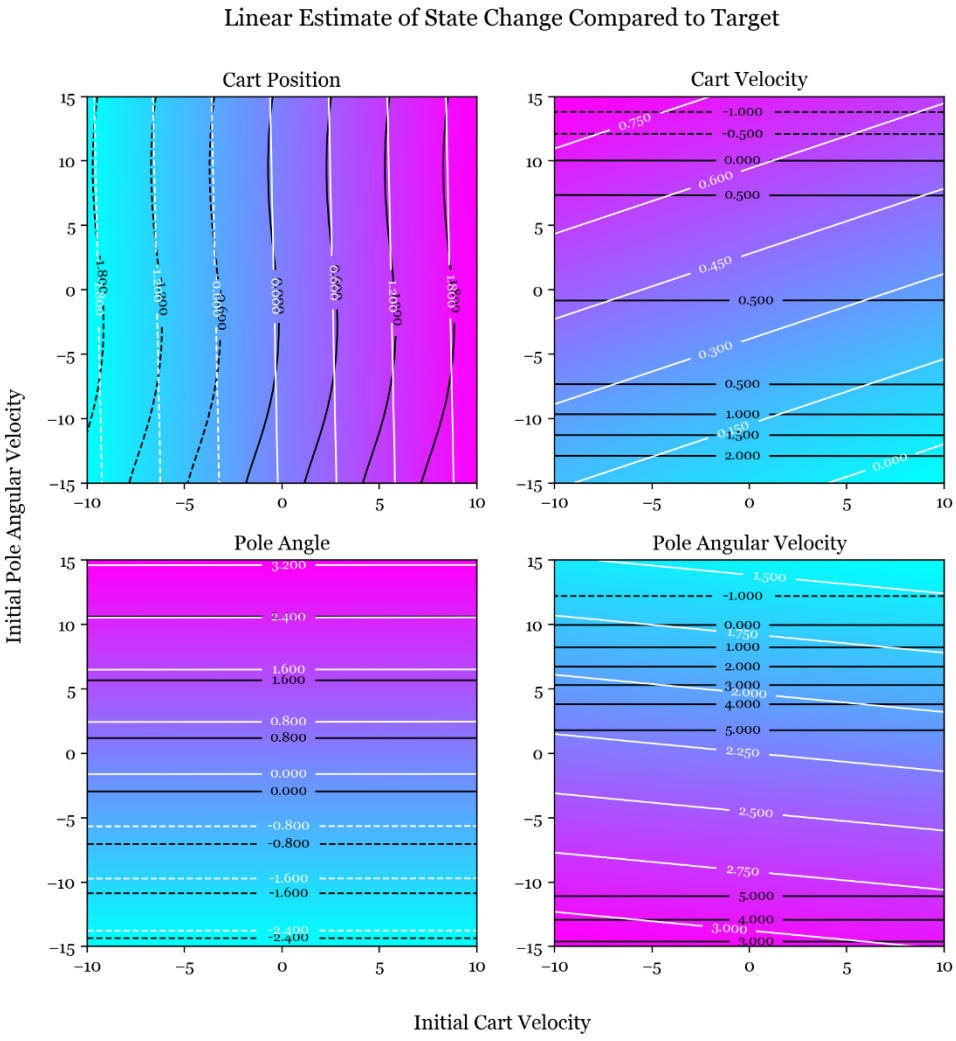
The linear model works nicely for small deflections as shown opposite, with all the points near to the ideal lines. Perhaps we could build a good model by using many linear models centred around different points, which provides motivation for moving to a nonlinear model.



Prediction Contour Plots vs. Target Contour Plots. I generated 4 contour plots, shown bottom right, for the predicted state change of each the state vector elements as predicted by the linear model. I am varying the initial pole angle and angular velocity again, so I am aiming for these contour plots to be identical to the final 4 I plotted in task 1.2. Note that I am back to using the linear estimate for the entire range of data.

The contours from the linear estimator are in white and the target contours are in black. It seems the nonlinearities in the data have been discarded and the linear estimator has picked up on only the lowest-frequency linear trends. I would expect this model to miss the nuance in the motion of the cartpole - especially the nonlinear rotational dynamics of the pole.





However, the linear estimator makes some good predictions when contours are plotted over varying initial cart velocity and pole angular velocity as shown left. This is the pair of variables I didn't plot contours for in task 1.2, but here we can see how the two trends on the left are closer to linear and the white and black contour lines are in close agreement. This mirrors the results from the scatter plots where the predictions for change in cart position and pole angle were most accurate.

Prediction Accuracy. The elements that are well predicted by the linear model are the change in cart location and pole angle. The change in cart location will be quite similar to the velocity of the cart times the timestep (neglecting acceleration during the timestep), so it is expected that we could predict this as a linear function of the velocity. The change in pole angle is similarly related to the angular velocity.

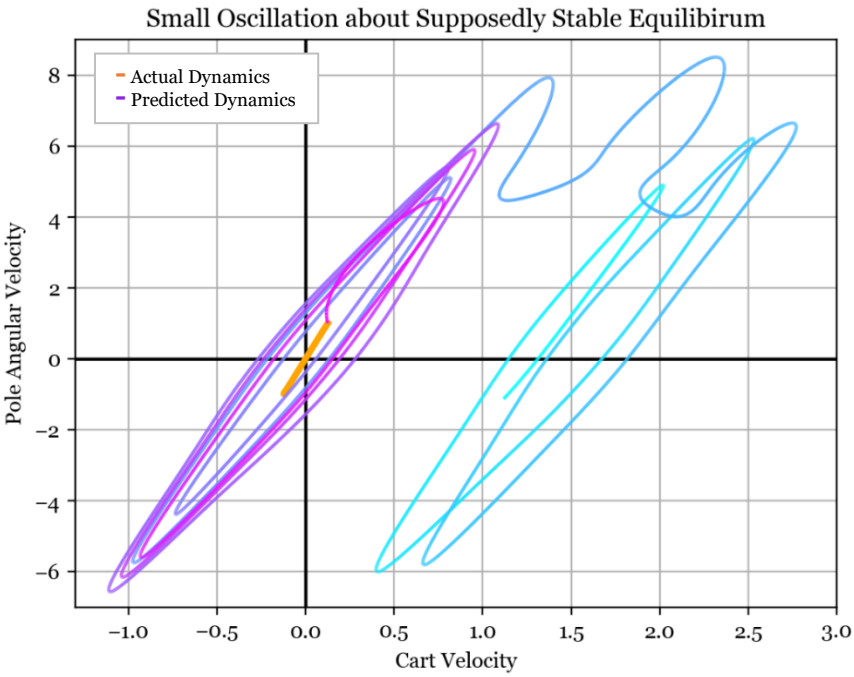
Task 1.4 - Prediction of System Evolution

Now the linear model can predict the system evolution by incrementing the current state by the prediction for the state change repeatedly. This can be used as an alternative to the analytic system dynamics, so I have made similar plots to task 1.1 but now using the linear model rather than the actual dynamics.

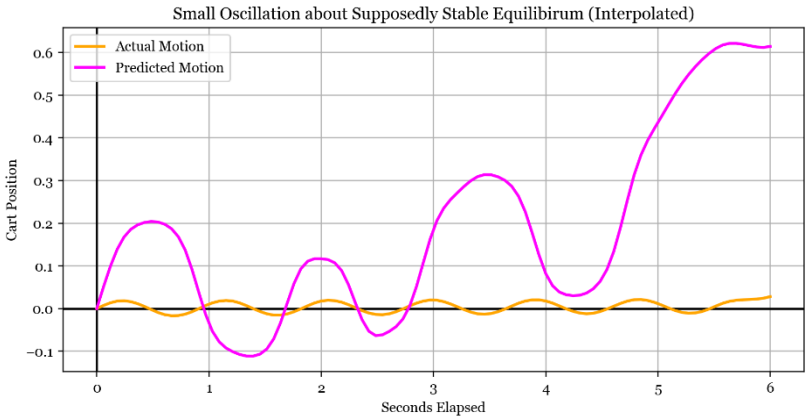
Remapping. It is vital to remap the pole angle to the range $-\pi$ to π during this process. If this is not done, the angle grows beyond anything seen in the training data and the result is an abnormally large input of energy to the system which causes divergence. If we don't remap, the state after $\boldsymbol{k}$ steps $\boldsymbol{X_{n+k} = X(C + I)^k}$ and diverges since $\boldsymbol{C + I}$ will likely have eigenvalues greater than 1.

Small Oscillations. The plot below and right is initialised the same as the small oscillations case from the first plot in this report, and shows the time evolution of the pole angular velocity plotted

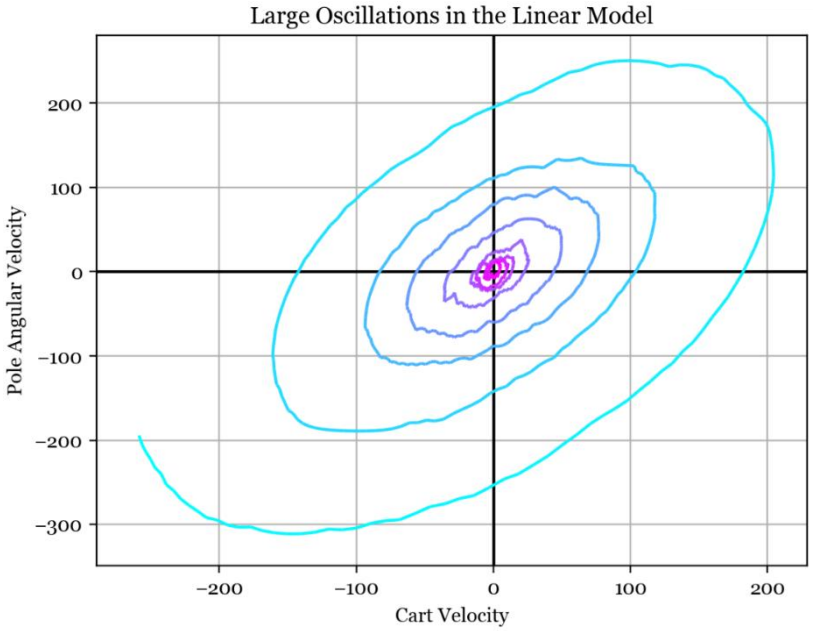
against cart velocity. The target behaviour is shown in orange, which is the data from the small oscillation plot using the analytic dynamics. The linear estimation time evolution is the curve that starts in magenta at initialisation and turns more cyan as time goes on.



The linear model successfully brings the system into an oscillating state, shown by the loops about the origin, although these oscillations are of much larger amplitude than they should be (the loops are larger than the orange line). After a few oscillations, the system spontaneously shifts into a regime where it is still oscillating but also moving to the right, shown by the loops shifted to the right. The random increase in kinetic energy indicates that conservation of energy is a nonlinear detail in the data that the linear model struggles to capture. It's important to note that this is a particularly lucky result based on favourable random generation of training data - most plots I made with other training data explosively diverged from the oscillating regime in fewer time steps.



The left plot shows the predicted and actual cart position over time. The linear model has managed to capture the oscillatory motion, but the amplitude and frequency are incorrect and the position diverges as time progresses.



Large Oscillations and Complete Revolutions. This plot is initialised with a larger pole angular velocity so that the oscillations start larger and grow to full revolutions. As time progresses the line goes from magenta to cyan, and both the pole angular velocity and cart velocity immediately diverge, reaching large values. However, the spiral structure conveys some understanding that a larger cart velocity implies a smaller pole angular velocity.

Task 2.1 – Nonlinear Modelling

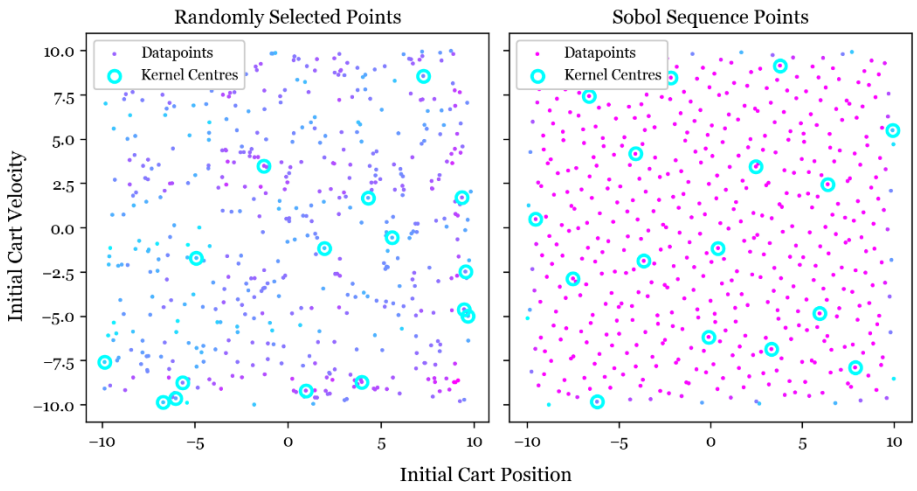
To improve the linear model, I will try to express the state change from a given state $\boldsymbol{f(X)}$ in the form:

$$\boldsymbol{f(X) = \sum_i \alpha_i K(X, X_i)}$$

This is a nonlinear model where the state change is found as a weighted sum of radial basis functions. Basis function $\boldsymbol{i}$ is centred at $\boldsymbol{X_i}$ and has weights $\boldsymbol{\alpha_i}$, and the basis function $\boldsymbol{K}$ is a Gaussian. The weights are found by regularised least-squares regression, so the regularisation constant $\boldsymbol{l}$ and the length scale of the Gaussians are hyperparameters to be tuned.

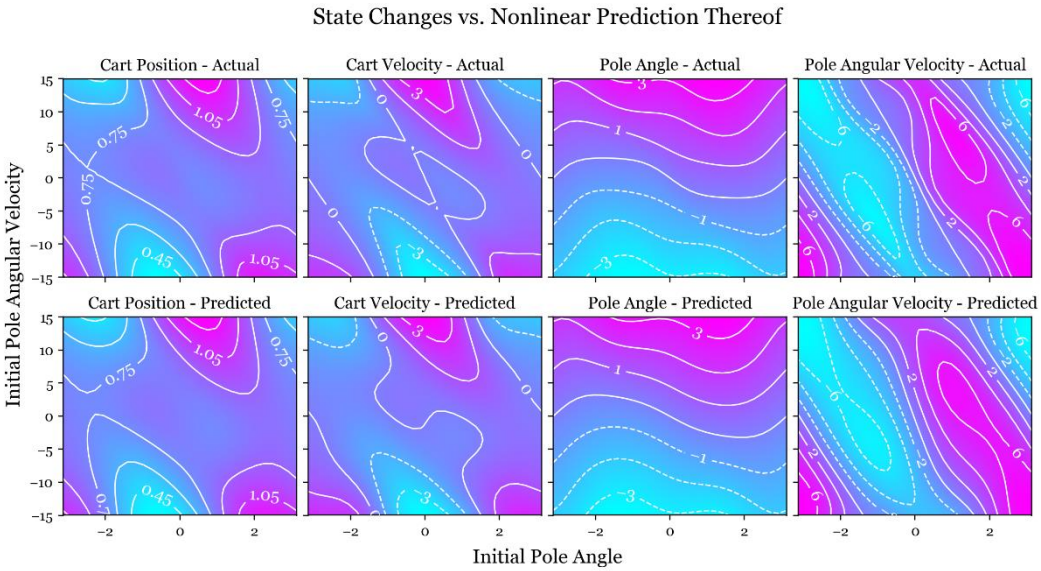
Generating Training Data. Now that I wish to improve the state change model with nonlinear basis functions, the first task is to generate training data. For task 1.3 I used uniformly distributed random datapoints, but this time I am using a Sobol sequence to generate more evenly spaced data.

Sobol Sequence. These are scatter plots of the datapoints generated by the two methods, projected onto the plane formed by the first two state vector dimensions. The dots represent datapoints, and the circled points represent basis function centres. The randomly selected points can be bunched up in some areas, and absent from others. The Sobol sequence provides much better coverage of the state space, meaning more useful information makes it into the model.



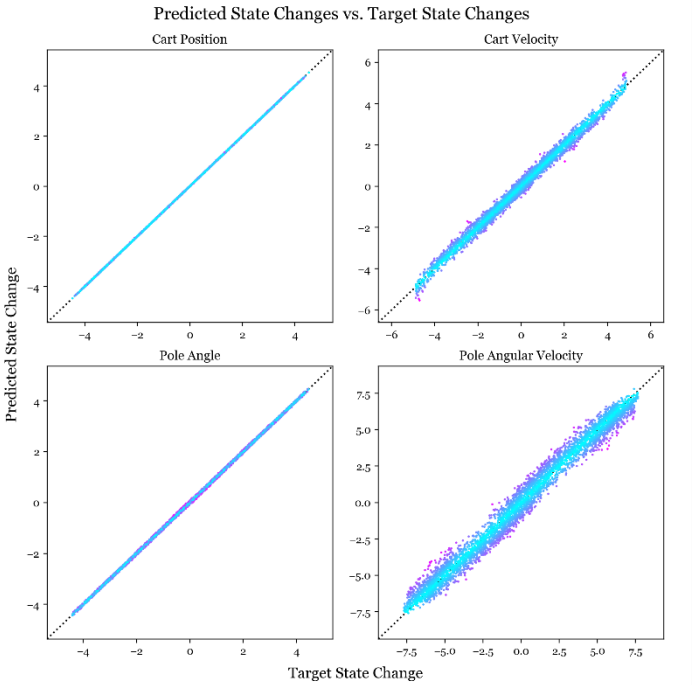
Selection of Kernel Centres. The Sobol sequence method has another advantage: the first few generated points are also evenly spaced over the state space so they work well as kernel centres (if two kernel centres were next to each other, they would explain the same part of the data so one would be enough). This works best when the number of kernel centres is a power of two; above I have selected the first 16 points as basis function centres. The left plot shows the result of random basis function centre generation, where some kernel centres are very close to each other and there is a large gap in the top left which would lead to poor modelling in that region.

Evaluation of Fitted Model. Below are the contour plots of the nonlinear predictions for the state changes, across varying initial pole angle and angular velocity. The nonlinear structure has been captured by the model quite well but it did take a large amount of data and many kernel functions to achieve this. The hyperparameters sigma and lambda were identified by trial and error. Improvements

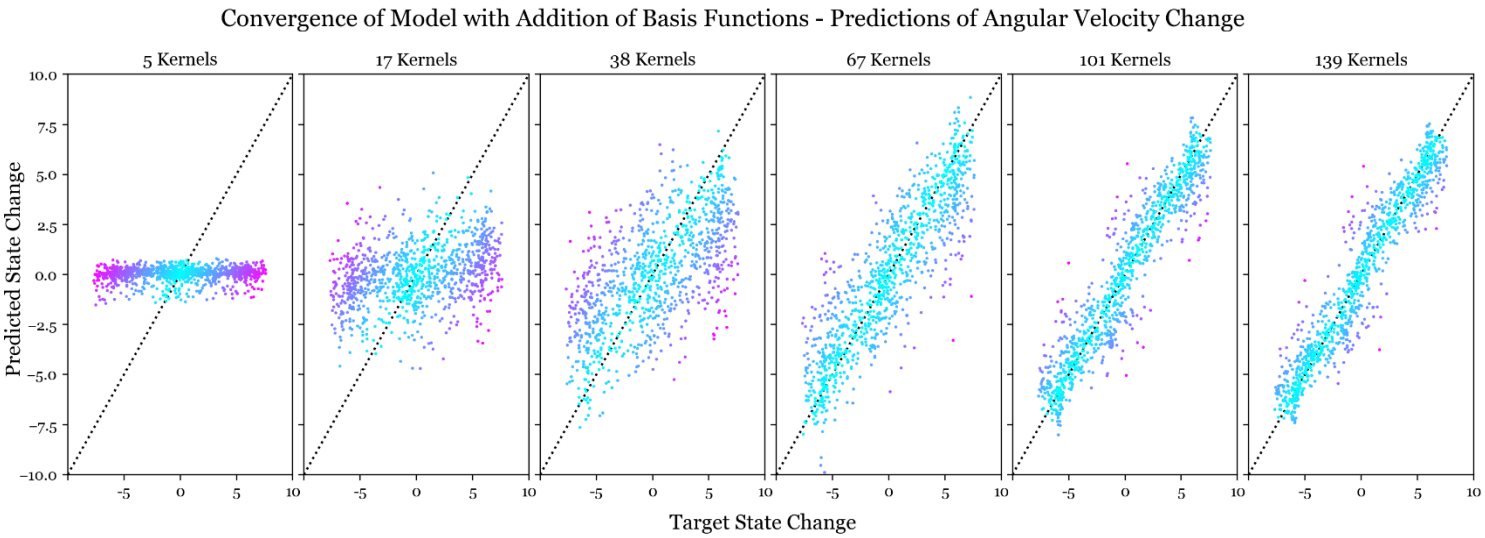


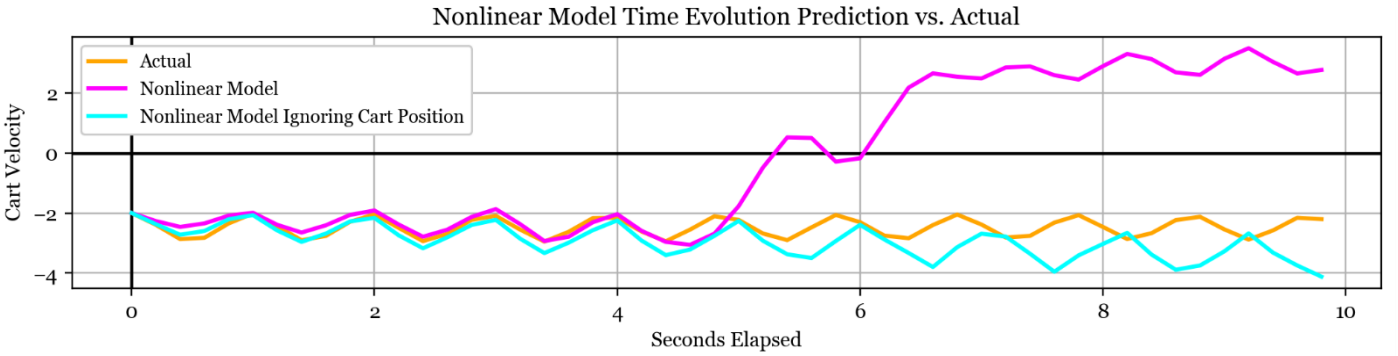
to the process might include collecting data further outside the region we would like to make predictions in, pushing errors further away from where we are using the model for prediction. Another improvement could be to collect more datapoints in regions with larger errors, or assign different "weights" to different data points, making them more or less important for the model to explain well.

To the right I have plotted the target state change against the predictions from the nonlinear model. The points are all lying much closer to the ideal line y=x, so we can expect this model to perform much better than the linear model from task 1.3.



Effect of Increasing Basis Function Density. Below I have plotted the prediction for change in pole angular velocity against the actual change for increasing numbers of basis functions. As the number of basis functions increases the fit converges, but plateaus before achieving a perfect fit. Increasing the number of datapoints fed into the model has a similar effect, but fitting the model with more datapoints provides a modelling improvement that only increases computation time during fitting whereas increasing the number of kernel centres increases computation time during fitting and model evaluation.

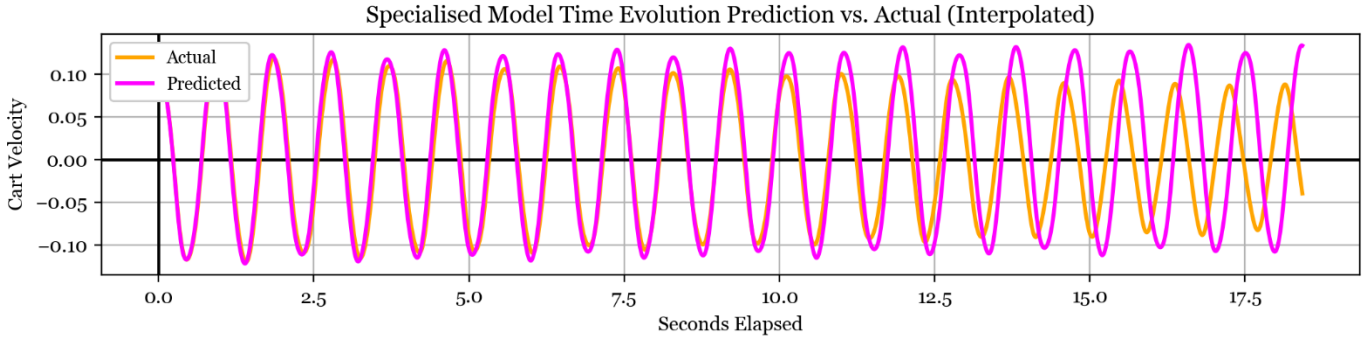




Nonlinear Model Time Evolution. Above, I have plotted the time evolution of the system as predicted by the nonlinear model in magenta, and according to the analytic equations in orange. I have plotted the evolution of the cart velocity only because I found while experimenting that if the time evolution of one of the states matches then they all will, and divergence of one state indicates total divergence. Therefore all the meaning I was interested in was contained in the time evolution of a single element of the state vector, and allows me to save space for the final report etc.

There is reasonable agreement up to 5 seconds, and disagreement beyond. The slight mispredictions up to 5 seconds are due to accumulating errors in the model, but the deviation at 5 seconds is indicative of a weakness in the model. At this point, the cart has been moving left at about 2 metres per second for 5 seconds, so has moved 10 metres. The training data and basis function centres were all generated with cart positions within -10 to 10 in state space, so after 5 seconds the cart has moved away from all the kernel centres into a region of the state space that is necessarily badly described by the model.

Model Ignoring Cart Position. So, I adjusted the kernel function to ignore the cart position dimension (which has no effect on the change in state anyway) and refitted the model. The resulting prediction is shown in cyan and is not an exact match to the real dynamics in orange but exhibits more similar overall behaviour. It is also worth noting that when in use, the controller will keep the cart position close to the origin so this adjustment may be inconsequential in that case.

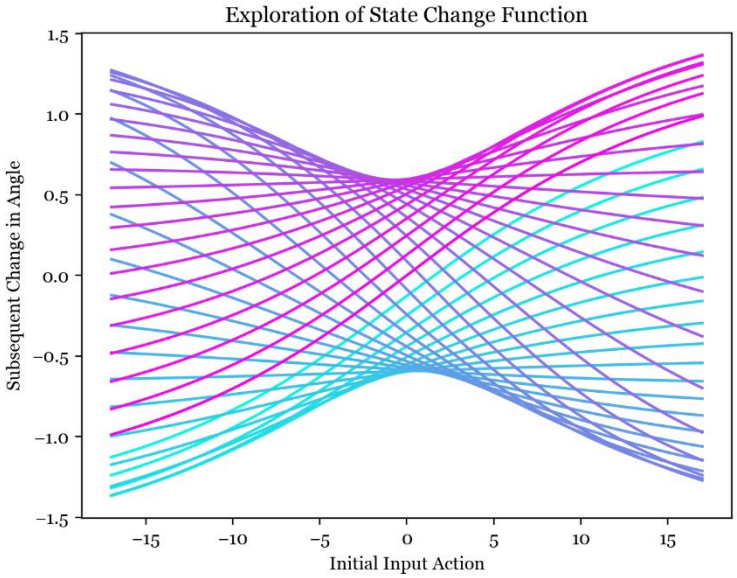


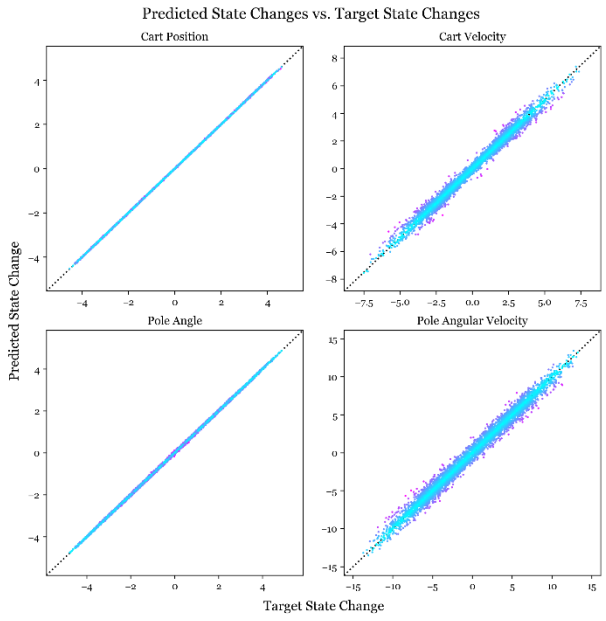
Descoping the Model. In order to improve the model's predictions, I restricted the training data to the range relevant to small amplitude oscillations and then examined the time evolution of the predictions exclusively within that same regime. The plot above shows the actual dynamics in orange, and the model predictions in magenta. The predictions are very good up to about 10 seconds, so training the model on only a subset of the data improves predictions in that neighbourhood significantly. Beyond about 7 seconds the real oscillations start to decay - the model struggles to capture this behaviour as it occurs over a larger timescale, while the model can predict only one timestep forward.

Nonlinear Model Evaluation. Above I found that the model is able to give agreement for many oscillations with a bit of tweaking, although errors always accumulate. This is to be expected: we trained the model with inputs and targets that were only 1 step apart, so intuitively the model can predict a single time step well but has no understanding that repeated evaluation amplifies undesirable errors. Although we see good predictions from the pointwise accuracy tests, the time evolution exacerbates these errors.

Task 2.2 – Modelling Effect of Force Input

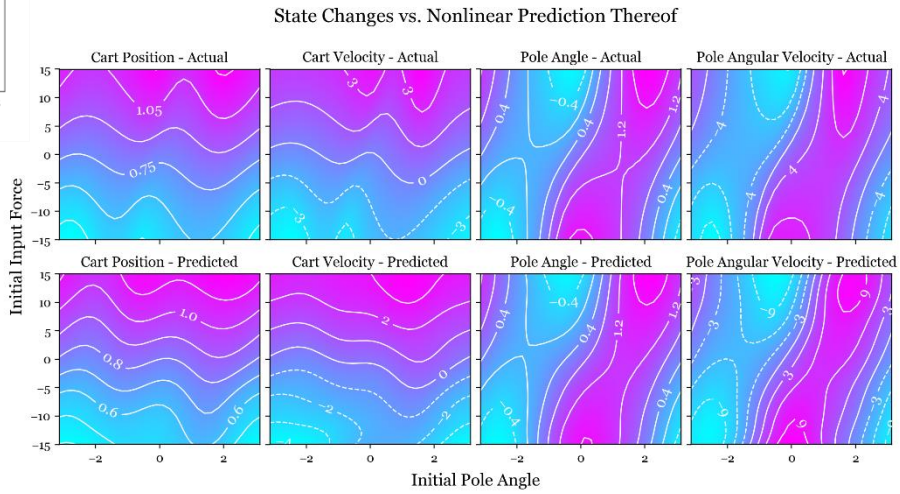
Effect of Input Force. The system now includes an input force, meaning the state space is one dimension larger. The input force affects the cartpole dynamics within the `perform_action` function, so by controlling this input it will be possible to hold the pendulum at its unstable equilibrium. In order to visualise the state change function with the force input, I plotted this graph which shows how the initial angle and input action affect the subsequent change in angle. The different lines are different initial angles, from -pi in magenta to pi in cyan, and the input force is varying along the lines.



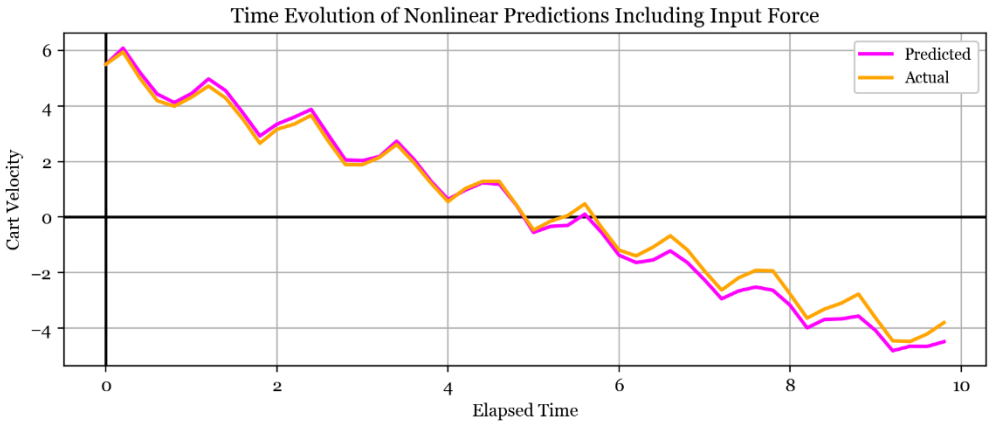


Target State Changes vs Predictions, Including Force. I was able to use essentially the same code as for task 2.1 for fitting the model including the force input, so the results are very similar; for example in these scatter plots comparing target to predicted state change the points again all lie close to the ideal line.

Below I have plotted contours of the target state changes as well as the predictions from the nonlinear model. I have varied the initial pole angle along the x axis and initial input force along the y axis, and there is good agreement between the two sets of contours. However, discrepancies are easier to find in these plots than in those without the force input.



Time Evolution Modelling. Below is the time evolution of the cart velocity with predictions in magenta and analytic in orange; the close agreement shows that the model has understood this motion well. Although this initial condition leads to good agreement, others lead to divergence more quickly. We might expect the predictions when including the force input to be worse since the state space has become much larger; I have set the input force to be between -15 and 15 in training data so the additional travel in this dimension effectively reduces the data density and basis function density in the state space. In other words, the model has to fit a larger and more complex state change function with sparser information about it and using the same number of kernels. Using more basis functions and datapoints to compensate for this effect makes the fitting process take much longer.



Task 2.3 – Linear Control

Now I am interested in controlling the cartpole system using a linear policy $\boldsymbol{p}(\boldsymbol{X}) = \boldsymbol{p} \cdot \boldsymbol{X}$ to set the action at each timestep. The vector $\boldsymbol{p}$ is determined by minimising a loss function $\boldsymbol{L}$, defined by summing a point loss function evaluated at each time step over a certain timeframe:

$$\boldsymbol{L} = \sum_{i=1}^{N} \boldsymbol{l}(\boldsymbol{X}_i)$$

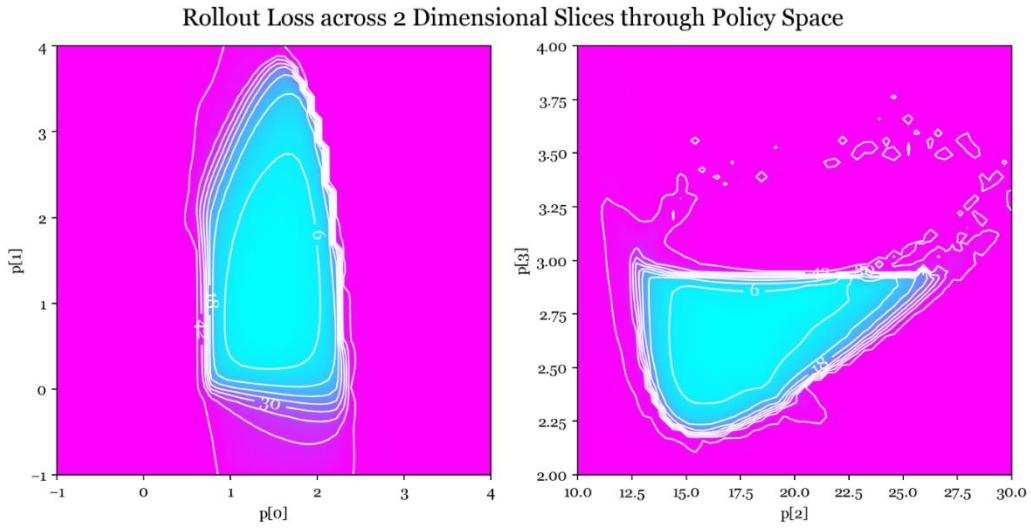
$$\boldsymbol{l}(\boldsymbol{X}) = \mathbf{1} - \exp\left(-\frac{|\boldsymbol{X}|^2}{2\sigma_l^2}\right)$$

First, note that finding L involves repeated evaluation of the cartpole dynamics as defined in `perform_action`. To speed up the fitting process, I reimplemented the Euler integrator using the RK4 method which improves accuracy and executes in roughly half the time – for details see `perform_action_RK4(state)` in `CartPole.py`.

Also note that the provided loss function can ascribe a large loss to a cartpole in the desired configuration if the pole angle is not zero but actually a multiple of 2π. Therefore an improved loss function could be periodic in the pole angle variable, similar to the correction used in the kernel functions in task 2.1. The same is true for the linear policy: $\boldsymbol{p} \cdot \boldsymbol{X}$ could be nonzero in the desired configuration if the pole angle is a multiple of 2π.

Next, I implemented the loss function L and created some contour plots to visualise how it varied across the policy space. I then used `scipy.optimize.minimize` with the Nelder-Mead method to find an optimal setting of the policy vector p. I made several calls to the minimize function, each starting

from a different initial value of the p vector, and took the one that gave the smallest loss. I also implemented an optimisation where the loss function L would return the maximum value if any of the elements of the state vector grew too large. This helped to reduce the number of `perform_action` calls that needed to be made as poor policies were identified earlier in the process.

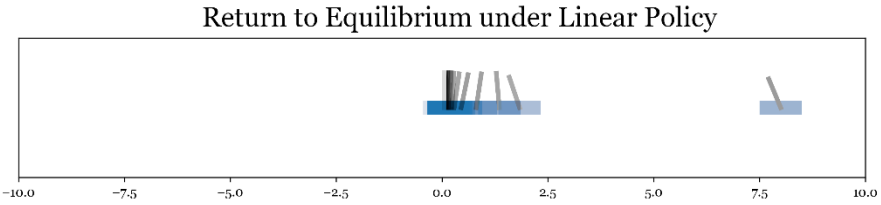


Loss Contour Plots. Above I have plotted the total loss evaluated at values of the p vector across two orthogonal slices through one of the minimums. The optimal value of the p vector found by the optimiser is the intersection of these two planes. I was lucky to find this minimum and will probably not find any more since the policy space is so large and the portion where the loss isn't maximal is a tiny fraction of it - magenta shows maximal loss and cyan anything below.

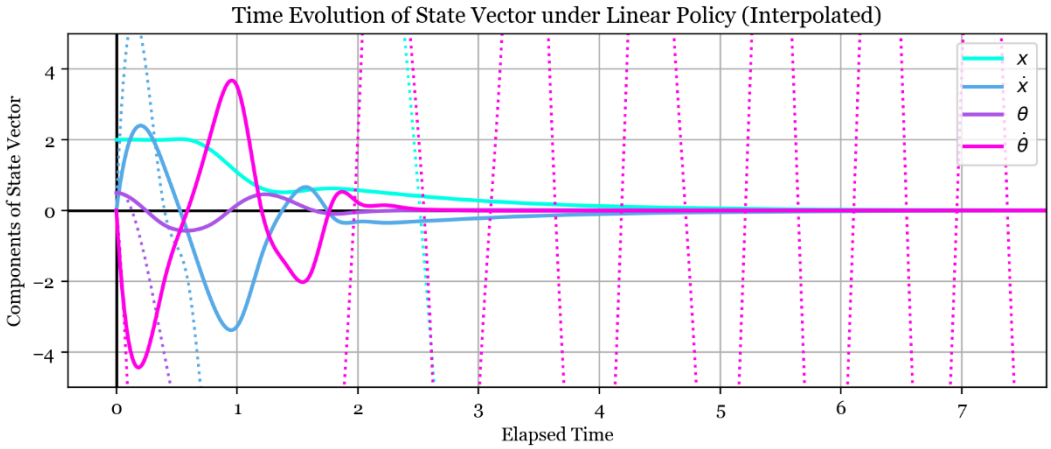
One thing about this loss function is that there are large areas with very small gradients, which would make classical machine learning techniques like gradient descent difficult to apply. One possibility is to try the log of the loss function which might help regularise the gradients, or adjust the loss function so that it doesn't reach a maximum as the current one does. I used the Nelder-Mead method so that the gradients weren't a factor, but noted that a lot of bad policies were found with this method. Using many different starting locations allowed me to find decent policies but if this hadn't worked I would next try to adjust the definition of the loss function to help the optimiser work better.

Stabilised Range of Initial Conditions. The optimised policy vector that my search returned is p = [0.76, 1.44, 17.06, 2.66]. This setting of p is able to stabilise the cartpole from initial angular displacements around 0.6 and initial cart displacements around 2 metres. These ranges can also be increased by favourable initial linear and angular velocities. Increasing the initial cart displacement beyond this resulted in a large initial force that caused the system to lose control, as the contribution from the cart position became too large. However, we can leverage the translational invariance of the system to avoid this: by clipping the first element of the state vector between -2 and 2 just before it is fed into the policy function $\boldsymbol{p}(\boldsymbol{X})$, the policy thinks it is only at most 2 metres from the equilibrium. So it still moves the cart towards the desired state at the origin, but without overdoing it. However, this is no longer a truly linear model.

I created a little graphic showing how my optimised policy brings the cartpole to the desired state, starting from the extreme right.



State Vector Evolution with Time. In the plot below each solid line is one component of the state vector as they vary under the linear policy, starting from an intial state X = [40, 0, 0.5, 0]. All the components are brought to zero as desired and you can also see the horizontal part of the cyan line (representing the cart position) where the position clipping is coming into effect. The dotted lines show the states without the position clipping, and are highly unstable.



Task 2.4 – Model Predictive Control

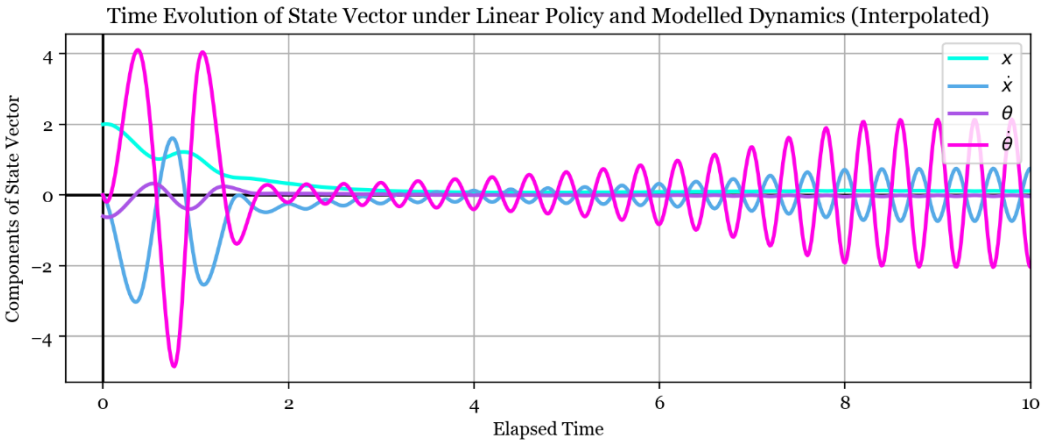
I tried to use the same optimised policy vector p to control the modelled dynamics, and found that the same policy vector was able to bring the modelled system to the desired state but took slightly longer to do so.

Model-Based Optimisation. I changed the rollout loss evaluation function to use the modelled dynamics, and then re-ran the optimisation process to find a new policy vector noting that the process was much faster because the model is cheaper to evaluate than the actual dynamics (this provides motivation for performing model predictive control). Thinking that the model dynamics aren't too different to the real ones, I used the previous optimal policy vector as the starting point for the optimisation and found the following optimal policy vector: p = [2.42, 3.82, 22.41, 3.96].

Below I have plotted the time evolution of the state vector under the model-optimised linear policy, and using the modelled dynamics. Although the cart and pole positions are relatively stable at the desired configuration, there are oscillations in the linear and angular velocities about the equilibrium, although these stop growing after 8 seconds. These oscillations are of period 0.4s so are formed by overshoot and correction during 2 subsequent calls to `perform_action`; I think these are acceptable as they don't cause the cartpole to really deviate from the desired configuration.

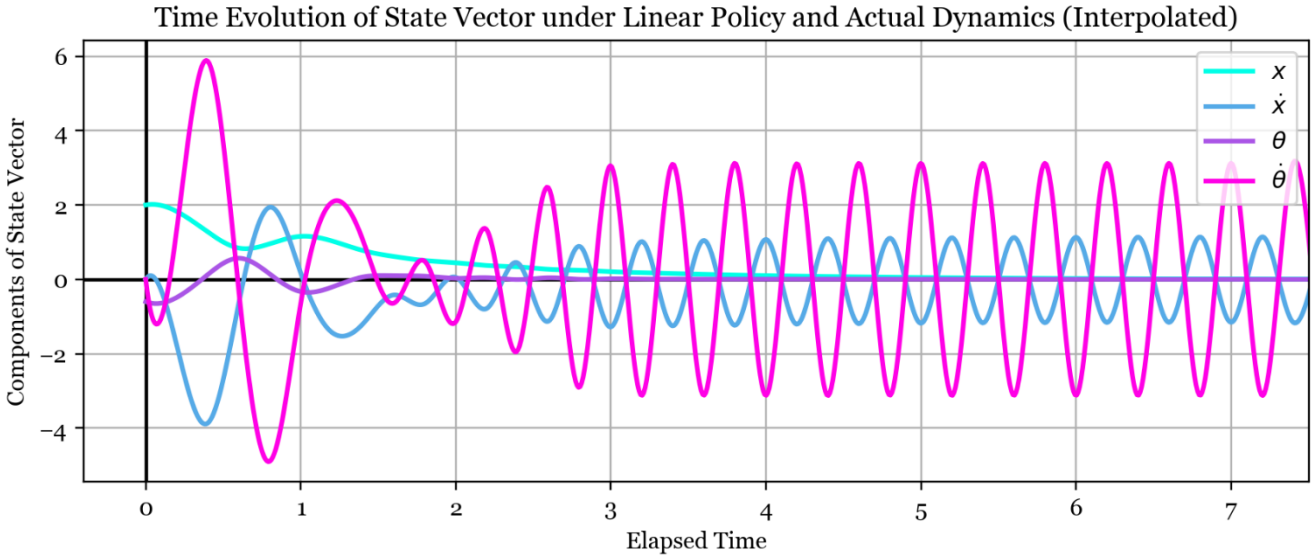
However, if I wanted to avoid this oscillation the loss function could be adjusted to discourage it. For example, the total rollout loss function could ignore the first few seconds and only include states afterwards. This could select for policies that take a bit longer to get the pole upright and stable, but are

subsequently able to hold it completely still. Other adjustments to the loss function could include penalising force input also, to encourage efficient control.



In general, the model predictive control method has not performed as well, with these oscillations proving very hard to avoid. The other big problem with the model-optimised policy is that it's much more sensitive to initial conditions: carefully selected small initial displacements can lead to divergence. So it seems that using the model for dynamics has caused the policy to miss some of the generality that the previous policy was able to capture, maybe due to the model being ignorant of certain features of the true data.

Return to Actual Dynamics. I then tried to use the policy vector optimised on the model to control the actual dynamics. As seen below the oscillations in linear and angular velocity have become larger. However, the pole and cart positions are successfully controlled to the desired state. Again, the policy is more sensitive to initial conditions. I would say that the model predictive control approach has worked reasonably well but definitely has suffered a loss in stability and generality.



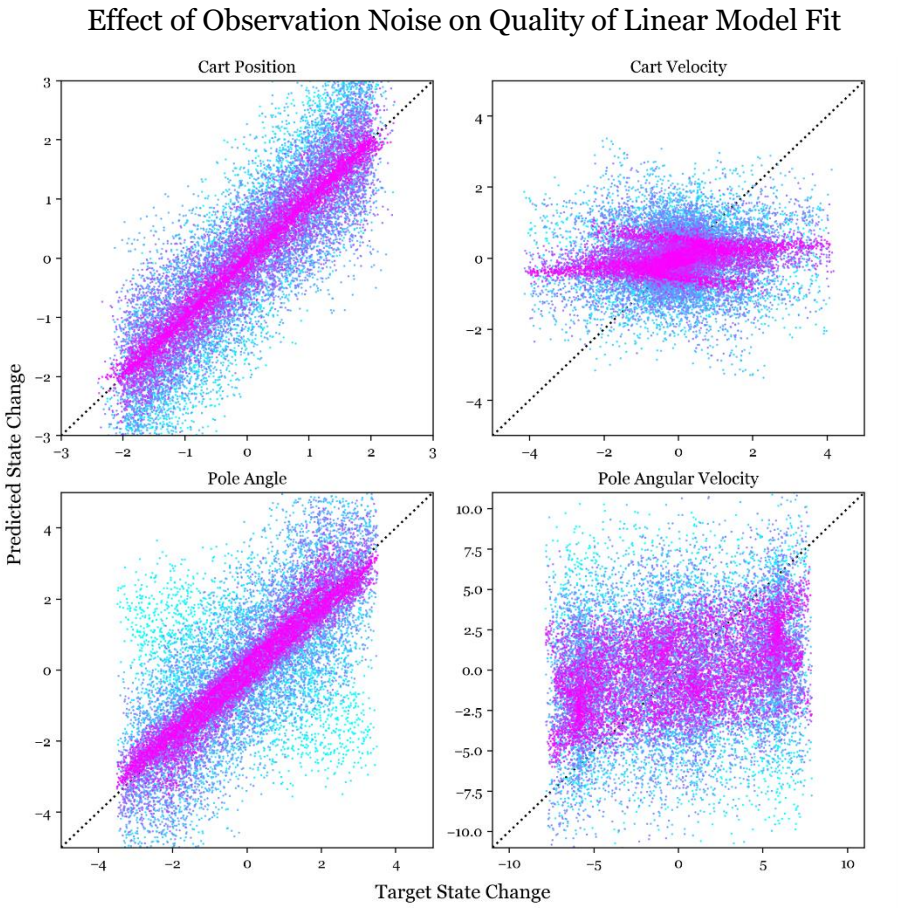
Task 3.1 – Introduction of Noise to Observed Dynamics

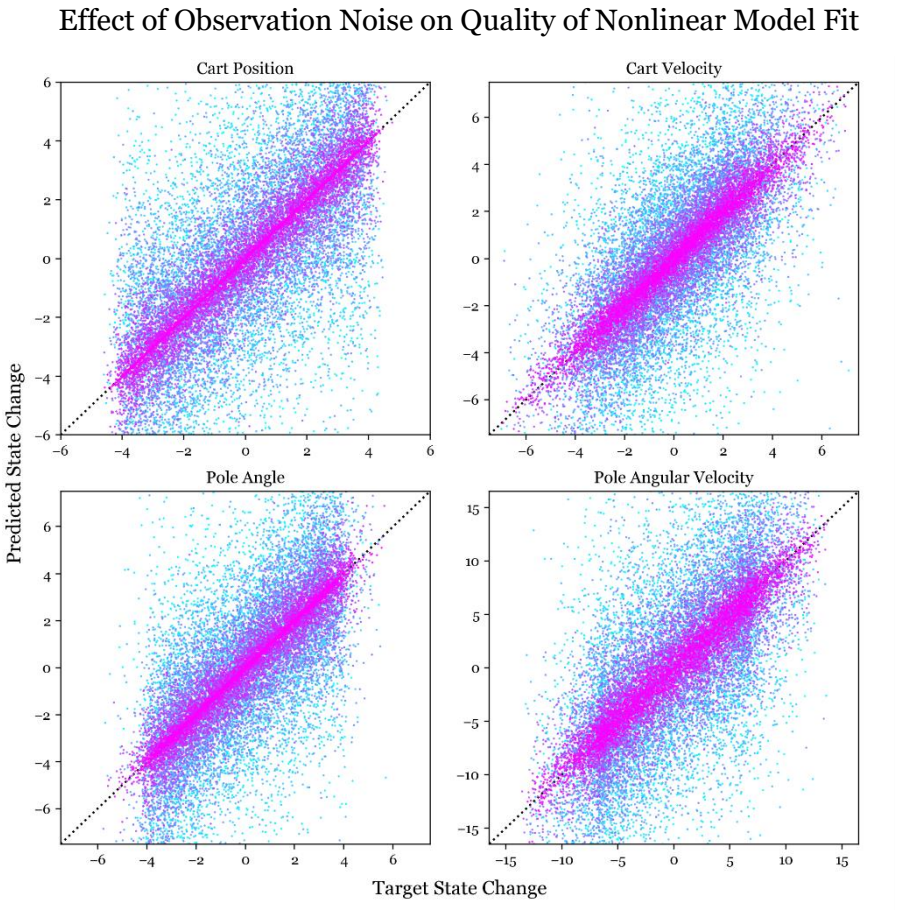
Linear Model Accuracy with Varying Observation Noise. Below are scatter plots of the predicted state change against target state change for linear models fitted with varying amounts of observation noise. Magenta represents models fit with little noise, and cyan with a lot of noise added. The cart position and pole angle plots show improved prediction accuracy with less noise, as does the cart velocity plot although the linear model can't achieve a very good fit for this variable even with zero noise. The pole angular velocity plot shows bad predictions for all noise levels so I suppose it is less

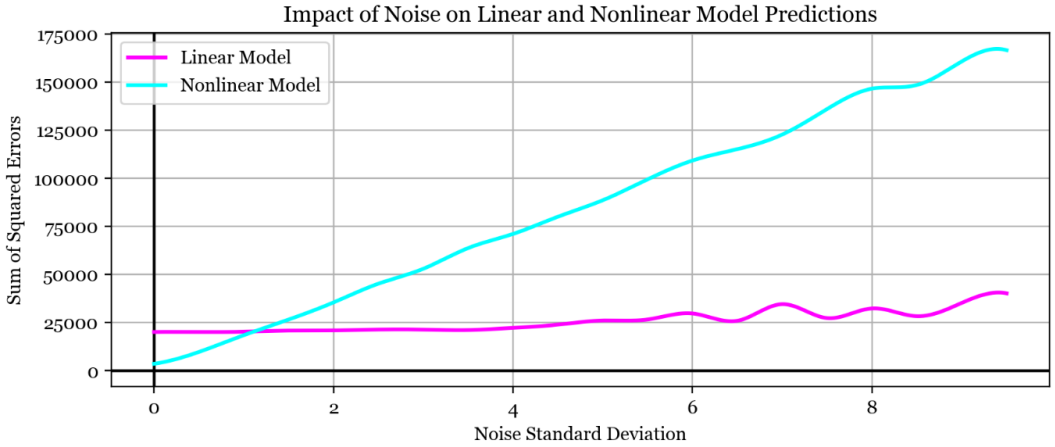
sensitive to noise. Also note that I scaled the noise for each variable based on the average magnitude of the points in the plots to make the comparison more fair.

Effect of Noise on the Nonlinear Model. In the bottom right I have made the same plot, but using the nonlinear model rather than the linear model. Again the low noise scenario in magenta shows better predictions than the cyan noisy scenario, but this time the zero noise predictions are much better. Interestingly the plots for the nonlinear model have one quarter as much noise added as those using the linear model, but still show similar deviation from the ideal line. This indicates that the nonlinear model is more sensitive to noise. I think this makes sense - for a linear function, the noise on either side of the desired fit almost cancels out so that the desired fit is less affected. The nonlinear model is actually able to fit to the noise which means its predictions of the true dynamics are worse.

Comparison of Models' Noise Tolerance. The plot overleaf shows how the sum of squared errors varies with the noise level for the linear and nonlinear models. The linear model is more resistant to the noise, while the nonlinear model is linearly impacted by the noise although its initial squared error is lower. So there is a crossover point where the optimal model flips from the nonlinear one to the linear one - at

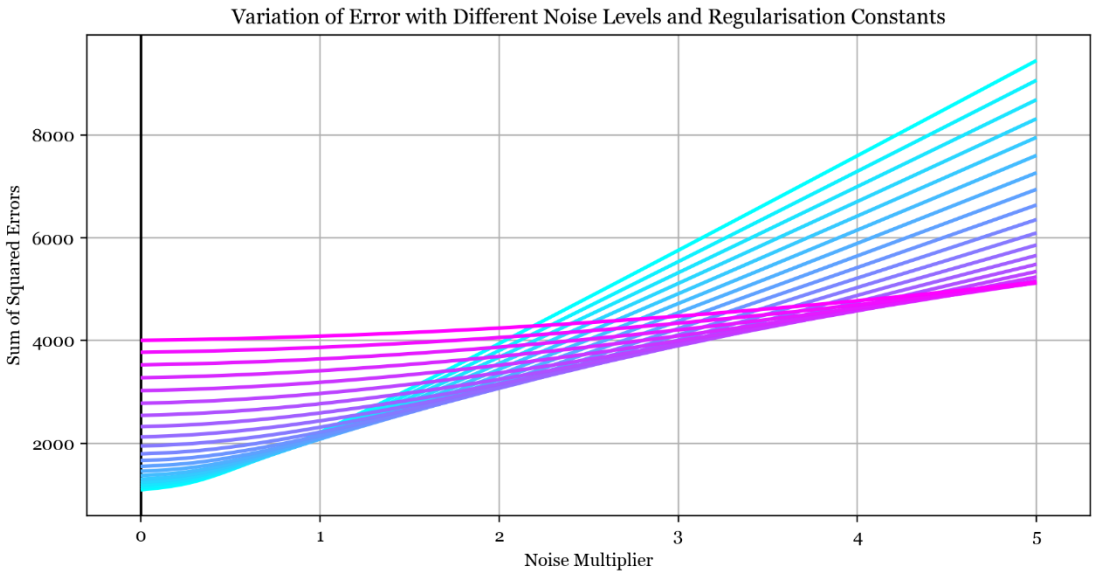




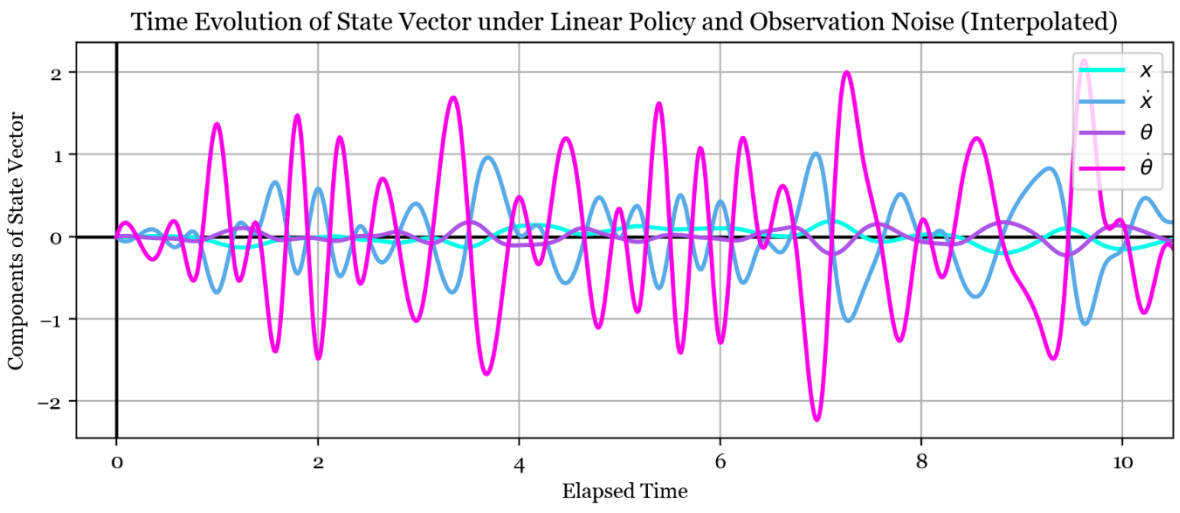


least for this regularisation strength. This is an important point as it indicates the nonlinear model has overfit to the training data. The more parsimonious linear model performs better because its small number of parameters forces it to capture real overarching structure in the data rather than the noisy quirks that the nonlinear model's huge number of parameters can overfit to.

Error vs Noise Level and Regularisation Strength. Here I have plotted the sum of squared errors against varying observation noise level for the nonlinear model, where each line has a different regularisation constant. The regularisation constants vary logarithmically from 2^{-4} in cyan to 2^{3} in magenta. As indicated by the gradient along the locus of the lowest points of the curves, the lowest line on the graph depends on the noise level - in other words, the optimal regularisation constant increases as the noise level increases. This cements the regularisation constant as a hyperparameter that helps to avoid overfitting. We also know from Gaussian Process theory that the regularisation constant is the variance of the noise, so this matches up well.



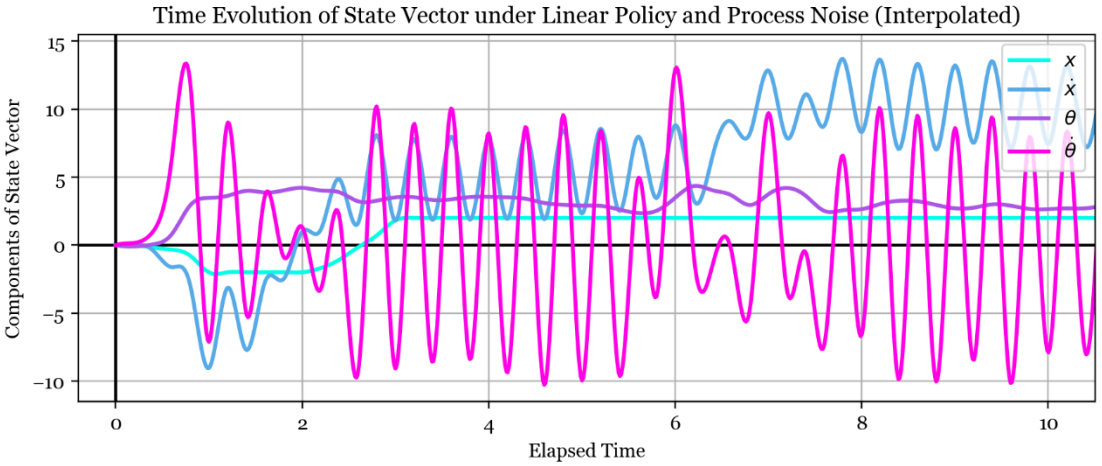
Linear Control with Noise. Below I have plotted a typical time evolution for the state vector under the linear policy with observation noise added. The policy is just about able to prevent the system from losing control at this noise level, so the states all fluctuate around 0.



Task 3.2 – Process Noise

The next experiment was to add noise to the actual dynamics of the cartpole rather than just observation noise on top of the actual dynamics. This type of noise is similar to observation noise over a single timestep, so the scatter plots are similar to those in task 3.1. However, the insertion of noise directly into the dynamics will affect the nature of rollouts performed under process noise.

Local Minimum. One interesting local minimum was found when optimising the policy in the presence of a lot of process noise. This policy tried to minimise the loss function by keeping the pole down and keeping the cart as close to the origin as possible. In this policy vector, the cart position and cart velocity entries were of similar magnitude to the pole angle and angular velocity entries. In the previous policies, the pole position and angular velocity entries were much larger relative to the others, indicating that keeping the pole upright was a higher priority under those regimes. The explanation for this behaviour is that in the presence of significant process noise the pole is almost impossible to keep upright; as soon as it falls over, a large pole position coefficient would cause the force to be huge and the cart to shoot off to one side (which would lead to a huge loss). So the policy avoids this by keeping the pole down, and just keeping the cart near the origin which is much more manageable.



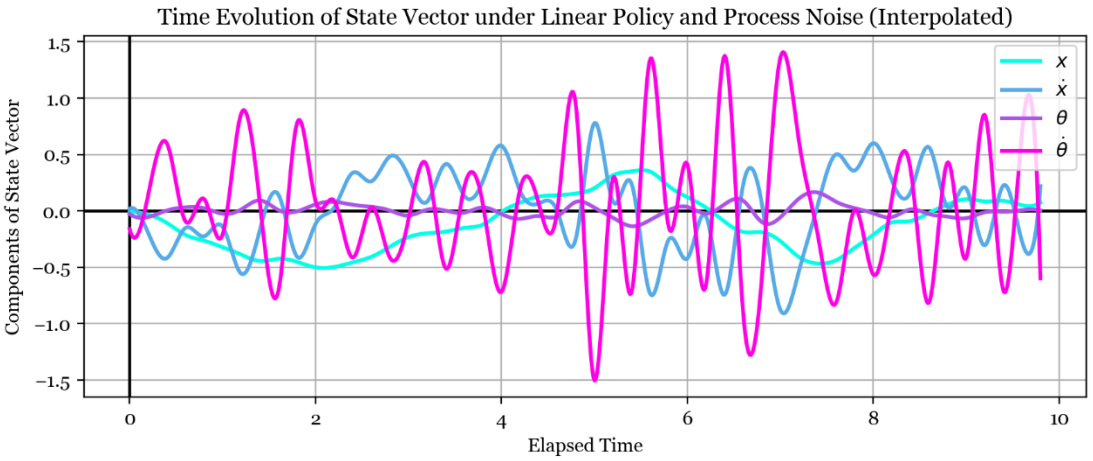
This is a typical state vector plotted over time under a large amount of process noise. The cart and pole position are both maintained at nonzero levels, and the large oscillations in pole and cart velocity are the reason why keeping the pole upright is a nonviable policy.

Note that the cart cannot be stationary at the origin with the pole down because then the force contribution from the pole angle would be nonzero while the contribution from the cart position would be zero, so the cart would move. This is why the policy keeps the cart a bit away from the origin, but a different angle definition or introduction of a bias term (common in machine learning) would be able to remove this effect.

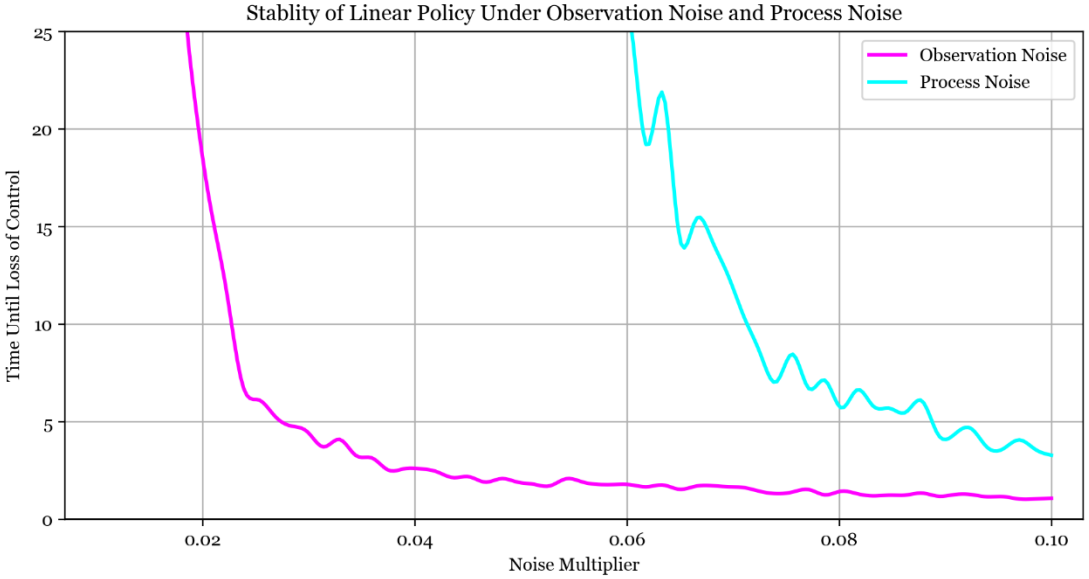
To break out of this local minimum I used typical techniques like using different initial conditions and parameters and trying to make the policy train over a larger portion of the state space. I also incorporated some of the other learned policies to weight the starting locations to try to focus in on portions of the state space that I thought would give good results (applying a prior). For small amounts of process noise, I found a policy very similar to the one trained under observation noise, and that performed similarly.

Control in Noise. In order to design a controller that can keep the pole up under a lot of noise, the timestep must be reduced below 0.2 seconds. It is the case that for any timestep length, there is a certain noise level that makes the cartpole completely uncontrollable. Imagine over one timestep the noise is such that the pole angle flips from $\mathbf{0}$ to $\boldsymbol{\pi}$ - there is no controller that could account for that, so there is a hard limit to how good our controller can be under observation noise and a given time step.

Time Evolution of State in Process Noise. Below is a typical state vector time evolution under process noise and an optimised linear policy. The effect of noise over subsequent timesteps is accumulating, so you can see the fluctuations in cart position occur over a much larger timescale as compared to those for the observation noise case. I think one interpretation is that the noise is now filtered through the cartpole dynamics, similar to an autoregressive process. The dynamics filter out the highest frequencies of the noise and leave lower frequency undulations behind.



Time Until Instability. I reoptimised the linear policy in the presence of observation and process noise, and then experimented with how long it could keep the pole upright at different noise levels. This plot shows that the linear policy could maintain control indefinitely for low noise levels, but after a certain point the time until loss of control appears to be inversely proportional to the noise level. The process noise was easier to control than the observation noise, as the cartpole was kept stable for longer with an equal amount of process noise than observation noise. I think this is because with process noise, the policy can correct for the actual state of the system and fight against the noise. With observation noise, the policy does not know the actual state of the system so could end up making corrections in the wrong direction.



The new linear policies were not able to control the system over as large a range of initial conditions as the original policy from task 2.3 - I think this is because I optimised them starting from the zero state and so they had experience over a smaller range of states than the original policy. The linear policies can only control the system within certain portions of the state space (around equilibria) - maybe a nonlinear policy could be capable of doing so over the entire space.

State Space Representation of the System and Closed-Form Control Solutions. The cartpole is a classical control system and as such a lot of the theory we learned in 3F2 can be applied directly. This is an approach that applies only to certain problems and is not a general machine learning methodology but since it applies so well to this system I have investigated a bit.

The first step is to create a discrete-time linear state-space model of the system with the following form, which assumes small deflections from the equilibrium at the zero state.

$$\boldsymbol{x_{k+1} = Ax_k + Bu_k + Ke_k}$$

In this equation, the $\boldsymbol{A}$ matrix is a state transition matrix that describes the effect of the current state $\boldsymbol{x_k}$ on the subsequent state $\boldsymbol{x_{k+1}}$. The $\boldsymbol{B}$ matrix is in this case a column vector, and converts the input force $\boldsymbol{u_k}$ to its effect on the subsequent state. The third term on the right describes noise input to the system.

To find the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices I could have tried to linearise the system by hand, but the equations of motion are complex and involve multiple integrator steps per state transition. Another approach is to generate them using least-squares regression similarly to task 1.3. First, set the force input $\boldsymbol{u_k}$ to zero and generate a set of initial state vectors, all close to the zero state. Then, use the cartpole dynamics to update these states. This gives the equation

$$\boldsymbol{X_{k+1} = AX_k}$$

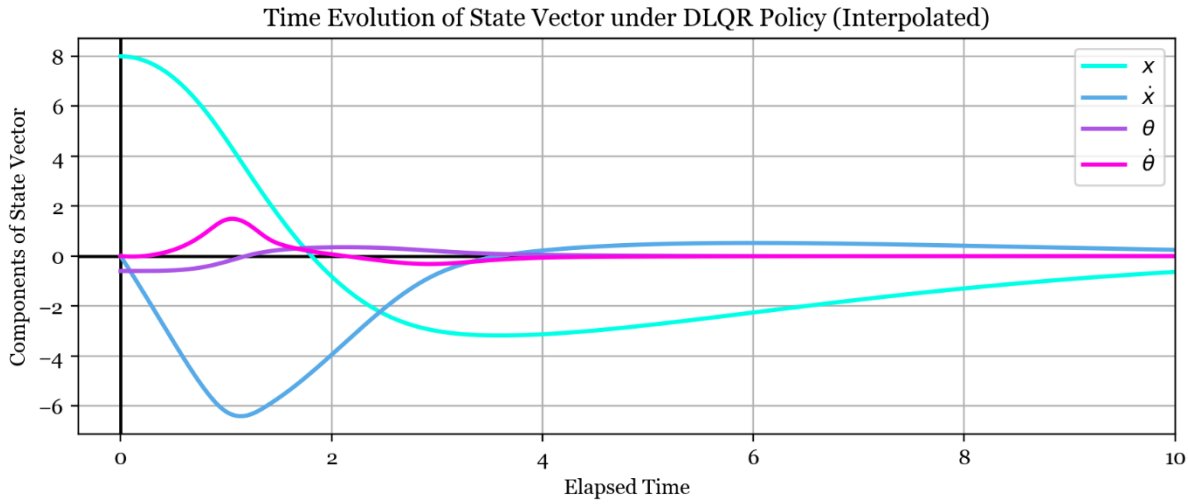
where the capital $\boldsymbol{X}$ represents a matrix formed by stacking the state vectors. This equation is then solved for $\boldsymbol{A}$ using `numpy.lstsq`. A similar approach works for the $\boldsymbol{B}$ matrix, keeping the initial state at $\mathbf{0}$ and generating a set of force inputs and subsequent states.

Computation of Policy. Having found these two matrices, a discrete linear quadratic regulator (DLQR) is used to find a linear state-feedback controller. This computation is involved, requiring solving the discrete algebraic Riccati equation, so I used the imported `control.dlqr` function. For the $\boldsymbol{Q}$ matrix I used the identity, and for the $\boldsymbol{R}$ matrix I found that the identity times 100 worked well. This whole process executes almost immediately, compared to the long optimisation process of task 2.3. The policy vector found is p = [0.02581708, 0.1272267, 11.95653773, 1.73874922], and it is able to stabilise the system.

The policy vector found by this method is similar to the optimised policy vector from task 2.3 in some ways, but different in others. For example, it ascribes a much lower weight to position error, and so the cart takes much longer to reach the origin. It is able to stabilise the cart from some initial

conditions that the previous policy vector could not, and unable to stabilise from other initial states which the previous policy vector could.

The $\boldsymbol{Q}$ and $\boldsymbol{R}$ matrices play the same role as the loss function from task 2.3 - by varying them, we can encourage the controller to behave in different ways. A large $\boldsymbol{Q}$ matrix encourages fast convergence of the state to zero, but risks reducing stability margins. By varying the diagonal elements of the $\boldsymbol{Q}$ matrix, different elements of the state vector can be given higher priorities to bring to zero. The $\boldsymbol{R}$ matrix similarly penalises the force input. The DLQR policy has not been able to learn the nonlinearities of the system, so could be at a disadvantage in performance. This manifests as the policy taking longer to bring all the states to zero - it cannot rely on nonlinear quirks to help speed up the convergence so is more cautious with the force input.



Above is a typical state vector time evolution under the DLQR policy. It is clear that this policy brings the states to zero much more gradually. So task 2.3 is solved to a reasonable standard in a few lines that execute almost instantly. The takeaway is that by applying our experience as data scientists, the underlying structure of many problems can be revealed to allow more elegant solutions.

Introduction of Noise. The state-space representation of the system enables application of Kalman Filtering or other noise handling techniques. Linear Quadratic Gaussian control is ideal for this scenario but I did not have time to implement this and I imagine the lag introduced by the Kalman filter would cause issues with controlling the cartpole.

One interesting effect I noticed is that by varying the $\boldsymbol{Q}$ and $\boldsymbol{R}$ matrices, the tolerance to noise can be increased. The above controller with the large $\boldsymbol{R}$ matrix was more tolerant to process noise than any other policy I had created so far. I think the different loss function definition led to the controller being less desperate to quickly bring the cart to the origin. It was then more able to keep the pole up since the force inputs to bring the cart to the centre were less erratic.

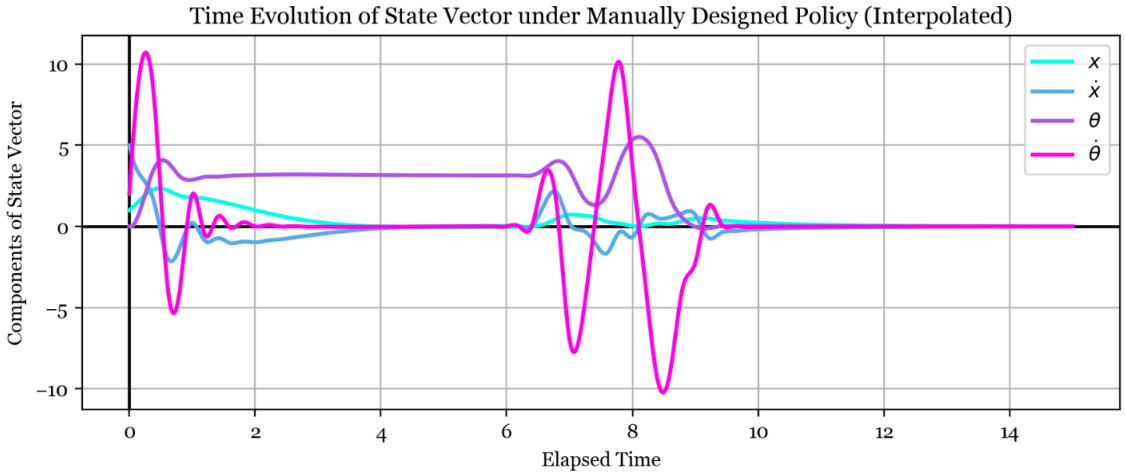
Task 4 – Nonlinear Control

Manually Designed Policy. I wanted to design a controller that can keep the pole upright starting from the stable equilibrium. As a first step, I manually designed such a policy as follows.

1. Use a linear policy to bring the cartpole to the stable equilibrium, at the origin.
2. Apply a certain sequence of force inputs to swing up the pole.
3. Use another linear policy to keep the pole upright.

For step 1, I used standard PD controller tuning techniques to find a policy vector p = [-1, -1.5, 0, -1]. For step 3, I used the policy vector I found in task 2.3, p = [0.76, 1.44, 17.06, 2.66]. For step two, I applied some intuition that the cart will need to swing back and forth to raise up the pole gradually over a few oscillations. I therefore started with a list of force values that alternated in sign every few time steps (frequency roughly matched to pole natural frequency), and then tuned the exact values until the second linear policy could "catch" the system in its resultant state and stabilise it. The list of force values I found is [5, 5, -5, -5, -5, -5, 4, 4, 4, 4, -2, -2].

I like this approach because it leverages a human, intuitive understanding of the system to simplify the problem, and works with the computer to solve it in small, manageable chunks. Designing this controller took about 15 minutes and gives ideal results. The drawback is that this type of controller is specialised to this system - the same method may not be tractable for another system/dataset.



This is the time evolution of the state vector under my controller. The first set of ripples is the first linear policy bringing the cartpole from an arbitrary state to the equilibrium with the pole downwards, shown by the purple line (pole angle) being brought to rest at pi. The second set of ripples is the swing up and subsequent stabilisation in the upright configuration - after this all the states are zero as desired.

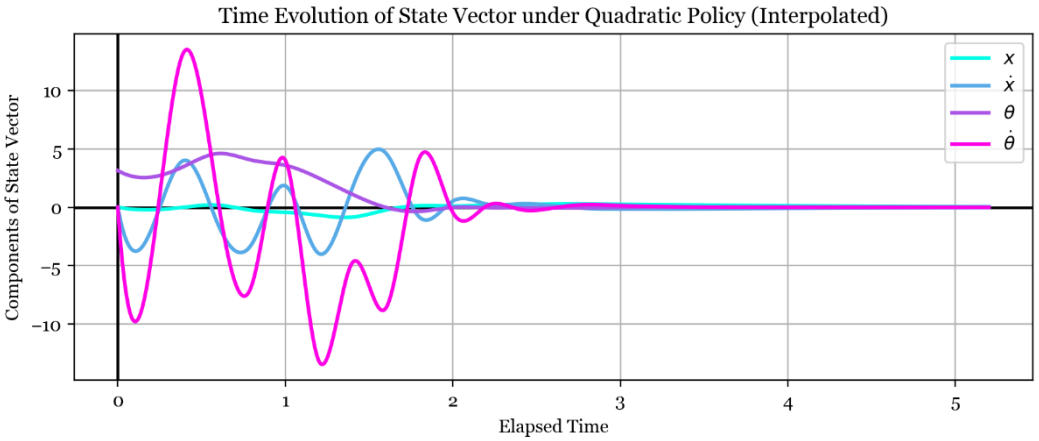
Quadratic Policy. So far I used a linear policy of the form $\boldsymbol{p} \cdot \boldsymbol{X}$. It then became clear that the linear policy could not handle the nonlinearities of the cartpole system. So my next step was to try a quadratic controller, of the form $\boldsymbol{p} \cdot \boldsymbol{X} + \boldsymbol{X}^{\boldsymbol{T}} \boldsymbol{P} \boldsymbol{X}$, hoping that it could capture some of the nonlinearity of the cartpole and enable swing-up. I used the initial condition where the cartpole is stationary at the origin with the pole down, since I know a simple linear policy can bring the system to this state. I could

not think of a good way to find the optimal $\boldsymbol{p}$ vector and $\boldsymbol{P}$ matrix, so I just randomly selected them with entries normally distributed around $\mathbf{0}$ and kept doing random selections until I found a good setting. That setting is

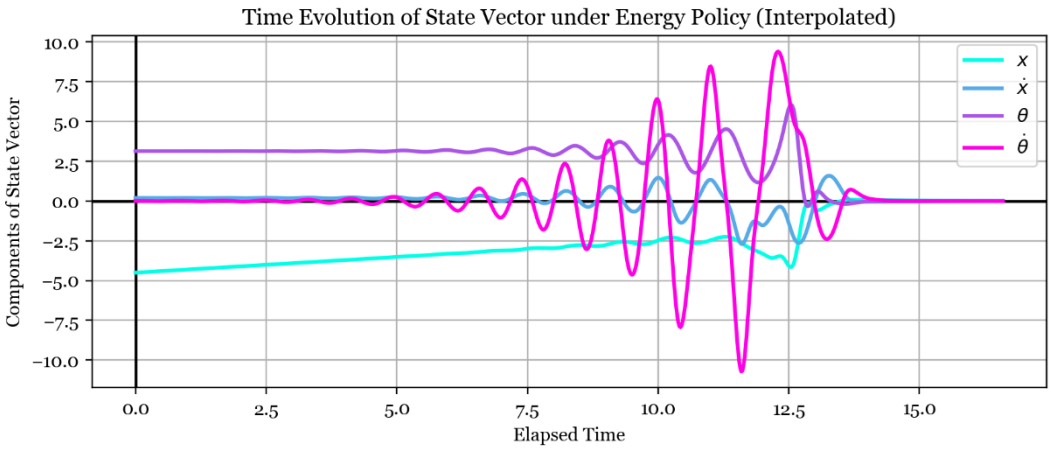
```
p = [ 0.79720335, -0.81892826, 0.16672051 , -0.09236256 ]

P = [[-0.34134063, -0.44538288, -0.54157946,  0.5954045 ]
     [ 0.48675709, -1.13294381, -0.951934  , -0.5663644 ]
     [ 1.37535627, -1.66183934, -0.8258103 , -1.3074058 ]
     [ 2.1116826 ,  1.3782299 , -1.72001146,  0.1383763 ]]
```

and the swing-up sequence that this controller results in is plotted below. There is one subtlety which is that I could not find a good quadratic controller for the entire state space, so after 2.2 seconds of quadratic control (after which the cartpole is near the desired state) the controller reverts to a known working linear controller. One other issue is that this controller leverages large and erratic pole angular velocities which would probably be unrealistic in practice.



Energy-Based Policy. Another type of swing-up controller I read about is an energy-based controller, where the force input is modulated to increase the energy of the swinging pole until it has just enough to reach the top, at which point a linear controller is employed. In my case I set the force to be proportional to the negative pole angle, remapped to be zero at the bottom. This way of setting the force ensured that the system was excited at its natural frequency. The controller keeps the force at this level until the system energy is high enough, at which point it sets the force input to zero and lets the pole swing up. The linear controller from task 2.3 is then used to maintain balance in the upright position. This controller can also handle some nonzero initial cart position and velocity, and required no long optimisations.



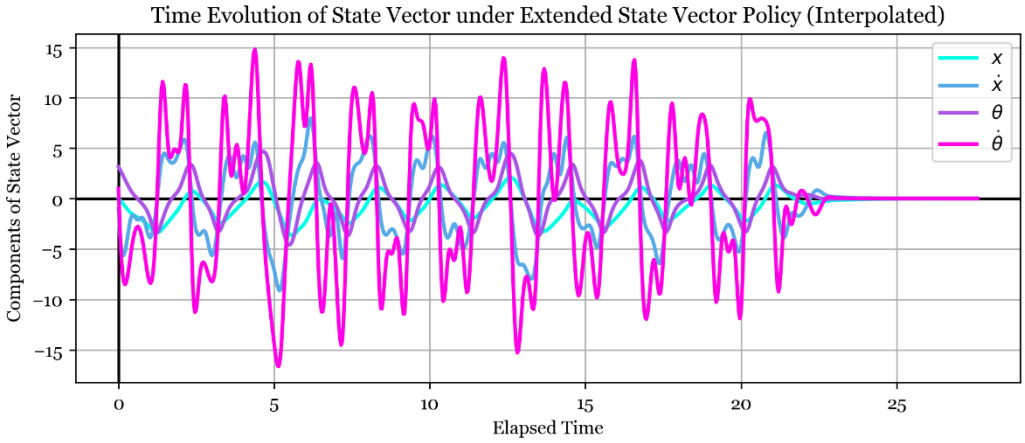
Extended State Vector Policy. The last type of controller I tried was another linear policy of the form $\boldsymbol{p} \cdot \widetilde{\boldsymbol{X}}$, but this time I extended the state vector with nonlinear combinations of states to create $\widetilde{\boldsymbol{X}}$. I then used `scipy.optimize` as before to find the best $\boldsymbol{p}$ vector. The expanded policy vector was:

$$\widetilde{\boldsymbol{X}} = \left[\boldsymbol{x}\ \dot{\boldsymbol{x}}\ \boldsymbol{\theta}\ \dot{\boldsymbol{\theta}}\ \sin(\boldsymbol{\theta})\ \dot{\boldsymbol{x}} \cdot \cos(\boldsymbol{\theta})\ \dot{\boldsymbol{\theta}} \cdot \cos(\boldsymbol{\theta})\ \boldsymbol{\theta}^{\boldsymbol{3}}\right]$$

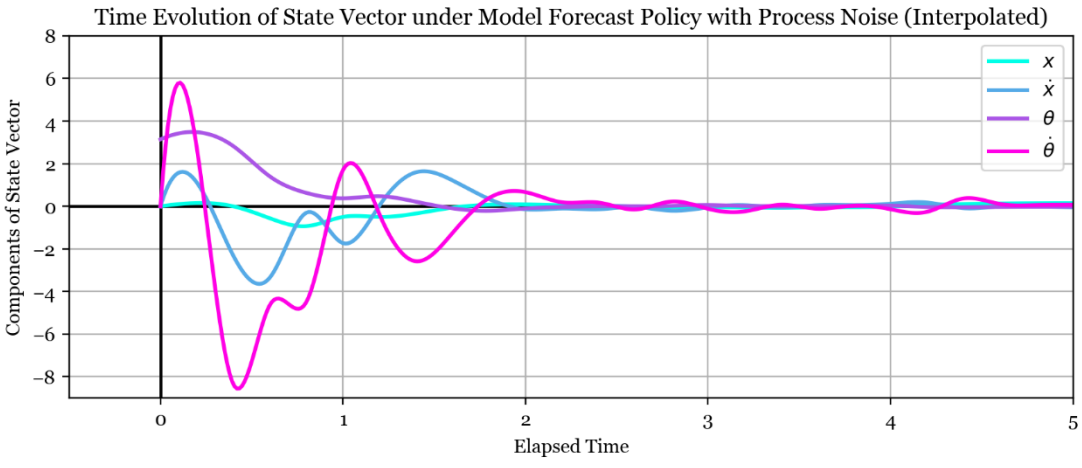
and the linear policy vector p was:

```
p = [-3.19540, -5.76562, 0.37423, 2.03079, 5.57070, 3.90909, 0.01235, -1.81513]
```

This controller throws the pole up multiple times, oscillating from left to right until one of its attempts results in the pole being almost stationary in the upright position. At this point the linear controller from task 2.3 takes over and stabilises the pole at the top. Because the motion is so erratic at the start, this controller is less sensitive to initial conditions and slightly less sensitive to noise.



Model Forecast Policy. The final controller I tried used the nonlinear model from task 2.1. At each timestep, I generated many possibilities for the force inputs over the next five timesteps. I then used the nonlinear model to estimate the state evolution over the next five timesteps with these force inputs, and evaluated the sum of the loss function over these estimated state evolutions. The controller then selects the next force input that leads to the lowest loss. It swings up the cartpole and again uses the linear policy when the pole is upright.



This controller was my favourite as it incorporated the nonlinear model from task 2, required no fine-tuning of parameters, and works with a bit of process noise. The big drawback is that so much computation is required at each timestep that it can't run in real time. The solution is to record the set of force inputs that swing up the pole and play them back when required.

Stability of Swing-up Controllers. I found that most of the swing-up controllers I could come up with were extremely sensitive to noise. I think this is partly due to the sampling period of the controller being so long - even a slight deviation from what the controller expects builds up over the 0.2 seconds before the next force input and becomes unrecoverable. This is especially true for the quadratic controller - the weights in the p and P matrices are specific to the exact states that the system passes through during the swing-up, so any deviation is disastrous. I think for a more noise-resistant swing-up controller, it really is necessary to use a finer timestep.

Conclusions

I enjoyed studying the cartpole system; it contains a lot of complexity within an easily understood system and a lot of machine learning techniques can be applied to it. The complex structure was clear from the poor performance of the linear model, and fitting the nonlinear model was an interesting study in regularised least-squares and hyperparameter tuning. Linear control offered a simple method to control the system but the limits on initial conditions revealed the limitations of a linear state-feedback policy. I learned how sensitive the models and linear policies can be to observation and process noise, and saw how over-complex models can overfit to noisy data as well as how this can be limited by selection of the regularisation constant. I briefly experimented with a DQLR policy, which offered a nice way to avoid long optimisations. Building the nonlinear swing-up policies was a fun challenge, with different advantages and disadvantages offered by each method. The project taught me a lot and was really enjoyable; I would like to give a big thankyou to Gabor and Yashar for supervising the project and answering all my questions along the way.

Appendix – Code Listing

CartPole.py

```
"""
fork from python-rl and pybrain for visualization
"""

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

# If theta has gone past our conceptual limits of [-pi,pi]
# map it onto the equivalent angle that is in the accepted range (by adding or subtracting 2pi)

def remap_angle(theta):
    return (theta - np.pi) % (2*np.pi) - np.pi


## loss function given a state vector. the elements of the state vector are
## [cart location, cart velocity, pole angle, pole angular velocity]

def loss(state):

    sigma_l = 0.5
    return 1 - np.exp( - state @ state / (2.0 * sigma_l**2) )


class CartPole:
    """Cart Pole environment. This implementation allows multiple poles,
    noisy action, and random starts. It has been checked repeatedly for
    'correctness', specifically the direction of gravity. Some implementations of
    cart pole on the internet have the gravity constant inverted. The way to check is to
    limit the force to be zero, start from a valid random start state and watch how long
    it takes for the pole to fall. If the pole falls almost immediately, you're all set. If it takes
    tens or hundreds of steps then you have gravity inverted. It will tend to still fall because
    of round off errors that cause the oscillations to grow until it eventually falls.
    """

    def __init__(self, visual=False, smooth=False, save_frames=False, fig_num=1):

        self.reset()

        self.visual = visual
        self.save_frames = save_frames
        self.frame = 0

        # Setup pole lengths and masses based on scale of each pole
        # (Papers using multi-poles tend to have them either same lengths/masses
        # or they vary by some scalar from the other poles)
        self.pole_length = 0.5
        self.pole_mass = 0.5

        self.mu_c = 0.001 # friction coefficient of the cart
        self.mu_p = 0.001 # friction coefficient of the pole
        self.sim_steps = 50 # number of Euler integration steps to perform in one go
        self.delta_time = 0.2 # time step of the Euler integrator
        self.max_force = 20.
        self.gravity = 9.8
        self.cart_mass = 0.5

        # set the euler integration settings to smaller steps to allow smooth rendering
        if smooth:
            self.sim_steps = 20
            self.delta_time = 0.08

        # for plotting
        self.cartwidth = 1.0
        self.cartheight = 0.2

        if self.visual:
            self.drawPlot( fig_num )


    # reset the state vector to the initial state (down-hanging pole)
    def reset(self):
        self.set_state( [ 0, 0, np.pi, 0 ] )

    def set_state(self, state):

        self.cart_position, self.cart_velocity, self.pole_angle, self.pole_angvel = state[:4]


    def get_state(self):

        return np.array([ self.cart_position, self.cart_velocity, self.pole_angle, self.pole_angvel ])


    def remap_angle(self):

        self.pole_angle = remap_angle( self.pole_angle )


    # the loss function that the policy will try to optimise (lower) as a member function
    def loss(self):
        return loss(self.getState())


    # This is where the equations of motion are implemented
    def perform_action( self, action=0.0 ):

        # prevent the force from being too large
        force = self.max_force * np.tanh(action/self.max_force)
        dt = self.delta_time / float(self.sim_steps)

        # integrate forward the equations of motion using the Euler method
        for step in range(self.sim_steps):

            s = np.sin(self.pole_angle)
            c = np.cos(self.pole_angle)

            m = 4.0 * ( self.cart_mass + self.pole_mass ) - 3.0 * self.pole_mass * c**2
```

```
            cart_accel = 1/m * (
                2.0 * ( self.pole_length * self.pole_mass * s * self.pole_angvel**2
                + 2.0 * ( force - self.mu_c * self.cart_velocity ) )
                - 3.0 * self.pole_mass * self.gravity * c*s
                + 6.0 * self.mu_p * self.pole_angvel * c / self.pole_length
            )

            pole_accel = 1/m * (
                - 3.*c * 2. / self.pole_length * (
                    self.pole_length / 2.0 * self.pole_mass * s * self.pole_angvel**2
                    + force
                    - self.mu_c * self.cart_velocity
                )
                + 6.0 * ( self.cart_mass + self.pole_mass ) / ( self.pole_mass * self.pole_length ) * \
                ( self.pole_mass * self.gravity * s - 2.0/self.pole_length * self.mu_p * self.pole_angvel )
            )

            # Update state variables
            # Do the updates in this order, so that we get semi-implicit Euler that is simplectic rather than forward-Euler which is not.
            self.cart_velocity += dt * cart_accel
            self.pole_angvel   += dt * pole_accel
            self.pole_angle    += dt * self.pole_angvel
            self.cart_position += dt * self.cart_velocity

        if self.visual:
            self._render()


    # the following are graphics routines
    def drawPlot(self, fig_num):

        plt.ion()
        self.fig, self.axes = plt.subplots( 1, 1, num=fig_num, figsize=(9.5,2) )

        # draw cart
        self.axes.get_yaxis().set_visible(False)
        self.box = plt.Rectangle(xy=(self.cart_position - self.cartwidth / 2.0, -self.cartheight / 2.0),
                                 width=self.cartwidth, height=self.cartheight)
        self.axes.add_artist(self.box)
        self.box.set_clip_box(self.axes.bbox)

        # draw pole
        self.pole = plt.Line2D([self.cart_position, self.cart_position + np.sin(self.pole_angle) * self.pole_length],
                               [0, np.cos(self.pole_angle) * self.pole_length], linewidth=3.5, color='black')
        self.axes.add_artist(self.pole)
        self.pole.set_clip_box(self.axes.bbox)

        # set axes limits
        self.axes.set_xlim(-10, 10)
        self.axes.set_ylim(-1, 1)
        #self.fig.tight_layout()

        self.fig.subplots_adjust( top=0.92, bottom=0.17, left=0.02, right=0.98 )

    def _render(self):

        self.box.set_x(self.cart_position - self.cartwidth / 2.0)
        self.pole.set_xdata([ self.cart_position, self.cart_position + np.sin(self.pole_angle) * self.pole_length ])
        self.pole.set_ydata([ 0, np.cos(self.pole_angle) * self.pole_length ])

        self.fig.canvas.draw()

        if self.save_frames:
            self.fig.savefig( f"frames/{self.frame}.png", dpi=300 )
            self.frame += 1

class Object(object):
    pass

# static version of perform action
def perform_action( state, action=0.0, better_angle=False ):

    self = Object()

    self.cart_position, self.cart_velocity, self.pole_angle, self.pole_angvel = state

    if better_angle:
        self.pole_angle += np.pi

    self.pole_length = 0.5
    self.pole_mass = 0.5

    self.mu_c = 0.001 # friction coefficient of the cart
    self.mu_p = 0.001 # friction coefficient of the pole
    self.sim_steps = 50 # number of Euler integration steps to perform in one go
    self.delta_time = 0.2 # time step of the Euler integrator
    self.max_force = 20.
    self.gravity = 9.8
    self.cart_mass = 0.5

    # prevent the force from being too large
    force = self.max_force * np.tanh(action/self.max_force)
    dt = self.delta_time / float(self.sim_steps)

    # integrate forward the equations of motion using the Euler method
    for step in range(self.sim_steps):

        s = np.sin(self.pole_angle)
        c = np.cos(self.pole_angle)

        m = 4.0 * ( self.cart_mass + self.pole_mass ) - 3.0 * self.pole_mass * c**2

        cart_accel = 1/m * (
            2.0 * ( self.pole_length * self.pole_mass * s * self.pole_angvel**2
            + 2.0 * ( force - self.mu_c * self.cart_velocity ) )
            - 3.0 * self.pole_mass * self.gravity * c*s
            + 6.0 * self.mu_p * self.pole_angvel * c / self.pole_length
        )

        pole_accel = 1/m * (
            - 3*c * 2/self.pole_length * (
                self.pole_length/2 * self.pole_mass * s * self.pole_angvel**2
                + force
                - self.mu_c * self.cart_velocity
            )
```

```
            + 6.0 * ( self.cart_mass + self.pole_mass ) / ( self.pole_mass * self.pole_length ) * \
            ( self.pole_mass * self.gravity * s - 2.0/self.pole_length * self.mu_p * self.pole_angvel )
        )

        # Update state variables
        # Do the updates in this order, so that we get semi-implicit Euler that is simplectic rather than forward-Euler which is not.
        self.cart_velocity += dt * cart_accel
        self.pole_angvel   += dt * pole_accel
        self.pole_angle    += dt * self.pole_angvel
        self.cart_position += dt * self.cart_velocity

    if better_angle:
        self.pole_angle -= np.pi

    return np.array( [ self.cart_position, self.cart_velocity, self.pole_angle, self.pole_angvel ] )


def perform_action5( state5 ):

    self = Object()

    self.cart_position, self.cart_velocity, self.pole_angle, self.pole_angvel, action = state5

    self.pole_length = 0.5
    self.pole_mass = 0.5

    self.mu_c = 0.001 # friction coefficient of the cart
    self.mu_p = 0.001 # friction coefficient of the pole
    self.sim_steps = 500 # number of Euler integration steps to perform in one go
    self.delta_time = 0.2 # time step of the Euler integrator
    self.max_force = 20.
    self.gravity = 9.8
    self.cart_mass = 0.5

    # prevent the force from being too large
    force = self.max_force * np.tanh(action/self.max_force)
    dt = self.delta_time / float(self.sim_steps)

    # integrate forward the equations of motion using the Euler method
    for step in range(self.sim_steps):

        s = np.sin(self.pole_angle)
        c = np.cos(self.pole_angle)

        m = 4.0 * ( self.cart_mass + self.pole_mass ) - 3.0 * self.pole_mass * c**2

        cart_accel = 1/m * (
            2.0 * ( self.pole_length * self.pole_mass * s * self.pole_angvel**2
            + 2.0 * ( force - self.mu_c * self.cart_velocity ) )
            - 3.0 * self.pole_mass * self.gravity * c*s
            + 6.0 * self.mu_p * self.pole_angvel * c / self.pole_length
        )

        pole_accel = 1/m * (
            - 3*c * 2/self.pole_length * (
                self.pole_length/2 * self.pole_mass * s * self.pole_angvel**2
                + force
                - self.mu_c * self.cart_velocity
            )
            + 6.0 * ( self.cart_mass + self.pole_mass ) / ( self.pole_mass * self.pole_length ) * \
            ( self.pole_mass * self.gravity * s - 2.0/self.pole_length * self.mu_p * self.pole_angvel )
        )

        # Update state variables
        # Do the updates in this order, so that we get semi-implicit Euler that is simplectic rather than forward-Euler which is not.
        self.cart_velocity += dt * cart_accel
        self.pole_angvel   += dt * pole_accel
        self.pole_angle    += dt * self.pole_angvel
        self.cart_position += dt * self.cart_velocity

    return np.array( [ self.cart_position, self.cart_velocity, self.pole_angle, self.pole_angvel, action ] )



pole_length = 0.5
pole_mass   = 0.5
mu_c        = 0.001
mu_p        = 0.001
max_force   = 20.
gravity     = 9.8
cart_mass   = 0.5


def perform_action_RK4( state ):

    h = 0.1 # RK4 step size

    # perform 2 RK4 steps
    for _ in range(2):

        k1 = dstate_dt( state            )
        k2 = dstate_dt( state + h/2 * k1 )
        k3 = dstate_dt( state + h/2 * k2 )
        k4 = dstate_dt( state + h   * k3 )

        state = state + h/6 * ( k1 + 2*k2 + 2*k3 + k4 )

    return state


def dstate_dt( state ):

    cart_position, \
    cart_velocity, \
    pole_angle,    \
    pole_angvel,   \
    action = state

    dstate = np.array( [0., 0, 0, 0, 0] )

    dstate[0] = state[1]
    dstate[2] = state[3]


    force = max_force * np.tanh( action / max_force )
```

```
    s = np.sin(pole_angle)
    c = np.cos(pole_angle)

    m = 4.0 * ( cart_mass + pole_mass ) - 3.0 * pole_mass * c**2

    dstate[1] = 1/m * (
        2.0 * ( pole_length * pole_mass * s * pole_angvel**2
        + 2.0 * ( force - mu_c * cart_velocity ) )
        - 3.0 * pole_mass * gravity * c*s
        + 6.0 * mu_p * pole_angvel * c / pole_length
    )

    dstate[3] = 1/m * (
        - 3.*c * 2./pole_length * (
            pole_length/2. * pole_mass * s * pole_angvel**2
            + force
            - mu_c * cart_velocity
        )
        + 6.0 * ( cart_mass + pole_mass ) / ( pole_mass * pole_length ) * \
        ( pole_mass * gravity * s - 2.0/pole_length * mu_p * pole_angvel )
    )

    return dstate
```

sf3.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


# import CartPole.py from local directory
import CartPole, sf3utility
import matplotlib.collections
import matplotlib.pyplot as plt
import numpy as np
import scipy.interpolate
import random

plt.rcParams["font.family"] = "Georgia"
#plt.rcParams['figure.figsize'] = [9.0, 7.0]
#plt.rcParams['figure.dpi'] = 400

# store results for later
cache = {}


# In[2]:


# allows nice plots that can be redrawn
get_ipython().run_line_magic('matplotlib', 'notebook')


# # Task 1.1 - Simulation of Rollout
#

# In[3]:


# instantiate a cartpole and use a small timestep to get smooth lines
rollout_cartpole = CartPole.CartPole()

rollout_cartpole.sim_steps = 1
rollout_cartpole.delta_time = 0.01


# ## Small Oscillation about Stable Equilibirum

# In[4]:


# small oscillations about stable equilibrium

rollout_cartpole.reset()

rollout_cartpole.cart_velocity = 0.126
rollout_cartpole.pole_angvel = 1

x, y = [], []
states = []

for i in range(50):

    rollout_cartpole.perform_action()

    x.append( rollout_cartpole.cart_velocity )
    y.append( rollout_cartpole.pole_angvel )

    states.append( rollout_cartpole.get_state() )


fig, ax = plt.subplots(1, 1, num=2)
sf3utility.setup_phase_portrait( ax )
ax.plot( x, y )

ax.set_title( "Small Oscillation about Stable Equilibirum" )
ax.set_xlabel( "Cart Velocity" )
ax.set_ylabel( "Pole Angular Velocity" )

states = np.array( states )
cache["states_small_oscillations"] = states



# ## Large Amplitude Oscillations

# In[5]:
```

```
# large oscillations about stable equilibrium

rollout_cartpole.reset()

rollout_cartpole.cart_velocity = 1.72
rollout_cartpole.pole_angvel = 12
rollout_cartpole.mu_p = 0.001 # increase friction to speed up convergence

x, y = [], []
states = []

for i in range(10000):

    rollout_cartpole.perform_action()

    x.append( rollout_cartpole.cart_velocity )
    y.append( rollout_cartpole.pole_angvel )

    states.append( rollout_cartpole.get_state() )


fig, ax = plt.subplots(1, 1, num=3)
sf3utility.setup_phase_portrait( ax )

ax.set_xlim( min(x) * 1.12, max(x) * 1.12 )
ax.set_ylim( min(y) * 1.12, max(y) * 1.12 )

points   = np.array([x, y]).T.reshape(-1, 1, 2)[::-1]
segments = np.concatenate( [points[:-1], points[1:]], axis=1 )
colouring_array =  np.linspace( 0.0, 1.0, len(x) ) ** 3

linecollection = matplotlib.collections.LineCollection( segments, array=colouring_array, cmap="cool", zorder=3, linewidths=1.2 )

ax.add_collection( linecollection )

ax.set_title( "Large Oscillations of the Cartpole System" )
ax.set_xlabel( "Cart Velocity" )
ax.set_ylabel( "Pole Angular Velocity" )

states = np.array( states )
cache["states_large_oscillations"] = states



# ## Swinging Over the Top

# In[6]:


# even larger oscillations

rollout_cartpole.reset()

rollout_cartpole.cart_velocity = 4
rollout_cartpole.pole_angvel = 20
rollout_cartpole.mu_p = 0.005 # increase friction to speed up convergence

x, y = [], []

for i in range( 5700 ):

    pole_angle = rollout_cartpole.pole_angle % (2*np.pi)

    if pole_angle > 6.1:
        x.append( np.nan )
        y.append( np.nan )

    else:
        x.append( pole_angle )
        y.append( rollout_cartpole.pole_angvel )

    rollout_cartpole.perform_action()


fig, ax = plt.subplots(1, 1, num=4)
sf3utility.setup_phase_portrait( ax )
ax.axvline( x=np.pi, color="black" )

ax.set_xlim( min(x) * 1.12, max(x) * 1.12 )
ax.set_ylim( min(y) * 1.12, max(y) * 1.12 )

points   = np.array([x, y]).T.reshape(-1, 1, 2)[::-1]
segments = np.concatenate( [points[:-1], points[1:]], axis=1 )
colouring_array =  np.linspace( 0.0, 1.0, len(x) ) ** 3

linecollection = matplotlib.collections.LineCollection( segments, array=colouring_array, cmap="cool", zorder=3 )

ax.add_collection( linecollection )

ax.set_xlim(0.2, 6)
ax.set_xticks( np.pi * np.array([0.5, 1, 1.5]) )
ax.set_xticklabels( ["π/2", "π", "3π/2"] )

ax.set_title( "Phase Plot with Complete Rotations" )
ax.set_xlabel( "Pole Angle" )
ax.set_ylabel( "Pole Angular Velocity" )



# # Task 1.2: Changes in State
#
#

# ## Effect of Varying Initial Cart Velocity and Pole Angle

# In[7]:


# sweep over different initial cart velocities and angles and find the subsequent change in state

# number of steps to vary the intial conditions across their range
Nsteps = 30

# setup some intial conditions to loop over, varying the intial pole angle and cart velocity
```

```
initial_cart_positions  = np.array( [1] )
initial_cart_velocities = np.linspace( -10, 10, num=Nsteps )
initial_pole_angles     = np.linspace( -np.pi, np.pi, num=Nsteps )
initial_pole_angvels    = np.array( [0] )

# create array of initial state vectors

initial_states = np.array( np.meshgrid(
    initial_cart_positions,
    initial_cart_velocities,
    initial_pole_angles,
    initial_pole_angvels
)).T.squeeze()

# get 2d array of subsquent state vectors

subsequent_states = [ CartPole.perform_action( state ) for state in initial_states.reshape( (Nsteps**2,4) ) ]
subsequent_states = np.array( subsequent_states ).reshape( (Nsteps, Nsteps, 4) )

state_changes = subsequent_states - initial_states


# In[8]:


fig, ax = plt.subplots(1, 1, num=5)

# create array of interpolated colours

col_lerp = np.linspace(0, 1, Nsteps)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255


for i, row in enumerate(state_changes):

    # convert to arrays and extract certain components from vectors

    x = initial_states[:,i,2] # extract initial angle
    y = state_changes[:,i,0] # extract change in cart position

    # code to smooth plot lines

    xnew = np.linspace( x.min(), x.max(), 300 )

    y_spline = scipy.interpolate.make_interp_spline(x, y, k=2)
    y_smooth = y_spline(xnew)

    # plot then move onto next line

    ax.plot( xnew, y_smooth, color=colours[i] )

ax.set_title( "Exploration of State Change Function" )
ax.set_xlabel( "Initial Pole Angle" )
ax.set_ylabel( "Subsequent Change in Cart Position" )


#

# In[9]:


fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, num=6, figsize=(9,9))

titles = ["Cart Position", "Cart Velocity", "Pole Angle", "Pole Angular Velocity"]

for i, ax in enumerate( [ax1, ax2, ax3, ax4] ):

    ax.imshow( state_changes[:,:,i], interpolation="bicubic", extent=(-10, 10, -np.pi, np.pi), aspect='auto', cmap="cool", origin='lower' )
    contour = ax.contour( initial_states[0,:,1], initial_states[:,0,2], state_changes[:,:,i], colors="white", linewidths=1 )
    ax.clabel( contour, contour.levels[1::2], inline=True, fontsize=12 )

    ax.set_title( titles[i] )

fig.text(0.5, 0.94, 'Changes in State as a Function of Initial State', ha='center', va='center', fontsize=16)
fig.text(0.5, 0.05, 'Initial Cart Velocity', ha='center', va='center', fontsize=14)
fig.text(0.06, 0.5, 'Initial Pole Angle', ha='center', va='center', rotation='vertical', fontsize=14)


#

# ## Effect of Varying Initial Pole Angle and Angular Velocity

# In[10]:


# sweep over different initial pole angles and angvels and find the subsequent change in state

# number of steps to vary the intial conditions across their range
Nsteps = 30

# setup some intial conditions to loop over, varying the intial pole angle and angular velocity

initial_cart_positions  = np.array( [2] )
initial_cart_velocities = np.array( [4] )
initial_pole_angles     = np.linspace( -np.pi, np.pi, num=Nsteps )
initial_pole_angvels    = np.linspace( -15, 15, num=Nsteps )

# create array of initial state vectors

initial_states = np.array( np.meshgrid(
    initial_cart_positions,
    initial_cart_velocities,
    initial_pole_angles,
    initial_pole_angvels
)).T.squeeze()

# get 2d array of subsquent state vectors

state_changes = [ CartPole.perform_action( state ) - state for state in initial_states.reshape( (Nsteps**2,4) ) ]
state_changes = np.array( state_changes ).reshape( (Nsteps, Nsteps, 4) )

# cache data for later

cache["state_changes_varying_angle_and_angvel"] = state_changes
```

```
# In[11]:


fig, ax = plt.subplots(1, 1, num=8)

# create array of interpolated colours

col_lerp = np.linspace(0, 1, Nsteps)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255


for i, row in enumerate(state_changes):

    # convert to arrays and extract certain components from vectors

    x = initial_states[:,i,3] # extract initial angular velocity
    y = state_changes[:,i,1] # extract change in cart velocity

    # code to smooth plot lines

    xnew = np.linspace( x.min(), x.max(), 300 )

    y_spline = scipy.interpolate.make_interp_spline(x, y, k=2)
    y_smooth = y_spline(xnew)

    # plot then move onto next line

    ax.plot( xnew, y_smooth, color=colours[i] )

ax.set_title( "Exploration of State Change Function" )
ax.set_xlabel( "Initial Pole Angular Velocity" )
ax.set_ylabel( "Subsequent Change in Cart Velocity" )



# In[12]:


fig, ax = plt.subplots(1, 1, num=9)

# create array of interpolated colours

col_lerp = np.linspace(0, 1, Nsteps)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255


for i, row in enumerate(state_changes):

    # convert to arrays and extract certain components from vectors

    x = initial_states[i,:,2] # extract initial angular velocity
    y = state_changes[i,:,1] # extract change in cart velocity

    # code to smooth plot lines

    xnew = np.linspace( x.min(), x.max(), 300 )

    y_spline = scipy.interpolate.make_interp_spline(x, y, k=2)
    y_smooth = y_spline(xnew)

    # plot then move onto next line

    ax.plot( xnew, y_smooth, color=colours[i] )

ax.set_title( "Exploration of State Change Function" )
ax.set_xlabel( "Initial Pole Angle" )
ax.set_ylabel( "Subsequent Change in Cart Velocity" )



# In[13]:


fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, num=11, figsize=(9,9))

titles = ["Cart Position", "Cart Velocity", "Pole Angle", "Pole Angular Velocity"]

for i, ax in enumerate( [ax1, ax2, ax3, ax4] ):

    ax.imshow( state_changes[:,:,i], interpolation="bicubic", extent=(-np.pi, np.pi, -15, 15), aspect='auto', cmap="cool", origin='lower' )
    contour = ax.contour( initial_states[0,:,2], initial_states[:,0,3], state_changes[:,:,i], colors="white", linewidths=1 )
    ax.clabel( contour, contour.levels[1::2], inline=True, fontsize=12 )
    ax.set_title( titles[i] )

fig.text(0.5, 0.94, 'Changes in State as a Function of Initial State', ha='center', va='center', fontsize=16)
fig.text(0.5, 0.05, 'Initial Pole Angle', ha='center', va='center', fontsize=14)
fig.text(0.06, 0.5, 'Initial Pole Angular Velocity', ha='center', va='center', rotation='vertical', fontsize=14)



# # Task 1.3: Linear Model
#
# I assume that the change in state is a linear function of the current state:
#
# $$
# Y = f(X) = XC
# $$
#

#
# ## Generating Training Data

# In[14]:


# set the random seed to make this cell deterministic
np.random.seed(4)

# generate random arrays of 500 values

random_positions  = np.random.rand( 500 ) * 10 - 5
random_velocities = np.random.rand( 500 ) * 20 - 10
random_angles     = np.random.rand( 500 ) * np.pi * 2 - np.pi
random_angvels    = np.random.rand( 500 ) * 30 - 15
```

```
# stack random values into 500 state vectors

X = initial_states = np.stack( [
    random_positions,
    random_velocities,
    random_angles,
    random_angvels
] ).T

Y = subsequent_states = np.array( [ CartPole.perform_action( state ) - state for state in initial_states ] )


# ## Finding the Least-Squares Fit
#
# The Moore-Penrose pseudoinverse can then be used to construct the least-squares matrix that maps $X$ to $Y$:
#
# $$
# Y = XC
# $$
# $$
# X^{+}Y = X^{+}XC = C
# $$

# In[15]:


Xplus = np.linalg.inv(X.T @ X) @ X.T
C = Xplus @ Y

cache["C_large_deviations"] = C


# ## Evaluating the Linear Estimator
#
# We can now compare our linear model to the experimental data we plotted in task 1.2.

# ## Plotting Predicted State Change Against Target State Change

# In[16]:


fig, ((ax1, ax2),(ax3,ax4)) = plt.subplots(2, 2, num=19, figsize=(9,9))

XC = X @ C

titles = ["Cart Position", "Cart Velocity", "Pole Angle", "Pole Angular Velocity"]

for i, ax in enumerate( [ax1, ax2, ax3, ax4] ):

    x, y = (XC)[:,i], Y[:,i]
    c = np.abs(x - y)

    extent = np.max( ( np.concatenate([x, y]) ) ) * 1.2

    ax.scatter( y, x, s=1, c=c, cmap="cool" )
    ax.set_xlim(-extent, extent)
    ax.set_ylim(-extent, extent)

    ax.plot( [-extent, extent], [-extent, extent], color="black", linestyle="dotted" )

    ax.set_title( titles[i] )

fig.text(0.5, 0.94, 'Predicted State Changes vs. Target State Changes', ha='center', va='center', fontsize=16)
fig.text(0.5, 0.05, 'Target State Change', ha='center', va='center', fontsize=14)
fig.text(0.06, 0.5, 'Predicted State Change', ha='center', va='center', rotation='vertical', fontsize=14)



#

# ## Linear Model for Small Deflections Only
#

# In[17]:


# generate random arrays of 500 values

random_positions  = np.random.rand( 500 ) * 0.5 - 0.25
random_velocities = np.random.rand( 500 ) * 0.5 - 0.25
random_angles     = np.random.rand( 500 ) * 0.5 - 0.25
random_angvels    = np.random.rand( 500 ) * 0.5 - 0.25

# stack random values into 500 state vectors

X = initial_states = np.stack( [
    random_positions,
    random_velocities,
    random_angles,
    random_angvels
] ).T

Y = subsequent_states = np.array( [ CartPole.perform_action( state ) - state for state in initial_states ] )


Xplus = np.linalg.inv(X.T @ X) @ X.T
C = Xplus @ Y


fig, ((ax1, ax2),(ax3,ax4)) = plt.subplots(2, 2, num=20, figsize=(9,9))

XC = X @ C

titles = ["Cart Position", "Cart Velocity", "Pole Angle", "Pole Angular Velocity"]

for i, ax in enumerate( [ax1, ax2, ax3, ax4] ):

    x, y = (XC)[:,i], Y[:,i]
    c = np.abs(x - y)

    extent = np.max( ( np.concatenate([x, y]) ) ) * 1.2

    ax.scatter( y, x, s=1, c=c, cmap="cool" )
    ax.set_xlim(-extent, extent)
```

```
    ax.set_ylim(-extent, extent)

    ax.plot( [-extent, extent], [-extent, extent], color="black", linestyle="dotted" )

    ax.set_title( titles[i] )

fig.text(0.5, 0.94, 'Predicted State Changes vs. Target State Changes', ha='center', va='center', fontsize=16)
fig.text(0.5, 0.05, 'Target State Change', ha='center', va='center', fontsize=14)
fig.text(0.05, 0.5, 'Predicted State Change', ha='center', va='center', rotation='vertical', fontsize=14)



# ## Comparing Prediction Contour Plots to Target Contour Plots
#

# In[18]:


C = cache["C_large_deviations"]

# sweep over different initial pole angles and angvels and find the predicted change in state

# number of steps to vary the intial conditions across their range
Nsteps = 30

# setup some intial conditions to loop over, varying the intial pole angle and angular velocity

initial_cart_positions  = np.array( [2] )
initial_cart_velocities = np.array( [4] )
initial_pole_angles     = np.linspace( -np.pi, np.pi, num=Nsteps )
initial_pole_angvels    = np.linspace( -15, 15, num=Nsteps )

# create array of initial state vectors

initial_states = np.array( np.meshgrid(
    initial_cart_positions,
    initial_cart_velocities,
    initial_pole_angles,
    initial_pole_angvels
)).T.squeeze()

# get 2d array of subsquent state vectors

predicted_changes = initial_states.reshape( (Nsteps**2,4) ) @ C
predicted_changes = predicted_changes.reshape( (Nsteps, Nsteps, 4) )

cache["linear_predicted_changes_varying_angle_and_angvel"] = predicted_changes


# In[19]:


state_changes = cache["state_changes_varying_angle_and_angvel"]

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, num=15, figsize=(9,9))

titles = ["Cart Position", "Cart Velocity", "Pole Angle", "Pole Angular Velocity"]

for i, ax in enumerate( [ax1, ax2, ax3, ax4] ):

    ax.imshow( predicted_changes[:,:,i], interpolation="bicubic", extent=(-np.pi, np.pi, -15, 15), aspect='auto', cmap="cool", origin='lower' )
    target_contour = ax.contour( initial_states[0,:,2], initial_states[:,0,3], state_changes[:,:,i], colors="black", linewidths=1 )

    ax.clabel( target_contour, target_contour.levels[1::2], inline=True, fontsize=12 )
    estimate_contour = ax.contour( initial_states[0,:,2], initial_states[:,0,3], predicted_changes[:,:,i], colors="white", linewidths=1 )

    ax.clabel( estimate_contour, estimate_contour.levels[1::2], inline=True, fontsize=12 )
    ax.set_title( titles[i] )

fig.text(0.5, 0.94, 'Linear Estimate of State Change Compared to Target', ha='center', va='center', fontsize=16)
fig.text(0.5, 0.06, 'Initial Pole Angle', ha='center', va='center', fontsize=14)
fig.text(0.06, 0.5, 'Initial Pole Angular Velocity', ha='center', va='center', rotation='vertical', fontsize=14)



# In[20]:


# sweep over different initial cart velocities and angles and find the subsequent change in state

# number of steps to vary the intial conditions across their range
Nsteps = 30

# setup some intial conditions to loop over, varying the intial cart velocity and pole angular velocity

initial_cart_positions  = np.array( [1] )
initial_cart_velocities = np.linspace( -10, 10, num=Nsteps )
initial_pole_angles     = np.array( [2] )
initial_pole_angvels    = np.linspace( -15, 15, num=Nsteps )

# create array of initial state vectors

initial_states = np.array( np.meshgrid(
    initial_cart_positions,
    initial_cart_velocities,
    initial_pole_angles,
    initial_pole_angvels
)).T.squeeze()

# get 2d array of subsquent state vectors

state_changes = [ CartPole.perform_action( state ) - state for state in initial_states.reshape( (Nsteps**2,4) ) ]
state_changes = np.array( state_changes ).reshape( (Nsteps, Nsteps, 4) )

predicted_changes = initial_states.reshape( (Nsteps**2,4) ) @ C
predicted_changes = np.array( predicted_changes ).reshape( (Nsteps, Nsteps, 4) )


# In[21]:


fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, num=16, figsize=(9,9))

titles = ["Cart Position", "Cart Velocity", "Pole Angle", "Pole Angular Velocity"]
```

```
for i, ax in enumerate( [ax1, ax2, ax3, ax4] ):

    ax.imshow( predicted_changes[:,:,i], interpolation="bicubic", extent=(-10, 10, -15, 15), aspect='auto', cmap="cool", origin='lower' )
    target_contour = ax.contour( initial_states[0,:,1], initial_states[:,0,3], state_changes[:,:,i], colors="black", linewidths=1 )

    ax.clabel( target_contour, inline=True, fontsize=8 )
    estimate_contour = ax.contour( initial_states[0,:,1], initial_states[:,0,3], predicted_changes[:,:,i], colors="white", linewidths=1 )

    ax.clabel( estimate_contour, inline=True, fontsize=8 )
    ax.set_title( titles[i] )

fig.text(0.5, 0.95, 'Linear Estimate of State Change Compared to Target', ha='center', va='center', fontsize=14)
fig.text(0.5, 0.04, 'Initial Cart Velocity', ha='center', va='center', fontsize=12)
fig.text(0.06, 0.5, 'Initial Pole Angular Velocity', ha='center', va='center', rotation='vertical', fontsize=12)


#

# # Task 1.4: Prediction of System Evolution

#

# ## Small Oscillations

# In[22]:


actual_states = cache["states_small_oscillations"]

fig, ax = plt.subplots(1, 1, num=22)
sf3utility.setup_phase_portrait( ax )

# small oscillations about stable equilibrium

state = np.array( [0, 0.126, np.pi, 1] )

prediction_states = []

for i in range(100):

    prediction_states.append( state )
    state = state @ ( C + np.identity(4) )
    state[2] = CartPole.remap_angle( state[2] )


prediction_states = np.array( prediction_states )

x = prediction_states[:,1]
y = prediction_states[:,3]

f, u = scipy.interpolate.splprep( [x, y], s=0, per=True )
xint, yint = scipy.interpolate.splev(np.linspace(0, 1, 10000), f)

centerx = (max(x) + min(x)) / 2
centery = (max(y) + min(y)) / 2

ax.set_xlim( centerx - (centerx - min(x)) * 1.12, centerx + (max(x) - centerx) * 1.12 )
ax.set_ylim( centery - (centery - min(y)) * 1.12, centery + (max(y) - centery) * 1.12 )

points   = np.array([xint, yint]).T.reshape(-1, 1, 2)[::-1]
segments = np.concatenate( [points[:-1], points[1:]], axis=1 )[500:]
colouring_array =  np.linspace( 0.0, 1.0, len(xint) )# ** 3

linecollection = matplotlib.collections.LineCollection( segments, array=colouring_array, cmap="cool", zorder=3 )

ax.add_collection( linecollection )

x = actual_states[:,1]
y = actual_states[:,3]

ax.plot( x, y, color="orange", linewidth=3 )

ax.set_title( "Small Oscillation about Supposedly Stable Equilibirum" )
ax.set_xlabel( "Cart Velocity" )
ax.set_ylabel( "Pole Angular Velocity" )


#

# In[65]:


fig, ax = plt.subplots(1, 1, figsize=(8,4.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.08, right=0.99)
sf3utility.setup_phase_portrait( ax )

# small oscillations about stable equilibrium

actual_state = np.array( [0, 0.126, np.pi, 1] )
prediction_state = np.array( [0, 0.126, np.pi, 1] )

actual_states = []
prediction_states = []

for i in range(30):

    prediction_states.append( prediction_state )
    actual_states.append( actual_state )

    prediction_state = prediction_state @ ( C + np.identity(4) )
    prediction_state[2] = CartPole.remap_angle( prediction_state[2] )

    actual_state = CartPole.perform_action( actual_state )

prediction_states = np.array( prediction_states )
actual_states = np.array( actual_states )


pred_interp = scipy.interpolate.make_interp_spline( np.arange(0,30*0.2,0.2), prediction_states[:,0], bc_type="natural" )
actual_interp = scipy.interpolate.make_interp_spline( np.arange(0,30*0.2,0.2), actual_states[:,0], bc_type="natural" )

xint = np.linspace( 0, 30*0.2, num=100 )
pred_int = pred_interp( xint )
actual_int = actual_interp( xint )
```

```
ax.plot( xint, actual_int, linewidth=2, label="Actual Motion", color="orange" )
ax.plot( xint, pred_int, linewidth=2, label="Predicted Motion", color="magenta" )

ax.set_title( "Small Oscillation about Supposedly Stable Equilibirum (Interpolated)" )
ax.set_xlabel( "Seconds Elapsed" )
ax.set_ylabel( "Cart Position" )
ax.legend( loc="upper left" )


# In[26]:


actual_states = cache["states_large_oscillations"]

# large oscillations about stable equilibrium

state = np.array( [0, 1.72, np.pi, 10] )

x, y = [], []

for i in range(500):

    x.append( state[1] )
    y.append( state[3] )

    state += state @ C
    state[2] = CartPole.remap_angle( state[2] )


fig, ax = plt.subplots(1, 1, num=23)
sf3utility.setup_phase_portrait( ax )

centerx = (max(x) + min(x)) / 2
centery = (max(y) + min(y)) / 2

ax.set_xlim( centerx - (centerx - min(x)) * 1.12, centerx + (max(x) - centerx) * 1.12 )
ax.set_ylim( centery - (centery - min(y)) * 1.12, centery + (max(y) - centery) * 1.12 )

points   = np.array([x, y]).T.reshape(-1, 1, 2)[::-1]
segments = np.concatenate( [points[:-1], points[1:]], axis=1 )
colouring_array =  np.linspace( 0.0, 1.0, len(x) )

linecollection = matplotlib.collections.LineCollection( segments, array=colouring_array, cmap="cool", zorder=3 )

ax.add_collection( linecollection )

ax.set_title( "Large Oscillations in the Linear Model" )
ax.set_xlabel( "Cart Velocity" )
ax.set_ylabel( "Pole Angular Velocity" )

states = np.array( states )
```

task2.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


# import CartPole.py from local directory
import CartPole, sf3utility
import matplotlib.collections
import matplotlib.pyplot as plt
import numpy as np
import scipy.interpolate, scipy.stats.qmc, scipy.optimize
import random, copy

plt.rcParams["font.family"] = "Georgia"
#plt.rcParams['figure.figsize'] = [9.0, 7.0]
#plt.rcParams['figure.dpi'] = 400

# store results for later
cache = {}


# In[2]:


# allows nice plots that can be redrawn
get_ipython().run_line_magic('matplotlib', 'notebook')


# # Task 2.1 - Nonlinear Modelling

# ## Generating Training Data
#

# In[3]:


N = 512

np.random.seed(4)

random_positions  = np.random.rand( N ) * 20 - 10
random_velocities = np.random.rand( N ) * 20 - 10
random_angles     = np.random.rand( N ) * np.pi * 2 - np.pi
random_angvels    = np.random.rand( N ) * 30 - 15

# stack random values into 512 state vectors

X_random = initial_states = np.stack( [
    random_positions,
    random_velocities,
    random_angles,
    random_angvels
] ).T


# set the random seed and create the sobol sequence generator
sobol_engine = scipy.stats.qmc.Sobol( d=4, seed=4 )
```

```
# get 512 initial states spaced in the recommended ranges
X = X_sobol = initial_states = (sobol_engine.random_base2( m=int(np.log2(N)) ) - 0.5 ) * 2 * np.array( [ 10, 10, np.pi, 15 ] )

# generate the changes in state from these states
Y = np.array( [ CartPole.perform_action( state ) - state for state in initial_states ] )


M = 16
kernel_centres = X[:M]


# In[4]:


fig, (ax1, ax2) = plt.subplots( 1, 2, num=1, sharey=True, figsize=(9,4.5) )
fig.subplots_adjust(wspace=0.05, bottom=0.15)

c = np.zeros( N )
for i, x in enumerate( X_random ):

    c[i] = np.linalg.norm( np.exp( -3*(X_random[:,0:2] - x[0:2])**2 ) )

ax1.scatter( X_random[:,0], X_random[:,1], s=2, c=c, cmap="cool", label="Datapoints" )
ax1.scatter( X_random[:M,0], X_random[:M,1], s=50, color=[0,0,0,0], edgecolors="cyan", linewidths=2, label="Kernel Centres" )

ax1.set_title( "Randomly Selected Points" )
ax1.legend( loc="upper left", prop={'size': 9}, facecolor='white', framealpha=1 )


c = np.zeros( N )
for i, x in enumerate( X_sobol ):

    c[i] = np.linalg.norm( np.exp( -3*(X_sobol[:,0:2] - x[0:2])**2 ) )


ax2.scatter( X_sobol[:,0], X_sobol[:,1], s=2, c=c, cmap="cool", label="Datapoints" )
ax2.scatter( kernel_centres[:,0], kernel_centres[:,1], s=50, color=[0,0,0,0], edgecolors="cyan", linewidths=2, label="Kernel Centres" )

ax2.set_title( "Sobol Sequence Points" )
ax2.legend( loc="upper left", prop={'size': 9}, facecolor='white', framealpha=1 )


fig.text( 0.06, 0.5, "Initial Cart Velocity", rotation="vertical", ha="center", va="center", fontsize=12 )
fig.text( 0.5, 0.05, "Initial Cart Position", ha="center", va="center", fontsize=12 )


# ## Sobol Sequence


# ## Selection of Kernel Centres


# ## Creating More Training Data

# In[5]:


N = 512 * 8

np.random.seed(4)

# set the random seed and create the sobol sequence generator
sobol_engine = scipy.stats.qmc.Sobol( d=4, seed=3 )

# get N initial states spaced in the recommended ranges
X = X_sobol = initial_states = (sobol_engine.random_base2( m=int(np.log2(N)) ) - 0.5 ) * 2 * np.array( [ 10, 20, np.pi, 20 ] )
#X[:,2] = np.array( [ CartPole.remap_angle(theta) for theta in X[:,2] ] )

# generate the changes in state from these states
Y = np.array( [ CartPole.perform_action( state ) - state for state in initial_states ] )


M = 32 * 32
kernel_centres = X[:M]

cache["X_Y_M_N_kc_fit"] = tuple( copy.deepcopy(x) for x in [X, Y, M, N, kernel_centres])


# ## Define the Nonlinear Kernel

# In[6]:


sigma = np.std( X, axis=0 )

def K( X, Xp, sigma = sigma ):

    # get squared differences and substitute angle one for periodic version
    d2 = ( (X - Xp) / sigma ) ** 2
    d2[:,2] = (np.sin( 0.5 * ( X[:,2] - Xp[:,2] ) ) / sigma[2] ) ** 2

    # divide rows by 2 sigma and return exponential of negative sum along rows
    return np.exp( - 0.5 * np.sum( d2, axis=1 ) )

cache["sigma_fitted"] = copy.deepcopy(sigma)


# ## Construction of Knm Matrices

# In[7]:


# loop over the kernel centres and evaluate the K function across all the Xs at each
Kmn = np.zeros( (M,N) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmn[i] = K( X, kernel_centre[np.newaxis] )

# same as above but only use first M vectors from X
Kmm = np.zeros( (M,M) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmm[i] = K( X[:M], kernel_centre[np.newaxis] )
```

```
# ## Finding the Kernel Coefficients

# In[8]:


l = 1e-4

KmnKnm = Kmn @ Kmn.T
a = KmnKnm + l * Kmm
b = Kmn @ Y

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

cache["alpha_m_fitted"] = copy.deepcopy( alpha_m )
cache["kernel_centres_fitted"] = copy.deepcopy( kernel_centres )


# ## Building the Model

# In[9]:


def nonlinear_model( state ):

    kernels = K( state[np.newaxis], kernel_centres )

    weighted_sums = kernels @ alpha_m

    return weighted_sums.squeeze()

cache["nonlinear_model_fitted"] = copy.deepcopy(nonlinear_model)


# ## Evaluating the Model

# In[10]:


# sweep over different initial pole angles and angvels and find the subsequent change in state

# number of steps to vary the intial conditions across their range
Nsteps = 30

# setup some intial conditions to loop over, varying the intial pole angle and angular velocity

initial_cart_positions  = np.array( [2] )
initial_cart_velocities = np.array( [4] )
initial_pole_angles     = np.linspace( -np.pi, np.pi, num=Nsteps )
initial_pole_angvels    = np.linspace( -15, 15, num=Nsteps )

# create array of initial state vectors

initial_states = np.array( np.meshgrid(
    initial_cart_positions,
    initial_cart_velocities,
    initial_pole_angles,
    initial_pole_angvels
)).T.squeeze()

# get 2d arrays of subsquent state changes and predictions

state_changes = [ CartPole.perform_action( state ) - state for state in initial_states.reshape( (Nsteps**2,4) ) ]
state_changes = np.array( state_changes ).reshape( (Nsteps, Nsteps, 4) )

modelled_changes = [ nonlinear_model( state ) for state in initial_states.reshape( (Nsteps**2,4) ) ]
modelled_changes = np.array( modelled_changes ).reshape( (Nsteps, Nsteps, 4) )


# In[11]:


fig, axs = plt.subplots(2, 4, num=4, figsize=(12,6), sharex=True, sharey=True )
fig.subplots_adjust(wspace=0.05, hspace=0.2, top=0.84, bottom=0.12, left=0.1, right=0.9)

titles = [["Cart Position - Actual", "Cart Velocity - Actual", "Pole Angle - Actual", "Pole Angular Velocity - Actual"],
          ["Cart Position - Predicted", "Cart Velocity - Predicted", "Pole Angle - Predicted", "Pole Angular Velocity - Predicted"]]

# plot nonlinear prediction contours

for row, axrow in enumerate(axs):

    changes = [ state_changes, modelled_changes ][row]

    for col, ax in enumerate(axrow):

        ax.imshow( changes[:,:,col], interpolation="bicubic", extent=(-np.pi, np.pi, -15, 15), aspect='auto', cmap="cool", origin='lower' )
        contour = ax.contour( initial_states[0,:,2], initial_states[:,0,3], changes[:,:,col], colors="white", linewidths=1 )
        ax.clabel( contour, contour.levels[1::2], inline=True, fontsize=12 )

        ax.set_title( titles[row][col] )

fig.text(0.5, 0.94, 'State Changes vs. Nonlinear Prediction Thereof', ha='center', va='center', fontsize=16)
fig.text(0.5, 0.05, 'Initial Pole Angle', ha='center', va='center', fontsize=14)
fig.text(0.06, 0.5, 'Initial Pole Angular Velocity', ha='center', va='center', rotation='vertical', fontsize=14)



# ## Plotting Predicted State Changes Against Target State Changes

# In[12]:


fig, ((ax1, ax2),(ax3,ax4)) = plt.subplots(2, 2, num=19, figsize=(9,9))
fig.subplots_adjust(wspace=0.16, hspace=0.16, top=0.92, bottom=0.08, left=0.08, right=0.96)

titles = ["Cart Position", "Cart Velocity", "Pole Angle", "Pole Angular Velocity"]

predictions = np.array( [ nonlinear_model(state) for state in X ] )

for i, ax in enumerate( [ax1, ax2, ax3, ax4] ):

    x, y = predictions[:,i], Y[:,i]
    c = np.abs(x - y)
```

```
    extent = np.max( ( np.concatenate([x, y]) ) ) * 1.2

    ax.scatter( y, x, s=1, c=c, cmap="cool", zorder=2 )
    ax.set_xlim(-extent, extent)
    ax.set_ylim(-extent, extent)

    ax.plot( [-extent, extent], [-extent, extent], color="black", linestyle="dotted", zorder=1 )

    ax.set_title( titles[i] )

fig.text(0.5, 0.97, 'Predicted State Changes vs. Target State Changes', ha='center', va='center', fontsize=16)
fig.text(0.52, 0.03, 'Target State Change', ha='center', va='center', fontsize=14)
fig.text(0.03, 0.5, 'Predicted State Change', ha='center', va='center', rotation='vertical', fontsize=14)
```

```
# In[13]:
```

```
X, Y, M, N, kernel_centres = cache["X_Y_M_N_kc_fit"]
sigma = cache["sigma_fitted"]
kernel_centres = cache["kernel_centres_fitted"]


fig, axs = plt.subplots(1, 6, num=9, figsize=(14,5), sharey=True, sharex=True)
fig.subplots_adjust(wspace=0.05, top=0.86, bottom=0.12, left=0.06, right=0.99)

Ms = [5,10,15,25,40,100]

for p, ax in enumerate( axs ):

    M = int( 5 + 12*p**1.5 )
    #M = Ms[p]
    kernel_centres = X[:M]

    # loop over the kernel centres and evaluate the K function across all the Xs at each
    Kmn = np.zeros( (M,N) )
    for i, kernel_centre in enumerate( kernel_centres ):

        Kmn[i] = K( X, kernel_centre[np.newaxis] )

    # same as above but only use first M vectors from X
    Kmm = np.zeros( (M,M) )
    for i, kernel_centre in enumerate( kernel_centres ):

        Kmm[i] = K( X[:M], kernel_centre[np.newaxis] )


    l = 1e-4

    KmnKnm = Kmn @ Kmn.T
    a = KmnKnm + l * Kmm
    b = Kmn @ Y

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

    Npoints = 1024

    predictions = np.array( [ nonlinear_model(state) for state in X[:Npoints] ] )

    x, y = predictions[:,3], Y[:Npoints,3]
    c = np.abs(x - y)

    extent = 10

    ax.scatter( y, x, s=1, c=c, cmap="cool", zorder=2 )
    ax.set_xlim(-extent, extent)
    ax.set_ylim(-extent, extent)

    ax.plot( [-extent, extent], [-extent, extent], color="black", linestyle="dotted", zorder=1 )

    ax.set_title( f'{M} Kernels' )
    ax.set_xticklabels( [" ","-5","0","5","10"] )

fig.text(0.5, 0.96, 'Convergence of Model with Addition of Basis Functions - Predictions of Angular Velocity Change', ha='center', va='center',
fontsize=16)
fig.text(0.52, 0.03, 'Target State Change', ha='center', va='center', fontsize=14)
fig.text(0.025, 0.5, 'Predicted State Change', ha='center', va='center', rotation='vertical', fontsize=14)
```

```
# ## Performing Rollouts with the Nonlinear Model
```

```
# In[14]:
```

```
X, Y, M, N, kernel_centres = cache["X_Y_M_N_kc_fit"]
kernel_centres = cache["kernel_centres_fitted"]
alpha_m = cache["alpha_m_fitted"]
sigma = cache["sigma_fitted"]
nonlinear_model = cache["nonlinear_model_fitted"]

fig, ax = plt.subplots(1, 1, num=15, figsize=(10,2.5))
fig.subplots_adjust(bottom=0.2, right=0.99, left=0.1)
sf3utility.setup_phase_portrait( ax )

# small oscillations about stable equilibrium

predicted_state = np.array( [0, -2, np.pi, 4] )
actual_state = np.array( [0, -2, np.pi, 4] )

prediction_states = []
actual_states = []
time = []

for i in range(50):

    prediction_states.append( predicted_state )
    actual_states.append( actual_state )
    time.append( i * 0.2 )

    predicted_state = nonlinear_model( predicted_state ) + predicted_state
    actual_state    = CartPole.perform_action( actual_state )
```

```
prediction_states = np.array( prediction_states )
actual_states = np.array( actual_states )
time = np.array( time )

x = time
y1 = prediction_states[:,1]
y2 = actual_states[:,1]


f, u = scipy.interpolate.splprep( [x, y1], s=0, per=True )
xint, y1int = scipy.interpolate.splev(np.linspace(0, 1, 10000), f)
xint, y1int = xint[:5000], y1int[:5000]

f, u = scipy.interpolate.splprep( [x, y2], s=0, per=True )
_, y2int = scipy.interpolate.splev(np.linspace(0, 1, 10000), f)
y2int = y2int[:5000]

ax.plot( x, y2, color="orange", linewidth=2, label="Actual" )
ax.plot( x, y1, color="magenta", linewidth=2, label="Nonlinear Model" )

ax.set_title("Nonlinear Model Time Evolution Prediction vs. Actual")
ax.set_xlabel("Seconds Elapsed")
ax.set_ylabel("Cart Velocity")


#

# ## Model Ignoring Cart Position


# In[15]:


N = 512 * 8

# set the random seed and create the sobol sequence generator
sobol_engine = scipy.stats.qmc.Sobol( d=4, seed=4 )

# get M initial states spaced in the recommended ranges
X = X_sobol = initial_states = (sobol_engine.random_base2( m=int(np.log2(N)) ) - 0.5 ) * 2 * np.array( [ 10, 20, np.pi, 20 ] )
#X[:,2] = np.array( [ CartPole.remap_angle(theta) for theta in X[:,2] ] )

# generate the changes in state from these states
Y = np.array( [ CartPole.perform_action( state ) - state for state in initial_states ] )


M = 32 * 32
kernel_centres = X[:M]


sigma = np.std( X, axis=0 ) * 0.9

def K( X, Xp, sigma = sigma ):

    # get squared differences and substitute angle one for periodic version
    d2 = ( (X - Xp) / sigma ) ** 2
    d2[:,0] = 0
    d2[:,2] = (np.sin( 0.5 * ( X[:,2] - Xp[:,2] ) ) / sigma[2] ) ** 2

    # divide rows by 2 sigma and return exponential of negative sum along rows
    return np.exp( - 0.5 * np.sum( d2, axis=1 ) )


# loop over the kernel centres and evaluate the K function across all the Xs at each
Kmn = np.zeros( (M,N) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmn[i] = K( X, kernel_centre[np.newaxis] )

# same as above but only use first M vectors from X
Kmm = np.zeros( (M,M) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmm[i] = K( X[:M], kernel_centre[np.newaxis] )


l = 1e-4

KmnKnm = Kmn @ Kmn.T
a = KmnKnm + l * Kmm
b = Kmn @ Y

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

def nonlinear_model( state ):

    kernels = K( state[np.newaxis], kernel_centres )

    weighted_sums = kernels @ alpha_m

    return weighted_sums.squeeze()

# small oscillations about stable equilibrium

predicted_state = np.array( [0, -2, np.pi, 3] )

prediction_states = []
time = []

for i in range(50):

    prediction_states.append( predicted_state )
    time.append( i * 0.2 )

    predicted_state = nonlinear_model( predicted_state ) + predicted_state

prediction_states = np.array( prediction_states )
time = np.array( time )

x = time
y1 = prediction_states[:,1]

ax.plot( x, y1, color="cyan", linewidth=2, label="Nonlinear Model Ignoring Cart Position" )
```

```
ax.legend( loc="upper left", prop={'size': 9}, facecolor='white', framealpha=1 )


# In[16]:


N = 512 * 8

# set the random seed and create the sobol sequence generator
sobol_engine = scipy.stats.qmc.Sobol( d=4, seed=4 )

# get M initial states spaced in the recommended ranges
X = X_sobol = initial_states = (sobol_engine.random_base2( m=int(np.log2(N)) ) - 0.5 ) * 2 * np.array( [ 5, 2, 1, 2 ] ) + np.array([0,0,np.pi,0])
#X[:,2] = np.array( [ CartPole.remap_angle(theta) for theta in X[:,2] ] )

# generate the changes in state from these states
Y = np.array( [ CartPole.perform_action( state ) - state for state in initial_states ] )


M = 32 * 32
kernel_centres = X[:M]


sigma = np.std( X, axis=0 ) * 0.9

def K( X, Xp, sigma = sigma ):

    # get squared differences and substitute angle one for periodic version
    d2 = ( (X - Xp) / sigma ) ** 2
    d2[:,2] = (np.sin( 0.5 * ( X[:,2] - Xp[:,2] ) ) / sigma[2] ) ** 2

    # divide rows by 2 sigma and return exponential of negative sum along rows
    return np.exp( - 0.5 * np.sum( d2, axis=1 ) )


# loop over the kernel centres and evaluate the K function across all the Xs at each
Kmn = np.zeros( (M,N) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmn[i] = K( X, kernel_centre[np.newaxis] )

# same as above but only use first M vectors from X
Kmm = np.zeros( (M,M) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmm[i] = K( X[:M], kernel_centre[np.newaxis] )


l = 1e-4

KmnKnm = Kmn @ Kmn.T
a = KmnKnm + l * Kmm
b = Kmn @ Y

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

def nonlinear_model( state ):

    kernels = K( state[np.newaxis], kernel_centres )

    weighted_sums = kernels @ alpha_m

    return weighted_sums.squeeze()


# In[17]:


#kernel_centres = cache["kernel_centres_fitted"]
#alpha_m = cache["alpha_m_fitted"]

fig, ax = plt.subplots(1, 1, num=14, figsize=(10,2.5))
fig.subplots_adjust(bottom=0.2, right=0.99, left=0.1)
sf3utility.setup_phase_portrait( ax )

# small oscillations about stable equilibrium

predicted_state = np.array( [0, 0.126, np.pi, 1] )
actual_state = np.array( [0, 0.126, np.pi, 1] )

prediction_states = []
actual_states = []
time = []

for i in range(100):

    prediction_states.append( predicted_state )
    actual_states.append( actual_state )
    time.append( i * 0.2 )

    predicted_state = nonlinear_model( predicted_state ) + predicted_state
    actual_state    = CartPole.perform_action( actual_state )

prediction_states = np.array( prediction_states )
actual_states = np.array( actual_states )
time = np.array( time )

x = time
y1 = prediction_states[:,1]
y2 = actual_states[:,1]


f, u = scipy.interpolate.splprep( [x, y1], s=0, per=True )
xint, y1int = scipy.interpolate.splev(np.linspace(0, 1, 10000), f)
xint, y1int = xint[:5000], y1int[:5000]

f, u = scipy.interpolate.splprep( [x, y2], s=0, per=True )
_, y2int = scipy.interpolate.splev(np.linspace(0, 1, 10000), f)
y2int = y2int[:5000]

ax.plot( xint, y2int, color="orange", linewidth=2, label="Actual" )
ax.plot( xint, y1int, color="magenta", linewidth=2, label="Predicted" )

ax.set_title("Specialised Model Time Evolution Prediction vs. Actual (Interpolated)")
ax.set_xlabel("Seconds Elapsed")
```

```
ax.set_ylabel("Cart Velocity")

ax.legend( loc="upper left", prop={'size': 9}, facecolor='white', framealpha=1 )


# ## Descoping the Model


# ## Nonlinear Model Evaluation
#

# # Task 2.2 - Modelling Effect of Force Input

# In[18]:


# sweep over different initial pole angles and angvels and find the subsequent change in state

# number of steps to vary the intial conditions across their range
Nsteps = 40

# setup some intial conditions to loop over, varying the intial pole angle and angular velocity

initial_cart_positions  = np.array( [2] )
initial_cart_velocities = np.array( [0] )
initial_pole_angles     = np.linspace( -np.pi, np.pi, num=Nsteps ) # np.array( [3] )
initial_pole_angvels    = np.array( [0] ) # np.linspace( -15, 15, num=Nsteps )
initial_actions         = np.linspace( -17, 17, num=Nsteps )

# create array of initial state vectors

initial_states = np.array( np.meshgrid(
    initial_cart_positions,
    initial_cart_velocities,
    initial_pole_angles,
    initial_pole_angvels,
    initial_actions
)).T.squeeze()

# get 2d array of subsquent state vectors

state_changes = [ CartPole.perform_action_RK4( state ) - state for state in initial_states.reshape( (Nsteps**2,5) ) ]
state_changes = np.array( state_changes ).reshape( (Nsteps, Nsteps, 5) )

# cache data for later

#cache["state_changes_varying_angvel_and_force"] = state_changes


# In[19]:


fig, ax = plt.subplots(1, 1, num=8)

# create array of interpolated colours

col_lerp = np.linspace(0, 1, Nsteps)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255


for i, row in enumerate(state_changes):

    # convert to arrays and extract certain components from vectors

    x = initial_states[:,i,4] # extract initial force
    y = state_changes[:,i,2] # extract change in angle

    # code to smooth plot lines

    xnew = np.linspace( x.min(), x.max(), 300 )

    y_spline = scipy.interpolate.make_interp_spline(x, y, k=2)
    y_smooth = y_spline(xnew)

    # plot then move onto next line

    ax.plot( xnew, y_smooth, color=colours[i] )

ax.set_title( "Exploration of State Change Function" )
ax.set_xlabel( "Initial Input Action" )
ax.set_ylabel( "Subsequent Change in Angle" )


# ## Effect of Input Force


# In[20]:


N = 512 * 8

# set the random seed and create the sobol sequence generator
sobol_engine = scipy.stats.qmc.Sobol( d=5, seed=4 )

# get M initial states spaced in the recommended ranges
X = X_sobol = initial_states = (sobol_engine.random_base2( m=int(np.log2(N)) ) - 0.5 ) * 2 * np.array( [ 10, 20, np.pi, 20, 15 ] )
#X[:,2] = np.array( [ CartPole.remap_angle(theta) for theta in X[:,2] ] )

# generate the changes in state from these states
Y = np.array( [ CartPole.perform_action5( state ) - state for state in initial_states ] )


M = 32 * 32
kernel_centres = X[:M]


sigma = np.std( X, axis=0 )

def K( X, Xp, sigma = sigma ):

    # get squared differences and substitute angle one for periodic version
    d2 = ( (X - Xp) / sigma ) ** 2
    d2[:,0] = 0
    d2[:,2] = (np.sin( 0.5 * ( X[:,2] - Xp[:,2] ) ) / sigma[2] ) ** 2
```

```
    # divide rows by 2 sigma and return exponential of negative sum along rows
    return np.exp( - 0.5 * np.sum( d2, axis=1 ) )


# loop over the kernel centres and evaluate the K function across all the Xs at each
Kmn = np.zeros( (M,N) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmn[i] = K( X, kernel_centre[np.newaxis] )

# same as above but only use first M vectors from X
Kmm = np.zeros( (M,M) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmm[i] = K( X[:M], kernel_centre[np.newaxis] )


l = 1e-4

KmnKnm = Kmn @ Kmn.T
a = KmnKnm + l * Kmm
b = Kmn @ Y

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

def nonlinear_model( state ):

    kernels = K( state[np.newaxis], kernel_centres )

    weighted_sums = kernels @ alpha_m

    return weighted_sums.squeeze()


# In[21]:


fig, ((ax1, ax2),(ax3,ax4)) = plt.subplots(2, 2, num=44, figsize=(9,9))
fig.subplots_adjust(wspace=0.16, hspace=0.16, top=0.92, bottom=0.08, left=0.08, right=0.96)

titles = ["Cart Position", "Cart Velocity", "Pole Angle", "Pole Angular Velocity"]

predictions = np.array( [ nonlinear_model(state) for state in X ] )

for i, ax in enumerate( [ax1, ax2, ax3, ax4] ):

    x, y = predictions[:,i], Y[:,i]
    c = np.abs(x - y)

    extent = np.max( ( np.concatenate([x, y]) ) ) * 1.2

    ax.scatter( y, x, s=1, c=c, cmap="cool", zorder=2 )
    ax.set_xlim(-extent, extent)
    ax.set_ylim(-extent, extent)

    ax.plot( [-extent, extent], [-extent, extent], color="black", linestyle="dotted", zorder=1 )

    ax.set_title( titles[i] )

fig.text(0.5, 0.97, 'Predicted State Changes vs. Target State Changes', ha='center', va='center', fontsize=16)
fig.text(0.52, 0.03, 'Target State Change', ha='center', va='center', fontsize=14)
fig.text(0.03, 0.5, 'Predicted State Change', ha='center', va='center', rotation='vertical', fontsize=14)



# ## Target State Change vs Predictions, Including Force


# In[22]:


# sweep over different initial pole angles and angvels and find the subsequent change in state

# number of steps to vary the intial conditions across their range
Nsteps = 30

# setup some intial conditions to loop over, varying the intial pole angle and angular velocity

initial_cart_positions  = np.array( [2] )
initial_cart_velocities = np.array( [4] )
initial_pole_angles     = np.linspace( -np.pi, np.pi, num=Nsteps )
initial_pole_angvels    = np.array( [3] )
initial_actions         = np.linspace( -15, 15, num=Nsteps )

# create array of initial state vectors

initial_states = np.array( np.meshgrid(
    initial_cart_positions,
    initial_cart_velocities,
    initial_pole_angles,
    initial_pole_angvels,
    initial_actions
)).T.squeeze()

# get 2d arrays of subsquent state changes and predictions

state_changes = [ CartPole.perform_action5( state ) - state for state in initial_states.reshape( (Nsteps**2,5) ) ]
state_changes = np.array( state_changes ).reshape( (Nsteps, Nsteps, 5) )

modelled_changes = [ nonlinear_model( state ) for state in initial_states.reshape( (Nsteps**2,5) ) ]
modelled_changes = np.array( modelled_changes ).reshape( (Nsteps, Nsteps, 5) )


# In[23]:


fig, axs = plt.subplots(2, 4, num=45, figsize=(12,6), sharex=True, sharey=True )
fig.subplots_adjust(wspace=0.05, hspace=0.2, top=0.84, bottom=0.12, left=0.1, right=0.9)

titles = [["Cart Position - Actual", "Cart Velocity - Actual", "Pole Angle - Actual", "Pole Angular Velocity - Actual"],
          ["Cart Position - Predicted", "Cart Velocity - Predicted", "Pole Angle - Predicted", "Pole Angular Velocity - Predicted"]]

# plot nonlinear prediction contours

for row, axrow in enumerate(axs):
```

```
    changes = [ state_changes, modelled_changes ][row]

    for col, ax in enumerate(axrow):

        ax.imshow( changes[:,:,col], interpolation="bicubic", extent=(-np.pi, np.pi, -15, 15), aspect='auto', cmap="cool", origin='lower' )
        contour = ax.contour( initial_states[0,:,2], initial_states[:,0,4], changes[:,:,col], colors="white", linewidths=1 )
        ax.clabel( contour, contour.levels[1::2], inline=True, fontsize=12 )

        ax.set_title( titles[row][col] )

fig.text(0.5, 0.94, 'State Changes vs. Nonlinear Prediction Thereof', ha='center', va='center', fontsize=16)
fig.text(0.5, 0.05, 'Initial Pole Angle', ha='center', va='center', fontsize=14)
fig.text(0.06, 0.5, 'Initial Input Force', ha='center', va='center', rotation='vertical', fontsize=14)



# In[24]:


fig, ax = plt.subplots(1, 1, num=43, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.08, right=0.99)
sf3utility.setup_phase_portrait( ax )

# small oscillations about stable equilibrium

actual_state = np.array(    [0, 5.5, np.pi - 1, 3, -1] )
predicted_state = np.array( [0, 5.5, np.pi - 1, 3, -1] )

prediction_states = []
actual_states = []
time = []

for i in range(50):

    prediction_states.append( predicted_state )
    actual_states.append( actual_state )
    time.append( i * 0.2 )

    predicted_state = nonlinear_model( predicted_state ) + predicted_state
    actual_state = CartPole.perform_action5( actual_state )

prediction_states = np.array( prediction_states )
actual_states = np.array( actual_states )
time = np.array( time )

x = time
y1 = prediction_states[:,1]
y2 = actual_states[:,1]

ax.plot( x, y1, color="magenta", linewidth=2, label="Predicted" )
ax.plot( x, y2, color="orange", linewidth=2, label="Actual" )

ax.set_title( "Time Evolution of Nonlinear Predictions Including Input Force" )
ax.set_xlabel( "Elapsed Time" )
ax.set_ylabel( "Cart Velocity" )

ax.legend( loc="upper right", prop={'size': 9}, facecolor='white', framealpha=1 )


# ## Time Evolution Modelling
#
```

task2.3.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


# import CartPole.py from local directory
import CartPole, sf3utility
import matplotlib.collections
import matplotlib.pyplot as plt
import numpy as np
import scipy.interpolate, scipy.stats.qmc, scipy.optimize
import random, copy

plt.rcParams["font.family"] = "Georgia"
#plt.rcParams['figure.figsize'] = [9.0, 7.0]
#plt.rcParams['figure.dpi'] = 400

# store results for later
cache = {}


# In[2]:


# allows nice plots that can be redrawn
get_ipython().run_line_magic('matplotlib', 'notebook')


# # Task 2.3 - Linear Control
#
#
#
#

# In[3]:


def rollout_loss( initial_state, p ):

    state = initial_state
    sim_seconds = 10
    sim_steps = int( sim_seconds / 0.2 )
    loss = 0

    for i in range( sim_steps ):

        if np.any( np.abs(state) > np.array([10,40,50,1e+3,1e+3]) ):
            return sim_steps
```

```
        state[4] = p @ state
        loss += CartPole.loss( state[:4] )
        state = CartPole.perform_action_RK4( state )

    return loss


# ## Create Loss Contour Plots

# In[4]:


# number of steps to vary the intial conditions across their range
Nsteps = 10

initial_state = np.array( [0.2,0,0.2,0,0] )

# setup some p vectors

p0 = np.linspace( -1, 4, num=Nsteps )
p1 = np.linspace( -1, 4, num=Nsteps )
p2 = np.array( [17.06] ) # np.linspace( 0, 30, num=Nsteps )
p3 = np.array( [2.66] ) # np.linspace( 0, 4, num=Nsteps )
p4 = np.array( [0] )

# create array of p vectors

ps1 = np.array( np.meshgrid( p0, p1, p2, p3, p4 )).T.squeeze()
ps1_flat = ps1.reshape( (Nsteps**2,5) )

losses1_flat = np.array([ rollout_loss( initial_state, p ) for p in ps1_flat ])
losses1 = losses1_flat.reshape( (Nsteps, Nsteps) )

# setup some p vectors

p0 = np.array( [0.75] ) # np.linspace( -4, 4, num=Nsteps )
p1 = np.array( [1.44] ) # np.linspace( -4, 4, num=Nsteps )
p2 = np.linspace( 10, 30, num=Nsteps )
p3 = np.linspace( 2, 4, num=Nsteps )
p4 = np.array( [0] )

# create array of p vectors

ps2 = np.array( np.meshgrid( p0, p1, p2, p3, p4 )).T.squeeze()
ps2_flat = ps2.reshape( (Nsteps**2,5) )

losses2_flat = np.array([ rollout_loss( initial_state, p ) for p in ps2_flat ])
losses2 = losses2_flat.reshape( (Nsteps, Nsteps) )


# In[5]:


fig, (ax1, ax2) = plt.subplots( 1, 2, num=51, figsize=(10,5) )
fig.subplots_adjust( top=0.9, bottom=0.11, left=0.08, right=0.96 )

ax1.imshow( losses1, interpolation="bicubic", extent=(-1, 4, -1, 4), aspect='auto', cmap="cool", origin='lower' )
contour1 = ax1.contour( ps1[:,0,0], ps1[0,:,1], losses1, colors="white", linewidths=1 )
ax1.clabel( contour1, contour1.levels[1::2], inline=True, fontsize=12 )
ax1.set_xlabel( "p[0]" )
ax1.set_ylabel( "p[1]" )

ax2.imshow( losses2, interpolation="bicubic", extent=(10, 30, 2, 4), aspect='auto', cmap="cool", origin='lower' )
contour2 = ax2.contour( ps2[0,:,2], ps2[:,0,3], losses2, colors="white", linewidths=1 )
ax2.clabel( contour2, contour2.levels[1::2], inline=True, fontsize=12 )
ax2.set_xlabel( "p[2]" )
ax2.set_ylabel( "p[3]" )



#

# ## Optimise Policy Vector

# In[6]:


%%time

initial_state = np.array( [ 0.2, 0, 0.2, 0, 0 ] )
#initial_state = np.array( [1,0,0.5,0,0] )

pad0 = lambda p : np.pad( p, (0,1) )
rollout_loss_from_initial = lambda p : rollout_loss( initial_state, pad0(p) )

min_loss = 999999
best_p = None

for i in range(100):

    p0     = np.random.rand( (4) ) - 0.5
    result = scipy.optimize.minimize( rollout_loss_from_initial, p0, method="Nelder-Mead" )

    end_loss = rollout_loss_from_initial( result["x"] )

    if end_loss < min_loss:
        best_p = result["x"]
        min_loss = end_loss

    print(best_p, min_loss)

p = best_p


## Simulation Using Optimised Policy Vector

# In[171]:


cart_pole = CartPole.CartPole( visual=True, save_frames=False, fig_num=48 )

state = np.array( [ 0, 0, 0.65, 0, 0 ] )
#p = np.array( [0.76187176,  1.19122763, 16.3309897 ,  2.49340076, 0] ) # optimised from theta=0.1
#p = np.array( [0.78060594,  1.22653498, 16.50730499,  2.52886043, 0] ) # optimised from theta=0.5
```

```
p = np.array( [0.75971638,  1.44284499, 17.05744679,  2.66183469, 0] ) # optimised from theta=0.5
#p = np.array ( [ -0.33143221, -0.42699904, -0.09783966,  0.10055861, 0 ] )
#p = np.array( [2.13646362,  2.594651,   17.62534208,  3.10145759, 0] ) # optimised on model
#p = np.array( [2.29932985, 3.80458197, 22.57529485, 3.9609772, 0] ) # better? optimised on model

#p = np.array( [2.29932985, 3.80458197, 22.57529485, 3.9609772, 0] ) # optimised on model

#p = - np.array( [-0.20792587, -0.4554855, -12.8760437,  -1.92181886, 0] ) # dlqr!!
#p = - np.array( [-0.42004789,  -0.754576  , -13.5778077 ,  -2.08834533, 0] ) # dlqr!! small R
#p = - np.array( [ -0.02581708,  -0.1272267 , -11.95653773,  -1.73874922, 0] ) # dlqr!! big R
p = - np.array( [ -0.02533298,  -0.14828819, -12.0167036 ,  -1.7515722, 0] ) # dlqr custom q

#p = np.array([5.15790716,  0.31123075, -0.43806554, -2.78868611, 0])

cart_pole.set_state( state )

cart_pole._render()


# In[172]:


for i in range(50):

    clipped_state = state
    clipped_state[0] = np.clip( state[0], -2, 2 )

    state[4] = p @ clipped_state
    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()


# ## Stabilised Range of Initial Conditions
#
#

# In[9]:


fig, ax = plt.subplots(1, 1, num=49, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.08, right=0.99)
sf3utility.setup_phase_portrait( ax )

# small oscillations about stable equilibrium

state = np.array( [40, 0, 0.5, 0, 0] )
unclipped_state = np.array( [40, 0, 0.5, 0, 0] )

states = []
unclipped_states = []
time = []

for i in range(40):

    states.append( state )
    unclipped_states.append( unclipped_state )
    time.append( i * 0.2 )

    clipped_state = state
    clipped_state[0] = np.clip( state[0], -2, 2 )

    state[4] = p @ clipped_state
    state = CartPole.perform_action_RK4( state )


    unclipped_state[4] = p @ unclipped_state
    unclipped_state = CartPole.perform_action_RK4( unclipped_state )


states = np.array( states )
unclipped_states = np.array( unclipped_states )
time = np.array( time )

x = time
y = states
y2 = unclipped_states

col_lerp = np.linspace(0, 1, 4)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255

labels = ["$x$","$\dot{x}$","$\theta$","$\dot{\theta}$"]

for i in range(4):

    f = scipy.interpolate.interp1d(x, y[:,i], kind='cubic')

    xnew = np.linspace(0, time[-1], 800)

    ax.plot( xnew, f(xnew), color=colours[i], label=labels[i], linewidth=2 )


    f2 = scipy.interpolate.interp1d(x, y2[:,i], kind='cubic')

    ax.plot( xnew, f2(xnew), color=colours[i], linestyle="dotted" )

ax.set_title( "Time Evolution of State Vector under Linear Policy (Interpolated)" )
ax.set_xlabel( "Elapsed Time" )
ax.set_ylabel( "Components of State Vector" )
ax.legend(loc="upper right")

ax.set_ylim( -5, 5 )
ax.set_xlim( -0.4, 7.7 )


# ## State Vector Evolution with Time
#

# # Task 2.4 - Model Predictive Control
#

# ## Refitting Nonlinear Model

# In[85]:
```

```
N = 512 * 8

# set the random seed and create the sobol sequence generator
sobol_engine = scipy.stats.qmc.Sobol( d=5, seed=4 )

# get N initial states spaced in the recommended ranges
X = X_sobol = initial_states = (sobol_engine.random_base2( m=int(np.log2(N)) ) - 0.5 ) * 2 * np.array( [ 10, 20, np.pi, 20, 15 ] )

# generate the changes in state from these states
Y = np.array( [ CartPole.perform_action_RK4( state ) - state for state in initial_states ] )


M = 32 * 32
kernel_centres = X[:M]


sigma = np.std( X, axis=0 )

def K( X, Xp, sigma = sigma ):

    # get squared differences and substitute angle one for periodic version
    d2 = ( (X - Xp) / sigma ) ** 2
    d2[:,0] = 0
    d2[:,2] = (np.sin( 0.5 * ( X[:,2] - Xp[:,2] ) ) / sigma[2] ) ** 2

    # divide rows by 2 sigma and return exponential of negative sum along rows
    return np.exp( - 0.5 * np.sum( d2, axis=1 ) )


# loop over the kernel centres and evaluate the K function across all the Xs at each
Kmn = np.zeros( (M,N) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmn[i] = K( X, kernel_centre[np.newaxis] )

# same as above but only use first M vectors from X
Kmm = np.zeros( (M,M) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmm[i] = K( X[:M], kernel_centre[np.newaxis] )


l = 1e-4

KmnKnm = Kmn @ Kmn.T
a = KmnKnm + l * Kmm
b = Kmn @ Y

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

def nonlinear_model( state ):

    kernels = K( state[np.newaxis], kernel_centres )

    weighted_sums = kernels @ alpha_m

    return weighted_sums.squeeze()


# ## Simulation Using Modelled Dynamics

# In[72]:


cart_pole = CartPole.CartPole( visual=True, save_frames=False, fig_num=58 )

state = np.array( [ 8, 0, -0.6, 0, 0 ] )
#p = np.array( [0.76187176,  1.19122763, 16.3309897 ,  2.49340076, 0] ) # optimised from theta=0.1
#p = np.array( [0.75971638,  1.44284499, 17.05744679,  2.66183469, 0] ) # optimised from theta=0.5
#p = np.array( [ -0.33143221, -0.42699904, -0.09783966,  0.10055861, 0 ] )
#p = np.array( [2.13646362,  2.594651,   17.62534208,  3.10145759, 0] ) # optimised on model
#p = np.array( [2.29932985, 3.80458197, 22.57529485, 3.9609772, 0] ) # better? optimised on model
p = np.array( [ 2.42158373, 3.81812947, 22.41281719, 3.95562641, 0 ] ) # optimised on model with 8 second horizon
#p = np.array( [11.91259675, 8.02276575, 29.81324002, 5.60813997, 0] ) # different initial state (bad)

cart_pole.set_state( state )

cart_pole._render()


# In[75]:


for i in range(50):

    clipped_state = state
    clipped_state[0] = np.clip( state[0], -2, 2 )

    state[4] = p @ clipped_state
    #state = nonlinear_model( state ) + state
    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()


# ## Create Model Rollout Loss Function

# In[13]:


def model_rollout_loss( initial_state, p ):

    state = initial_state
    sim_seconds = 8
    sim_steps = int( sim_seconds / 0.2 )
    loss = 0

    for i in range( sim_steps ):

        if np.any( np.abs(state) > np.array([10,40,50,1e+3,1e+3]) ):
            return sim_steps

        state[4] = p @ state
        loss += CartPole.loss( state[:4] )
```

```
        state = nonlinear_model( state ) + state

    return loss


# ## Optimise the Policy Vector using Modelled Dynamics

# In[84]:


%%time

# initial_state = np.array( [ 0.2, 0, 0.2, 0, 0 ] )
initial_state = np.array( [ 0.4, 0, -0.4, 0, 0 ] ) # larger initial state for better generality
#initial_state = np.array( [1,0,0.5,0,0] )

pad0 = lambda p : np.pad( p, (0,1) )
model_rollout_loss_from_initial = lambda p : model_rollout_loss( initial_state, pad0(p) )

min_loss = 999999
best_p = None

for i in range(100):

    p0     = np.array( [0.78060594,  1.22653498, 16.50730499,  2.52886043] ) + np.random.rand( 4 ) * 3 - 1.5
    result = scipy.optimize.minimize( model_rollout_loss_from_initial, p0, method="Nelder-Mead" )

    end_loss = result["fun"]

    if end_loss < min_loss:
        best_p = result["x"]
        min_loss = end_loss

    print(best_p, min_loss)

p = pad0(best_p)


# ## Model-Based Optimisation
#

# In[87]:


fig, ax = plt.subplots(1, 1, num=66, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.08, right=0.99)
sf3utility.setup_phase_portrait( ax )

# small oscillations about stable equilibrium

state = np.array( [ 8, 0, -0.6, 0, 0 ] )

states = []
time = []

for i in range(55):

    states.append( state )
    time.append( i * 0.2 )

    clipped_state = state
    clipped_state[0] = np.clip( state[0], -2, 2 )

    state[4] = p @ clipped_state
    state = nonlinear_model( state ) + state

states = np.array( states )
time = np.array( time )

x = time
y = states

col_lerp = np.linspace(0, 1, 4)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255

labels = ["$x$","$\dot{x}$","$\theta$","$\dot{\theta}$"]

for i in range(4):

    f2 = scipy.interpolate.interp1d(x, y[:,i], kind='cubic')

    xnew = np.linspace(0, time[-1], 800)

    ax.plot( xnew, f2(xnew), color=colours[i], label=labels[i], linewidth=2 )

ax.set_title( "Time Evolution of State Vector under Linear Policy and Modelled Dynamics (Interpolated)" )
ax.set_xlabel( "Elapsed Time" )
ax.set_ylabel( "Components of State Vector" )
ax.legend(loc="upper right")

# ax.set_ylim( -5, 5 )
ax.set_xlim( -0.4, 10 )


#
#

# In[88]:


fig, ax = plt.subplots(1, 1, num=64, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.08, right=0.99)
sf3utility.setup_phase_portrait( ax )

# small oscillations about stable equilibrium

state = np.array( [ 8, 0, -0.6, 0, 0 ] )

states = []
time = []

for i in range(40):

    states.append( state )
    time.append( i * 0.2 )
```

```
    clipped_state = state
    clipped_state[0] = np.clip( state[0], -2, 2 )

    state[4] = p @ clipped_state
    state = CartPole.perform_action_RK4( state )

states = np.array( states )
time = np.array( time )

x = time
y = states

col_lerp = np.linspace(0, 1, 4)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255

labels = ["$x$","$\dot{x}$","$\theta$","$\dot{\theta}$"]

for i in range(4):

    f2 = scipy.interpolate.interp1d(x, y[:,i], kind='cubic')

    xnew = np.linspace(0, time[-1], 800)

    ax.plot( xnew, f2(xnew), color=colours[i], label=labels[i], linewidth=2  )

ax.set_title( "Time Evolution of State Vector under Linear Policy and Actual Dynamics (Interpolated)" )
ax.set_xlabel( "Elapsed Time" )
ax.set_ylabel( "Components of State Vector" )
ax.legend(loc="upper right")

# ax.set_ylim( -5, 5 )
ax.set_xlim( -0.4, 7.5 )


# ## Return to Actual Dynamics
#
```

task3.py

```
#!/usr/bin/env python
# coding: utf-8

# # Task 3.1 - Introduction of Noise to Observed Dynamics

# In[45]:


# import CartPole.py from local directory
import CartPole, sf3utility
import matplotlib.collections
import matplotlib.pyplot as plt
import numpy as np
import scipy.interpolate, scipy.stats.qmc, scipy.optimize
import random, copy

plt.rcParams["font.family"] = "Georgia"
#plt.rcParams['figure.figsize'] = [9.0, 7.0]
#plt.rcParams['figure.dpi'] = 400

# store results for later
cache = {}


# In[46]:


get_ipython().run_line_magic('matplotlib', 'notebook')


# ## Generate Training Data

# In[47]:


N = 512 * 2

np.random.seed(4)

# set the random seed and create the sobol sequence generator
sobol_engine = scipy.stats.qmc.Sobol( d=4, seed=4 )

# get N initial states spaced in the recommended ranges
X = X_sobol = initial_states = (sobol_engine.random_base2( m=int(np.log2(N)) ) - 0.5 ) * 2 * np.array( [ 10, 20, np.pi, 20 ] )

pad0 = lambda p : np.pad( p, (0,1) )

# generate the changes in state from these states
Y = np.array( [ CartPole.perform_action_RK4( pad0(state) )[:4] - state for state in initial_states ] )


# ## Scatter Plots

# In[48]:


fig, ((ax1, ax2),(ax3,ax4)) = plt.subplots(2, 2, num=19, figsize=(9,9))
fig.subplots_adjust(wspace=0.16, hspace=0.16, top=0.92, bottom=0.08, left=0.08, right=0.96)

titles = ["Cart Position", "Cart Velocity", "Pole Angle", "Pole Angular Velocity"]

colours = matplotlib.cm.get_cmap('cool')( np.linspace(0,1,20) )

extents = np.array([ 3, 5, 5, 11 ])

pad0 = lambda p : np.pad( p, (0,1) )


N = 512 * 2

np.random.seed(4)


for j in range(20):
```

```
    # get N initial states spaced in the recommended ranges
    X = initial_states = ( np.random.rand( N, 4 ) - 0.5 ) * 2 * np.array( [ 10, 10, np.pi, 15 ] )

    # generate the changes in state from these states
    Y = np.array( [ CartPole.perform_action_RK4( pad0(state) )[:4] - state for state in initial_states ] )

    noise_level = (19-j) * 1.05

    Yn = Y + np.random.randn( *Y.shape ) * noise_level * np.mean(np.abs(Y), axis=0)

    Xplus = np.linalg.inv(X.T @ X) @ X.T
    C = Xplus @ Yn

    XC = X @ C

    for i, ax in enumerate( [ax1, ax2, ax3, ax4] ):

        x, y = XC[:,i], Y[:,i]

        ax.scatter( y, x, s=0.2, color=colours[j], zorder=2 )

        if j > 0:
            continue

        extent = extents[i]

        ax.set_xlim(-extent, extent)
        ax.set_ylim(-extent, extent)

        ax.plot( [-extent, extent], [-extent, extent], color="black", linestyle="dotted", zorder=1 )

        ax.set_title( titles[i] )

fig.text(0.5, 0.97, 'Predicted State Changes vs. Target State Changes', ha='center', va='center', fontsize=16)
fig.text(0.52, 0.03, 'Target State Change', ha='center', va='center', fontsize=14)
fig.text(0.03, 0.5, 'Predicted State Change', ha='center', va='center', rotation='vertical', fontsize=14)



# ## Linear Model Accuracy with Varying Observation Noise
#

# In[49]:


fig, ((ax1, ax2),(ax3,ax4)) = plt.subplots(2, 2, num=21, figsize=(9,9))
fig.subplots_adjust(wspace=0.16, hspace=0.16, top=0.92, bottom=0.08, left=0.08, right=0.96)

titles = ["Cart Position", "Cart Velocity", "Pole Angle", "Pole Angular Velocity"]

colours = matplotlib.cm.get_cmap('cool')( np.linspace(0,1,20) )

extents = np.array([ 4, 5, 5, 11 ]) * 1.5

pad0 = lambda p : np.pad( p, (0,1) )

np.random.seed(4)


for j in range(20):

    N = 512 * 8

    # set the random seed and create the sobol sequence generator
    sobol_engine = scipy.stats.qmc.Sobol( d=5, seed=j )

    # get N initial states spaced in the recommended ranges
    X = X_sobol = initial_states = (sobol_engine.random_base2( m=int(np.log2(N)) ) - 0.5 ) * 2 * np.array( [ 10, 20, np.pi, 20, 15 ] )

    # generate the changes in state from these states
    Y = np.array( [ CartPole.perform_action_RK4( state ) - state for state in initial_states ] )


    noise_level = (19-j) * 0.25

    Yn = Y + np.random.randn( *Y.shape ) * noise_level * np.mean(np.abs(Y), axis=0)


    M = 32 * 32
    kernel_centres = X[:M]


    sigma = np.std( X, axis=0 )

    def K( X, Xp, sigma = sigma ):

        # get squared differences and substitute angle one for periodic version
        d2 = ( (X - Xp) / sigma ) ** 2
        d2[:,0] = 0
        d2[:,2] = (np.sin( 0.5 * ( X[:,2] - Xp[:,2] ) ) / sigma[2] ) ** 2

        # divide rows by 2 sigma and return exponential of negative sum along rows
        return np.exp( - 0.5 * np.sum( d2, axis=1 ) )


    # loop over the kernel centres and evaluate the K function across all the Xs at each
    Kmn = np.zeros( (M,N) )
    for i, kernel_centre in enumerate( kernel_centres ):

        Kmn[i] = K( X, kernel_centre[np.newaxis] )

    # same as above but only use first M vectors from X
    Kmm = np.zeros( (M,M) )
    for i, kernel_centre in enumerate( kernel_centres ):

        Kmm[i] = K( X[:M], kernel_centre[np.newaxis] )


    l = 1e-4

    KmnKnm = Kmn @ Kmn.T
    a = KmnKnm + l * Kmm
    b = Kmn @ Yn
```

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


    predictions = Kmn.T @ alpha_m


    for i, ax in enumerate( [ax1, ax2, ax3, ax4] ):

        x, y = predictions[:1024,i], Y[:1024,i]

        ax.scatter( y, x, s=0.2, color=colours[j], zorder=2 )

        if j > 0:
            continue

        extent = extents[i]

        ax.set_xlim(-extent, extent)
        ax.set_ylim(-extent, extent)

        ax.plot( [-extent, extent], [-extent, extent], color="black", linestyle="dotted", zorder=1 )

        ax.set_title( titles[i] )


fig.text(0.5, 0.97, 'Predicted State Changes vs. Target State Changes', ha='center', va='center', fontsize=16)
fig.text(0.52, 0.03, 'Target State Change', ha='center', va='center', fontsize=14)
fig.text(0.03, 0.5, 'Predicted State Change', ha='center', va='center', rotation='vertical', fontsize=14)



# ## Effect of Noise on the Nonlinear Model
#

# In[6]:


linear_errors = []
nonlinear_errors = []

for j in range(20):

    N = 512 * 8

    # set the random seed and create the sobol sequence generator
    sobol_engine = scipy.stats.qmc.Sobol( d=5, seed=j )

    # get N initial states spaced in the recommended ranges
    X = X_sobol = initial_states = (sobol_engine.random_base2( m=int(np.log2(N)) ) - 0.5 ) * 2 * np.array( [ 10, 20, np.pi, 20, 15 ] )

    # generate the changes in state from these states
    Y = np.array( [ CartPole.perform_action_RK4( state ) - state for state in initial_states ] )


    noise_level = (19-j)

    Yn = Y + np.random.randn( *Y.shape ) * noise_level * np.mean(np.abs(Y), axis=0)


    M = 32 * 32
    kernel_centres = X[:M]


    # linear fit

    Xl = X[:1024]
    Ynl = Yn[:1024]

    Xplus = np.linalg.inv(Xl.T @ Xl) @ Xl.T
    C = Xplus @ Ynl


    # nonlinear fit

    sigma = np.std( X, axis=0 )

    def K( X, Xp, sigma = sigma ):

        # get squared differences and substitute angle one for periodic version
        d2 = ( (X - Xp) / sigma ) ** 2
        d2[:,0] = 0
        d2[:,2] = (np.sin( 0.5 * ( X[:,2] - Xp[:,2] ) ) / sigma[2] ) ** 2

        # divide rows by 2 sigma and return exponential of negative sum along rows
        return np.exp( - 0.5 * np.sum( d2, axis=1 ) )


    # loop over the kernel centres and evaluate the K function across all the Xs at each
    Kmn = np.zeros( (M,N) )
    for i, kernel_centre in enumerate( kernel_centres ):

        Kmn[i] = K( X, kernel_centre[np.newaxis] )

    # same as above but only use first M vectors from X
    Kmm = np.zeros( (M,M) )
    for i, kernel_centre in enumerate( kernel_centres ):

        Kmm[i] = K( X[:M], kernel_centre[np.newaxis] )


    l = 1e-4

    KmnKnm = Kmn @ Kmn.T
    a = KmnKnm + l * Kmm
    b = Kmn @ Yn

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


    # get the predictions

    linear_predictions = X @ C
    nonlinear_predictions = Kmn.T @ alpha_m
```

```
    # evaluate the models

    linear_error    = np.sum( np.linalg.norm(    linear_predictions - Y, axis=1 ) )
    nonlinear_error = np.sum( np.linalg.norm( nonlinear_predictions - Y, axis=1 ) )

    linear_errors.append( linear_error )
    nonlinear_errors.append( nonlinear_error )


# In[7]:


fig, ax = plt.subplots(1, 1, num=66, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.12, right=0.99)
sf3utility.setup_phase_portrait( ax )

j = np.arange( 20 )
x = noise_level = (19-j)

y1 = np.array( linear_errors )
y2 = np.array( nonlinear_errors )

f1 = scipy.interpolate.interp1d( x, y1, kind='cubic' )
f2 = scipy.interpolate.interp1d( x, y2, kind='cubic' )

xnew = np.linspace(x[0], x[-1], 800)

ax.plot( xnew, f1(xnew), color="magenta", label="Linear Model", linewidth=2 )
ax.plot( xnew, f2(xnew), color="cyan", label="Nonlinear Model", linewidth=2 )


ax.set_title( "Impact of Noise on Linear and Nonlinear Model Predictions" )
ax.set_xlabel( "Noise Multiplier" )
ax.set_ylabel( "Sum of Squared Errors" )
ax.legend(loc="upper left")



# In[50]:


np.random.seed(4)

ls = np.logspace( -4, 3, num=20, base=2 )
noise_levels = np.linspace( 0, 5, num=20 )

errors = np.zeros( (20, 20) )

N = 512 * 2

# set the random seed and create the sobol sequence generator
sobol_engine = scipy.stats.qmc.Sobol( d=5, seed=4 )

# get N initial states spaced in the recommended ranges
X = X_sobol = initial_states = (sobol_engine.random_base2( m=int(np.log2(N)) ) - 0.5 ) * 2 * np.array( [ 10, 20, np.pi, 20, 15 ] )

# generate the changes in state from these states
Y = np.array( [ CartPole.perform_action_RK4( state ) - state for state in initial_states ] )


M = 32 * 16
kernel_centres = X[:M]


# nonlinear fit

sigma = np.std( X, axis=0 )

def K( X, Xp, sigma = sigma ):

    # get squared differences and substitute angle one for periodic version
    d2 = ( (X - Xp) / sigma ) ** 2
    d2[:,0] = 0
    d2[:,2] = (np.sin( 0.5 * ( X[:,2] - Xp[:,2] ) ) / sigma[2] ) ** 2

    # divide rows by 2 sigma and return exponential of negative sum along rows
    return np.exp( - 0.5 * np.sum( d2, axis=1 ) )


# loop over the kernel centres and evaluate the K function across all the Xs at each
Kmn = np.zeros( (M,N) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmn[i] = K( X, kernel_centre[np.newaxis] )

# same as above but only use first M vectors from X
Kmm = np.zeros( (M,M) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmm[i] = K( X[:M], kernel_centre[np.newaxis] )


KmnKnm = Kmn @ Kmn.T

randoms = np.random.randn( *Y.shape )
means = np.mean(np.abs(Y), axis=0)
pertubations = randoms * means


for j, noise_level in enumerate( noise_levels ):

    Yn = Y + pertubations * noise_level
    b = Kmn @ Yn


    for i, l in enumerate(ls):

        a = KmnKnm + l * Kmm

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

        nonlinear_predictions = Kmn.T @ alpha_m

        # evaluate the models
```

```
        nonlinear_error = np.sum( np.linalg.norm( nonlinear_predictions - Y, axis=1 ) )
        errors[i, j] = nonlinear_error


# In[51]:


fig, ax = plt.subplots(1, 1, num=12, figsize=(9,5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.12, right=0.99)
sf3utility.setup_phase_portrait( ax )

colours = matplotlib.cm.get_cmap('cool')( np.linspace(0,1,20) )

ls = np.logspace( -4, 3, num=20, base=2 )
noise_levels = np.linspace( 0, 5, num=20 )

xx = np.broadcast_to( noise_levels, (20, 20) )
yy = errors

interpolant = scipy.interpolate.make_interp_spline( noise_levels, yy, axis=1, bc_type="natural" )

xxint = np.linspace( min(noise_levels), max(noise_levels), num=100 )
yyint = interpolant( xxint )
xxint = np.broadcast_to( xxint, (20, 100) )

segments = np.stack( (xxint, yyint), axis=2 )

linecollection = matplotlib.collections.LineCollection( segments, colors=colours, linewidths=2 )

ax.add_collection( linecollection )

ax.set_title( "Variation of Error with Different Noise Levels and Regularisation Constants" )
ax.set_xlabel( "Noise Multiplier" )
ax.set_ylabel( "Sum of Squared Errors" )

centerx = (np.max(xx) + np.min(xx)) / 2
centery = (np.max(yy) + np.min(yy)) / 2

ax.set_xlim( centerx - (centerx - np.min(xx)) * 1.12, centerx + (np.max(xx) - centerx) * 1.12 )
ax.set_ylim( centery - (centery - np.min(yy)) * 1.12, centery + (np.max(yy) - centery) * 1.12 )


# ## Error vs Noise Level and Regularisation
#

# ## Stability of Linear Policy
#
#

# In[52]:


fig, ax = plt.subplots(1, 1, num=33, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.08, right=0.99)
sf3utility.setup_phase_portrait( ax )

# small oscillations about stable equilibrium

state = np.array( [ 0, 0, 0, 0, 0 ] )
p = np.array( [1.10893787, 2.01832636, 16.99658894, 2.91580321, 0] ) # fit on noise

noise_level = 0.01

states = []
times = []

np.random.seed(4)

for i in range(55):

    states.append( state )
    times.append( i * 0.2 )

    clipped_state = state
    clipped_state[0] = np.clip( state[0], -2, 2 )
    clipped_state = clipped_state + np.random.randn( 5 ) * means * noise_level

    state[4] = p @ clipped_state
    state = CartPole.perform_action_RK4( state )

states = np.array( states )
times = np.array( times )

x = times
y = states

col_lerp = np.linspace(0, 1, 4)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255

labels = ["$x$","$\dot{x}$","$\theta$","$\dot{\theta}$"]

for i in range(4):

    f2 = scipy.interpolate.interp1d(x, y[:,i], kind='cubic')

    xnew = np.linspace(0, times[-1], 800)

    ax.plot( xnew, f2(xnew), color=colours[i], label=labels[i], linewidth=2 )

ax.set_title( "Time Evolution of State Vector under Linear Policy and Observation Noise (Interpolated)" )
ax.set_xlabel( "Elapsed Time" )
ax.set_ylabel( "Components of State Vector" )
ax.legend(loc="upper right")

# ax.set_ylim( -5, 5 )
ax.set_xlim( -0.4, 10.5 )


# ## Linear Policy with Noise
#

# In[ ]:
```

```
def noisy_rollout_loss( initial_state, p ):

    state = initial_state
    sim_seconds = 8
    sim_steps = int( sim_seconds / 0.2 )
    loss = 0
    noise_level = 0.01

    for i in range( sim_steps ):

        if np.any( np.abs(state) > np.array([10,40,50,1e+3,1e+3]) ):
            return sim_steps


        noisy_state = state + np.random.randn( 5 ) * means * noise_level

        state[4] = p @ noisy_state
        loss += CartPole.loss( state[:4] )
        state = CartPole.perform_action_RK4( state )

    return loss


# In[ ]:


%%time

# initial_state = np.array( [ 0.2, 0, 0.2, 0, 0 ] )
initial_state = np.array( [ 0.0, 0, 0, 0, 0 ] ) # larger initial state for better generality
#initial_state = np.array( [1,0,0.5,0,0] )

pad0 = lambda p : np.pad( p, (0,1) )
model_rollout_loss_from_initial = lambda p : noisy_rollout_loss( initial_state, pad0(p) )

min_loss = 999999
best_p = None

for i in range(100):

    p0     = np.array( [0.75971638,  1.44284499, 17.05744679,  2.66183469] ) + np.random.rand( 4 ) * 3 - 1.5
    result = scipy.optimize.minimize( model_rollout_loss_from_initial, p0, method="Nelder-Mead" )

    end_loss = result["fun"]

    if end_loss < min_loss:
        best_p = result["x"]
        min_loss = end_loss

    print(best_p, min_loss)

p = pad0(best_p)


# In[ ]:


cart_pole = CartPole.CartPole( visual=True, save_frames=False, fig_num=58 )

state = np.array( [ 0, 0, 0, 0, 0 ] )
#p = np.array( [0.76187176,  1.19122763, 16.3309897 ,  2.49340076, 0] ) # optimised from theta=0.1
p = np.array( [0.75971638,  1.44284499, 17.05744679,  2.66183469, 0] ) # optimised from theta=0.5
#p = np.array( [ -0.33143221, -0.42699904, -0.09783966,  0.10055861, 0 ] )
#p = np.array( [2.13646362,  2.594651,   17.62534208,  3.10145759, 0] ) # optimised on model
#p = np.array( [2.29932985, 3.80458197, 22.57529485, 3.9609772, 0] ) # better? optimised on model
#p = np.array( [ 2.42158373, 3.81812947, 22.41281719, 3.95562641, 0 ] ) # optimised on model with 8 second horizon
#p = np.array( [11.91259675, 8.02276575, 29.81324002, 5.60813997, 0] ) # different initial state (bad)
#p = np.array( [1.10893787, 2.01832636, 16.99658894, 2.91580321, 0] ) # fit on noise
#p = np.array([ 0.088,-0.274,-31.37,-4.66,0 ])

cart_pole.set_state( state )

cart_pole._render()


# In[ ]:


for i in range(50):

    noise_level = 0.02

    clipped_state = state
    clipped_state[0] = np.clip( state[0], -2, 2 )
    clipped_state = clipped_state + np.random.randn( 5 ) * means * noise_level

    state[4] = p @ clipped_state
    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()


# In[ ]:


estimated_state = state
smoothing = 0.1

for i in range(50):

    noise_level = 0.02

    estimated_state = estimated_state * smoothing + (state + np.random.randn( 5 ) * means * noise_level) * (1-smoothing)
    state[4] = p @ estimated_state
    state = CartPole.perform_action_RK4( state )

    cart_pole.set_state( state[:4] )
    cart_pole._render()


# In[12]:


p = np.array( [1.10893787, 2.01832636, 16.99658894, 2.91580321, 0] ) # fit on noise
```

```
# small oscillations about stable equilibrium

means = np.array([ 2.00951924, 1.964584, 1.98959769, 4.76029465, 0. ])

stable_times = np.ones( 60 ) * 40 * 0.2
noise_levels = np.linspace( 0.01, 0.1, num=60 )

for j, noise_level in enumerate( noise_levels ):

    stable_time_sum = 0

    for _ in range(50):

        state = np.array( [ 0, 0, 0, 0, 0 ] )

        for i in range(400):

            clipped_state = state
            clipped_state[0] = np.clip( state[0], -2, 2 )
            clipped_state = clipped_state + np.random.randn( 5 ) * means * noise_level

            state[4] = p @ clipped_state
            state = CartPole.perform_action_RK4( state )

            stable_time_sum += 0.2

            if abs(state[2]) > 2:
                break

    stable_times[j] = stable_time_sum


# In[38]:


p = np.array( [1.10893787, 2.01832636, 16.99658894, 2.91580321, 0] ) # fit on noise

# small oscillations about stable equilibrium

means = np.array([ 2.00951924, 1.964584, 1.98959769, 4.76029465, 0. ])

stable_times_process = np.ones( 60 ) * 40 * 0.2
noise_levels = np.linspace( 0.01, 0.1, num=60 )

for j, noise_level in enumerate( noise_levels ):

    stable_time_sum = 0

    for _ in range(50):

        state = np.array( [ 0., 0, 0, 0, 0 ] )

        for i in range(400):

            state += np.random.randn( 5 ) * means * noise_level

            clipped_state = state
            clipped_state[0] = np.clip( state[0], -2, 2 )
            clipped_state = clipped_state

            state[4] = p @ clipped_state
            state = CartPole.perform_action_RK4( state )

            stable_time_sum += 0.2

            if abs(state[2]) > 2:
                break

    stable_times_process[j] = stable_time_sum


# In[53]:


fig, ax = plt.subplots(1, 1, num=18, figsize=(9,5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.12, right=0.99)
sf3utility.setup_phase_portrait( ax )


noise_levels = np.linspace( 0.01, 0.1, num=60 )


interpolant = scipy.interpolate.make_interp_spline( noise_levels, stable_times / 50, bc_type="natural" )

xint = np.linspace( min(noise_levels), max(noise_levels), num=300 )
yint = interpolant( xint )

ax.plot( xint, yint, color="magenta", linewidth=2, label="Observation Noise" )
ax.set_xlim( 0.007, 0.103 )
ax.set_ylim( 0, 15 )


interpolant = scipy.interpolate.make_interp_spline( noise_levels, stable_times_process / 50, bc_type="natural" )

xint = np.linspace( min(noise_levels), max(noise_levels), num=300 )
yint = interpolant( xint )


ax.plot( xint, yint, color="cyan", linewidth=2, label="Process Noise" )
ax.set_xlim( 0.007, 0.103 )
ax.set_ylim( 0, 25 )

ax.set_title( "Stablity of Linear Policy Under Observation Noise and Process Noise" )
ax.set_xlabel( "Noise Multiplier" )
ax.set_ylabel( "Time Until Loss of Control" )
ax.legend(loc="upper right")


# In[18]:


noise_levels = np.linspace( 0.01, 0.1, num=60 )

interpolant = scipy.interpolate.make_interp_spline( noise_levels, stable_times_process / 50, bc_type="natural" )
```

```
xint = np.linspace( min(noise_levels), max(noise_levels), num=300 )
yint = interpolant( xint )


ax.plot( xint, yint, color="cyan", linewidth=2 )
ax.set_xlim( 0.007, 0.103 )
ax.set_ylim( 0, 15 )


# ## Time Until Instability
#
#
```

task3.2.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


# import CartPole.py from local directory
import CartPole, sf3utility
import matplotlib.collections
import matplotlib.pyplot as plt
import numpy as np
import scipy.interpolate, scipy.stats.qmc, scipy.optimize
import random, copy

plt.rcParams["font.family"] = "Georgia"
#plt.rcParams['figure.figsize'] = [9.0, 7.0]
#plt.rcParams['figure.dpi'] = 400

# store results for later
cache = {}


# In[2]:


get_ipython().run_line_magic('matplotlib', 'notebook')


# In[3]:


def noisy_rollout_loss( initial_state, p ):

    state = initial_state
    sim_seconds = 8
    sim_steps = int( sim_seconds / 0.2 )
    loss = 0
    noise_level = 0.01

    for i in range( sim_steps ):

        if np.any( np.abs(state) > np.array([10,40,50,1e+3,1e+3]) ):
            return sim_steps


        state = state + np.random.randn( 5 ) * means * noise_level

        state[4] = p @ state
        loss += CartPole.loss( state[:4] )
        state = CartPole.perform_action_RK4( state )

    return loss


# In[8]:


get_ipython().run_cell_magic('time', '', '\ninitial_state = np.array( [ 0.2, 0, 0.2, 0, 0 ] )\n#initial_state = np.array( [1,0,0.5,0,0] )\n\npad0 = lambda p : np.pad( p, (0,1) )\nrollout_loss_from_initial = lambda p : noisy_rollout_loss( initial_state, pad0(p) )\n\nmeans = np.array([ 2.00903004, 1.9578942, 1.99085053, 4.74956371, 0. ])\n\n\nmin_loss = 999999\nbest_p = None\n\nfor i in range(1000):\n    p0      = np.random.rand( (4) ) * 10 - 5\n    result = scipy.optimize.minimize( rollout_loss_from_initial, p0, method="Nelder-Mead" )\n    \n    end_loss = rollout_loss_from_initial( result["x"] )\n    \n    if end_loss < min_loss:\n        best_p = result["x"]\n        min_loss = end_loss\n        \n    print(best_p, min_loss)\n\np = best_p\n')


# # Task 3.2 - Process Noise
#

# ## Local Minimum
#

# ## Control in Noise
#

# In[195]:


cart_pole = CartPole.CartPole( visual=True, save_frames=False, fig_num=58 )

state = np.array( [ 0, 0, 0, 0, 0 ] )
#p = np.array( [0.76187176,  1.19122763, 16.3309897 ,  2.49340076, 0] ) # optimised from theta=0.1
#p = np.array( [0.75971638,  1.44284499, 17.05744679,  2.66183469, 0] ) # optimised from theta=0.5
#p = np.array( [ -0.33143221, -0.42699904, -0.09783966,  0.10055861, 0 ] )
#p = np.array( [2.13646362,  2.594651,   17.62534208,  3.10145759, 0] ) # optimised on model
#p = np.array( [2.29932985, 3.80458197, 22.57529485, 3.9609772, 0] ) # better? optimised on model
#p = np.array( [ 2.42158373, 3.81812947, 22.41281719, 3.95562641, 0 ] ) # optimised on model with 8 second horizon
#p = np.array( [11.91259675, 8.02276575, 29.81324002, 5.60813997, 0] ) # different initial state (bad)
#p = np.array( [1.10893787, 2.01832636, 16.99658894, 2.91580321, 0] ) # fit on noise
#p = np.array([ 0.088,-0.274,-31.37,-4.66,0 ])
#p = np.array([3.91092163, -3.24638638, 4.9457018, -4.80275196, 0])
#p = np.array([-2.38362354, -4.94276337, -3.48514146, -3.39752314, 0])
#p = np.array([0.52867083, 1.34935861, 15.04318042, 2.4385310, 0])
#p = np.array([ 0.52867083, 1.34935861, 15.04318042, 2.43853109, 0])
#p = - np.array( [-0.20792587, -0.4554855, -12.8760437,  -1.92181886, 0] ) # dlqr!!
#p = - np.array( [-0.42004789,  -0.754576  , -13.5778077 ,  -2.08834533, 0] ) # dlqr!! small R
p = - np.array( [ -0.02581708,  -0.1272267 , -11.95653773,  -1.73874922, 0] ) # dlqr!! big R

cart_pole.set_state( state )
```

```
cart_pole._render()
```

```
# In[199]:
```

```
for i in range(50):

    noise_level = 0.02

    clipped_state = state
    clipped_state[0] = np.clip( state[0], -2, 2 )
    clipped_state = clipped_state# + np.random.randn( 5 ) * means * noise_level

    state = state + np.random.randn( 5 ) * means * noise_level
    state[4] = p @ clipped_state
    state = CartPole.perform_action_RK4( state )# + np.random.randn( 5 ) * means * noise_level
    cart_pole.set_state( state[:4] )
    cart_pole._render()
```

```
# In[202]:
```

```
fig, ax = plt.subplots(1, 1, num=33, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.08, right=0.99)
sf3utility.setup_phase_portrait( ax )

# small oscillations about stable equilibrium

state = np.array( [ 0, 0, 0, 0, 0 ] )
p = np.array([3.91092163, -3.24638638, 4.9457018, -4.80275196, 0])

noise_level = 0.05

states = []
times = []

np.random.seed(4)

for i in range(55):

    states.append( state )
    times.append( i * 0.2 )

    clipped_state = state
    clipped_state[0] = np.clip( state[0], -2, 2 )
    clipped_state = clipped_state

    state[4] = p @ clipped_state
    state = CartPole.perform_action_RK4( state ) + np.random.randn( 5 ) * means * noise_level

states = np.array( states )
times = np.array( times )

x = times
y = states

col_lerp = np.linspace(0, 1, 4)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255

labels = ["$x$","$\dot{x}$","$\theta$","$\dot{\theta}$"]

for i in range(4):

    f2 = scipy.interpolate.interp1d(x, y[:,i], kind='cubic')

    xnew = np.linspace(0, times[-1], 800)

    ax.plot( xnew, f2(xnew), color=colours[i], label=labels[i], linewidth=2 )

ax.set_title( "Time Evolution of State Vector under Linear Policy and Process Noise (Interpolated)" )
ax.set_xlabel( "Elapsed Time" )
ax.set_ylabel( "Components of State Vector" )
ax.legend(loc="upper right")

# ax.set_ylim( -5, 5 )
ax.set_xlim( -0.4, 10.5 )
```

```
#
#
```

```
# In[203]:
```

```
np.random.seed(3)

fig, ax = plt.subplots(1, 1, num=35, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.08, right=0.99)
sf3utility.setup_phase_portrait( ax )

p = np.array([0.75971638,  1.44284499, 17.05744679,  2.66183469, 0])

obs_state = np.array([0., 0, 0, 0, 0])
proc_state = np.array([0., 0, 0, 0, 0])

times = []
obs_states = []
proc_states = []

noise_level = 0.01

for i in range(50):

    obs_state[4] = p @ (obs_state + np.random.randn( 5 ) * means * noise_level)
    obs_state = CartPole.perform_action_RK4( obs_state ) # + np.random.randn( 5 ) * means * noise_level

    proc_state[4] = p @ proc_state
    proc_state = proc_state + np.random.randn( 5 ) * means * noise_level
    proc_state = CartPole.perform_action_RK4( proc_state )

    times.append( i * 0.2 )
    obs_states.append( obs_state )
    proc_states.append( proc_state )
```

```
times = np.array( times )
obs_states = np.array( obs_states )
proc_states = np.array( proc_states )

x = times
y = proc_states

col_lerp = np.linspace(0, 1, 4)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255

labels = ["$x$","$\dot{x}$","$\theta$","$\dot{\theta}$"]

for i in range(4):

    f2 = scipy.interpolate.interp1d(x, y[:,i], kind='cubic')

    xnew = np.linspace(0, times[-1], 800)

    ax.plot( xnew, f2(xnew), color=colours[i], label=labels[i], linewidth=2 )

ax.set_title( "Time Evolution of State Vector under Linear Policy and Process Noise (Interpolated)" )
ax.set_xlabel( "Elapsed Time" )
ax.set_ylabel( "Components of State Vector" )
ax.legend(loc="upper right")



# In[60]:


def noisy_rollout_loss( initial_state, p ):

    state = initial_state
    sim_seconds = 8
    sim_steps = int( sim_seconds / 0.2 )
    loss = 0
    noise_level = 0.001

    for i in range( sim_steps ):

        if np.any( np.abs(state) > np.array([10,40,50,1e+3,1e+3]) ):
            return sim_steps


        state = state + np.random.randn( 5 ) * means * noise_level

        state[4] = p @ state
        loss += CartPole.loss( state[:4] )
        state = CartPole.perform_action_RK4( state )

    return loss


# In[61]:


get_ipython().run_cell_magic('time', '', '\ninitial_state = np.array( [ 0.0, 0, 0.0, 0, 0 ] )\n#initial_state = np.array( [1,0,0.5,0,0] )\n\npad0 = lambda p : np.pad( p, (0,1) )\nrollout_loss_from_initial = lambda p : noisy_rollout_loss( initial_state, pad0(p) )\n\nmeans = np.array([ 2.00903004, 1.9578942, 1.99085053, 4.74956371, 0. ])\n\n\nmin_loss = 999999\nbest_p = None\n\nfor i in range(1000):\n\n    p0      = np.array( [0.75971638, 1.44284499, 17.05744679,  2.66183469] ) + np.random.rand( (4) ) - 0.5\n    result = scipy.optimize.minimize( rollout_loss_from_initial, p0, method="Nelder-Mead" )\n    \n    end_loss = rollout_loss_from_initial( result["x"] )\n    \n    if end_loss < min_loss:\n        best_p = result["x"]\n        min_loss = end_loss\n        \n    print(best_p, min_loss)\n\np = best_p\n')
```

statespace.py

```
#!/usr/bin/env python
# coding: utf-8

# In[240]:


# import CartPole.py from local directory
import CartPole, sf3utility
import matplotlib.collections
import matplotlib.pyplot as plt
import numpy as np
import scipy.interpolate, scipy.stats.qmc, scipy.optimize
import random, copy, control
import filterpy.kalman, filterpy.common

plt.rcParams["font.family"] = "Georgia"
#plt.rcParams['figure.figsize'] = [9.0, 7.0]
#plt.rcParams['figure.dpi'] = 400

# store results for later
cache = {}

get_ipython().run_line_magic('matplotlib', 'notebook')


# $$
# x_{k+1} = Ax_k + Bu_k + Ke_k
# $$
# $$
# y_k = x_k
# $$

# In[ ]:


xk1 = A @ xk + B @ uk + K @ ek
xk = yk


# In[170]:


N = 512 * 2

# set the random seed and create the sobol sequence generator
np.random.seed(4)
sobol_engine4d = scipy.stats.qmc.Sobol( d=4, seed=4 )
```

```
sobol_engine1d = scipy.stats.qmc.Sobol( d=1, seed=4 )

# get N initial states spaced in the recommended ranges and also N initial forces
X = (sobol_engine4d.random_base2( m=int(np.log2(N)) ).T - 0.5 ) * 2. * np.array( [[ 0.25, 0.25, 0.25, 0.25 ]] ).T# + np.array([0,0,np.pi,0])
U = (sobol_engine1d.random_base2( m=int(np.log2(N)) ).T - 0.5 ) * 2. * 0.25 #np.array( [ 0, 0, 0, 0.25 ] )


# generate the changes in state from these states
Xn_x = np.array( [ CartPole.perform_action_RK4( pad0( state) )[:4] for state in X.T ] ).T
Xn_u = np.array( [ CartPole.perform_action_RK4( fill0(force) )[:4] for force in U.T ] ).T


# In[ ]:


A @ X = Xn_x
(A @ X).T = X.T @ A.T = Xn_x.T

B @ U = Xn_u
(B @ U).T = U.T @ B.T = Xn_u.T


# In[201]:


AT, Aresiduals, rank, s = np.linalg.lstsq( X.T, Xn_x.T, rcond=None )
BT, Bresiduals, rank, s = np.linalg.lstsq( U.T, Xn_u.T, rcond=None )

A = AT.T
B = BT.T

Bpadded = np.zeros((4,4))
Bpadded[:,3] = B[:,0]


# In[202]:


np.around(A, 6), np.around(B, 6), np.around(Bpadded, 6)


# # State Space Representation of the System and Closed-Form Control Solutions
#
#
# $$
# x_{k+1} = Ax_k + Bu_k + Ke_k
# $$
#
#
#
# $$
# X_{k+1} = AX_k
# $$
#

# In[228]:


control.dlqr(A, Bpadded, np.eye(4) * np.array([0.1,1,5,1]), np.eye(4) * 10)


# ## Computation of Policy
#
#
#

# In[341]:


fig, ax = plt.subplots(1, 1, num=66, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.08, right=0.99)
sf3utility.setup_phase_portrait( ax )

# small oscillations about stable equilibrium

state = np.array( [ 8, 0, -0.6, 0, 0 ] )
#p = - np.array( [ -0.20792587, -0.4554855, -12.8760437,  -1.92181886, 0 ] ) # dlqr!!
p = - np.array( [ -0.02581708,  -0.1272267 , -11.95653773,  -1.73874922, 0] ) # dlqr!! big R

states = []
time = []

for i in range(55):

    states.append( state )
    time.append( i * 0.2 )

    clipped_state = state
    #clipped_state[0] = np.clip( state[0], -2, 2 )

    state[4] = p @ clipped_state
    state = CartPole.perform_action_RK4( state )

states = np.array( states )
time = np.array( time )

x = time
y = states

col_lerp = np.linspace(0, 1, 4)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255

labels = ["$x$","$\dot{x}$","$\theta$","$\dot{\theta}$"]

for i in range(4):

    f2 = scipy.interpolate.interp1d(x, y[:,i], kind='cubic')

    xnew = np.linspace(0, time[-1], 800)

    ax.plot( xnew, f2(xnew), color=colours[i], label=labels[i], linewidth=2 )

ax.set_title( "Time Evolution of State Vector under DLQR Policy (Interpolated)" )
ax.set_xlabel( "Elapsed Time" )
ax.set_ylabel( "Components of State Vector" )
```

```
ax.legend(loc="upper right")

# ax.set_ylim( -5, 5 )
ax.set_xlim( -0.4, 10 )



# ## Introduction of Noise
#
#

# In[339]:


cart_pole = CartPole.CartPole( visual=True, save_frames=False, fig_num=58 )

state = np.array( [[ 0, 0, 0., 0 ]] ).T
#p = - np.array( [[-0.20792587, -0.4554855, -12.8760437,  -1.92181886]] ) # dlqr!!
#p = - np.array( [[-0.42004789,  -0.754576  , -13.5778077 ,  -2.08834533]] ) # dlqr!! small R
p = - np.array( [[ -0.02581708,  -0.1272267 , -11.95653773,  -1.73874922]] ) # dlqr!! big R

force = 0

cart_pole.set_state( state )
cart_pole._render()


# In[340]:


means = np.array([[ 2.00951924, 1.964584, 1.98959769, 4.76029465]]).T

kalman_filter = filterpy.kalman.KalmanFilter( dim_x=4, dim_z=4 )
kalman_filter.x = state # initial state
kalman_filter.F = A # state transition matrix
kalman_filter.H = np.eye(4) # measurement function
kalman_filter.P = np.eye(4) * 0 # uncertainty
kalman_filter.R = 0.02 # measurement noise
kalman_filter.Q = 0 # filterpy.common.Q_discrete_white_noise(dim=4, dt=0.2, var=0.02**2) # process noise


# In[313]:


for i in range(50):

    noisy_state = state + np.random.randn(4, 1) * 0.02

    force = p @ noisy_state
    state = A @ state + B * force
    cart_pole.set_state( state )
    cart_pole._render()


# In[337]:


means = np.array([ 2.00951924, 1.964584, 1.98959769, 4.76029465, 0 ])

np.random.seed(3)

fig, ax = plt.subplots(1, 1, num=35, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.08, right=0.99)
sf3utility.setup_phase_portrait( ax )

p = np.array([0.75971638,  1.44284499, 17.05744679,  2.66183469, 0])
p_dlqr = - np.array( [ -0.02581708,  -0.1272267 , -11.95653773,  -1.73874922, 0] ) # dlqr!! big R

state = np.array([0., 0, 0, 0, 0])
dlqr_state = np.array([0., 0, 0, 0, 0])

times = []
states = []
dlqr_states = []

noise_level = 0.07

for i in range(50):

    state = state + np.random.randn( 5 ) * means * noise_level
    dlqr_state = dlqr_state + np.random.randn( 5 ) * means * noise_level

    state[4] = p @ state
    dlqr_state[4] = p_dlqr @ dlqr_state

    state = CartPole.perform_action_RK4( state )
    dlqr_state = CartPole.perform_action_RK4( dlqr_state )


    times.append( i * 0.2 )
    states.append( state )
    dlqr_states.append( dlqr_state )

times = np.array( times )
states = np.array( states )
dlqr_states = np.array( dlqr_states )

x = times
y = dlqr_states

col_lerp = np.linspace(0, 1, 4)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255

labels = ["$x$","$\dot{x}$","$\theta$","$\dot{\theta}$"]

for i in range(4):

    f2 = scipy.interpolate.interp1d(x, y[:,i], kind='cubic')

    xnew = np.linspace(0, times[-1], 800)

    ax.plot( xnew, f2(xnew), color=colours[i], label=labels[i], linewidth=2 )

ax.set_title( "Time Evolution of State Vector under Linear Policy and Process Noise (Interpolated)" )
ax.set_xlabel( "Elapsed Time" )
```

```
ax.set_ylabel( "Components of State Vector" )
ax.legend(loc="upper right")
```

task4.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


# import CartPole.py from local directory
import CartPole, sf3utility
import matplotlib.collections
import matplotlib.pyplot as plt
import numpy as np
import scipy.interpolate, scipy.stats.qmc, scipy.optimize
import random, copy, time

plt.rcParams["font.family"] = "Georgia"
#plt.rcParams['figure.figsize'] = [9.0, 7.0]
#plt.rcParams['figure.dpi'] = 400

# store results for later
cache = {}

get_ipython().run_line_magic('matplotlib', 'notebook')


# In[2]:


cart_pole = CartPole.CartPole( visual=True, save_frames=False, fig_num=58 )

state = np.array( [ 1., 0, 0, 0, 0 ] )
cart_pole.set_state( state )
cart_pole._render()


# In[116]:


p = np.array([-1, -1.5, 0, -1, 0]) # steady at middle

for i in range(50):

    state[4] = p @ state
    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()

forces = [5, 5, -5, -5, -5, -5, 4, 4, 4, 4, -2, -2]

for i, force in enumerate(forces):

    state[4] = force
    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()

p = np.array( [0.75971638,  1.44284499, 17.05744679,  2.66183469, 0] ) # upright at middle

for i in range(50):

    state[4] = p @ state
    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()


# # Task 4 - Nonlinear Control
#
# 1. Use a linear policy to bring the cartpole to the stable equilibrium, at the origin.
# 2. Apply a certain sequence of force inputs to swing up the pole.
# 3. Use another linear policy to keep the pole upright.
#
#

# In[18]:


fig, ax = plt.subplots(1, 1, num=33, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.08, right=0.99)
sf3utility.setup_phase_portrait( ax )

state = np.array( [ 1., 5, 0, 2, 0 ] ) # initial state
p = np.array([-1, -1.5, 0, -1, 0]) # steady at middle

states = []
times = []
t = 0

for i in range(32):

    states.append(state)
    times.append(t)

    state[4] = p @ state
    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()
    t += 0.2

forces = [5, 5, -5, -5, -5, -5, 4, 4, 4, 4, -2, -2]

for i, force in enumerate(forces):

    states.append(state)
    times.append(t)

    state[4] = force
    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()
```

```
    t += 0.2

p = np.array( [0.75971638, 1.44284499, 17.05744679, 2.66183469, 0] ) # upright at middle

for i in range(32):

    states.append(state)
    times.append(t)

    state[4] = p @ state
    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()
    t += 0.2

states = np.array( states )
times = np.array( times )

x = times
y = states

col_lerp = np.linspace(0, 1, 4)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255

labels = ["$x$","$\dot{x}$","$\theta$","$\dot{\theta}$"]

for i in range(4):

    f2 = scipy.interpolate.interp1d(x, y[:,i], kind='cubic')

    xnew = np.linspace(0, times[-1], 800)

    ax.plot( xnew, f2(xnew), color=colours[i], label=labels[i], linewidth=2 )

ax.set_title( "Time Evolution of State Vector under Manually Designed Policy (Interpolated)" )
ax.set_xlabel( "Elapsed Time" )
ax.set_ylabel( "Components of State Vector" )
ax.legend(loc="upper right")



# ## Radial Basis Function Approach
#
#
# The policy is now a sum over a few kernel functions, given by
#
# $$
# p(X) = \sum_i w_i e^{-0.5 (X-X_i)^T W (X-X_i)}
# $$
#
# I have decided to start with W as the identity matrix to reduce the number of parameters to worry about.

# ## Quadratic Controller???

# In[105]:


cart_pole_exp = CartPole.CartPole( visual=True, save_frames=False, fig_num=59 )


# In[108]:


p = np.array(
 [ 0.79720335, -0.81892826, 0.16672051 , -0.09236256 ])
P = np.array(
[[-0.34134063, -0.44538288, -0.54157946,  0.5954045  ],
 [ 0.48675709, -1.13294381, -0.951934   , -0.56636443 ],
 [ 1.37535627, -1.66183934, -0.8258103 , -1.3074058  ],
 [ 2.1116826 ,  1.3782299 , -1.72001146,  0.1383763  ]])


# In[114]:


state = np.array( [ 0, 0, np.pi, 0, 0 ] )
cart_pole_exp.set_state( state )
cart_pole_exp._render()

means = np.array([ 2.00951924, 1.964584, 1.98959769, 4.76029465, 0. ])
noise_level = 0.00

for i in range(12):

    # quadratic controller!!
    state[4] = p @ state[:4] + state[:4] @ P @ state[:4]

    state = CartPole.perform_action_RK4( state )
    cart_pole_exp.set_state( state[:4] )
    cart_pole_exp._render()

    state += np.random.randn( 5 ) * means * noise_level

linear_p = np.array( [0.75971638, 1.44284499, 17.05744679, 2.66183469] )

for i in range(50):

    state[4] = linear_p @ state[:4]

    state = CartPole.perform_action_RK4( state )
    cart_pole_exp.set_state( state[:4] )
    cart_pole_exp._render()


# In[8]:


p = np.array(
 [-0.39141334, -0.35567795,  0.44813621, -0.46802521]
)
P = np.array(
[[-0.05847553, -1.11992684, -0.44995409, -1.25715311],
 [-0.41638631,  0.10466255,  1.10751617,  1.18754805],
 [ 0.69309034, -0.51847106, -0.04314128, -0.8760343 ],
 [-0.74069395,  1.78405652, -0.94659632, -0.45347616]]
```

```
)

state = np.array( [ 0, 0, np.pi, 0, 0 ] )
cart_pole_exp.set_state( state )
cart_pole_exp._render()

for i in range(50):

    # quadratic controller!!
    state[4] = p @ state[:4] + state[:4] @ P @ state[:4]

    state = CartPole.perform_action_RK4( state )
    cart_pole_exp.set_state( state[:4] )
    cart_pole_exp._render()


# In[24]:


fig, ax = plt.subplots(1, 1, num=3, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.08, right=0.99)
sf3utility.setup_phase_portrait( ax )

state = np.array( [ 0, 0, np.pi, 0, 0 ] )

states = []
times = []
t = 0

for i in range(12):

    states.append(state)
    times.append(t)
    t += 0.2

    state[4] = p @ state[:4] + state[:4] @ P @ state[:4]
    state = CartPole.perform_action_RK4( state )

linear_p = np.array( [0.75971638, 1.44284499, 17.05744679, 2.66183469] )

for i in range(15):

    states.append(state)
    times.append(t)
    t += 0.2

    state[4] = linear_p @ state[:4]
    state = CartPole.perform_action_RK4( state )


states = np.array( states )
times = np.array( times )

x = times
y = states

col_lerp = np.linspace(0, 1, 4)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255

labels = ["$x$","$\dot{x}$","$\theta$","$\dot{\theta}$"]

for i in range(4):

    f2 = scipy.interpolate.interp1d(x, y[:,i], kind='cubic')

    xnew = np.linspace(0, times[-1], 800)

    ax.plot( xnew, f2(xnew), color=colours[i], label=labels[i], linewidth=2 )

ax.set_title( "Time Evolution of State Vector under Quadratic Policy (Interpolated)" )
ax.set_xlabel( "Elapsed Time" )
ax.set_ylabel( "Components of State Vector" )
ax.legend(loc="upper right")


# ## Quadratic Policy
#
#
# p =
# [ 0.79720335, -0.81892826, 0.16672051 , -0.09236256 ]
#
# P =
# [[-0.34134063, -0.44538288, -0.54157946,  0.5954045  ]
#  [ 0.48675709, -1.13294381, -0.951934  , -0.56636443 ]
#  [ 1.37535627, -1.66183934, -0.8258103 , -1.3074058  ]
#  [ 2.1116826 ,  1.3782299 , -1.72001146,  0.1383763  ]]
#

# ## Stability of Swing-up Controllers
#

# In[10]:


def quadratic_swingup_loss( p, P ):

    state = np.array( [0, 0, np.pi, 0, 0] )
    sim_steps = 12

    for i in range( sim_steps ):

        state[4] = p @ state[:4] + state[:4] @ P @ state[:4]
        state = CartPole.perform_action_RK4( state )

    return CartPole.loss( state[:4] )


# In[11]:


def quadratic_swingup_loss( p, P ):

    state = np.array( [0, 0, np.pi, 0, 0] )
    loss = 0
```

```
    for i in range( 10 ):

        state[4] = p @ state[:4] + state[:4] @ P @ state[:4]
        state = CartPole.perform_action_RK4( state )

    for i in range( 10 ):

        state[4] = p @ state[:4] + state[:4] @ P @ state[:4]
        state = CartPole.perform_action_RK4( state )
        loss += CartPole.loss( state[:4] )

    return loss

""" best so far

p =
 [-0.39141334 -0.35567795  0.44813621 -0.46802521]
P =
 [[-0.05847553 -1.11992684 -0.44995409 -1.25715311]
 [-0.41638631  0.10466255  1.10751617  1.18754805]
 [ 0.69309034 -0.51847106 -0.04314128 -0.8760343 ]
 [-0.74069395  1.78405652 -0.94659632 -0.45347616]]
loss = 5.685331822985375
loss = 9.503086460035675 """


# In[12]:


def linear_swingup_loss( p ):

    state = np.array( [0, 0, np.pi, 0, 0] )
    sim_steps = 12

    for i in range( sim_steps ):

        state[4] = p @ state[:4]
        state = CartPole.perform_action_RK4( state )

    return CartPole.loss( state[:4] )


# In[13]:


# best_p = None
# best_P = None
# best_loss = 99999

# for i in np.arange(1e+9):

#     p = np.random.randn( 4 )
#     P = np.random.randn( 4, 4 )

#     loss = quadratic_swingup_loss( p, P )

#     if loss < best_loss:

#         best_p = p
#         best_P = P
#         best_loss = loss

#         print("p = \n", p)
#         print("P = \n", P)
#         print("loss =", loss, "\n")


# In[14]:


# best_p = None
# best_loss = 1

# for i in np.arange(1e+9):

#     p = np.random.randn( 4 )

#     loss = linear_swingup_loss( p )

#     if loss < best_loss:

#         best_p = p
#         best_loss = loss

#         print("p = \n", p)
#         print("loss =", loss, "\n")


# In[15]:


cart_pole_exp = CartPole.CartPole( visual=True, save_frames=False, fig_num=60 )


# In[16]:


state = np.array( [ 0.5, 0.2, np.pi, 0.01, 0 ] )
cart_pole_exp.set_state( state )
cart_pole_exp._render()

m = cart_pole.pole_mass
l = cart_pole.pole_length
I = 1/3 * m * l**2

target_V = 1/2 * m * 9.8 * l

V = lambda state : 1/2 * I * state[3] ** 2 + 1/2 * m * 9.8 * l * np.cos(state[2])
remap = lambda theta : theta % (2*np.pi) - np.pi

for i in range(64):

    if V(state) < 1.2 and abs(state[2]) > 2:
        state[4] = -3.9 * remap(state[2])
    else:
```

```
        state[4] = 0

    state = CartPole.perform_action_RK4( state )
    cart_pole_exp.set_state( state[:4] )
    cart_pole_exp._render()

linear_p = np.array( [0.75971638, 1.44284499, 17.05744679, 2.66183469] )

for i in range(20):

    clipped_state = state[:]
    clipped_state[2] = CartPole.remap_angle(clipped_state[2])
    clipped_state[0] = np.clip(clipped_state[0], -0.5, 0.5)

    state[4] = linear_p @ state[:4]

    state = CartPole.perform_action_RK4( state )
    cart_pole_exp.set_state( state[:4] )
    cart_pole_exp._render()


# In[92]:


fig, ax = plt.subplots(1, 1, num=34, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.1, right=0.99)
sf3utility.setup_phase_portrait( ax )

states = []
times = []
t = 0

state = np.array( [ -4.5, 0.2, np.pi, 0.01, 0 ] )

m = cart_pole.pole_mass
l = cart_pole.pole_length
I = 1/3 * m * l**2

target_V = 1/2 * m * 9.8 * l

V = lambda state : 1/2 * I * state[3] ** 2 + 1/2 * m * 9.8 * l * np.cos(state[2])
remap = lambda theta : theta % (2*np.pi) - np.pi

for i in range(64):

    states.append(state)
    times.append(t)
    t += 0.2

    if V(state) < 1.2 and abs(state[2]) > 2:
        state[4] = -3.9 * remap(state[2])
    else:
        state[4] = 0

    state = CartPole.perform_action_RK4( state )

linear_p = np.array( [0.75971638, 1.44284499, 17.05744679, 2.66183469] )

for i in range(20):

    states.append(state)
    times.append(t)
    t += 0.2

    clipped_state = state[:]
    clipped_state[2] = CartPole.remap_angle(clipped_state[2])
    clipped_state[0] = np.clip(clipped_state[0], -0.5, 0.5)

    state[4] = linear_p @ state[:4]

    state = CartPole.perform_action_RK4( state )

states = np.array( states )
times = np.array( times )

x = times
y = states

col_lerp = np.linspace(0, 1, 4)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255

labels = ["$x$","$\dot{x}$","$\theta$","$\dot{\theta}$"]

for i in range(4):

    f2 = scipy.interpolate.interp1d(x, y[:,i], kind='cubic')

    xnew = np.linspace(0, times[-1], 800)

    ax.plot( xnew, f2(xnew), color=colours[i], label=labels[i], linewidth=2 )

ax.set_title( "Time Evolution of State Vector under Energy Policy (Interpolated)" )
ax.set_xlabel( "Elapsed Time" )
ax.set_ylabel( "Components of State Vector" )
ax.legend(loc="upper right")



# ## Energy-Based Policy
#

# ## Extended State Vector Control

# In[28]:


cart_pole_ext = CartPole.CartPole( visual=True, save_frames=False, fig_num=61 )


# In[128]:


state = np.array( [ 0, 0, np.pi, 0, 0 ] )
cart_pole_ext.set_state( state )
cart_pole_ext._render()
```

```
p = np.array( [-1.43277565, -5.09585954, -0.49529056,  0.67997629, -3.40671644, -1.29766318, 0.61857502, -0.18109689, -1.67954534, -1.82307116,
0.00907654] )
means = np.array([ 2.00951924, 1.964584, 1.98959769, 4.76029465, 0. ])
noise_level = 0.

for i in range(50):

    extended_state = np.pad( state[:4], (0, 7) )

    extended_state[0]  = np.clip(extended_state[0], -0.5, 0.5)
    extended_state[4]  = np.sin(state[2])
    extended_state[5]  = np.cos(state[2])
    extended_state[6]  = extended_state[5] * state[1]
    extended_state[7]  = extended_state[5] * state[3]
    extended_state[8]  = extended_state[4] * state[1]
    extended_state[9]  = extended_state[4] * state[3]
    extended_state[10] = state[2] ** 3

    state[4] = p @ extended_state

    state = CartPole.perform_action_RK4( state )
    cart_pole_ext.set_state( state[:4] )
    cart_pole_ext._render()

    state += np.random.randn( 5 ) * means * noise_level


# In[129]:


unbounded_loss = lambda state : np.linalg.norm( state[:4] * np.array([0.4, 0.2, 1, 0.2]) )


# In[130]:


def extended_swingup_loss( p ):

    state = np.array( [0, 0, np.pi, 0, 0] )
    sim_steps = 50
    loss = 0

    for i in range( sim_steps ):

        if np.any( np.abs(state) > 200 ):
            return 999999

        extended_state = np.pad( state[:4], (0, 4) )

        extended_state[0]  = np.clip(extended_state[0], -0.5, 0.5)
        extended_state[4]  = np.sin(state[2])
        extended_state[5]  = np.cos(state[2]) * state[1]
        extended_state[6]  = np.cos(state[2]) * state[3]
        extended_state[7]  = state[2] ** 3

        state[4] = p @ extended_state
        state = CartPole.perform_action_RK4( state )
        loss += unbounded_loss( state )

    return loss


# In[ ]:


get_ipython().run_cell_magic('time', '', '\nmin_loss = 99999999999\nbest_p = None\n\nfor i in np.arange(1e+9):\n\n    p      = np.random.randn( 8 ) *
3\n    result = scipy.optimize.minimize( extended_swingup_loss, p )\n    \n    end_loss = result["fun"] # extended_swingup_loss(p)\n    \n    if
end_loss < min_loss:\n        best_p = result["x"]\n        #best_p = p\n        min_loss = end_loss\n        \n        print(best_p, min_loss)\n')
```

mpc.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


# import CartPole.py from local directory
import CartPole, sf3utility
import matplotlib.collections
import matplotlib.pyplot as plt
import numpy as np
import scipy.interpolate, scipy.stats.qmc, scipy.optimize
import random, copy, time

plt.rcParams["font.family"] = "Georgia"
#plt.rcParams['figure.figsize'] = [9.0, 7.0]
#plt.rcParams['figure.dpi'] = 400

# store results for later
cache = {}

%matplotlib notebook

# In[2]:


N = 512 * 8

# set the random seed and create the sobol sequence generator
sobol_engine = scipy.stats.qmc.Sobol( d=5, seed=4 )

# get M initial states spaced in the recommended ranges
X = X_sobol = initial_states = (sobol_engine.random_base2( m=int(np.log2(N)) ) - 0.5 ) * 2 * np.array( [ 10, 20, np.pi, 20, 15 ] )
#X = X_sobol = initial_states = (sobol_engine.random_base2( m=int(np.log2(N)) ) - 0.5 ) * 2 * np.array( [ 2, 2, 1, 2, 1 ] ) +
np.array([0,0,np.pi,0,0])
#X[:,2] = np.array( [ CartPole.remap_angle(theta) for theta in X[:,2] ] )

# generate the changes in state from these states
Y = np.array( [ CartPole.perform_action5( state ) - state for state in initial_states ] )


M = 32 * 32
```

```
kernel_centres = X[:M]


sigma = np.std( X, axis=0 )

def K( X, Xp, sigma = sigma ):

    # get squared differences and substitute angle one for periodic version
    d2 = ( (X - Xp) / sigma ) ** 2
    d2[:,0] = 0
    d2[:,2] = (np.sin( 0.5 * ( X[:,2] - Xp[:,2] ) ) / sigma[2] ) ** 2

    # divide rows by 2 sigma and return exponential of negative sum along rows
    return np.exp( - 0.5 * np.sum( d2, axis=1 ) )


# loop over the kernel centres and evaluate the K function across all the Xs at each
Kmn = np.zeros( (M,N) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmn[i] = K( X, kernel_centre[np.newaxis] )

# same as above but only use first M vectors from X
Kmm = np.zeros( (M,M) )
for i, kernel_centre in enumerate( kernel_centres ):

    Kmm[i] = K( X[:M], kernel_centre[np.newaxis] )


l = 1e-4

KmnKnm = Kmn @ Kmn.T
a = KmnKnm + l * Kmm
b = Kmn @ Y

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

def nonlinear_model( state ):

    kernels = K( state[np.newaxis], kernel_centres )

    weighted_sums = kernels @ alpha_m

    return weighted_sums.squeeze()

# In[8]:


unbounded_loss = lambda state : np.linalg.norm( state[:4] * np.array([0.4, 0.2, 1, 0.2]) )

# In[43]:



def estimate_next_loss( state, action ):

    adj_state = state[:]
    adj_state[4] = action
    state = adj_state + nonlinear_model(adj_state)

    return unbounded_loss( state )


def estimate_next_losses( state, actions ):

    loss = 0

    for action in actions:

        adj_state = state[:]
        adj_state[4] = action
        state = adj_state + nonlinear_model(adj_state)
        loss += unbounded_loss( state )

    return loss


# In[50]:


cart_pole = CartPole.CartPole( visual=True, save_frames=False, fig_num=58 )

# In[89]:


state = np.array( [ 0, 0, np.pi, 0, 0 ] )
cart_pole.set_state( state )
cart_pole._render()

time.sleep(1)

for i in range(200):

    state[4] = forces[i]

    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()

    if abs(CartPole.remap_angle(state[2])) < 0.4 and abs(state[1]) < 2.5 and abs(state[3]) < 2.5:
        break


linear_p = np.array( [0.75971638, 1.44284499, 17.05744679, 2.66183469] )

for i in range(50):

    clipped_state = state[:]
    clipped_state[0] = np.clip(clipped_state[0], -0.5, 0.5)

    state[4] = linear_p @ clipped_state[:4]

    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()
```

```
# In[64]:


state = np.array( [ 0, 0, np.pi, 0, 0 ] )
cart_pole.set_state( state )
cart_pole._render()

#states = []

def find_action( state ):

    actions = np.linspace( -20, 20, num=11 )

    actionses = np.array(np.meshgrid( actions, actions, actions, actions, actions )).T.reshape( (-1,5) )
    losses = np.array([ estimate_next_losses(state, actions) for actions in actionses ])

    mindex = np.argmin(losses)

    best_action = actionses[mindex, 0]

    return best_action


for i in range(200):

    state[4] = find_action( state )

    #states.append(state)

    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()

# In[111]:


fig, ax = plt.subplots(1, 1, num=33, figsize=(8,3.5))
fig.subplots_adjust(top=0.9, bottom=0.15, left=0.08, right=0.99)
sf3utility.setup_phase_portrait( ax )

np.random.seed(0)

states = []
times = []
t = 0

means = np.array([ 2.00951924, 1.964584, 1.98959769, 4.76029465, 0. ])
noise_level = 0.01

state = np.array( [ 0, 0, np.pi, 0, 0 ] )
cart_pole.set_state( state )
cart_pole._render()

for i in range(200):

    states.append(state)
    times.append(t)
    t += 0.2

    state[4] = forces[i]

    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()

    state += np.random.randn( 5 ) * means * noise_level

    if abs(CartPole.remap_angle(state[2])) < 0.4 and abs(state[1]) < 2.5 and abs(state[3]) < 2.5:
        break

linear_p = np.array( [0.75971638, 1.44284499, 17.05744679, 2.66183469] )

for i in range(50):

    states.append(state)
    times.append(t)
    t += 0.2

    clipped_state = state[:]
    clipped_state[0] = np.clip(clipped_state[0], -0.5, 0.5)

    state[4] = linear_p @ clipped_state[:4]


    state = CartPole.perform_action_RK4( state )
    cart_pole.set_state( state[:4] )
    cart_pole._render()

    state += np.random.randn( 5 ) * means * noise_level

states = np.array( states )
times = np.array( times )

x = times
y = states

col_lerp = np.linspace(0, 1, 4)[np.newaxis].T
colours = ( 1 - col_lerp ) * np.array( [0, 255, 231, 255] )/255 + col_lerp * np.array( [255, 0, 230, 255] )/255

labels = ["$x$","$\dot{x}$","$\theta$","$\dot{\theta}$"]

for i in range(4):

    f2 = scipy.interpolate.interp1d(x, y[:,i], kind='cubic')

    xnew = np.linspace(0, times[-1], 800)

    ax.plot( xnew, f2(xnew), color=colours[i], label=labels[i], linewidth=2 )

ax.set_title( "Time Evolution of State Vector under Model Forecast Policy with Process Noise (Interpolated)" )
ax.set_xlabel( "Elapsed Time" )
ax.set_ylabel( "Components of State Vector" )
ax.set_xlim(-0.5,5)
ax.set_ylim(-9,8)
```

```
ax.legend(loc="upper right")

# In[68]:


np.array(states).shape[0]*0.2
```

sf3utility.py

```
import matplotlib.pyplot as plt


def setup_phase_portrait( ax ):

    # show the grid
    ax.grid( visible = True, zorder=1 )

    ax.axhline( color="black", zorder=2 )
    ax.axvline( color="black", zorder=2 )
```