Cambridge University Engineering Department
Engineering Tripos Part IIA
PROJECTS: Interim and Final Report Coversheet

TO BE COMPLETED BY THE STUDENT(S)

Project:	SF3 – Machine Learning
Title of report:	Sf3 – Machine Learning ~~Group Report~~ / Individual Report (delete as appropriate)

Name(s): (capitals)	crsID(s):	College(s):
HARVEY HUGHES	Hh458	

Declaration for: ~~Interim Report 1 / Interim Report 2~~ / Final Report (delete as appropriate)

I/~~we~~ confirm that, except where indicated, the work contained in this report is my/~~our~~ own original work.

Instructions to markers of Part IIA project reports:

Grading scheme

Grade	A / A*	B	C	D	E
Standard	Very Good / Excellent	Good	Acceptable	Minimum acceptable for Honours	Below Honours

Grade the reports on the scale A* to D by marking the appropriate Overall Assessment box, and provide feedback against as many of the criteria as are applicable (or add your own). Feedback is particularly important for work graded C-E. Students should be aware that different projects and reports will require different characteristics.

Penalties for lateness: Interim Reports: 3 marks per weekday; Final Reports: 0 marks awarded – late reports not accepted.

Overall assessment (circle grade)	**A***	**A**	**B**	**C**	**D**	**E**
Guideline standard	> 80%	70-80%	60-70%	50-60%	40-50%	< 40%

Marker:		Date:	

Delete (1) or (2) as appropriate (for marking in hard copy – different arrangements apply for feedback on Moodle):

(1) Feedback from the marker is provided on the report itself.

(2) Feedback from the marker is provided on second page of cover sheet.

	Typical Criteria	Feedback comments
Project Skills, Initiative, Originality	Appreciation of problem, and development of ideas	
	Competence in planning and record-keeping	
	Practical skill, theoretical work, programming	
	Evidence of originality, innovation, wider reading (with full referencing), or additional research	
	Initiative, and level of supervision required	
Report	Overall planning and layout, within set page limit	
	Clarity of introductory overview and conclusions	
	Logical account of work, clarity in discussion of main issues	
	Technical understanding, competence and accuracy	
	Quality of language, readability, full referencing of papers and other sources	
	Clarity of figures, graphs and tables, with captions and full referencing in text	

SF3 : Machine Learning

Harvey Hughes
Emmanuel College
hh458@cam.ac.uk

June 7, 2019

Abstract

Investigations into a cart and pendulum system were carried out to determine the suitability of various models, and to then learn to control the system to a desired point. The system proved to be highly non linear in velocity and angular velocity across the range of motion. A linear model was unable to predict this behaviour at all points, unlike a non linear model which was. The non linear model can b optimised in various aspects to improve accuracy. Using the trained model policies can be optimised that can flip the pendulum upright and balance it there.

I. Introduction

Investigations in python were carried out on a cart and pendulum system throughout this project. The system can be described using state vector $\mathbf{x} = [x, \dot{x}, \theta, \dot{\theta}]^T$. The project aims are:

- Simulate the system behaviour in python for varying initial conditions
- Model the system behaviour using both linear and non-linear methods
- Control the system with linear and non-linear models in order to get the system into its unstable equilibrium position, where the pendulum is inverted
- Consider the effect of noise on modelling

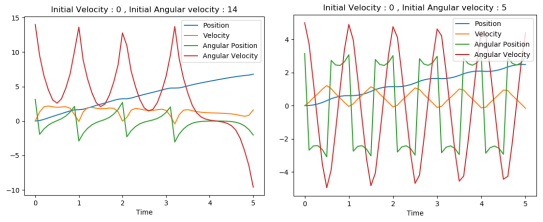
The python code written to implement these features is listed in the appendix.

II. Investigations

i. Dynamic Simulations

i.1 Rollout

Using the dynamic simulation provided in cart-pole.py state trajectories can be calculated for various starting conditions. Figures 1a and 1b show the trajectory for a system looping round three times before changing direction, and osculations about $\theta = \pi$ respectively. These simulations are what is expected from a cart-pole system with friction. Oscillations are about the stable equilibrium at $\theta = \pi$, therefore gravity is modelled the correct way around.



(a) *Three full loops* **(b)** *Oscillations*

Figure 1: *State trajectories in two scenarios.*

i.2 Change of state

Using random initialisation and modelling the dynamics for 0.1s the next state can be calculated. Figure 2 shows how scanning one state variable in turn effects the next state. Horizontal lines when scanning position show that this state variable has no effect on the next step's $\dot{x}$, θ or $\dot{\theta}$. Velocity and angular velocity are observed to be very non-linear no matter the state

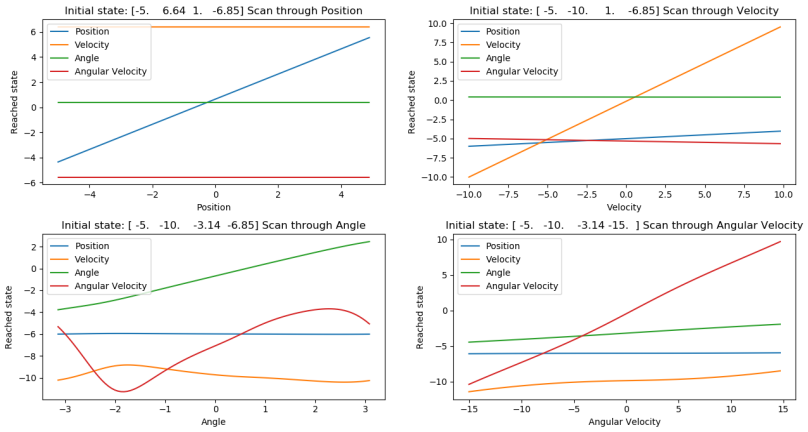


Figure 2: *Scan through each state variable to see the effect on the next state*

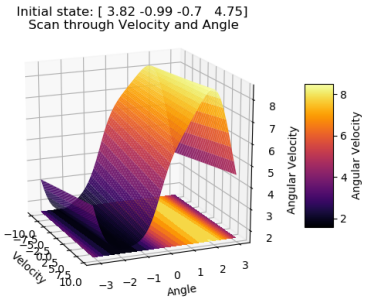


Figure 3: *Scan through $\dot{x}$ and θ to see the effect on the next step's $\dot{\theta}$.*

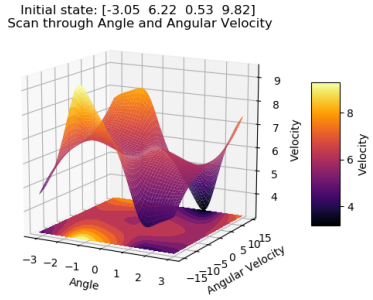


Figure 4: *Scan through $\dot{\theta}$ and θ to see the effect on the next step's $\dot{x}$.*

initialisation. These non-linearities are best shown by the sine like behaviour of $\dot{\theta}$ when scanning θ. Smaller amplitude sinusoidal relationships or other non-linear behaviours are observed between $\dot{x}$ and θ, and $\dot{x}$ and $\dot{\theta}$. These can be visualised as a surface plot as in figures 3 and 4, additionally these plots show a contour plot below the surface plot in order to show information on hidden areas of the surface. In figure 3 the sinusoidal relationship between θ and $\dot{\theta}$ is once again observed alongside an approximately linear relationship between $\dot{x}$ and $\dot{\theta}$.

Figure 4 shows a complex non-linear behaviour between θ, and $\dot{\theta}$ and $\dot{x}$. These relationships change depending on the initialisation of θ and $\dot{\theta}$. By taking different slices through the surface plot these non-linear behaviours can be imagined and range from sinusoidal to quadratic trends.

Similar plots to figure 2 can be constructed using the change in state, the same trends are observed and can be seen in figure 7.

ii. Modelling

ii.1 Data collection

Data pairs **x,y** can be generated by initialising in the suitable region $x \in [-5, 5]$, $\dot{x} \in [-10, 10]$, $\theta \in [-\pi, \pi]$ and $\dot{\theta} \in [-15, 15]$. $\mathbf{x}$ is the initial state $\mathbf{x}(0)$, and $\mathbf{y}$ is the change in state after 0.1s, $\mathbf{x}(1) - \mathbf{x}(0)$.

ii.2 Train linear model

Through observing the relationships in section i.2 the system is shown to be highly non-linear in multiple aspects. Therefore a linear model is

expected to have downfalls at predicting state transitions. These will be considered in this section.
Using the 1000 generated **x,y** pairs a linear model can be generated that minimises the lease squared error given in equation 1. **X,Y** are (4x1000) matrices with each column containing an **x,y** pair.

$$\begin{aligned} \mathbf{C} &= argmin(\mathbf{e}^T\mathbf{e}) \\ \mathbf{e} &= \mathbf{Y} - \mathbf{Y}_p = \mathbf{Y} - \mathbf{C}\mathbf{X} \end{aligned} \tag{1}$$

Figure 5 shows how the predictions using the model compare to the true training data. An approximately linear trend is observed with some variance about $\mathbf{y} = \mathbf{y}_p$ for the angle and position. These variables are well approximated with the linear model. However, velocity and angular velocity show a highly non-linear behaviour with the predictions not accurately matching the observed data. Gradients and determination coefficients can be seen in table 1. For a good fit m=1 and r^2=1, the regression values thus further show the non-linear behaviour of velocity and angular velocity.

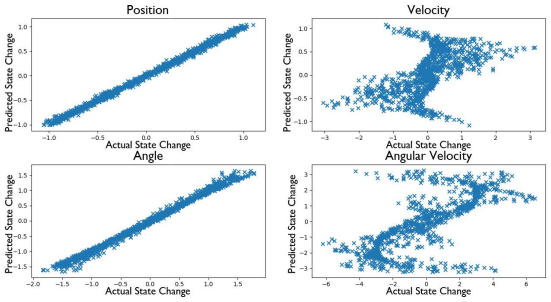


Figure 5: *Linear model performance on the training set*

If the model is trained and initialised only about the stable equilibrium point $\theta = \pi$ the trend becomes more linear, see figure 6. This is due to small motion about this equilibrium being better approximated by a linear model. Table 1 shows that the fit greatly improves as all regression values nearly equal one.

The predicted change in state and actual change can be plotted as each variable is scanned through it's range, these are shown

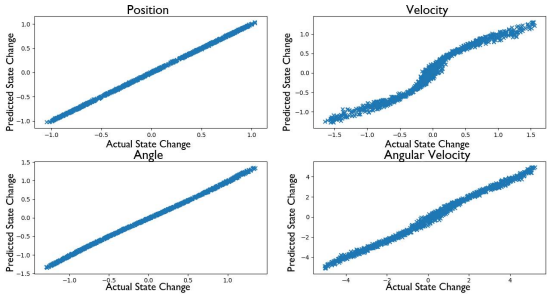


Figure 6: *Linear model performance only about $\theta = \pi$*

State	Whole Motion		Stable equilibrium	
	m	r^2	m	r^2
x	0.996	0.996	0.9996	0.9998
$\dot{x}$	0.285	0.287	0.950	0.951
θ	0.991	0.991	0.9996	0.9994
$\dot{\theta}$	0.481	0.482	0.991	0.993

Table 1: *Gradient (m) and determination coefficient (r^2) of linear predictions*

in figure 7. Each predicted relationship is linear as expected, with the predictions agreeing closely for x and θ showing the near linear relationships in these variables. However, $\dot{x}$ and $\dot{\theta}$ are once again badly predicted by the liner model, either through a straight line attempting to fit to a curved line as shown when scanning θ or $\dot{\theta}$, or as a constant offset seen when scanning x and $\dot{x}$.

ii.3 Test linear model

In order to fully test the generality of the model state trajectories can be compared against actual trajectories. Care must be taken to remap θ to remain in the range $[-\pi, \pi]$, otherwise when θ increases all other state variables explode as the model is only trained in the remaped range.

Figure 8 shows an initialisation which causes five loops before decaying into oscillations. The state trajectory predictions disagree to a large extent with the actual trajectory. This shows that the linear models inaccuracies highlighted in section ii.2 all add up and cause divergence or different oscillation periods in the state variables, these non-linearities occur in x and θ because of the model linking v and $\dot{\theta}$ to x and

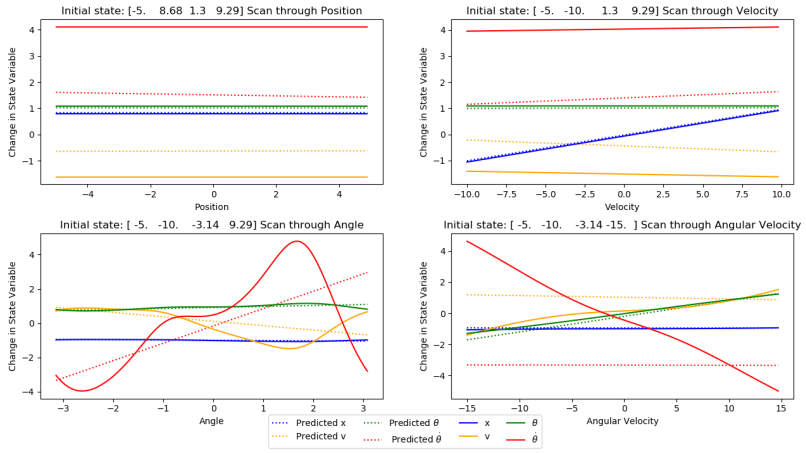


Figure 7: *Predicted state change and real state change when scanning through each variable*

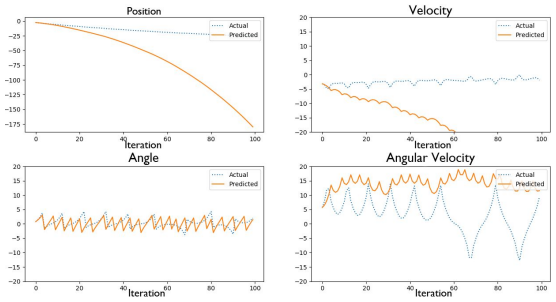


Figure 8: *Predicted state trajectory with loops*

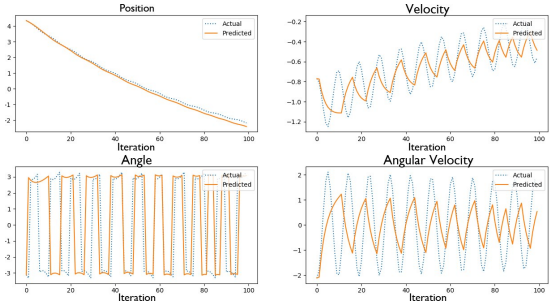


Figure 9: *Predicted state trajectory about $\theta = \pi$*

θ. Bad predictions were typical for all types of motion, osculations or loops.

In order to check for overfitting about the training region the model was retrained on larger suitable ranges for **x**. However, this didn't solve the build-up of errors and bad predictions were still observed.

In task ii.2 it was shown that motion about the stable equilibrium was more linear. Figure 9 shows a trajectory calculated on data trained and initialised about $\theta = \pi$. While still not perfect especially for oscillation magnitude the prediction matches the actual trajectory to a far greater degree than when the whole motion is considered. This further shows that motion here is approximately linear.

ii.4 Quasi-random data collection

The collection of data points was previously done using a unique random seed to generate each **x** location. A more efficient method of selecting points is to quasi-randomly distribute them over the training volume. This involves generating subsequent data points based on the generation history. The effect of this is basis locations and training data that provide observations throughout the whole range of **x** without overlap. The python module sobol_seq was used to generate this sequence. The performance marginally increased for the same number of training points after implementing this change.

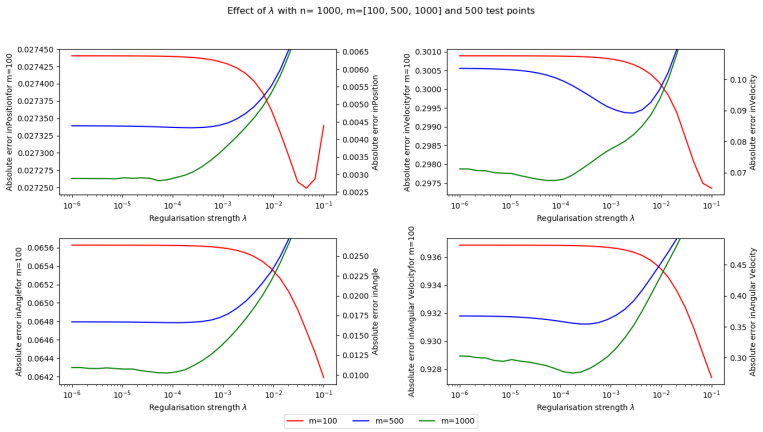


Figure 10: *Scan through λ with 500 test points*

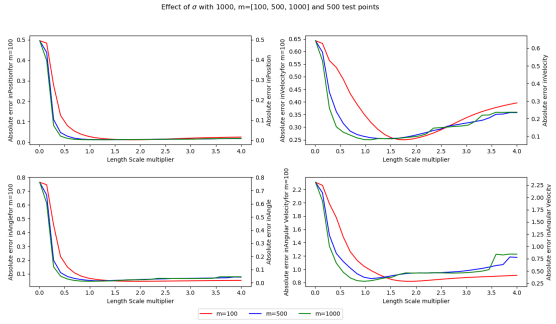


Figure 11: *Scan through length scale σ*

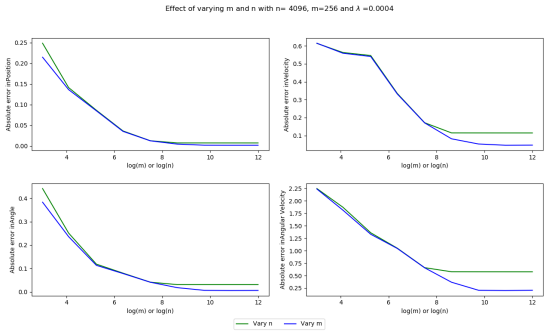


Figure 12: *Scan through m and n*

ii.5 Train non-linear model

As discussed previously the system behaviour is highly non-linear which requires a suitable non linear modelling technique for accurate predictions of trajectory. M Gaussian basis functions were implemented as a kernel for this modelling based on N training points. Less basis locations are chosen than data points to reduce overfitting to training data, as the model could perfectly learn the behaviour at N points with N basis locations. Equation 2 shows how the state change is generated. Each basis location will allow localised learning of the dynamics. If these cover the whole range of $\mathbf{x}$ then the non-linearities can be captured. As θ is periodic its useful to use $sin^2((\theta - \theta')/2)$ in the Gaussian kernel so that basis locations effect points at the opposite end of the range.

$$K(X, X') = e^{-\sum_j \frac{(X_j - X'_j)^2}{2\sigma_j^2}} \qquad (2)$$
$$\mathbf{Y}_p = \boldsymbol{\alpha}_{4M}\mathbf{K}_M$$

Regularised least squares is used to train $\boldsymbol{\alpha}$ for the model. Equation 3 shows how the solution to this can be set-up in standard least squares form to avoid the need to invert an MxM matrix. The regularisation constant λ improves generality of the model by limiting overfitting. This form arises when you minimise the sum of the squared error and the RKHS (Reproducing kernel Hilbert space) norm.

$$(\mathbf{K}_{MN}\mathbf{K}_{NM} + \lambda\mathbf{K}_{MM})\boldsymbol{\alpha}_{M4} = \mathbf{K}_{MN}\mathbf{Y}_{N4} \qquad (3)$$

By varying λ appropriately, accuracy in test data can be improved. Figure 10 shows this

generalisation relationship. There was an observed dip in error especially for $\dot{x}$ and $\dot{\theta}$, this arises because generality improves as λ increases until its large enough that the model is starting to be dominated by the regularisation. The dip was observed to be very small showing that the model has a good fit normally. The optimum value for lambda varied depending on M, these are shown in table 2.

M	Optimum λ	Optimum σ
100	3e-2	1.75
500	4e-4	1.1
1000	2e-4	1

Table 2: *The optimum values for λ and σ to achieve generality in training.*

Using these optimised regularisation coefficients the Gaussian basis length scale can be optimised. Figure 11 shows a scan of σ where 1 corresponds to the standard deviation in **X** for that state variable. Table 2 lists the optimum values. σ approaches one as M increases, this is because the proximity of basis locations decreases so a smaller length scale is required.
The last variables to investigate are number of training points (n) and number of basis locations (m). Figure 12 shows the result of scanning these variables. The additional benefit in both variables exponentially decreases with number, while simultaneously increasing computation time. Increasing n past the number of basis locations (256) showed little decrease in error. However, having large n only effects the computation time to build the model and not the time to use it. When m was larger than 1000 little error reduction was observed. A model using 500 or 1000 basis locations and over twice the number of training locations would be suitable and is used in later investigations.

The trained model was tested for accuracy in predictions against the training data as investigated earlier on the linear model. Figure 13 shows this for m=500, n =10000 with table 3 listing the gradient and coefficient of determination for various models. As before both

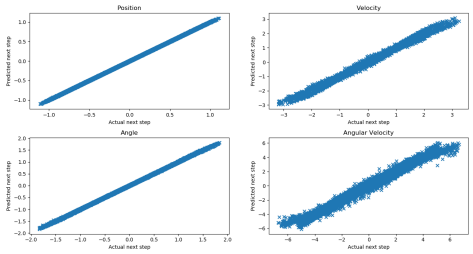


Figure 13: *Model performance on training set m=500, n=10000*

of these parameters are close to one for accurate predictions, which is observed in models where m=500 or 1000. These results show that a Gaussian kernel model is appropriate and accurate at predicting one time-step.

M		State x	$\dot{x}$	θ	$\dot{\theta}$
		No Force			
500	m	0.9999	0.9880	0.9997	0.980
	r^2	0.9999	0.9884	0.9998	0.981
1000	m	0.9999	0.9940	0.9999	0.990
	r^2	0.9999	0.9945	0.9999	0.991
		With Force			
500	m	0.9992	0.9534	0.9961	0.903
	r^2	0.9992	0.9534	0.9961	0.903
1000	m	0.9997	0.9803	0.9984	0.950
	r^2	0.9998	0.9804	0.9984	0.950

Table 3: *Gradient (m) and determination coefficient (r^2) of non-linear predictions with M basis locations*

The python module Pickle was used to save the model variables in order to speed up future investigations.

ii.6 Test non-linear model

Figures 14 and 15 show the predicted trajectory for full loops and oscillations respectively. This model used 1000 basis locations and 10000 training points. The model accurately follows the trajectory for around 30-40 time-steps. After this the build-up of errors causes the model to diverge. It was though this may be due to the cart position falling outside of the training region of [-5,5], therefore causing the trans-

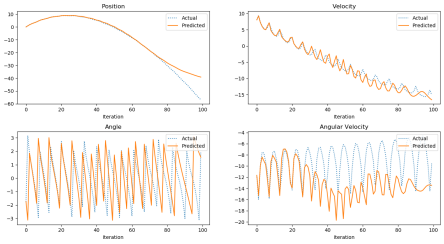


Figure 14: *Trajectory predictions when m=1000, n =10000 for a non linear model*

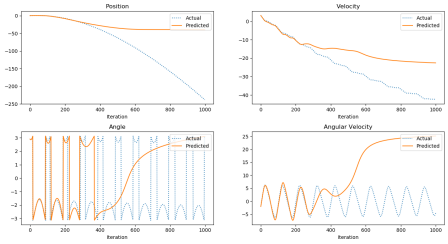


Figure 15: *Trajectory predictions when m=1000, n =10000 for a non linear model, training time-step was 0.01s.*

formed kernel to be zero at each location, and the calculated state change to also be zero. The model was tested using a larger range of [-15,15] which showed a minor improvement. When controlling the model in later investigations the x position would be controlled meaning this change wouldn't affect the accuracy in later tasks. Training the model on time-steps of 0.01s instead of 0.1s was also considered. For a smaller time-step it was believed the behaviour would be less complex and therefore easier to model. However, it was observed in figure 15 and other simulations that the error also scaled by a factor of 10 and trajectory prediction was only accurate for 3s as before.

The non linear model trained with 1000 basis locations on 10000 data points was tested for error against the true dynamics. Figures 17 and 18 show surface contour plots of this error. The error in $\dot{x}$ is close to 0 at the centre of the region with periodic spikes at the extremes of the testing range in $\dot{\theta}$. The same periodic peaks is observed in the error for $\dot{\theta}$ however the large error occurs at all velocities in this case. The basis locations plotted in blue show this error doesn't arise from a lack of information trained in this region. When the training region was extended to double the range the errors in $\dot{x}$ occurred at the new extremities, this means that the error in the desired range is better. These peaks are very thin and the model was still able to predict trajectories for a long enough time period. The average error in each state variable can be seen in figures 10 and 11 which are nearly zero in x and θ and 0.1, 0.25 in $\dot{x}$, $\dot{\theta}$ respectively.

Due to time constraints the following idea was never tested but may have improved the error. If basis locations are drawn more often in areas of high error and less often where the model accurately predicts, similar to importance sampling, then the peak errors could be reduced without using more basis locations. The locations could be drawn to follow the cosine pattern in error as θ is scanned. This would enable better learning of the system behaviour where it most most needed.

Figure 16 demonstrates the accuracy of change of state predictions for a model with m=1000, n=10000. Unlike the linear model discussed previously the complex non-linearities are predicted with the model, except at the peaks in $\dot{\theta}$ when scanning θ.

ii.7 Inclusion of force

To learn to control the system to a desired point the non linear model must know the effect an imputed force has on the state variables. To do this force was added as the fifth state variable and the model was retrained. As the state variables now cover 5D space instead of 4D the number of basis locations required for accurate predictions was increased. Figures 19 and 20 show the models success at predicting state change and trajectories respectively. Gradients and determination coefficients of predicted state change vs actual state change on the training data can be seen in table 3. Predictions still appear to capture most of the complex behaviour and trajectory predictions are

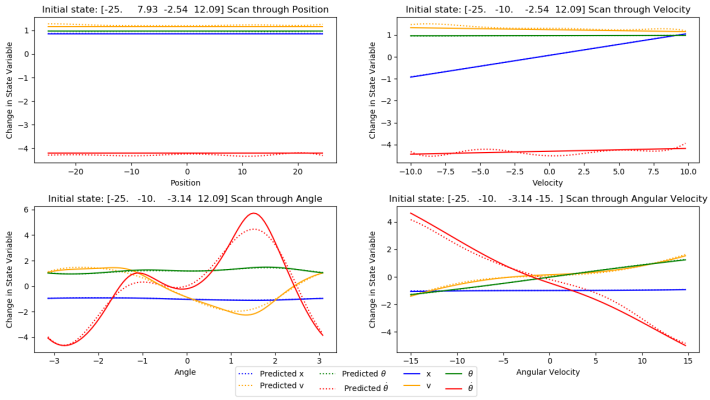


Figure 16: *Predicted state change and real state change for non linear model where m=1000, n=10000*

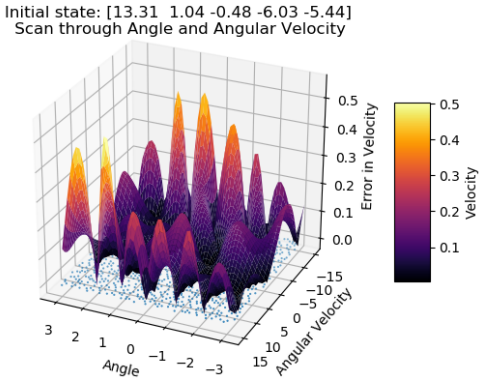


Figure 17: *Scan for error when m=1000, n=10000. Blue dots indicate basis locations.*

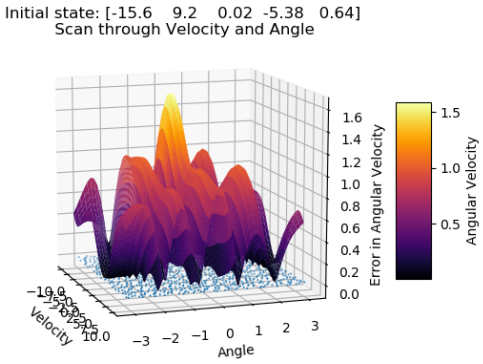


Figure 18: *Scan for error when m=1000, n=10000. Blue dots indicate basis locations.*

accurate for 30 time-steps. The non linear behaviours are still mostly captured when scanning through the state variables as in figure 19. Correlation values are all close to one with the most benefit from 1000 basis locations observed when predicting $\dot{\theta}$ as r^2 increases from 0.9 to 0.95 which could be significant when predicting. To improve model accuracy without increasing m, x could be removed from the state variables as it has no effect on the other states. This would reduce the space for basis functions back to 4D as before. However, this was never implemented as it was thought of towards the project end.

iii. Control

The aim was to balance the pendulum in its unstable equilibria position at $\theta = 0$, with small oscillations and without large velocities. This would be achieved by optimising a policy function (equation 4) using the trained model and a loss function evaluated over a trajectory (equation 5). The trajectory would be limited to 20 time-steps to maintain accuracy in the non linear model. Minima for **P** found during optimisation would be tested against the true dynamics to evaluate the actual performance if the policy was implemented in the real world.

$$F = \mathbf{PX} \tag{4}$$

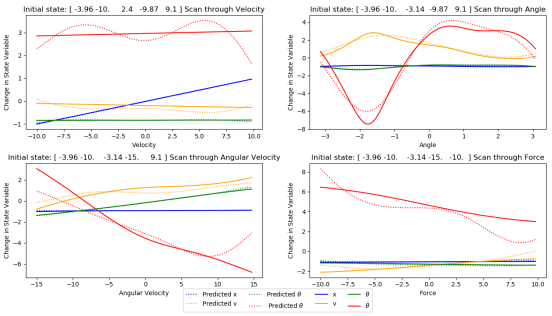


Figure 19: *Predicted state change and real state change for non linear model where m=1000, n=10000 including force*

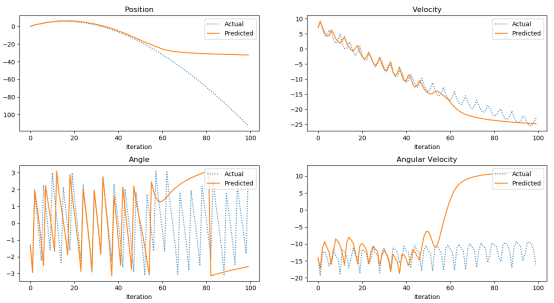


Figure 20: *Trajectory predictions with m=1000, n=10000 including force*

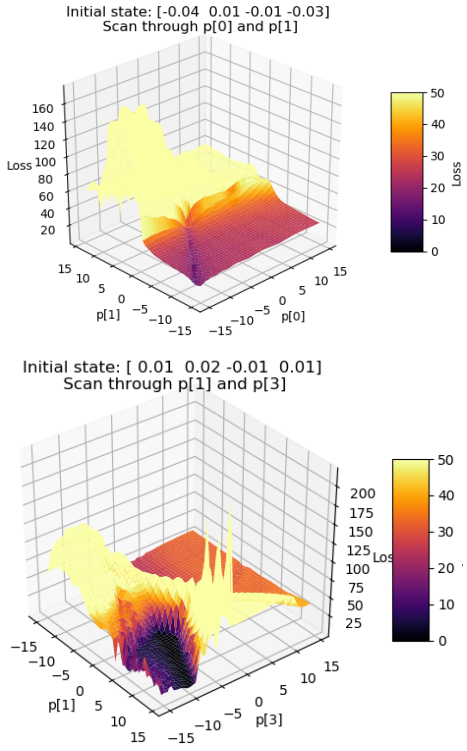


Figure 21: *Scanning the loss function*

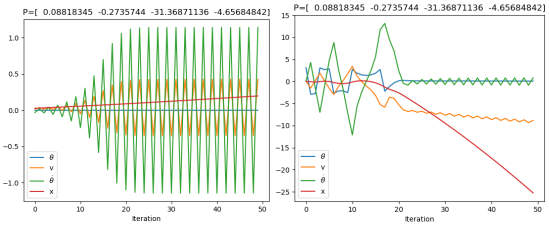
$$L = \sum_{t=0}^{t=20} [1 + e^{-0.5\frac{\theta}{s_\theta}^2} (\dot{\theta}^2 s_{\dot{\theta}} - 1)] \qquad (5)$$

iii.1 Linear control

A model with m=500, n=10000 was initially chosen to use for training, this was due to the time saving when calculating transformed state variables many times, the small difference in accuracy and the more linear behaviour about the desired point at θ = 0.
Initialisation for the loss function trajectory was originally at $\mathbf{x}$ = [0,0.2,0.05,-0.2]^T. The resulting loss function is a complex hyperplane with multiple minima. Figure 21 illustrates the function as 2D contour scans. These plots can be used to select sensible initialisation points and help to understand locations where convergence is likely to not occur.

To optimise **P** a gradient decent algorithm was initially written and tested with a momentum term to make subsequent search directions conjugate. However, minima that achieved the aim were hard to find due to the following: getting stuck in long plateaus in the loss function, looping in 0.1s often minimises the loss, and if started too close to θ=0 then the policy would do nothing and let the pendulum slowly fall. An inbuilt python library was then used: scipy.optimise. A minima was located at **P**=[0.088,-0.274,-31.37,-4.66] after initialising at **P**=[-1,1,-3,-6], this value was able to keep the pendulum steady at the top with some drift figure 22a, and able to bring the pendulum up from $\theta = \pi$ but the cart would shoot off very quickly, figure 22b.

In order to reduce this drift and increase the initialisations near the top that are stable, the loss function was adjusted. After adding in loss terms favouring x=0 , and v=0 as in equation 6 new minima were located. Care must be taken when adjusting the scal-



(a) *Initialise at top* **(b)** *Initialise at bottom*

Figure 22: *Rollout with* ***P****=[0.088,-0.274,-31.37,-4.66]*

ing of each of these terms as often unwanted behaviour becomes optimum. The scaling $[s_x, s_v, s_\theta, s_{\dot{\theta}}] = [0, s_{\dot{\theta}}/20, \pi/2, 0.3]$ provided a good optimum at **P**=[-1.792, 1.366, -38.372, -2.865]. This position was able to stabilise initialisations drawn normally with σ=0.5 from the top position, however drift still occurred after 30 time-steps. Initialising from this new point the model with m=1000 was then used, along with increasing the loss initialisation to be 10 times further from the top. This was in attempt to use better modelling and harder initialisation in order to increase the possible working initial trajectories. The final policy was **P**=[4.262, 10.236, -80.795, -10.032]. Figure 23 shows a typical trajectory using this policy, showing the pole remaining upright and x tending to 0. The range of initialisations this model worked for are **x** in ranges [2.9,1.4,0.54,4.2] either side of **0**, each calculated with the other states = 0. Failure typically happened when the initial $\dot{\theta}$ took the pole further away from vertical.

$$\begin{aligned} v_{loss} &= (1 - \theta_{loss})v^2 s_v \\ x_{loss} &= (1 - \theta_{loss})x^2 s_x \end{aligned} \tag{6}$$

iii.2 Non-linear control

Different control functions are required across the range of motion in order to achieve the aim of starting with the pendulum hanging below and ending balanced above. For this a non linear policy can be defined, this policy would use 11 sensibly placed basis functions across the

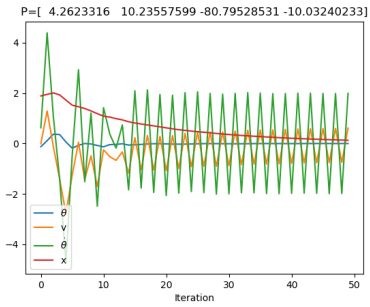


Figure 23: *Rollout using linear policy with* ***P****=[4.262, 10.236, -80.795, -10.032]*

motion. The basis functions are the same Gaussian used earlier, however the length scale is optimised according to the basis location and purpose. Once working basis locations had been found their exact locations could be optimised, doing this is stages helped to reduce computation time for each minimisation, and increased the likelihood of reaching a viable solution. Common unwanted solutions found otherwise resulted in very small oscillations about $\theta = \pi$ or looping behaviour with a time period of 0.1s .

Due to the accuracy of the dynamic model being limited to 20 time-steps multiple initial conditions for computing trajectory loss would be used. The whole problem could be broken down into three tasks, each involving separate optimisations. Upright balancing, rotating the pendulum upwards, and centring the cart at x=0. Separating into sub problems helps by reducing the search space dimensionality, thus increasing the likelihood of convergence. Scans of loss function were used to visualise the minima and choose initial **P**'s as done in the linear model.

Pickle was used once again to save policy variables for later used, this was easily implemented as the policy python class was the same as the non linear model used earlier.

iii.3 Non-linear control training

The first task in control is the same as achieved during linear model training, balancing the

pendulum above. This was considered first because its the most important step in minimising loss in the long term, without being able to balance here flipping the pole upwards wouldn't minimise loss by much, meaning desired solutions are unlikely to be found.
Due to the symmetry of a Gaussian basis function locations chosen were either side of θ=0 as the force required on either side would be opposite. Additional symmetry in the task was taken advantage of when training as the points to either side should be in mirror positions with the same length scales but opposite control value (p). This reduction further reduced solution search space to speed up convergence. Two locations were first tested at $\mathbf{x}$=$[0, 0, \pm 0.25, 0]^T$, σ=$[3, 5, 0.25, 0.5]^T$ and with loss scales set to $[s_x, s_v, s_\theta, s_{\dot{\theta}}] = [0, s_{\dot{\theta}}/20, \pi/2, 0.3]$. The result of this was **P**=[-13.13,13.13]. However this set-up caused oscillations to increase and stability to be lost. Two additional basis functions were added next, this was to have different policies for decreasing $\dot{\theta}$ and for recentering the pendulum. This change required the locations to be set to $\dot{\theta} = \pm 2.5$ with σ=2.5, and $\sigma_x = \sigma_v = 30$ as the cart drifted. After training on a larger initialisation, and changing s_x to $s_{\dot{\theta}}/10$ to reduce drift the resulting policy was found: **P**=[-27.669,27.669,-10.98,10.98]. Figure 24 shows a typical trajectory for this. The working initialization range was [anything, 30, 0.68, 5.1] either side of 0 when all other states were 0. This is an improvement on the linear policy already, however drift was a major problem to be solved later.
Basis locations were then optimised with **P** constant, resulting in valid initialisations in the range [anything, 30, 0.77, 5.1] . This increases the acceptable range for θ. One problem with keeping variables fixed is true minima may not be found due to the dependence between variables, however convergence speed is far better when minimising less variables. The final locations were $\mathbf{x}$=$\pm$[-0.01,-0.04,0.575,2.654]^T and $\pm$[0.005,0.093,0.882,-1.321]^T.

The next process to control is flipping the pendulum up from the downward position.

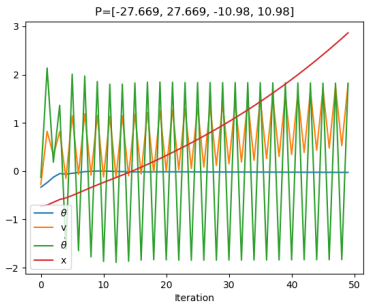


Figure 24: *Rollout using non-linear policy with* ***P****=[-27.669,27.669,-10.98,10.98], about top*

For this the top basis functions were kept constant. Difficulties arose due to the small time period where the dynamic model was accurate. This meant the desired motion should bring the pendulum to the top with few oscillations in order for the loss function to be evaluated after the pendulum is balanced. This was realised with one small oscillation before launching the pendulum upwards. One downside to this was the large cart velocities generated which caused drift. Slowly building up oscillation height would have solved this issue, and could have been done using a new loss functions with this as the goal. Then initialising in a state from these oscillations and using the original loss function. This was never implemented and instead the large velocities were dealt with once balanced upright.
This was achieved with two basis functions located either side of $\theta = \pi$, to generate a small oscillation and some located part way around the loop at $\dot{\theta}'s$ corresponding to rotating round that direction. After initial optimisation some of these locations were shown to not be useful as the optimised control value was near to 0. These were subsequently removed from training until only one remained at $\mathbf{x}$=[0, -1, 1.2, -10]^T, σ=[5., 5., 0.2, 8]^T. The two bottom functions were at $\mathbf{x}$=$\pm[0, 0, \pi - 0.3, -1.5]^T$, σ=[7., 7., .5, 5.]^T. Training was done using 30 time steps as the model was accurate enough for this long, and 3s is long enough for the pendulum to come to rest at the top. The loss scales were set to $[s_x, s_v, s_\theta, s_{\dot{\theta}}] = [s_{\dot{\theta}}/10, s_{\dot{\theta}}/20, \pi/2, 0.3]$.

Giving the policy: **P**=[76,-77,-37.025] , a typical rollout can be seen in figure 25. This suffers from drifting off due to the large forces required to swing the pendulum upright. This drift is fixed later.

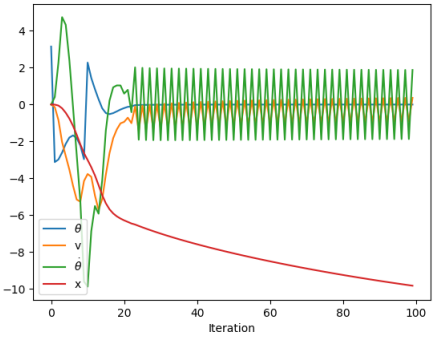


Figure 25: *Rollout using non-linear policy with **P**=[-27.669,27.669,-10.98,10.98,76,-77,-37.025], from bottom*

The final problem to be solved was the drift, for this four basis locations would be added. Two of these would be to push the cart back when at x=$\pm$6 using a sharp impulse. Care had to be taken with initialised length scale as pendulum sensitivity to looping was very high therefore requiring careful basis selection. These were located at **x**=$\pm[-6, .25, 0, 0]^T$, σ=[0.2, 0.5, 0.2, 3]^T. The last two locations were located near x=0 in order to bring the cart into a stable equilibrium here. They were at **x**=$\pm[1, 5, 0, 0]^T$, σ=[1., 3, 0.2, 3.]^T. The resulting policy was **P**=[-20,20,-18.95,18.95], a typical trajectory can seen in figure 26. The drift was successfully corrected however oscillations were observed about x=0. The basis x location were consequently moved to $\pm$0.5 which fixed this issue as seen in figure 27a. To optimise the policy within the accurate time frame of our model trajectories were initialised at **x**=$[-5.5, -3.36, 0, 0.42]^T$

The major downfall of this policy is the lack of slowing down when initialised in random positions, the trajectory will often end up looping. The recentering of the cart is also not perfect, ways of improving this are discussed later on. Additionally had there been more time optimising all basis locations and length scales would have been performed.

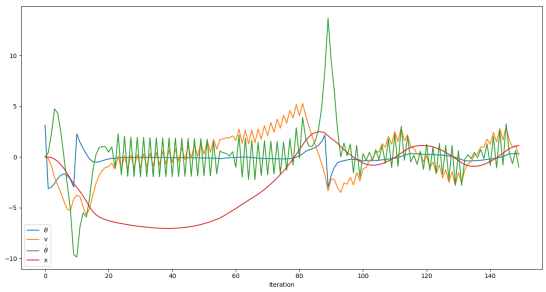


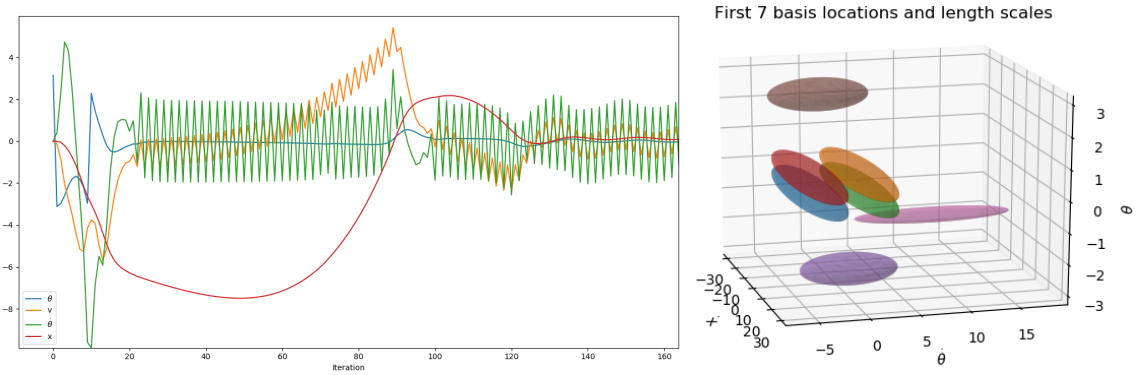
Figure 26: *Correction for drift but large oscillations are observed*

iii.4 Basis locations

The final basis locations, length scales and policy is listed in equations 7 - 9. A graphical visualisation in figure 27b was used throughout training to consider sensible placements and length scales.

iii.5 Extensions

Various extensions were thought about that could improve the policy function.
Training the model to build up oscillation height slowly in order to not deviate away from x=0 when the pendulum is swung. This could be done using a new loss function that values this motion.
Using different kernel shapes that are asymmetric or the untransformed state inputs in order to affect the cart at large x values and therefore recentre it.
Initialise the training with a different start point to θ=π to increase the generalisation of the model.
Add basis functions which slow down fast initialisations as these cause constant looping behaviour with the current policy. However, this may not occur when pendulum friction is considered.
Force could be added to the loss function in order to find controller values which give the designed motion most efficiently.



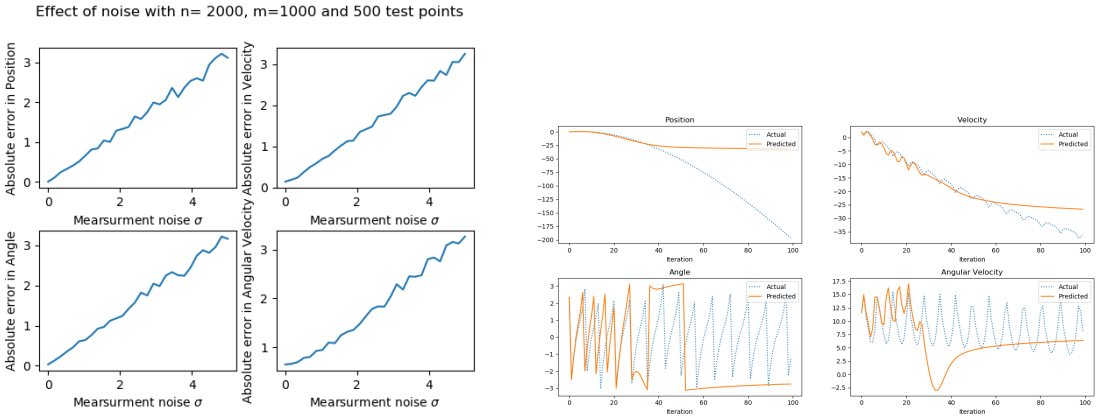
(a) *Desired behaviour* **(b)** *Basis locations*

Figure 27: *successful rollout using non linear policy, and visualisation of basis locations*

$$\mathbf{X}' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & -6 & 6 & -.5 & .5 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & .25 & -.25 & -5 & 5 \\ .25 & -.25 & .25 & -.25 & 2.84 & -2.84 & 1.2 & 0 & 0 & 0 & 0 \\ 2.5 & -2.5 & -2.5 & 2.5 & -1.5 & 1.5 & -10 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{7}$$

$$\sigma = \begin{bmatrix} 30 & 30 & 30 & 30 & 7 & 7 & 5 & .2 & .2 & 1 & 1 \\ 30 & 30 & 30 & 30 & 7 & 7 & 5 & .5 & .5 & 3 & 3 \\ .25 & .25 & .25 & .25 & 5 & 5 & .2 & .2 & .2 & .2 & .2 \\ 2.5 & 2.5 & 2.5 & 2.5 & 5 & 5 & 8 & 3 & 3 & 3 & 3 \end{bmatrix} \tag{8}$$

$$\mathbf{P} = \begin{bmatrix} -27.669 & 27.669 & -10.98 & 10.98 & 76.0 & 77.0 & -37.025 & -19.9 & 19.9 & -18.95 & 18.95 \end{bmatrix} \tag{9}$$



(a) *Error* **(b)** *Trajectory: σ=0.05*

Figure 28: *Effect of noise on error and trajectory predictions, 2m=n=2000 in a and 10m=n=10000 in b*

iv. Noise

Gaussian noise was added to the measured next state when training the model. This is to mimic real life sensors having uncertainties. The effect of this noise on average error in predictions was tested. Figure 28a shows this relationship. The is the absolute error on data points measured without noise. As predicted a strong positive correlation was shown. However, the small dips in error were unexpected as noise should always make modelling harder. These are likely caused by a sample of 2000 causing mean deviation to not always increase with noise. Trajectories were tested when σ=0.05, and can be seen in figure 28b. The model is accurate for about 25 iterations which is slightly less than before.
As the noise is known it can be incorporated into the predictions in order to get better estimates. This was never implemented however it could be done in a few ways. The length scales of each basis location should have the noise σ added on, to account for the inaccurate locations trained on. When plotting trajectories the belief about current position could be modelled as a likelihood which is updated with a prior at each time step. This prior is the noise. From the posterior generated, techniques could be used to determine true position more accurately.

v. Bayesian learning

An interesting extension to this project if more time was available would be to use Gaussian processing tools to learn how to control the system in very little simulation time. This reinforced learning technique is particularly useful when system behaviour data is very limited. Gaussian processing tools are effective with small data as they efficiently incorporate prior knowledge and uncertainties into modelling.[1] The model efficiently learns by evaluating probabilities that a given force will reduce the equivalent to our loss function, and by occasionally trying 'bad' moves in order to get data from unseen areas. [2][3]

III. Conclusion

- Complex non-linear relations ships exist between $\dot{x}$, θ and $\dot{\theta}$
- x has no effect on the other state variables
- Predicting x and θ shows good correlation with a linear model
- Bad correlation is observed when predicting $\dot{x}$ and $\dot{\theta}$ when using a linear model, the correlation improves when training about the stable equilibrium point
- Predicting state trajectories using a linear model is very inaccurate
- Small motion about θ=π is approximately linear meaning state trajectories predictions are nearly correct
- Extending the training region has no effect on the predictive accuracy of the linear model, meaning the errors are not due to overfitting and instead arise from the non-linear system behaviour
- A non linear model using between 500-1000 basis locations is able to accurately predict trajectories for 30 time-steps and state changes with or without the inclusion of force
- The model should be trained with n at least twice m
- Accuracy can be increased by optimising regularisation constant and basis length scales for each choice of m. The regularisation helps to limit overfitting
- Largest errors in prediction occur at training bounds and sinusoidally with angle
- A linear policy function can be optimised using a loss function and the trained dynamic model in order to balance the pendulum upright
- A non linear policy function can flip the pendulum above, balance it there and recentre the cart using 11 carefully places basis functions
- When optimising policy values different initialisations, loss functions, and splitting the task into sub tasks are required to more efficiently find good solutions
- Noise is shown to degrade model predictions

IV. References

[1]:Hildo Bijl, Jan-Willem van Wingerden and Michel Verhaegen. Applying Gaussian Processes to Reinforcement Learning for Fixed-Structure Controller Synthesis. 2014
[2]:Marc Peter Deisenroth and Carl Edward Rasmussen. PILCO: A Model-Based and Data-Efficient Approach to Policy Search. 2011
[3]:Carl Edward Rasmussen. http://mlg.eng.cam.ac.uk/carl/ewrl08/ . 2008

V. Appendix

i. Task_1_Functions.py

```
import numpy as np
from CartPole import *
import matplotlib.pyplot as plt
import random
from mpl_toolkits.mplot3d import Axes3D
import sobol_seq

labels = [ "Position" , "Velocity" , "Angle" , "Angular␣Velocity"]
ranges = [50.,20.,2*np.pi,30. , 20] ## the size of suitable ranage
#ranges = [100.,40.,2*np.pi,40. , 20]


def test_force_start(f=0,n=30, all=False):
    system = CartPole(visual=False)
    system.setState([0, 0, np.pi, 0]) ## initialise
    state_history = np.empty((n + 1, 4,n))
    for i in range(n):
        state_history[0, :,i] = [0, 0, np.pi,0]
    time = np.arange(0, (n + 0.5) * 0.1, 0.1)
    for fn in range(n):
        system.setState([0, 0, np.pi, 0])
        for i in range(n): # update dynamics n times
            if i < fn+1:
                system.performAction(f) ## runs for 0.1 secs with no
            else:
                system.performAction(0)
            system.remap_angle()
            state_history[i + 1, :,fn] = system.getState()#
        if fn%3==0:
            plt.plot(time, state_history[:, 2,fn], Label=str(fn+1) + '␣Steps')
            if all:
                plt.plot(time, state_history[:, 0, fn], Label=str(fn + 1) + '␣Steps-
                    ↪ x')
                plt.plot(time, state_history[:, 1, fn], Label=str(fn + 1) + '␣Steps-
                    ↪ v')
```

```
            plt.plot(time, state_history[:, 3, fn], Label=str(fn + 1) + r'␣
                ↪ Steps-$\dot{\theta}$')
    plt.xlabel('Time')
    plt.ylabel(r'$\theta$')
    plt.legend(loc='lower␣left')
    plt.title('Constant␣force␣effect␣for␣start')
    plt.show()


def rollout (initial_state, n = 100, visual =False,f=0,fn=0):
    ## takes an intitial state array for the initial cart and angular velocity,
    ## itialises at stable equilibrium theta = pi , x =0
    ## runs for n iterations of euler dynamics each iteration is 0.1 seconds
    ## no applied force
    ## plots state variables

    system = CartPole(visual)
    system.setState([0, initial_state[0], np.pi, initial_state[1]]) ## initialise
    state_history= np.empty((n+1,4))
    state_history[0,:] = [0, initial_state[0], np.pi, initial_state[1]]
    for i in range(n): #update dynamics n times
        if i<fn:
            system.performAction(f) ## runs for 0.1 secs with no
        else: system.performAction(0)
        system.remap_angle()
        state_history[i+1,:] = system.getState()
    time=np.arange(0,(n+0.5)*0.1,0.1)


    plt.plot(time,state_history[:,0], Label = 'Position')
    plt.plot(time,state_history[:,1], Label = 'Velocity')
    plt.plot(time,state_history[:,2], Label = 'Angular␣Position')
    plt.plot(time,state_history[:,3], Label = 'Angular␣Velocity')
    plt.xlabel('Time')
    plt.legend(loc='upper␣right')
    plt.title('Initial␣Velocity␣:␣'+ str(initial_state[0])+ '␣,␣Initial␣Angular␣
        ↪ velocity␣:␣'+ str(initial_state[1]))
    plt.show()

    # plt.plot(state_history[:,0], state_history[:, 1])
    # plt.xlabel('Position')
    # plt.ylabel('Velocity')
    # plt.show()

def quasi_init(seed,f=False):
    v =4
    if f:
        v=5
```

```
    x, seed = sobol_seq.i4_sobol(v, seed)
    for i in range(v):
        x[i] = x[i]* ranges[i] - ranges[i]/2 # produces rando float in the ranges
            ↪ band described above
    return x,seed

def random_init(x,stable_equ=False,ext =False,f=False):
    # generates a random start state in the correct range
    v=4
    if f:
        v=5
    for i in range(v):
        x[i] = random.random() * ranges[i] - ranges[i]/2 # produces rando float in
            ↪  the ranges band described above
    if stable_equ==True:
        x[2] = random.random() * 0.2
        if x[2] > 0.1:
            x[2]=np.pi-x[2]
        else: x[2] = -np.pi+x[2]
    elif ext == True: ## get 2 times the range
            x[0] = x[0]*2
            x[1] = x[1]*2
            x[3] = x[3]*2
    return x

def scan_step(to_scan,n,visual=False,f=False):
    # scnes through one state variable with n points scan variable given by
        ↪ to_scan = [0,3]
    # plots the next state vector


    system = CartPole(visual)
    initial_state = np.zeros(4)
    initial_state = random_init(initial_state,f=f)

    initial_state[to_scan] = 0 - ranges[to_scan]/2 ##makes it scan from the start
        ↪ of a variables range
    scanned_values = np.zeros(n)
    reached_states = np.zeros((n,4))
    step = ranges[to_scan]/n
    for i in range(n): ##number of scanning variables
        x = initial_state.copy() ##takes a copy of this state
        x[to_scan] = x[to_scan]+ i * step ##changes one state variable
        system.setState(x)
        if to_scan == 2:
            system.remap_angle() ##remaps the angle
            x[to_scan] = remap_angle(x[to_scan])
        scanned_values[i] = x[to_scan]
```

```
        system.performAction(x[4])
        reached_states[i,:] = system.getState()

    plt.plot(scanned_values, reached_states[:, 0], Label='Position')
    plt.plot(scanned_values, reached_states[:, 1], Label='Velocity')
    plt.plot(scanned_values, reached_states[:, 2], Label='Angle')
    plt.plot(scanned_values, reached_states[:, 3], Label='Angular␣Velocity')


    plt.xlabel(labels[to_scan])
    plt.ylabel('Reached␣state')
    plt.legend(loc='upper␣left')
    plt.title(
            'Initial␣state:␣' + str(np.round(initial_state,2)) + '␣Scan␣through␣' +
                ↪  labels[to_scan] )
    plt.show()

def scan_all(n,type=0,visual=False, c =None,f=False,nlm=False): ##type 0 is a
    ↪ regular scan, type 1 is a change in variable scan
    ## c iks a 4x4 linear coefficent matrix for plotting model scans
    # scans through all state variables and plots either the next state or the
        ↪ change in state

    system = CartPole(visual)
    if f:
        initial_state = np.zeros(5)
    else:
        initial_state = np.zeros(4)
    initial_state = random_init(initial_state,f=f)
    line_lables=[None,None,None,None,None,None,None,None]
    fig = plt.figure()
    for to_scan in range(4):
        plot_no=to_scan
        scanned_state = np.zeros((4,n))
        if nlm:
            if c.v==5:
                to_scan+=1
                scanned_state = np.zeros((5, n))
        initial_state[to_scan] = 0 - ranges[to_scan]/2 ##makes it scan from the
            ↪ start of a variables range
        scanned_values = np.zeros(n)
        reached_states = np.zeros((n,4))

        Y = np.zeros((n,4))
        step = ranges[to_scan]/n
        for i in range(n): ##number of scanning variables
            x = initial_state.copy() ##takes a copy of this state
```

```
        x[to_scan] = x[to_scan]+ i * step ##changes one state variable
        system.setState(x)
        if to_scan == 2:
            system.remap_angle() ##remaps the angle
            x[to_scan] = remap_angle(x[to_scan])
        scanned_values[i] = x[to_scan]
        scanned_state[:,i] = x

        if f:
            system.performAction(x[4])
        else:
            system.performAction(0)
        reached_states[i,:] = system.getState()
        Y[i,:] = reached_states[i,:] - x[:4]

    fig.add_subplot(2,2,plot_no+1)
    plt.subplots_adjust(hspace=0.3)
    if plot_no == 3: #add labels
        line_lables=['x','v',r"$\theta$",r"$\dot{\theta}$",'Predicted␣x','
            ↪ Predicted␣v',r"Predicted␣$\theta$",r"␣Predicted␣$\dot{\theta}$"]

    if type == 0 : #plot regular scan
        plt.plot(scanned_values, reached_states[:, 0], Label=line_lables[0])
        plt.plot(scanned_values, reached_states[:, 1], Label=line_lables[1])
        plt.plot(scanned_values, reached_states[:, 2], Label=line_lables[2])
        plt.plot(scanned_values, reached_states[:, 3], Label=line_lables[3])
        plt.ylabel('Reached␣state')

    elif type ==1: #plot change
        if np.any(c)!=None:
            if nlm:
                scanned_state=c.transform_x(scanned_state)
                predictions = np.matmul(c.alpha.T,scanned_state)
            else:
                predictions = np.matmul(c,scanned_state)

            plt.plot(scanned_values[:], predictions[0, :],linestyle=':',color=
                ↪ 'b', Label=line_lables[4])
            plt.plot(scanned_values[:], predictions[1, :],linestyle=':',color=
                ↪ 'orange', Label=line_lables[5])
            plt.plot(scanned_values[:], predictions[2, :],linestyle=':',color=
                ↪ 'g', Label=line_lables[6])
            plt.plot(scanned_values[:], predictions[3, :],linestyle=':',color=
                ↪ 'r', Label=line_lables[7])

        plt.plot(scanned_values[:], Y[:, 0],color= 'b', Label=line_lables[0])
        plt.plot(scanned_values[:], Y[:, 1],color= 'orange', Label=line_lables
            ↪ [1])
```

```
            plt.plot(scanned_values[:], Y[:, 2],color= 'g', Label=line_lables[2])
            plt.plot(scanned_values[:], Y[:, 3],color= 'r', Label=line_lables[3])
            plt.ylabel('Change in State Variable')


        labels = [ "Position" , "Velocity" , "Angle" , "Angular Velocity", "Force"
            ↪ ]
        plt.xlabel(labels[to_scan])
        #plt.legend(loc='upper left')
        plt.title(
                'Initial state: ' + str(np.round(initial_state,2)) + ' Scan through
                    ↪  ' + labels[to_scan] )
    fig.legend(loc='lower center', ncol=4)
    plt.show()


def contour_plot(n,x_var, y_var , cont, visual = False , model = None,error=False,
    ↪ f=False):
    # plots a 3d surface plot and contoru plot where x_var , Y_var are scanned
        ↪ through the suitable range
    # cont is the variable to be plotted on the contours
    #n is the number of points to evaluate at in the range
    # x_var,y_var and cont are either 0,1,2,3
    v=4
    if f:
        v=5
    system = CartPole(visual)
    initial_state = np.zeros(v)
    initial_state = random_init(initial_state,f=f)
    force=0
    if f:
        force = initial_state[4]

    scanned_x = np.arange( - ranges[x_var]/2, ranges[x_var]/2, ranges[x_var]/n ) #
        ↪ #scanning variable 1yield
    scanned_y = np.arange( - ranges[y_var]/2, ranges[y_var]/2, ranges[y_var]/n ) #
        ↪ #scanning variable 1yield

    X, Y = np.meshgrid(scanned_x, scanned_y)

    rav_X=np.ravel(X)
    rav_Y=np.ravel(Y)
    rav_Z=np.zeros(rav_X.shape[0])
    for i in range(rav_X.shape[0]):
        x = initial_state.copy()
        x[x_var] = rav_X[i]
        x[y_var] = rav_Y[i]
```

```
        system.setState(x)

        system.performAction(force)
        rav_Z[i] = system.getState()[cont]

        if error ==True:
            change = np.zeros((1,v))
            change[0,0:4] = np.matmul(model.alpha.T, model.transform_x(x)).T

            next_state = x + change

            rav_Z[i] = abs(next_state[0,cont] - rav_Z[i])

    Z = rav_Z.reshape(X.shape)

    colour = 'inferno'
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1 , cmap=colour)
    # Add a color bar which maps values to colors.
    cbar =fig.colorbar(surf, shrink=0.5, aspect=5)
    cbar.set_label(labels[cont])
    plt.xlabel(labels[x_var])
    plt.ylabel(labels[y_var])
    ax.set_zlabel(labels[cont])
    plt.title('Initial␣state:␣' + str(np.round(initial_state,2)) + '\n␣Scan␣
        ↪ through␣' + labels[x_var]+ '␣and␣' + labels[y_var] )

    if error == True:
        ax.set_zlabel('Error␣in␣' + labels[cont])
        plt.scatter(model.basis[x_var,:],model.basis[y_var,:],np.ones(model.basis[
            ↪ x_var,:].shape[0])*0.3)
    else:
        cont_plot = plt.tricontourf(rav_X, rav_Y, rav_Z, levels=14, cmap=colour,
            ↪ offset = np.amin(rav_Z))

    plt.show()

    cont_plot = plt.tricontourf(rav_X,rav_Y,rav_Z,levels=14, cmap=colour)
    cbar = plt.colorbar(cont_plot,shrink=0.5, aspect=5)
    cbar.set_label(labels[cont])
    plt.xlabel(labels[x_var])
    plt.ylabel(labels[y_var])
    plt.title(
        'Initial␣state:␣' + str(np.round(initial_state, 2)) + '\n␣Scan␣through␣' +
            ↪  labels[x_var] + '␣and␣' + labels[
            y_var])
```

```
    plt.show()
```

ii. Linear_Modelling.py

```
import numpy as np
from CartPole import *
import matplotlib.pyplot as plt
import random
from scipy import stats
from Task_1_Functions import *
#from Non_linear_modelling import *
import sobol_seq



def get_xy_pair(n,system,force=0,f=False,noise=None): ## returns the nth
    ↪ iternation and the difference bween nth and n+1th for an intialised system
    # x and y both 4x1 matrices
    v=4
    if f:
        v=5

    for i in range(n):
        system.performAction(force) ## runs for 0.1 secs with no force
    x= np.zeros(v)
    x[0:4] = system.getState()
    if f:
        x[4] = force
    system.performAction(force)
    #system.remap_angle()
    state=system.getState()
    if noise!= None:
        for i in range(4):
            state[i] += random.normalvariate(0,noise)
    y= state- x[0:4]
    return x[0:v],y

def get_random_data_pairs(n,visual=False,stable_equ=False,ext=False, quasi =
    ↪ False, f = False,noise=None):
    # generates n random state initialisations and their change in state
    # stable_equ if generation is only about theta = pi
    # returns 4xn matrices xs and ys for the initial state and change in state
        ↪ respectivvly
    system = CartPole(visual)

    # initial_state = random_init(initial_state,stable_equ)
    # system.setState(initial_state)
```

```
    v=4
    force=0
    if f:
        v=5
    initial_state = np.zeros(v)
    xs = np.zeros((v,n))
    ys = np.zeros((4,n))
    seed=1
    for i in range(n):
        if quasi == False:
            initial_state = random_init(initial_state,stable_equ,ext,f=f)
        else:
            initial_state, seed = quasi_init(seed,f=f)

        system.setState(initial_state[0:4])
        if f:
            force=initial_state[4]
        xs[:,i],ys[:,i] = get_xy_pair(0,system,f=f,force=force,noise=noise) ##
            ↪ each column is a sample
    return xs,ys


def linear_regression(x,y, f=False):
    #x is (vxn) matrix of states, y is (4xn) matrix of state changes
    # minimise Y - CX
    # returns the 4xv coefficent matrix c
    # plots state predictions for y against true y
    c = np.linalg.lstsq(x.T,y.T,rcond=None)[0]
    c=c.T
    predicted_y = np.matmul(c,x)
    v=4
    if f:
        v=5

    for i in range(4):
        plt.subplot(2, 2, i + 1)
        plt.subplots_adjust(hspace=0.3)
        plt.scatter(y[i,:],predicted_y[i,:], marker = 'x')
        plt.ylabel('Predicted␣next␣step')
        plt.xlabel('Actual␣next␣step')
        plt.title (labels[i])

        grad,int,r,p,se = stats.linregress(y[i,:],predicted_y[i,:])
        print(labels[i] , '␣gradient␣=␣', grad)
        print(labels[i], '␣inter␣=␣', int)
        print(labels[i], '␣r␣=␣', r)
        print(labels[i], '␣r^2␣=␣', r**2)
    plt.show()
```

```
    return c
```

iii. Non_Linear_Modelling.py

```
import numpy as np
from CartPole import *
import matplotlib.pyplot as plt
import random
from scipy import stats
from Task_1_Functions import *
from Linear_Modeling import *
import pickle

class non_linear_model:
    def __init__(self,m,f=False,policy=False):
        self.type=policy

        if policy==False:
            self.alpha = np.zeros((m,4))
            self.v = 4
            if f:
                self.v = 5
            self.sd = np.zeros((self.v))
        else:
            self.p = np.zeros((m))
            self.v=4
            self.sd=np.zeros((self.v,m))

        self.basis = np.zeros((self.v, m))
        self.m = m

    def transform_x(self,x):
        # if x.ndim == 1:
        # x=x.reshape((x.shape[0],1)) ## reshapes it to a column vector

        #return kernel_matrix(x, self.m, self.sd, self.basis)[1]
        if x.ndim == 1:
            n = 1
        else: n = x.shape[1]
        Km = np.zeros((self.m,n))
        for i in range(self.m):
            for j in range(n):
                if x.ndim == 1:
                    input=x
                else:
```

```
                input = x[:,j]
            if self.type==False:
                kernel = gaussian_kernel(input, self.basis[:, i], self.sd)
            else:
                kernel = gaussian_kernel(input, self.basis[:, i], self.sd[:,i])
            Km[i,j] = kernel

        if Km.ndim == 1:
            Km=Km.reshape((x.shape[0],1)) ## reshapes it to a column vector
        return Km

    def pp(self):
        print(self.m)
        print(self.basis)
        print(self.sd)
        print(self.v)
        if self.type==False:
            print(self.alpha)
        else:
            print(self.p)
    def write_to_file(self,filename):
        with open(filename, 'wb') as f:
            if self.type == False:
                pickle.dump([self.m,self.v,self.alpha,self.basis,self.sd],f)
            else:
                pickle.dump([self.m, self.v, self.p, self.basis, self.sd], f)

    def read_from_file(self,filename):
        with open(filename,'rb') as f:
            if self.type== False:
                self.m,self.v,self.alpha,self.basis,self.sd = pickle.load(f)
            else:
                self.m, self.v, self.p, self.basis, self.sd = pickle.load(f)



def gaussian_kernel(x,x_basis,sd):
    ## takes a single input x and evaluates the kernel function K(X,Xi) for one
        ↪ basis location

    k = (x - x_basis)
    kernel = np.e** ( - 0.5*(np.sin(k[2]/2)/sd[2])**2)
    for i in range(x.shape[0]):
        if i !=2: #not angle
            kernel *= np.e** ( -0.5*(k[i] / sd[i])**2)

    return kernel
```

```
def kernel_matrix(x,m,sd,original_basis=None):
    ## builds the kernal matrices Kmm and Kmn
    # each row is a basis location, each col is a data location
    # original basis is for transforming new unseen test data'
    n=x.shape[1]

    Kmm = np.zeros((m,m))
    Kmn = np.zeros ((m,n))
    for j in range(m): ## each row
        for i in range(n): ## each col
            if np.all(original_basis) == None:
                kernel = gaussian_kernel(x[:,i],x[:,j],sd)
            else:
                kernel = gaussian_kernel(x[:,i],original_basis[:,j],sd)

            Kmn[j,i]=kernel
            if i< m:
                Kmm[j,i] = kernel
    return Kmm, Kmn

def get_basis(x,m):
    X = x.T
    #np.random.shuffle(X)## shuffle the inputs
    x=X.T
    return x[:,0:m]

def non_linear_regression(x,y,m,sd,reg,f=False):
    # x is 4xn original points
    # y is 4xn change of state
    # m is number of basis locations
    # sd is 4x1 array of standard deviations for each state variable
    # reg is the regularisation strength between 1e-6 1e-1

    # function returns mx1 array of alpha
    v=4
    if f:
        v=5
    n = x.shape[1]
    if m > n:
        m = n
    model = non_linear_model(m,f=f)

    model.basis = get_basis(x,m)
    model.sd = sd

    Y = y.T

    Kmm,Kmn = kernel_matrix(x,m,sd,model.basis)
```

```
    ## Ca = b

    C = np.matmul(Kmn,Kmn.T)+reg*Kmm ## mxm
    b = np.matmul(Kmn,Y) ## mx4

    model.alpha = np.linalg.lstsq(C, b, rcond=None)[0]

    transformed_x = Kmn # make a mxn transformed input
    #transformed_x =model.transform_x(x)

    predicted_y = np.matmul(model.alpha.T,transformed_x) ## this is a nx4 matrix
    # plt.plot(predicted_y[1,:])
    # plt.show()
    for i in range(4):
        plt.subplot(2, 2, i + 1)
        plt.subplots_adjust(hspace=0.3)
        plt.scatter(y[i, :], predicted_y[i, :], marker='x')
        plt.ylabel('Predicted␣next␣step')
        plt.xlabel('Actual␣next␣step')
        plt.title(labels[i])

        grad, int, r, p, se = stats.linregress(y[i, :], predicted_y[i, :])
        print(labels[i], '␣gradient␣=␣', grad)
        print(labels[i], '␣inter␣=␣', int)
        print(labels[i], '␣r␣=␣', r)
        print(labels[i], '␣r^2␣=␣', r ** 2)
    plt.show()

    return model


def test_regularisation(n, m ,p , t):
    # n training points
    # m basis centeres
    # p evaluation points for regurlasiation
    # t x,y pairs to test on extended extended

    model = non_linear_model(m)

    x, y = get_random_data_pairs(n,quasi=True)
    sd = np.std(x, axis=1)

    model.basis = get_basis(x,m)
    model.sd = sd

    lambdas = np.logspace(-6,-1,p)
    error = np.zeros((4,p))
    Y = y.T
```

```
    Kmm, Kmn = kernel_matrix(x, m, sd,model.basis)
    b = np.matmul(Kmn, Y) ## mx4
    x_test, y_test = get_random_data_pairs(t) ## extended the search space can
        ↪ cause most transformed x's to be 0 because of the exponential kernel,
        ↪ and only one of the four variabkes needing to be out

    transformed_x = kernel_matrix(x_test,m,sd,model.basis)[1] #get Kmn for this
        ↪ data

    for i in range(p):
    ## Ca = b
        C = np.matmul(Kmn, Kmn.T) + lambdas[i] * Kmm ## mxm

        model.alpha = np.linalg.lstsq(C, b, rcond=None)[0]

        predicted_y = np.matmul(model.alpha.T, transformed_x)
        # for j in range(4):
        # plt.subplot(2, 2, j + 1)
        # plt.subplots_adjust(hspace=0.3)
        # plt.scatter(y_test[j, :], predicted_y[j, :], marker='x')
        # plt.ylabel('Predicted next step')
        # plt.xlabel('Actual next step')
        # plt.title(labels[j])
        # plt.show()
        e = y_test - predicted_y
        error[:,i] = np.average(np.absolute(e) , axis=1)

    for i in range(4):
        plt.subplot(2 ,2 ,i+1)
        plt.subplots_adjust(hspace=0.3)
        plt.semilogx(lambdas,error[i,:], label = labels[i])
        plt.xlabel(r'Regularisation␣strength␣$\lambda$')
        plt.ylabel('Absolute␣error␣in' + labels[i])
    plt.title(r'Effect␣of␣$\lambda$␣with␣n=␣' + str(n) + ',␣m=' +str(m)+'␣and␣'+
        ↪ str(t)+'␣test␣points')
    plt.show()


def test_regularisations(n,p , t):
    # n training points
    # p evaluation points for regurlasiation
    # t x,y pairs to test on extended extended

    ms = [100,500,1000]
    mno=3
    x, y = get_random_data_pairs(n, quasi=True)
    sd = np.std(x, axis=1)
    lambdas = np.logspace(-6, -1, p)
```

```
error = np.zeros((4, p,mno))
x_test, y_test = get_random_data_pairs(t)

for j in range(mno):
    m=ms[j]
    model = non_linear_model(m)
    model.basis = get_basis(x,m)
    model.sd = sd
    Y = y.T
    Kmm, Kmn = kernel_matrix(x, m, sd,model.basis)
    b = np.matmul(Kmn, Y) ## mx4
    transformed_x = kernel_matrix(x_test,m,sd,model.basis)[1] #get Kmn for
        ↪ this data

    for i in range(p):
    ## Ca = b
        C = np.matmul(Kmn, Kmn.T) + lambdas[i] * Kmm ## mxm

        model.alpha = np.linalg.lstsq(C, b, rcond=None)[0]

        predicted_y = np.matmul(model.alpha.T, transformed_x)
        e = y_test - predicted_y
        error[:,i,j] = np.average(np.absolute(e) , axis=1)

fig = plt.figure()
labs = [None,None,None]
for i in range(4):
    if i==3:
        labs = ms
    ax1 = fig.add_subplot(2, 2, i + 1)
    ax2 = ax1.twinx()
    plt.subplots_adjust(hspace=0.3,wspace=0.4)
    for j in range(mno):
        if j==2:
            ax2.semilogx(lambdas, error[i, :, j], label='m=' + str(labs[j]),
                ↪ color = 'G')
        elif j==1:
            ax2.semilogx(lambdas, error[i, :, j], label='m=' + str(labs[j]),
                ↪ color='B')
        else:
            ax1.semilogx(lambdas,error[i,:,j], label = 'm='+ str(labs[j]),color
                ↪ ='R')
    ax1.set_xlabel(r'Regularisation␣strength␣$\lambda$')
    ax2.set_ylabel('Absolute␣error␣in' + labels[i])
    ax1.set_ylabel('Absolute␣error␣in' + labels[i] + 'for␣m=100')
    #ax1.ticklabel_format(axis='y',style='sci')
    #ax1.set_ylim(top=error[i,25,0])
    ax2.set_ylim(top=error[i, 25, 1])
```

```
        #ax2.ticklabel_format(axis='y', style='sci')
    fig.suptitle(r'Effect␣of␣$\lambda$␣with␣n=␣' + str(n) + ',␣m=' +str(ms) + '␣
        ↪ and␣'+str(t)+'␣test␣points')
    h1, l1 = ax1.get_legend_handles_labels()
    h2, l2 = ax2.get_legend_handles_labels()
    fig.legend(h1+h2, l1+l2,loc='lower␣center', ncol=3)
    plt.show()

def test_length_scales(n ,p ,t ):
    # n training points

    # p evaluation points for length scale
    # t x,y pairs to test on extended extended
    ms = [100,500,1000]
    mno=3
    regs= [3e-2,4e-4,2e-4]
    x, y = get_random_data_pairs(n, quasi=True)
    sd = np.std(x, axis=1) ## the middle sd to test
    sd_multip = np.linspace(0.01, 4, p)
    error = np.zeros((4, p,mno))
    x_test, y_test = get_random_data_pairs(t)
    for j in range(mno):
        m=ms[j]
        reg = regs[j]
        model = non_linear_model(m)
        sds= np.zeros((4,p))
        for i in range(4):
            sds[i,:] = sd_multip * sd[i]
        model.basis = get_basis(x,m)
        model.sd = sd
        Y = y.T
        for i in range(p):
        ## Ca = b
            Kmm, Kmn = kernel_matrix(x, m, sds[:,i])
            b = np.matmul(Kmn, Y) ## mx4
            transformed_x = kernel_matrix(x_test, m, sds[:,i], model.basis)[1] #
                ↪ get Kmn for this data

            C = np.matmul(Kmn, Kmn.T) + reg * Kmm ## mxm

            model.alpha = np.linalg.lstsq(C, b, rcond=None)[0]

            predicted_y = np.matmul(model.alpha.T, transformed_x)
            e = y_test - predicted_y
            error[:,i,j] = np.average(np.absolute(e) , axis=1)

    fig = plt.figure()
    labs = [None, None, None]
```

```
    for i in range(4):
        if i == 3:
            labs = ms
        ax1 = fig.add_subplot(2, 2, i + 1)
        ax2 = ax1.twinx()
        plt.subplots_adjust(hspace=0.3, wspace=0.4)
        for j in range(mno):
            if j == 2:
                ax2.plot(sd_multip, error[i, :, j], label='m=' + str(labs[j]),
                    ↪ color='G')
            elif j == 1:
                ax2.plot(sd_multip, error[i, :, j], label='m=' + str(labs[j]),
                    ↪ color='B')
            else:
                ax1.plot(sd_multip, error[i, :, j], label='m=' + str(labs[j]),
                    ↪ color='R')
        ax1.set_xlabel(r'Length␣Scale␣multiplier')
        ax2.set_ylabel('Absolute␣error␣in' + labels[i])
        ax1.set_ylabel('Absolute␣error␣in' + labels[i] + 'for␣m=100')
        # ax1.ticklabel_format(axis='y',style='sci')
        # ax1.set_ylim(top=error[i,25,0])
        #ax2.set_ylim(top=error[i, 25, 1])
        # ax2.ticklabel_format(axis='y', style='sci')
    fig.suptitle(r'Effect␣of␣$\sigma$␣with␣' + str(n) + ',␣m=' + str(ms) + '␣and␣'
        ↪  + str(t) + '␣test␣points')
    h1, l1 = ax1.get_legend_handles_labels()
    h2, l2 = ax2.get_legend_handles_labels()
    fig.legend(h1 + h2, l1 + l2, loc='lower␣center', ncol=3)
    plt.show()

def test_length_scale(n, m ,p ,t , reg):
    # n training points
    # m basis centeres
    # p evaluation points for length scale
    # t x,y pairs to test on extended extended
    # reg is the regularisation

    model = non_linear_model(m)

    x, y = get_random_data_pairs(n,quasi=True)
    sd = np.std(x, axis=1) ## the middle sd to test
    sd_multip = np.linspace(0.01,4,p)

    sds= np.zeros((4,p))
    for i in range(4):
        sds[i,:] = sd_multip * sd[i]

    model.basis = get_basis(x,m)
```

```
    model.sd = sd

    error = np.zeros((4,p))
    Y = y.T

    x_test, y_test = get_random_data_pairs(t) ## extended the search space can
        ↪ cause most transformed x's to be 0 because of the exponential kernel,
        ↪ and only one of the four variabkes needing to be out



    for i in range(p):
    ## Ca = b
        Kmm, Kmn = kernel_matrix(x, m, sds[:,i])
        b = np.matmul(Kmn, Y) ## mx4
        transformed_x = kernel_matrix(x_test, m, sds[:,i], model.basis)[1] # get
            ↪ Kmn for this data

        C = np.matmul(Kmn, Kmn.T) + reg * Kmm ## mxm

        model.alpha = np.linalg.lstsq(C, b, rcond=None)[0]

        predicted_y = np.matmul(model.alpha.T, transformed_x)
        # for j in range(4):
        # plt.subplot(2, 2, j + 1)
        # plt.subplots_adjust(hspace=0.3)
        # plt.scatter(y_test[j, :], predicted_y[j, :], marker='x')
        # plt.ylabel('Predicted next step')
        # plt.xlabel('Actual next step')
        # plt.title(labels[j])
        # plt.show()
        e = y_test - predicted_y
        error[:,i] = np.average(np.absolute(e) , axis=1)
    for i in range(4):
        plt.subplot(2 ,2 ,i+1)
        plt.subplots_adjust(hspace=0.3)
        plt.plot(sd_multip,error[i,:], label = labels[i])
        plt.xlabel(r'Length␣Scale␣multiplier')
        plt.ylabel('Absolute␣error␣in' + labels[i])
    plt.title(r'Effect␣of␣$\sigma$␣with␣n=␣' + str(n) + ',␣m=' +str(m)+'␣and␣'+str
        ↪ (t)+'␣test␣points')
    plt.show()


def test_mn(t ):
    # n training points

    # p evaluation points for length scale
```

```
# t x,y pairs to test on extended extended
p=9 #9
n_test=4096
m_test = 256
reg = 4e-4
error = np.zeros((4, p,2))
#ns = np.linspace(100, 3000, p)
ns=np.logspace(3,3+p,num=p,base=2,dtype='int16')
#ms = np.linspace(10, 2000, p)
ms = np.logspace(3, 3 + p,num=p,base=2,dtype='int16')
x_test, y_test = get_random_data_pairs(t)
#get n data
x, y = get_random_data_pairs(4096, quasi=True)
for i in range(p):
    #print(ns[i])
    x_train = x[:,:ns[i]]
    sd = np.std(x_train, axis=1) ## the middle sd to test
    if ns[i]>=m_test:
        m=m_test
    else:
        m=ns[i]
    model = non_linear_model(m)
    model.basis = get_basis(x_train,m)
    model.sd = sd
    Y = y.T

    ## Ca = b
    Kmm, Kmn = kernel_matrix(x, m, model.sd)
    b = np.matmul(Kmn, Y) ## mx4
    transformed_x = kernel_matrix(x_test, m, model.sd, model.basis)[1] # get
        ↪ Kmn for this data

    C = np.matmul(Kmn, Kmn.T) + reg * Kmm ## mxm

    model.alpha = np.linalg.lstsq(C, b, rcond=None)[0]

    predicted_y = np.matmul(model.alpha.T, transformed_x)
    e = y_test - predicted_y
    error[:,i,0] = np.average(np.absolute(e) , axis=1)

#get m data
for i in range(p):
    sd = np.std(x[:,:n_test], axis=1) ## the middle sd to test
    m=ms[i]
    model = non_linear_model(m)
    model.basis = get_basis(x[:,:n_test],m)
    model.sd = sd
    Y = y.T
```

```
        ## Ca = b
        Kmm, Kmn = kernel_matrix(x[:,:n_test], m, model.sd)
        b = np.matmul(Kmn, Y) ## mx4
        transformed_x = kernel_matrix(x_test, m, model.sd, model.basis)[1] # get
            ↪ Kmn for this data

        C = np.matmul(Kmn, Kmn.T) + reg * Kmm ## mxm

        model.alpha = np.linalg.lstsq(C, b, rcond=None)[0]

        predicted_y = np.matmul(model.alpha.T, transformed_x)
        e = y_test - predicted_y
        error[:,i,1] = np.average(np.absolute(e) , axis=1)

    fig = plt.figure()
    labs = [None, None]
    for i in range(4):
        if i == 3:
            labs = ['Vary n','Vary m']
        ax1 = fig.add_subplot(2, 2, i + 1)
        #ax2 = ax1.twinx()
        plt.subplots_adjust(hspace=0.3, wspace=0.4)
        for j in range(2):
            if j == 0:
                ax1.plot(np.log2(ns), error[i, :, j], label= str(labs[j]), color='G
                    ↪ ')
            elif j == 1:
                ax1.plot(np.log2(ms), error[i, :, j], label=str(labs[j]), color='B'
                    ↪ )
        ax1.set_xlabel(r'log(m) or log(n)')
        #ax2.set_ylabel('Absolute error in' + labels[i])
        ax1.set_ylabel('Absolute error in' + labels[i])
        # ax1.ticklabel_format(axis='y',style='sci')
        # ax1.set_ylim(top=error[i,25,0])
        #ax2.set_ylim(top=error[i, 25, 1])
        # ax2.ticklabel_format(axis='y', style='sci')
    fig.suptitle(r'Effect of varying m and n with n= ' + str(n_test) + ', m=' +
        ↪ str(m_test) + ' and $\lambda$= ' + str(reg))
    h1, l1 = ax1.get_legend_handles_labels()
    #h2, l2 = ax2.get_legend_handles_labels()
    fig.legend(h1,l1,loc='lower center', ncol=2)
    #fig.legend(h1 + h2, l1 + l2, loc='lower center', ncol=3)
    plt.show()

def test_model(c,n,visual=False,loop = False, osc = False, stable_equ=False,
    ↪ model = 0,f=False):
    # c is 4x4 linear coefficent matrix, or contais alpha and basis location for
```

```
    ↪ non linear model
# n is number of time iterations to predict
# stable_equ if motion is about theta = pi
# plots real state trajectory and linear model predictions
# model = 0 if linear model
#modoel = 1 if non linear model
v= 4
if f:
    v=5
system = CartPole(visual)
initial_state = np.zeros(v)
initial_state = random_init(initial_state,stable_equ,f=f)
initial_state[0] = 0
if loop == True:
    initial_state = np.array([0,0,np.pi,15])
elif osc == True:
    initial_state = np.array([0, 0, np.pi, 5])


force = 0
if f:
    force=initial_state[4]
system.setState(initial_state[0:4])

state_history = np.empty((v, n))
state_history[:, 0] = initial_state[0:v]
for i in range(n-1): # update dynamics n times
    system.performAction(force) ## runs for 0.1 secs with no force
    system.remap_angle()
    state_history[0:4,i + 1] = system.getState()
    if f:
        state_history[4]=force
#time = np.arange(0, (n ) * 0.1, 0.1)
prediction = state_prediction(c,n,initial_state,model,f=f)

for i in range(4):
    plt.subplot(2 ,2 ,i+1)
    plt.subplots_adjust(hspace=0.3)
    plt.plot( state_history[i,:],linestyle=':' ,label = 'Actual')
    plt.plot(prediction[i, :], label='Predicted')
    plt.xlabel('Iteration')
    plt.title(labels[i])
    plt.legend(loc='upper␣right')
    if i != 2:
        plt.ylim = (np.min(state_history[i,:])-0.5*np.min(state_history[i,:]),
            ↪ np.max(state_history[i,:])+0.5*np.max(state_history[i,:]) )

    # if i !=0 and i!=2:
```

```
        # plt.ylim(-20, 20)

    plt.show()

def state_prediction(c,n,initial,model=0,f=False):
    # c is 4x4 linear coefficent matrix, n is number of time interations, initial
        ↪ is 4x1 initial state
    # returns predicted trajectory from linear model
    v=4
    if f:
        v=5
    prediction = np.zeros((v,n))
    prediction[:,0] = initial
    for i in range(n-1):
        change = np.zeros((1,v))
        if model == 0: #linear model
            change[0,0:4]=np.matmul(c,prediction[:,i])
            prediction[:,i+1] = prediction[:,i] + change
        else:
            change[0, 0:4] = np.matmul(c.alpha.T, c.transform_x(prediction[:,i])).T
            prediction[:, i + 1] = prediction[:, i] + change
        theta = remap_angle(prediction[2,i+1])
        prediction[2,i+1] = theta
    return prediction


def test_noise(p ,t ,correct=False,f=False):
    # n training points
    # m basis centeres
    # p evaluation points for length scale
    # t x,y pairs to test on extended extended
    # reg is the regularisation

    m=1000
    n=1000
    reg = 2e-4
    model = non_linear_model(m)
    noise = np.linspace(0., 5., p)
    error = np.zeros((4,p))
    fig=plt.figure()
    x_test, y_test = get_random_data_pairs(t,f=f) #test on same points
    for i in range(p):
        print(i)
        x, y = get_random_data_pairs(n,quasi=True,noise=noise[i],f=f)
        sd = np.std(x, axis=1) ## the middle sd to test

        model.basis = get_basis(x,m)
        model.sd = sd
```

```
    Y = y.T

     ## extended the search space can cause most transformed x's to be 0
         ↪ because of the exponential kernel, and only one of the four
         ↪ variabkes needing to be out

## Ca = b
    Kmm, Kmn = kernel_matrix(x, m, model.sd)
    b = np.matmul(Kmn, Y) ## mx4
    transformed_x = kernel_matrix(x_test, m, model.sd, model.basis)[1] # get
        ↪ Kmn for this data

    C = np.matmul(Kmn, Kmn.T) + reg * Kmm ## mxm

    model.alpha = np.linalg.lstsq(C, b, rcond=None)[0]

    predicted_y = np.matmul(model.alpha.T, transformed_x)
    # for j in range(4):
    # plt.subplot(2, 2, j + 1)
    # plt.subplots_adjust(hspace=0.3)
    # plt.scatter(y_test[j, :], predicted_y[j, :], marker='x')
    # plt.ylabel('Predicted next step')
    # plt.xlabel('Actual next step')
    # plt.title(labels[j])
    # plt.show()
    e = y_test - predicted_y
    error[:,i] = np.average(np.absolute(e) , axis=1)
for i in range(4):
    plt.subplot(2 ,2 ,i+1)
    plt.subplots_adjust(hspace=0.3)
    plt.plot(noise,error[i,:], label = labels[i])
    plt.xlabel(r'Mearsurment␣noise␣$\sigma$')
    plt.ylabel('Absolute␣error␣in␣' + labels[i])
fig.suptitle(r'Effect␣of␣noise␣with␣n=␣' + str(n) + ',␣m=' +str(m)+'␣and␣'+str
    ↪ (t)+'␣test␣points')
plt.show()

with open('noisedata', 'wb') as f:
    pickle.dump([error, noise], f)
```

iv. Control.py

```
import autograd.numpy as np
from autograd import grad
from CartPole import *
```

```
import matplotlib.pyplot as plt
import random
from scipy import stats
from Task_1_Functions import *
from Linear_Modeling import *
from Non_linear_modelling import *
import random
from scipy import optimize


p_range = [30.,30.,30.,30.]
training_model = None #model



def loss_trajectory(initial,n,p,model=None,visual=False,nlp=None):
    loss = 0
    if model == None:
        system = CartPole(visual)
        system.setState(initial)
        loss += system.loss()
    else:
        loss += loss_pos(initial)
        x = initial[0:4]
        x_extended =np.zeros(5)

    for i in range(n):
        if model == None: #using model dynamics for initial scans
            x = system.getState()
            force = np.matmul(p,x)
            system.performAction(force)
            system.remap_angle()
            loss += system.loss()
        else: # using non linear modeelling for training
            if nlp==None:
                force = np.matmul(p, x)
            else:
                force = np.matmul(p,nlp.transform_x(x))
            x_extended[0:4] = x
            x_extended[4] = force
            change = np.matmul(model.alpha.T, model.transform_x(x_extended)).T
            x +=change
            x[2] = remap_angle(x[2])
            loss += loss_pos(x)
    return loss
```

```
def scan_policy(pos_1,pos_2=None,n=20,points=30, model=None,top=False,bot=False,
    ↪ nlp=None):

    x0= np.zeros(4)
    x0 = random_init(x0)
    x0[0]= 0

    if top==True:
        x0=np.array([random.normalvariate(0,0.03),random.normalvariate(0,0.03),
            ↪ random.normalvariate(0,0.03),random.normalvariate(0,0.03)])
    if bot:
        x0 = np.array([random.normalvariate(0, 0.03), random.normalvariate(0,
            ↪ 0.03), np.pi - random.normalvariate(0, 0.03),
                       random.normalvariate(0, 0.03)])

    p=np.zeros(4)

    # p[1]=5
    # p[3]=-8

    loop=False
    osc=False

    if loop == True:
        x0 = np.array([0,0,np.pi,15])
    elif osc == True:
        x0 = np.array([0, 0, np.pi, 5])

    scanned_x = np.arange(- p_range[pos_1] / 2, p_range[pos_1] / 2, p_range[pos_1]
        ↪  / points) ##scanning variable 1yield


    if pos_2 != None: ## 2d contour plots
        scanned_y = np.arange(- p_range[pos_2] / 2, p_range[pos_2] / 2, p_range[
            ↪ pos_2] / points) ##scanning variable 1yield
        X, Y = np.meshgrid(scanned_x, scanned_y)
        rav_X = np.ravel(X)
        rav_Y = np.ravel(Y)
        rav_Z = np.zeros(rav_X.shape[0])
        for i in range(rav_X.shape[0]):
            p[pos_1] = rav_X[i]
            p[pos_2] = rav_Y[i]

            rav_Z[i] = loss_trajectory(x0,n,p,model,nlp)

        Z = rav_Z.reshape(X.shape)

        colour = 'inferno'
```

```
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=colour,vmin=0,
            ↪ vmax=50)
        # Add a color bar which maps values to colors.
        cbar = fig.colorbar(surf, shrink=0.5, aspect=5)
        cbar.set_label('Loss')
        plt.xlabel('p[' + str(pos_1) +']')
        plt.ylabel('p[' + str(pos_2) +']')
        ax.set_zlabel('Loss')
        plt.title(
            'Initial␣state:␣' + str(np.round(x0, 2)) + '\n␣Scan␣through␣' + 'p[' +
                ↪ str(pos_1) +']␣and␣p[' + str(pos_2) +']')
        plt.show()

        cont_plot = plt.tricontourf(rav_X, rav_Y, rav_Z, levels=14, cmap=colour,
            ↪ vmin=0, vmax=50 )
        cbar = plt.colorbar(cont_plot, shrink=0.5, aspect=5)
        cbar.set_label('Loss')
        plt.xlabel('p[' + str(pos_1) +']')
        plt.ylabel('p[' + str(pos_2) +']')
        plt.title(
            'Initial␣state:␣' + str(np.round(x0, 2)) + '\n␣Scan␣through␣' + 'p[' +
                ↪ str(pos_1) + ']␣and␣p[' + str(
                pos_2) + ']')

        plt.show()

def loss_function(p):
    model = non_linear_model(10, f=True)
    #model.read_from_file('m=500n=10000')
    model.read_from_file('m=1000n=10000')

    loss = 0
    #initial =np.array([random.normalvariate(0,0.03),random.normalvariate(0,0.03),
        ↪ random.normalvariate(0,0.03),random.normalvariate(0,0.03)]) #+ random.
        ↪ normalvariate(0,0.01)
    initial=np.array([0,0.2,0.05,-0.2])#*10
    n = 20
    loss =loss + loss_pos(initial)
    x=np.zeros((4,1))
    x[:,0] = initial[0:4]
    x_extended = np.zeros((5))

    system = CartPole(visual=False)
    system.setState(initial)

    for i in range(n):
```

```
      # # using non linear modeelling for training
      change = np.zeros((1,4))
      if isinstance(p, (list, tuple, np.ndarray)):
         force = np.matmul(p,x)
      else:
         force = np.matmul(p._value, x)

      x_extended[0:4] = x[:,0]
      x_extended[4] = force
      test = np.matmul(model.alpha.T, model.transform_x(x_extended))
      x =x + test

      x[2] = remap_angle(x[2])
      loss += loss_pos(x)
   return loss



def compute_gradient(p):
   dp = 0.01

   gradient = np.zeros((p.shape))

   loss_original = loss_function(p)
   p = p + dp
   for i in range(p.shape[0]):
      p_new = p.copy()
      p_new[i] = p[i]+dp
      gradient[i] = (loss_function(p_new) - loss_original)/dp
   return gradient
def grad_decent(n,model,p):

   training_model = model
   #loss_function_grad = grad(loss_function)
   a=0.01
   m = 0.01
   #print(loss_function(p))
   loss_history = np.zeros((n))
   grad_old=np.zeros(p.shape)
   for i in range(n): #do n loops of gradient decent
      #gradient = loss_function_grad(p)
      gradient = compute_gradient(p)
      #print(gradient)
      #p = p - a* gradient
      grad_old = a*gradient
      p-= a* gradient +grad_old*m
      loss_history[i] = loss_function(p)
```

```
    plt.plot(loss_history)
    plt.show()

    print(p)
    print(loss_function(p))
    return p

def test_policy(p,n,visible=False,sd=0.03,bottom=False,nlm=None,Top=False):
    system = CartPole(visual=visible)
    #system.setState(np.array([0,0.01,-0.01,-0.01]))
    system.setState(np.array([random.normalvariate(0,sd),random.normalvariate(0,sd
        ↪ ),random.normalvariate(0,sd),random.normalvariate(0,sd)]))
    if nlm != None and Top==False: bottom =True
    if bottom:
        system.setState(np.array([0,0,np.pi,0]))
        #sd=1
        #system.setState(np.array([random.normalvariate(0, sd), random.
            ↪ normalvariate(0, sd), np.pi ,random.normalvariate(0, sd)]))
        #system.setState(np.array([-5.5,-3.36,-0.56,0.42]))

    state_history = np.empty((n,4))
    #system.setState(np.array([0, 0, 0,5.1]))
    #system.setState(np.array([0, 0.2, 0.05, -0.2])*5)
    # system.setState(np.array([0, 0.2, 1, -4]) ) #get it to 1 rad at -4rads^1 and
        ↪  itll settle
    for i in range(n):
        x = system.getState()
        if x.ndim == 1:
            state_history[i,:]=x
        else:
            state_history[i, :] = x.reshape((4))

        if nlm==None:
            force = np.matmul(p, x)
        else:
            x_ext = nlm.transform_x(x)
            force = np.matmul(p, x_ext)
            #state_history[i,2]=force

        system.performAction(force)
        system.remap_angle()

    plt.plot(state_history[:,2],label=r'$\theta$')
    plt.plot(state_history[:, 1],label='v')
    plt.plot(state_history[:, 3],label=r'$\dot{\theta}$')
    plt.plot(state_history[:, 0], label='x')
    plt.xlabel('Iteration')
    #plt.ylabel(r'$\theta$')
```

```
    plt.legend(loc='lower left')
    plt.title('P=' + str(p))
    #plt.ylim([-0.2,0.2])
    plt.show()


def non_linear_loss_function(p):
    model = non_linear_model(10, f=True)
    #model.read_from_file('m=500n=10000')
    model.read_from_file('m=1000n=10000')

    policy = non_linear_model(10,policy=True)
    #policy.read_from_file('nonzero') ### change at somepoint so that sd's are
        ↪ part of the p input?
    #policy.read_from_file('justtop')
    policy.read_from_file('wholetest')
   # policy.read_from_file('flickup')

    #different optmisation situations
    optimise_loc = False
    sym = False
    top_constant = False
    bot_constant = False
    bounce_back = True

    if optimise_loc == True:
        m = int((p.shape[0]-2)/4) #policy.m
        #policy = non_linear_model(m,policy=True)
        #policy.sd = np.ones((4,m))
        policy.basis[:, 0] = p[: 4].T
        policy.basis[:, 1] = -policy.basis[:, 0]
        policy.basis[:, 2] = p[4:4 + 4].T
        policy.basis[:, 3] = -policy.basis[:, 2]
        p = [-27.669,27.669,-10.98,10.98]
        policy.sd[0, :] = [30, 30, 30, 30]
        policy.sd[1, :] = [30, 30, 30, 30]

    if sym:
        policy.basis[:,0] = p[2:6].T
        policy.basis[:,1]=-policy.basis[:,0]
        policy.basis[:, 2] = p[6:10].T
        policy.basis[:, 3] = -policy.basis[:, 2]
        #print(policy.basis)
        p = [p[0],-p[0],p[1],-p[1]]

    if top_constant:
        p=np.array([-27.669,27.669,-10.98,10.98,p[0],p[1],0,p[2]]) #for updating 3
            ↪ on lhs
```

```
    if bot_constant:
        p = [p[0], -p[0], p[1], -p[1],76, -77,0, -37.025]

    if bounce_back:
        p=np.array([-27.669,27.669,-10.98,10.98,76,-77,p[0],-37.025,-p[0],p[1],-p
            ↪ [1]])


    loss = 0
    initial=np.array([0,0,np.pi,0])
    #initial = np.array([0, 0.2, 0.05, -0.2])*5
    #initial = np.array([-5.5,-3.36,-0.56,0.42])
    n = 30
    loss =loss + loss_pos(initial)
    x=np.zeros((4,1))
    x[:,0] = initial[0:4]
    x_extended = np.zeros((5))

    system = CartPole(visual=False)
    system.setState(initial)

    for i in range(n):
       # # using non linear modeelling for training
       change = np.zeros((1,4))
       x_policy = policy.transform_x(x)
       if isinstance(p, (list, tuple, np.ndarray)):
           force = np.matmul(p,x_policy)
       else:
           force = np.matmul(p._value, x_policy)

       x_extended[0:4] = x[:,0]
       x_extended[4] = force
       test = np.matmul(model.alpha.T, model.transform_x(x_extended))
       x =x + test
       x[2] = remap_angle(x[2])

       loss += loss_pos(x)
    return loss




def generate_non_linear_policy(m):


    m=11
```

```
nlp = non_linear_model(m,policy=True)

basis = np.zeros((4,m)) #no force nessary
sd = np.zeros((4,m))

#top section
# to slow down
basis[:, 0] = [0, 0, 0.25, 2.5] # this is the overall aim
#basis[:,0]=[-0.00988998, -0.03803768, 0.57465364, 2.65458026]
sd[:, 0] = [30, 30, .25, 2.5] # small sd #change velocity sd to huge, and x to
    ↪ hughe
basis[:, 1] = [0, 0, -0.25, -2.5] # this is the overall aim
#basis[:,1]=[-0.00988998, -0.03803768, 0.57465364, 2.65458026]
sd[:, 1] = [30, 30, .25, 2.5] # small sd

# to push back
basis[:, 2] = [0, 0, 0.25, -2.5] # this is the overall aim
#basis[:,2]=[0.00525632, 0.09308213, 0.88202638, -1.32182546]
sd[:, 2] = [30, 30, .25, 2.5] # small sd
basis[:, 3] = [0, 0, -0.25, 2.5] # this is the overall aim
sd[:, 3] = [30, 30, .25, 2.5] # small sd
#basis[:,3]=[0.00525632, 0.09308213, 0.88202638, -1.32182546]


#bottom , try and increase oscilation height
basis[:, 4] = [0, 0, np.pi - 0.3, -1.5] # in start position still # generates
    ↪ the initial force
sd[:, 4] = [7., 7., .5, 5.]

basis[:, 5] = [0, 0, -np.pi + 0.3, 1.5] # in start position still # generates
    ↪ the initial force
sd[:, 5] = [7., 7., .5, 5.]


#push it back (from -ve x)
basis[:, 6] = [-6, 0.25, 0, 0] # in start position still # generates the
    ↪ initial force
sd[:, 6] = [.2, .5, 0.2, 3.] # small angle sd as its like an impulse


#2/3 round
basis[:, 7] = [0, -1, 1.2, -10]
sd[:, 7] = [5., 5., 0.2, 8]

# push it back (from +ve x)
basis[:, 8] = [6, -0.25, 0, 0] # in start position still # generates the
    ↪ initial force
```

```
    sd[:, 8] = [.2, .5, 0.2, 3.] # small angle sd as its like an impulse

    # slow doum (from +ve x)
    basis[:, 9] = [-0.5, -5, 0, 0] # in start position still # generates the
        ↪ initial force
    sd[:, 9] = [1., 3, 0.2, 3.] # small angle sd as its like an impulse

    # slow dow (from -ve x)
    basis[:, 10] = [0.5, 5, 0, 0] # in start position still # generates the
        ↪ initial force
    sd[:, 10] = [1., 3, 0.2, 3.] # small angle sd as its like an impulse

    nlp.basis = basis#5xm
    #
    nlp.sd = sd#5xm

    nlp.write_to_file('wholetest')
    #nlp.write_to_file('flickup')

    plot_basis_locations(basis,sd)

    return nlp


def plot_basis_locations(basis,sd):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    plt.xlabel('v')
    plt.ylabel(r'$\dot{\theta}$')
    ax.set_zlabel(r'$\theta$')
    ax.set_zlim([-np.pi,np.pi])

    m = basis.shape[1]
    for i in range(m): # for each basis location
        phi = np.linspace(0, 2 * np.pi, 256).reshape(256, 1) # the angle of the
            ↪ projection in the xy-plane
        theta = np.linspace(0, np.pi, 256).reshape(-1, 256) # the angle from the
            ↪ polar axis, ie the polar angle

        # Transformation formulae for a spherical coordinate system.
        x = sd[1,i] * np.sin(theta) * np.cos(phi) - basis[1,i]
        y = sd[3,i] * np.sin(theta) * np.sin(phi) - basis[3,i]
        z = sd[2,i] * np.cos(theta) - basis[2,i]

        # fig = plt.figure(figsize=plt.figaspect(1)) # Square figure
        # ax = fig.add_subplot(111, projection='3d')
        if i!= 10 and i!= 9 and i!= 8 and i!= 6:
            ax.plot_surface(x, y, z,alpha=0.5)
```

```
    plt.xlabel(r'$\dot{x}$')
    ax.set_zlabel(r'$\theta$')
    ax.set_ylabel(r'$\dot{\theta}$')
    plt.title('First␣7␣basis␣locations␣and␣length␣scales')
    plt.show()
```

v. Cartpole.py

```
"""
fork from python-rl and pybrain for visualization
"""
import numpy as np
from pylab import ion, draw, Rectangle, Line2D
import pylab as plt

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):
        self.cart_location = 0.0
        self.cart_velocity = 0.0
        self.pole_angle = np.pi # angle is defined to be zero when the pole is
            ↪ upright, pi when hanging vertically down
        self.pole_velocity = 0.0
        self.visual = visual

        # 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 = 0.1 # 0.005 # friction coefficient of the cart
        self.mu_p = 0.0000 # 0.000002 # friction coefficient of the pole
```

```
        self.sim_steps = 200 # number of Euler steps to perform in one go
        self.delta_time = 0.1 # time step of the Euler integrator
        self.max_force = 10.
        self.gravity = 9.8
        self.cart_mass = 0.5

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

        if self.visual:
            self.drawPlot()

    def setState(self, state):
        self.cart_location = state[0]
        self.cart_velocity = state[1]
        self.pole_angle = state[2]
        self.pole_velocity = state[3]

    def getState(self):
        return np.array([self.cart_location,self.cart_velocity,self.pole_angle,
            ↪ self.pole_velocity])

    def reset(self):
        self.cart_location = 0.0
        self.cart_velocity = 0.0
        self.pole_angle = np.pi
        self.pole_velocity = 0.0


    def drawPlot(self):
        ion()
        self.fig = plt.figure()
        # draw cart
        self.axes = self.fig.add_subplot(111, aspect='equal')
        self.box = Rectangle(xy=(self.cart_location - self.cartwidth / 2.0, -self.
            ↪ cartheight),
                             width=self.cartwidth, height=self.cartheight)
        self.axes.add_artist(self.box)
        self.box.set_clip_box(self.axes.bbox)

        # draw pole
        self.pole = Line2D([self.cart_location, self.cart_location + np.sin(self.
            ↪ pole_angle)],
                           [0, np.cos(self.pole_angle)], linewidth=3, 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(-0.5, 2)


    def _render(self):
        self.box.set_x(self.cart_location - self.cartwidth / 2.0)
        self.pole.set_xdata([self.cart_location, self.cart_location + np.sin(self.
            ↪ pole_angle)])
        self.pole.set_ydata([0, np.cos(self.pole_angle)])
        draw()

        plt.pause(0.015)


    def performAction(self, action):
        force = self.max_force * np.tanh(action/self.max_force)

        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 = (-2.0*self.pole_length*self.pole_mass*(self.pole_velocity
                ↪ **2)*s+3.0*self.pole_mass*self.gravity*c*s+4.0*(force-self.mu_c*
                ↪ self.cart_velocity) )/m

            pole_accel = (-3.0*self.pole_length*self.pole_mass*(self.pole_velocity
                ↪ **2)*s*c + 6.0*(self.cart_mass+self.pole_mass)*self.gravity*s +
                ↪ 6.0*(force-self.mu_c*self.cart_velocity)*c)/(m*self.pole_length)

            # Update state variables
            df = (self.delta_time / float(self.sim_steps))
            self.cart_location += df * self.cart_velocity
            self.cart_velocity += df * cart_accel
            self.pole_angle += df * self.pole_velocity
            self.pole_velocity += df * pole_accel

        if self.visual:
            self._render()


    def remap_angle(self):
        # 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)
        while self.pole_angle < -np.pi:
            self.pole_angle += 2. * np.pi
        while self.pole_angle > np.pi:
```

```
            self.pole_angle -= 2. * np.pi


    # the loss function that the policy will try to optimise (lower)
    def loss(self):
        # first of all, we want the pole to be upright (theta = 0), so we penalise
            ↪  theta away from that
        loss_angle_scale = np.pi/2.0
        loss_angle = 1.0-np.exp(-0.5*self.pole_angle**2/loss_angle_scale**2)
        # but also, we want to HOLD it upright, so we also penalise large angular
            ↪ velocities, but only near
        # the upright position
        loss_velocity_scale = 0.1
        loss_velocity = (1.0-loss_angle)*(self.pole_velocity**2)*
            ↪ loss_velocity_scale
        return loss_angle + loss_velocity

    def terminate(self):
        """Indicates whether or not the episode should terminate.

        Returns:
            A boolean, true indicating the end of an episode and false indicating
                ↪ the episode should continue.
            False is returned if either the cart location or
            the pole angle is beyond the allowed range.
        """
        return np.abs(self.cart_location) > self.state_range[0, 1] or \
               (np.abs(self.pole_angle) > self.state_range[2, 1]).any()

def remap_angle(theta):
    while theta < -np.pi:
        theta += 2. * np.pi
    while theta > np.pi:
        theta -= 2. * np.pi
    return theta


def loss_pos(x):
    # first of all, we want the pole to be upright (theta = 0), so we penalise
        ↪ theta away from that
    loss_angle_scale = np.pi/2. #make division larger for larger penly
    loss_angle = 1.0-np.exp(-0.5*x[2]**2/loss_angle_scale**2)
    # but also, we want to HOLD it upright, so we also penalise large angular
        ↪ velocities, but only near
    # the upright position
    loss_velocity_scale = 0.1
    loss_velocity =(1.0-loss_angle)*(x[3]**2)*loss_velocity_scale #(1.0-loss_angle
        ↪ )*(x[3]**2)*loss_velocity_scale
```

```
    loss_v = (1.0-loss_angle) * x[1]**2 * loss_velocity_scale/20
    loss_pos = (1.0-loss_angle) * x[0]**2 * loss_velocity_scale/10
    return loss_angle + loss_velocity +loss_v +loss_pos #abs(x[1])/5 #+ loss_v
```

vi. Plotting test file

This file is quite messy as it involved all the testing and optimisations performed.

```
import numpy as np
from CartPole import *
import matplotlib.pyplot as plt
import random
from Task_1_Functions import *
from Linear_Modeling import *
from Non_linear_modelling import *
from Control import *
from scipy import optimize

###task 1
##full loop
# rollout([0,14],50,False)
#
# ##oscilations
# rollout([0,5],50,False)



# ##task 1.22
# for i in range(4):
# scan_step(i,100)

# scan_all(100,1) #plots change in state
# scan_all(100,0) #plots next state1)

#contour_plot(50,2,3,1)

##task 2
# x,y =get_random_data_pairs(1000,f=True)
# c = linear_regression(x,y,f=True)
# # scan_all(100,1,False,c)
# print(c)
# test_model(c,100,f=True)
f=True
##non linear regression
# x,y =get_random_data_pairs(10000,quasi=True,f=f)
#x1,y1 = get_random_data_pairs(1000)
# plt.scatter(x[0,:],x[1,:])
# plt.scatter(x1[0,:],x1[1,:])
# plt.show()
```

```
# plt.scatter(x[2,:],x[3,:])
# plt.scatter(x1[2,:],x1[3,:])
# plt.show()
# sd = np.std(x, axis = 1)
#model = non_linear_regression(x,y,100,sd*1.75,3e-2,f=f)
# model = non_linear_regression(x,y,500,sd*1.1,4e-4,f=f)
# model.write_to_file('m=500n=10000')
# model = non_linear_regression(x,y,1000,sd,2e-4,f=f)
# model.write_to_file('m=1000n=1000noforce')
# sd = np.std(x, axis = 1)
# model = non_linear_regression(x,y,5,sd,4e-4,f=f)
# model.write_to_file('test_new_class')
# model.pp()
#
#scan_all(100,1,False,model,nlm=True)


# model = non_linear_model(10,f=f)
# model.read_from_file('m=1000n=10000')
#scan_all(100,1,False,model,nlm=True,f=True)
# test_model(model,100,f=f,model=1)

# transformed_x = model.transform_x(x)
# print(transformed_x.shape)

# a = non_linear_regression(x,y,750,sd,1e-6)
# a = non_linear_regression(x,y,750,sd,0)
#test_regularisation(1000,500,30,500) # for m=100 3e-2 , for m =500 4e-4 , m =
    ↪ 1000 2e-4 # if its a good model then lambda makes little difference
#test_regularisations(1000,30,500)
#test_length_scales(1000,30,500)
# test_length_scale(1000,100,30,500, 3e-2) #1.75
# test_length_scale(1000,500,30,500, 1e-3) ##1.1
#test_length_scale(1000,1000,10,500, 2e-4) ## 1 is best

# contour_plot(50,1,2,3,model = model, error = True,f=f)
# contour_plot(50,2,3,1,model = model, error = True,f=f)
#
#
# test_model(model,100,model=1,f=f)
# test_model(model,100,model=1,f=f)
# test_model(model,100,model=1,f=f)
# test_model(model,100,model=1,f=f)

#test_mn(500)
#scan_policy(0,1,points=50,top=True)
# scan_policy(2,3,points=50,top=True)
#scan_policy(0,2,points=50,top=True)
```

```
# scan_policy(0,3,points=50,top=True)
# scan_policy(1,2,points=50,top=True)
# scan_policy(1,3,points=50,top=True)
# scan_policy(2,3,points=50,top=True)

#p = grad_decent(20,model,np.array([-5. , 5., 10., -8.]))


#trough that extends forever
#[-2.9 , 2.9, -5, -7.5]




#try and force it to oscillate
#[-1.1293577 1.18508497 -4.44289383 -6.28098848]
#[-1.10047104 1.03441207 -5.19856927 -5.39793644]
# up the penalyty for being slightly off centre



#[5,0,0,-5]
#[ 4.25288791 -0.93788629 -0.53291293 -5.62347163]
#[ 4.05333611 -1.16506632 -0.66591379 -5.7730857 ]
#[ 3.69363813 -1.54454497 -0.92180985 -6.04883493]
#[ 2.7255605 -1.2124378 -1.86443561 -6.47899126]
#[ 2.09098295 -0.8883427 -2.38997734 -6.60517805]
#[ 0.46086604 0.08204547 -3.5850825 -6.78087206]
#[-0.45145868 0.70110939 -4.08845974 -6.68875464]
#[-1.1293577 1.18508497 -4.44289383 -6.28098848]



#training on random start
#[-0.48,-0.74,-6,-5.8])
#[-0.36698448 -0.50132806 -6.14096113 -5.84895659]
#[-0.22131246 -0.28809543 -5.93875405 -5.69678041]

#[ 0.05438369 0.25645456 -5.68277981 -5.43769725]


# [-0.2330697 0.53157727 -1.04678576 -3.97326816]
# [-0.1803402 0.51218414 -0.95866025 -3.91482031]
#test_policy(p,50)


#[-0.43582614 0.71224902 -3.07496319 -5.76620956]
```

```
#[-0.4805147 0.74308739 -3.11652141 -5.8005056 ]


#on actual function
#[-0.42695853 0.83341675 -1.48886871 -4.5490731 ]
#[-0.62433402 0.91229899 -1.74282501 -4.84296058]
#[-0.83964223 1.01560017 -1.99773593 -5.12171169]
#[-0.99351887 1.09727579 -2.1612305 -5.29091299]
#[-1.10211907 1.15713825 -2.28483694 -5.40929849]
#[-1.35252253 1.30237887 -2.57434577 -5.67442529]
#[-1.43196324 1.34941848 -2.73536873 -5.80675097]
#[-1.45618002 1.36338369 -2.84721507 -5.89223597]
#[-1.46048033 1.36472933 -2.93289326 -5.94044096] no furter improvment , try new
    ↪ point , needs to be larger as this one never changes direction



#[ 4.39955929 25.7004174 1.12594272 7.04127632]
#[ 6.03126329 23.94430181 2.96117616 12.94970198]
#[13.24146069 25.67375723 10.51650612 30.68875856]
#[16.59400259 27.87177418 14.15819829 38.21578741]

# res= optimize.minimize(loss_function,x0=np.array([-1.,1.,-3.,-6.]) ,method='
    ↪ Nelder-Mead')
# print(res)
# test_policy(res.x,50)


# res= optimize.minimize(loss_function,x0=np.array([0.08818345, -0.2735744 ,
    ↪ -31.36871136, -4.65684842]) ,method='Nelder-Mead')
# res= optimize.minimize(loss_function,x0=np.array([ -1.79225088, 1.36575015,
    ↪ -38.37208003, -2.86461199]) ,method='Nelder-Mead')
# print(res)
# test_policy(res.x,50)
# test_policy(res.x,50,True)


#good one for mormal running, suffers from shooting off for ages, but it can get
    ↪ the pendulum upright from the bottom
# p = np.array([0.08818345, -0.2735744 , -31.36871136, -4.65684842])
# test_policy(p,50,bottom=True)
# test_policy(p,50)
# test_policy(p,50)
# test_policy(p,50)
# test_policy(p,50)
# test_policy(p,50)
# test_policy(p,50)
```

```
## from [-1.,1.,-3.,-6.] started here because it was the point that made the
    ↪ oendulum very still at the start
#to [ 0.08818345, -0.2735744 , -31.36871136, -4.65684842]


##talk about sensitivity to loss function
#all extra penalty ones generated about the good one from before [0.08818345,
    ↪ -0.2735744 , -31.36871136, -4.65684842]


# #adding in x penalty abs(x)/5

# p=np.array([ 23.4712012 , -12.56236927, -23.16326871, 2.70409019])
# test_policy(p,50,True)


# p=np.array([ 0.3033434 , 1.30803855, 1.26010285, -6.8872815 ])
# test_policy(p,50,True)
#v penalty
# #linear abs(v)/5
# p=np.array([ 1.01139182, -9.3456029 , -25.64859931, 1.68480808])
# test_policy(p,50,True)
#still shoots off

#squared
# p=np.array([ -1.92506945, -7.38263384, -30.21323364, 0.95301129])
# test_policy(p,50,True) ## itinially too strong as loss_v = (1.0-loss_angle) * x
    ↪ [1]**2 * loss_velocity_scale its using the pendulum swinging round to keep
    ↪  it stationary

#try 10* smaller :
#p=np.array([ -14.13264246, 13.19196525, -138.72905826, -14.65152857])
#test_policy(p,50,True)
#managed to keep it oscilationg for a while then shoots off

#try weaker still : 20*
#p=np.array([ -14.97029792, 15.01466064, -156.24642981, -15.8959439 ])
#test_policy(p,50,True)
#is better but oscilates very very fast ,

#try stronger limit on the angular velocity (three times)
# p=np.array([ -1.79225088, 1.36575015, -38.37208003, -2.86461199])
# test_policy(p,50,False)
###whooooooooooop gets it very still and stable for points near to the start
# test_policy(p,50,True,sd=1)
# test_policy(p,50,True,bottom=True)
# test_policy(p,50,True,sd=1)
# test_policy(p,50,True,sd=1)
```

```
# ## doesnt for for sd= 1 or from the bottom
# test_policy(p,50,True,sd=0.5)
# test_policy(p,50,True,sd=0.5)
# test_policy(p,50,True,sd=0.5)
# test_policy(p,50,True,sd=0.5)
#0.5 is ok but still difts off slowly away from x=0 at about 30 timesteps


#try running for 30 timesteps in the loss function instead of 20
#p=np.array([ -6.39707597, 11.58085495, -127.68793243, -12.44826855])


# try model where m =1000, start from where prev model ended,
#p=np.array([ 0.30355566, 14.7607993 , -91.53447232, -12.62519575])
#test_policy(p,50,True,sd=0.5)
# test_policy(p,50,True,bottom=True)
# test_policy(p,50,True,sd=1)
# test_policy(p,50,True,sd=1)
## doesnt for for sd= 1 or from the bottom
# test_policy(p,50,True,sd=0.5)
# test_policy(p,50,True,sd=0.5)
# test_policy(p,50,True,sd=0.5)
# test_policy(p,50,True,sd=0.5)
#0.5ok most of the time

#try training on point further away by 10* (trained on [0.08818345, -0.2735744 ,
    ↪ -31.36871136, -4.65684842])
# p=np.array([ 0.14286246, -0.16538587, -11.14879924, -3.11638685])
# test_policy(p,50,True,sd=0.5)
# test_policy(p,50,True,sd=0.5)
# test_policy(p,50,True,sd=0.5)
# test_policy(p,50,True,sd=0.5)
# test_policy(p,50,True,bottom=True)
# test_policy(p,50,True,sd=1)
# test_policy(p,50,True,sd=1)
# can get it up from the bottom but shotts off,
#other ones appear to shooot off

# train on # p=np.array([ -1.79225088, 1.36575015, -38.37208003, -2.86461199])
    ↪ instead
# p=np.array([ 4.2623316 , 10.23557599, -80.79528531, -10.03240233])#didnt finish
# test_policy(p,50,False,sd=0.75)
# test_policy(p,50,True,sd=0.75)
# test_policy(p,50,False,sd=0.75)
# test_policy(p,50,False,sd=0.75)
# test_policy(p,50,False,sd=0.75)
# test_policy(p,50,False,bottom=True)
# test_policy(p,50,False,sd=1)
```

```
# test_policy(p,50,False,sd=1)
## migh be the best yet
#cant get it up from botom, but can keep ot there for 0.5 , and maybe a bit more
    ↪ try 0.75 also keeps it very near to x=0

# nlm = generate_non_linear_policy(11)
# res= optimize.minimize(non_linear_loss_function,x0=np.ones((11)) ,method='
    ↪ Nelder-Mead')
# print(res)
# test_policy(res.x,50,nlm=nlm)

# p = np.array([-250.23134627, -55.56979565, 38.01778897, 30.88543913,
# -104.44498873, 42.26508438, 319.7611134 , -26.40337195,
# -16.56199577, -33.06672362]) ## just didnt work


#with 11 didnt work array([ -0.72926725, -7.13655484, 15.40050596, -8.49481581,
     # -29.78226858, 3.86383929, 9.67926091, 8.67618803,
     # 0.74186436, 16.02190835, 1.88707695])


# test_policy(p,50,nlm=nlm)
# test_policy(p,50,True,nlm=nlm)
# test_force_start(10,9)
# test_force_start(20,9)

# test_force_start(10,7,False)
# test_force_start(20,7,True)



#try optimise 5 locations
# res= optimize.minimize(non_linear_loss_function,x0=[1,1,1,1,1 , 0,0,np.pi,0
    ↪ ,0,0,0,0, 0,0,np.pi/2,0 , 0,0,-np.pi/2,0 , 0, 3,0,3] ,method='Nelder-Mead')
    ↪
# print(res)
# [-6.59383036e+00, 6.78458229e+00, -1.51580341e+00, -6.09327591e+00,1.80406865e
    ↪ +01
# , 3.89023227e-03, 5.21471762e-02, 1.43035453e+00,1.07703253e-02
# , 7.33327374e-03, 4.43271917e-03, 3.99075698e-02,-2.99869444e-02
# , -1.33294311e-02, -8.96068834e-04, 1.76919500e+01, -2.39294072e-03
# , 1.15807067e-02, -1.28465198e-01, -5.63118894e-01,-9.62780792e-02,
# 1.59111089e-02, 9.28812961e-01, -1.12125948e-02,1.02995201e+00]

# nlm = non_linear_model(5,policy=True)
# nlm.sd=np.ones((4,5))
# nlm.basis = np.array([[3.9e-3,7.3e-3,-1.33e-2,1.15e-2,1.59e-2],[5.21e-2,4.43e
    ↪ -3,-8.96e-4,-1.28e-1,9.28e-1],[1.43,4e-2,1.77e1,-5.63e-1,-1.12e-2],[1.08e
```

```
    ↪ -2,-3e-2,-2.39e-3,-9.63e-2,1.03]])
#
# p=np.array([-6.59383036e+00, 6.78458229e+00, -1.51580341e+00, -6.09327591e
    ↪ +00,1.80406865e+01])
#
# test_policy(p,50,nlm=nlm)
# test_policy(p,50,True,nlm=nlm)
#only did small oscilations about base


# nlm = generate_non_linear_policy(10)
# res= optimize.minimize(non_linear_loss_function,x0=np.ones((10)) ,method='
    ↪ Nelder-Mead')
# print(res)
# test_policy(res.x,50,nlm=nlm)

#about top
# p=np.array([ -1.31362979, 0.25518872, -12.80932662, 11.16310449,
# 15.87181763, -9.17540978, 9.21673611, -12.86441258,
# 2.6984682 , -4.25570855])
# # res= optimize.minimize(non_linear_loss_function,x0=p ,method='Nelder-Mead')
# # print(res)
# nlm = generate_non_linear_policy(10)
# nlm.read_from_file('nonzero')
# test_policy(p,50,nlm=nlm,Top=True,visible=False)


#try with optimise top two positions
#
# nlm = generate_non_linear_policy(10)
# options = {"disp": True, "maxiter": 5000,"maxfev":10000}
# res= optimize.minimize(non_linear_loss_function,x0=[-5,-5] ,method='Nelder-Mead
    ↪ ',options=options)
# print(res)
# pres = np.array([-3.01099963, -0.52534223, 0.06382456, -0.01904216, 0.26624947,
# 1.48003872, -0.00567868, 0.89882945, 0.23362329, 0.26767945]) # when points can
    ↪  move
# pres=[res.x[0],-res.x[0],res.x[1],-res.x[1]]
# nlm.read_from_file('justtop')
# test_policy(pres,50,nlm=nlm,Top=True,visible=False)


#symetric points about the range , witth 4* init p=np.array([-1.15711946,
    ↪ 1.15711946])
#symetric points and p
# nlm.read_from_file('justtop')
# nlm.basis[:,0] = res.x[1:5].T
# nlm.basis[:,1] = -res.x[1:5].T
```

```
# p=[res.x[0],-res.x[0]]
# test_policy(p,50,nlm=nlm,Top=True,visible=False)
#array([-1.12426169e+00, -3.62897394e-04, 3.09121301e-01, 3.51876147e
    ↪ -01,6.59403563e-01]) ## which oscilates , try intiilising at a different p
    ↪  val (not -1)
#start at 2, didnt work x: array([11.70325351, -0.13364436, 0.12660699,
    ↪ -0.04405935, -0.82851625])
#start at -5 array([-5.47268559e+00, -1.09839130e-03, 5.24551120e-01, 2.68130009e
    ↪ -01,5.02517841e-01]) # seemed to try
#try -10 x: array([-9.85352053e+00, -4.96117708e-05, 5.16162925e-01, 2.59056507e
    ↪ -01,4.93672245e-01]) #changes direction too strongly
#try -7 array([-6.93450059e+00, 6.54290704e-05, 5.06569486e-01, 2.49215232e
    ↪ -01,5.11567391e-01]) # not strong enugh
#try -8 x: array([-8.26406333e+00, 6.48441315e-04, 4.17808749e-01, 3.44625895e
    ↪ -01,4.33674897e-01]) did 5k iterations....
#try -8.26 with thee locations

#trying various situagions for the loss func, including penalise velocity
    ↪ everywhere


#try 4 points as when the pendulum changes direction its not working
#didnt work
#try centering 2 points on 0v, 0theta dot
# almost works but theta dot is still getting too big, try increasing sd from 3
    ↪ to 5 (p=-13.15 before)
#p=-13.13 almost works, try 7



#tried it on the real model and it looks like its just accelerating it loads
# so try 4 pints again
#on real model it actually worked x: array([-8.74489119, -2.66876906])
#x: array([-31.1264486 , -19.99398314]) on trained model very fast oscillations
    ↪ tho
#try lowerr intiialisation point (-5,-5) not(-14,-14) x: array([-37.61391782,
    ↪ -20.38823026]) is slicghtly better
# increase size of start to 5* init x: array([-23.2721317 , -6.67715398])

# p=np.array([-23.2721,23.2721,-6.677,6.677]) #large init
# #p=np.array([-37.62,37.62,-20.388,20.388]) #small init
# p=np.array([-21.17,21.17,-5.17,5.17]) #large init more v penalty
# p=np.array([-27.669,27.669,-10.98,10.98])
# nlm = generate_non_linear_policy(10)
# nlm.read_from_file('justtop')
# test_policy(p,50,False,sd=0.5,Top=True,nlm=nlm)
# test_policy(p,50,False,sd=0.5,Top=True,nlm=nlm)
# test_policy(p,50,False,sd=0.5,Top=True,nlm=nlm)
```

```
# test_policy(p,50,False,sd=0.5,Top=True,nlm=nlm)
# test_policy(p,50,False,bottom=True,nlm=nlm)
# test_policy(p,50,False,sd=1,Top=True,nlm=nlm)
# test_policy(p,50,False,sd=1,Top=True,nlm=nlm)
#had problems with sliding off slowly, but worked alrigght at 0.5, and some of
    ↪ the 1's which lauchned the pole upwards
# the one trained on the smaller init had more problems with x gtrailing off ,
    ↪ out of the range and therefore cauing the pendulum to fall


#try and penalise a velocity more , with the larger region # use 10 not 20, (note
    ↪  these were trained with angular velocity penalised eerywhere, which is
    ↪ fine because atm i only want to be in the top section moving slowly
#x: array([-21.16835599, -5.16637694])
# didnt work, i should really be penalising x poition not bvelocity as velocity
    ↪ is required to move the thing and keep steady
#return v penalty to 20, and add position penalty
#x: array([-3.9254275 , -5.57901667]) initially wayy too strong , add a dive by
    ↪ 10
#x: array([-27.66941614, -10.98365538])
#seems a bit better



#try optimising locations as well
# nlm = generate_non_linear_policy(10)
# options = {"disp": True, "maxiter": 5000,"maxfev":10000}
# res= optimize.minimize(non_linear_loss_function,x0=[0,0,0.25,2.5,
    ↪ 0,0,0.25,-2.5] ,method='Nelder-Mead',options=options)
# print(res)
# pres=[res.x[0],-res.x[0],res.x[1],-res.x[1]]
# nlm.read_from_file('justtop')
# nlm.pp()
# nlm.sd[0,:]=[30,30,30,30]
# nlm.sd[1,:]=[30,30,30,30]
# # pres=[-27.669,27.669,-10.98,10.98]
# nlm.basis[:, 0] = res.x[ : 4].T
# nlm.basis[:, 1] = -nlm.basis[:,0]
# nlm.basis[:, 2] = res.x[ 4:4+ 4].T
# nlm.basis[:, 3] = -nlm.basis[:,2]
# test_policy(pres,50,nlm=nlm,Top=True,visible=False,sd=0.68)

#x: array([-0.00988998, -0.03803768, 0.57465364, 2.65458026,
    ↪ 0.00525632,0.09308213, 0.88202638, -1.32182546])


# p=np.array([-27.669,27.669,-10.98,10.98])
# nlm = generate_non_linear_policy(10)
```

```
# nlm.read_from_file('justtop')
# nlm.sd[0,:]=[30,30,30,30]
# nlm.sd[1,:]=[30,30,30,30]
# nlm.basis[:,0] = np.array([ -0.00988998, -0.03803768, 0.57465364, 2.65458026]).
    ↪ T
# nlm.basis[:,1] = -nlm.basis[:,0]
# nlm.basis[:,2] = np.array([ 0.00525632,0.09308213, 0.88202638, -1.32182546 ]).T
# nlm.basis[:,3] = -nlm.basis[:,2]
# test_policy(p,50,False,sd=0.5,Top=True,nlm=nlm)
# test_policy(p,50,False,sd=0.5,Top=True,nlm=nlm)
# test_policy(p,50,False,sd=0.5,Top=True,nlm=nlm)
# test_policy(p,50,False,sd=0.5,Top=True,nlm=nlm)
# test_policy(p,50,False,bottom=True,nlm=nlm)
# test_policy(p,50,False,sd=1,Top=True,nlm=nlm)
# test_policy(p,50,False,sd=1,Top=True,nlm=nlm)
# #didnt work



# now try starting from bottom and fliging upwards into the top four which remain
    ↪  constant
#try it with just righthand side basis location (posive theta) as direction
    ↪ shouldnt matter
#make it so that fast theta dot isnt penalised everywhere
# nlm = generate_non_linear_policy(10)
# options = {"disp": True, "maxiter": 5000,"maxfev":10000}
# res= optimize.minimize(non_linear_loss_function,x0=[100,-100, -75,-15] ,method='
    ↪ Nelder-Mead',options=options)
# print(res)
# pres=[-27.669,27.669,-10.98,10.98,res.x[0],res.x[1],res.x[2],res.x[3]]
# pres=[-27.669,27.669,-10.98,10.98,76, -77,0, -37.025]
# nlm.read_from_file('wholetest')
# test_policy(pres,100,nlm=nlm,Top=False,visible=False)

#x: array([23.75144614, 309.11040632, -154.83442517]) didnt work (with only one
    ↪ point at the bottom, in a attempt to get it up in one swing

#x: array([ 9.11438853e+01, -8.47006718e+01, -7.72495253e+02, 1.28177381e-01])
    ↪ gets it up in two swings, however its wayyy too fast , change the near top
    ↪  one to be a slow down one centered on -10 theta dot
# the last point has a value of 0.04 so is pretty useless, try itialising it at
    ↪ 50 # made the thing just oscilalte
#try at 10 with a smaller sd for theta x: array([ 100.74749168, -101.83204991,
    ↪ -77.14597527, 9.72082445])
##looks vaguly promising hwoever it first goes over at x=6 so extend sd on the
    ↪ top 4 from 3 to 7, made no difference
#try intiialising at 5 #didnt work
```

```
#try penalising angle more
#x: array([-14.83120117, 18.16611363, 42.71462915, 33.59296929]) this didnt work,
    ↪  probably because of the effect of loss angle on the other penaltyies


#pres=[-27.669,27.669,-10.98,10.98,100, -101,-77, -15]
#change sd of x,v to 30 and it works for a while, need to fix a constant velocity
    ↪  of 9 tho
#run this throuhg optimisation with the large sd
#x: array([ 1656.47761715, -1689.18218664, 1389.80530118, 121.90091981]) still
    ↪ shoots off

#changing from the two being 80, -80 to -75,75 makes the velocity change
    ↪ direction
#pres=[-27.669,27.669,-10.98,10.98,75, -75,-77, -15]
#77 is a good value and almost works, clearly it was trying to do it wayyy too
    ↪ fast
#try reducing the impulse at 1/2 at about , 69 it seems better

# reduce all of them as its still going very fast at the start?

#problem is the pendulum isnt at the top until after 20 iterations
#up to 40 or initialise at a mid point



#pres=[-27.669,27.669,-10.98,10.98,76, -77,0, -37.025] and change the 2/3 point
    ↪ to be at 0.2 sd for angle
#gets it up in half a swing

# run it for with initialisation at about 20 iteration [-5.5,-3.36,-0.56,0.42]
# nlm = generate_non_linear_policy(10)
# options = {"disp": True, "maxiter": 5000,"maxfev":10000}
# res= optimize.minimize(non_linear_loss_function,x0=[76,-77, 0,-37.025] ,method='
    ↪ Nelder-Mead',options=options)
# print(res)
# pres=[-27.669,27.669,-10.98,10.98,res.x[0],res.x[1],res.x[2],res.x[3]]
# pres=[-27.669,27.669,-10.98,10.98,76, -77,0, -37.025]
# nlm.read_from_file('wholetest')
# test_policy(pres,100,nlm=nlm,Top=False,visible=False)

#x: array([ 7.95716748e+01, -8.06053706e+01, -2.32638157e-04, -3.77144461e+01]) #
    ↪ not useful tho

# nlm = generate_non_linear_policy(10)
# options = {"disp": True, "maxiter": 30,"maxfev":50}
# res= optimize.minimize(non_linear_loss_function,x0=[76,-77,-37.025] ,method='
    ↪ Nelder-Mead',options=options)
```

```
# print(res)
# pres=[-27.669,27.669,-10.98,10.98,res.x[0],res.x[1],0,res.x[2]]
#pres=[-27.669,27.669,-10.98,10.98,76, -77,0, -37.025]
# pres=[-27.669,27.669,-10.98,10.98,77, -77,0, -30.575]
#pres=[-27.669,27.669,-10.98,10.98,37.2 ,38.8,0, -108.1]
#pres = [-27.669,27.669,-10.98,10.98,71.71793911, -79.06023329, 0,-38.85691662]
# nlm.read_from_file('flickup')
# test_policy(pres,100,nlm=nlm,Top=False,visible=False)

# if bottom are different then x: array([ 71.71793911, -79.06023329,
    ↪ -38.85691662])
#else it failed

# go back to optimising the top sections, and intiialise at the
    ↪ [-5.5,-3.36,-0.56,0.42]
# also go back to using model with 1000 basis points that i stopped using for
    ↪ some reason
# nlm = generate_non_linear_policy(10)
# # options = {"disp": True, "maxiter": 5000,"maxfev":10000}
# # res= optimize.minimize(non_linear_loss_function,x0=[-27.669,-10.98] ,method='
    ↪ Nelder-Mead',options=options)
# # print(res)
# # pres=[res.x[0],-res.x[0],res.x[1],-res.x[1],76, -77,0, -37.025]
# pres=[-27.669,27.669,-10.98,10.98,76, -77,-20, -37.025]
# nlm.read_from_file('wholetest')
# test_policy(pres,100,nlm=nlm,Top=False,visible=False)

#x: array([-27.77622433, -11.66771078])

#add in bounce back
# nlm = generate_non_linear_policy(10)
# options = {"disp": True, "maxiter": 50,"maxfev":10}
# res= optimize.minimize(non_linear_loss_function,x0=[-20,-18.95] ,method='Nelder-
    ↪ Mead',options=options)
# print(res)
# pres=np.array([-27.669,27.669,-10.98,10.98,76,-77,res.x[0],-37.025,-res.x[0],
    ↪ res.x[1],-res.x[1]])
#pres=[-27.669,27.669,-10.98,10.98,76, -77,-20, -37.025,20
    ↪ ,-18.95,18.95]#-12.74,12.74]
# pres=[-27.669,27.669,-10.98,10.98,76, -77,-19.9, -37.025,19.9
    ↪ ,-18.95,18.95]#-12.74,12.74]
# nlm.read_from_file('wholetest')
# test_policy(pres,250,nlm=nlm,Top=False,visible=False,sd=0.1)
# test_policy(pres,250,nlm=nlm,Top=False,visible=False,sd=0.1)
# test_policy(pres,250,nlm=nlm,Top=False,visible=False,sd=0.1)
# test_policy(pres,250,nlm=nlm,Top=False,visible=False,sd=0)
```

```
# need to adjust the the velocity basis position of the slow down so that it
    ↪ doesnt act when the cart goes the opposite direction
#pres=[-27.669,27.669,-10.98,10.98,76, -77,-20, -37.025,20 ,-32.95,32.95] changed
    ↪  x positions to be at +-0.5 to try and stop oscilations, this setup caused
    ↪  a flip but a perfect stop afterwards
#pres=[-27.669,27.669,-10.98,10.98,76, -77,-20, -37.025,20
    ↪ ,-18.95,18.95]#-12.74,12.74] has no flip

#doesnt seem that robust to random starts, might need to change the bounch back
    ↪ basis to be active over larger velocity ranges, or get the swing up to be
    ↪ better and not involve a massive change in x

#test_noise(30,500,f=True)

x,y = get_random_data_pairs(10000,quasi=True,f=True,noise=0.2)
sd = np.std(x, axis = 1)
model = non_linear_regression(x,y,1000,sd,4e-4,f=f)
model.write_to_file('noisymodel2')
model.pp()
test_model(model,100,model=1,f=f)
test_model(model,100,model=1,f=f)
test_model(model,100,model=1,f=f)
test_model(model,100,model=1,f=f)
```