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:	~~Group Report~~ / Individual Report

Name(s): (capitals)	crsID(s):	College(s):
HARVEY HUGHES	hh458	Emmanuel

Declaration for: Interim Report 1 / ~~Interim Report 2 / Final Report~~

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

May 26, 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. Therefore the accuracy of predictions from the linear model was low, except around the stable equilibria where the model is approximately linear.

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

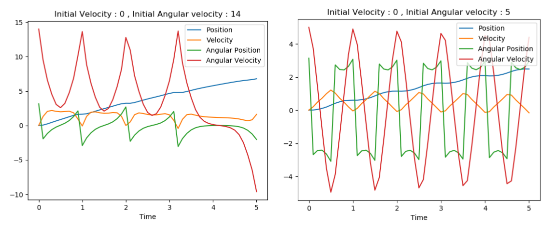
The python code written to implement these features is listed in the appendix.

II. Investigations

i. Dynamic Simulations

i.1 Rollout

Using the dynamic simulation provided in cartpole.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 initialisation. These non-linearities are best shown by the sine like behaviour of $\dot{\theta}$ when scanning θ. Smaller amplitude sinusoidal re-

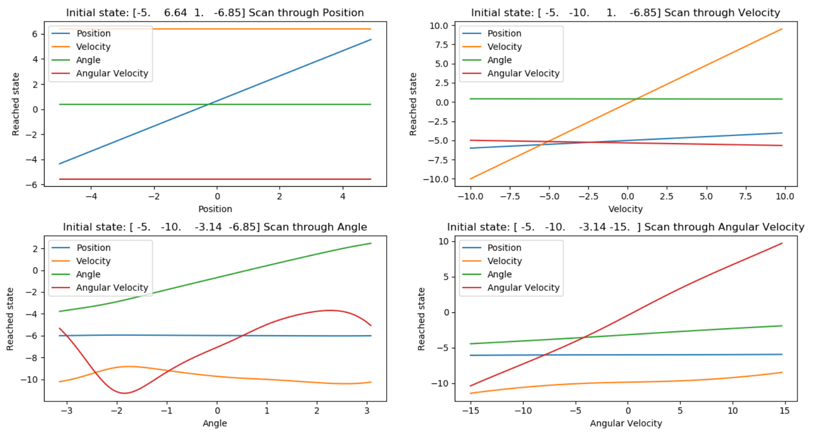


Figure 2: *Scan through each state variable to see the effect on the next state*

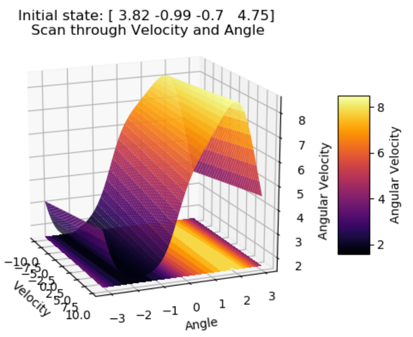


Figure 3: *Scan through $\dot{x}$ and θ to see the effect on the next step's $\dot{\theta}$.*

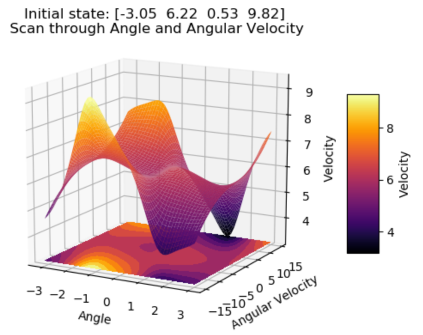


Figure 4: *Scan through $\dot{\theta}$ and θ to see the effect on the next step's $\dot{x}$.*

lationships 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 $\dot{\theta} \in [-15, 15]$. **x** is the initial state $\mathbf{x}(0)$, and **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.

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

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.

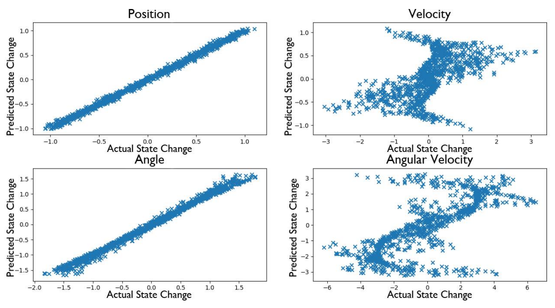


Figure 5: *Linear model performance on the training set*

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 in figure 7. Each predicted relationship is linear as expected, with the predictions agreeing

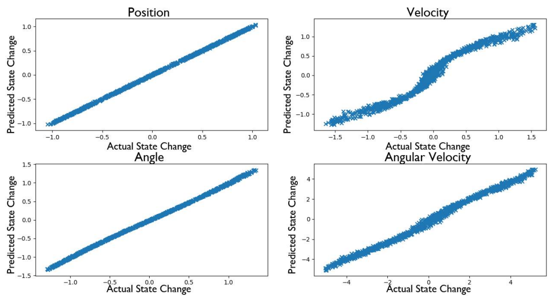


Figure 6: *Linear model performance only about $\theta = \pi$*

State	Whole Motion m	Whole Motion r^2	Stable equilibrium m	Stable equilibrium 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*

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

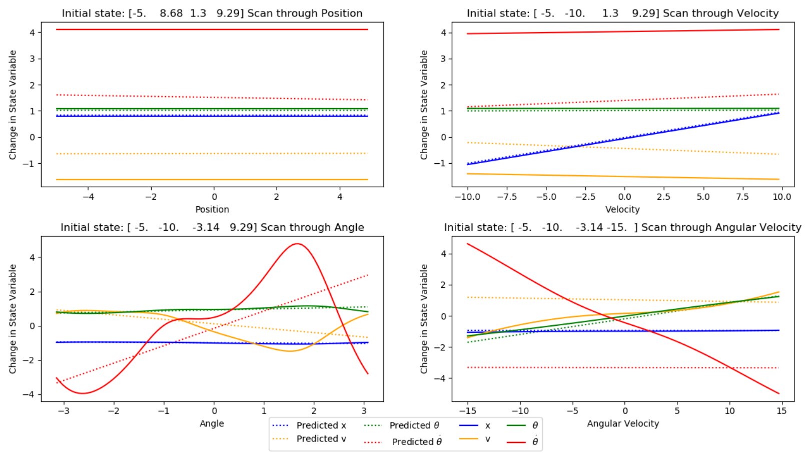


Figure 7: *Predicted state change and real state change when scanning through each variable*

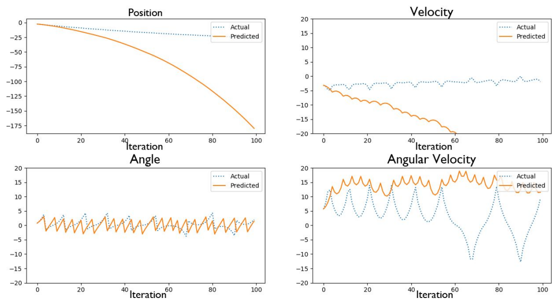


Figure 8: *Predicted state trajectory with loops*

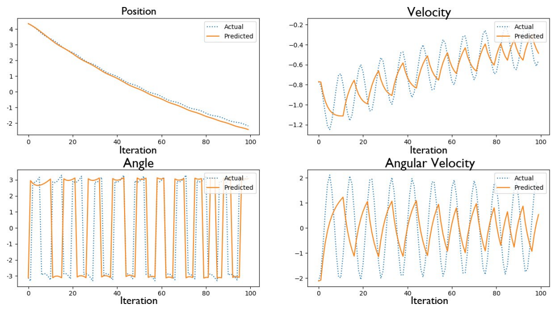


Figure 9: *Predicted state trajectory about $\theta = \pi$*

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.

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
- Non-linear modelling will be required for accurate predictions

IV. Appendix

i. Plotting test file

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

###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 state

contour_plot(50,2,3,1)

##task 2
x,y =get_random_data_pairs(1000)
c = linear_regression(x,y)
scan_all(100,1,False,c)
print(c)
test_linear_model(c,100)
```

ii. Task_1_Functions.py

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


labels = [ "Position" , "Velocity" , "Angle" , "Angular Velocity"]
ranges = [10.,20.,2*np.pi,30. ] ## the size of suitable ranage
```

```
def rollout (initial_state , n = 100, visual =False):
    ## 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
        system.performAction(0)  ## runs for 0.1 secs with no force
        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 random_init(x,stable_equ=False):
    # generates a random start state in the correct range

    for i in range(4):
        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]
    return x

def scan_step(to_scan,n,visual=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)
    ranges = [10.,20.,2*np.pi,30. ] ## the size of suitable ranage
    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(0)
        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):  ##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)
    initial_state = np.zeros(4)
    initial_state = random_init(initial_state)
    line_lables=[None,None,None,None,None,None,None,None]
    fig = plt.figure()
    for to_scan in range(4):
        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))
        scanned_state = np.zeros((4,n))
        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

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

        fig.add_subplot(2,2,to_scan+1)
        plt.subplots_adjust(hspace=0.3)
        if to_scan == 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 c.any()!=None:
                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"]
        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 ):
    # 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
    system = CartPole(visual)
    initial_state = np.zeros(4)
    initial_state = random_init(initial_state)

    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((0))
        rav_Z[i] = system.getState()[cont]
    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] )

    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()
```

iii. 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 *

def get_xy_pair(n,system):  ## returns the nth iternation and the
#difference bween nth and n+1th for an intialised system
    # x and y both 4x1 matrices
    for i in range(n):
        system.performAction(0)  ## runs for 0.1 secs with no force
    x = system.getState()
    system.performAction(0)
    y= system.getState() - x
    return x,y

def get_random_data_pairs(n,visual=False,stable_equ=False):
    # 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 respectively
    system = CartPole(visual)
    initial_state = np.zeros(4)
    initial_state = random_init(initial_state,stable_equ)
    system.setState(initial_state)

    xs = np.zeros((4,n))
    ys = np.zeros((4,n))
    for i in range(n):
        initial_state = random_init(initial_state)
        system.setState(initial_state)

        xs[:,i],ys[:,i] = get_xy_pair(0,system)
## each column is a sample
    return xs,ys


def linear_regression(x,y):
    #x is (4xn) matrix of states, y is (4xn) matrix of state changes
    # minimise Y - CX
    # returns the 4x4 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)


    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

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



def test_linear_model(c,n,visual=False,loop = False, osc = False,
 stable_equ=False):
    # c is 4x4 linear coefficent matrix , n is number of time
    # iterations to predict
    # stable_equ if motion is about theta = pi
    # plots real state trajectory and linear model predictions

    system = CartPole(visual)
    initial_state = np.zeros(4)
    initial_state = random_init(initial_state,stable_equ)
    if loop == True:
        initial_state = np.array([0,0,np.pi,15])
    elif osc == True:
        initial_state = np.array([0, 0, np.pi, 5])

    system.setState(initial_state)
```

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

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 !=0 and i!=2:
        plt.ylim(-20, 20)

plt.show()
```