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

IIA Projects

TO BE COMPLETED BY THE STUDENT(S)

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

Name(s): (capitals)	crsID(s):	College(s):
OSCAR SAHAROY	os408	St. Johns

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	Excellent	Very Good	Good	Acceptable	Minimum acceptable for Honours	Below Honours

Grade the reports by ticking the appropriate guideline assessment box below, 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.

Guideline assessment (tick one box)

A*/A	A/B	B/C	C/D	D/E

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	

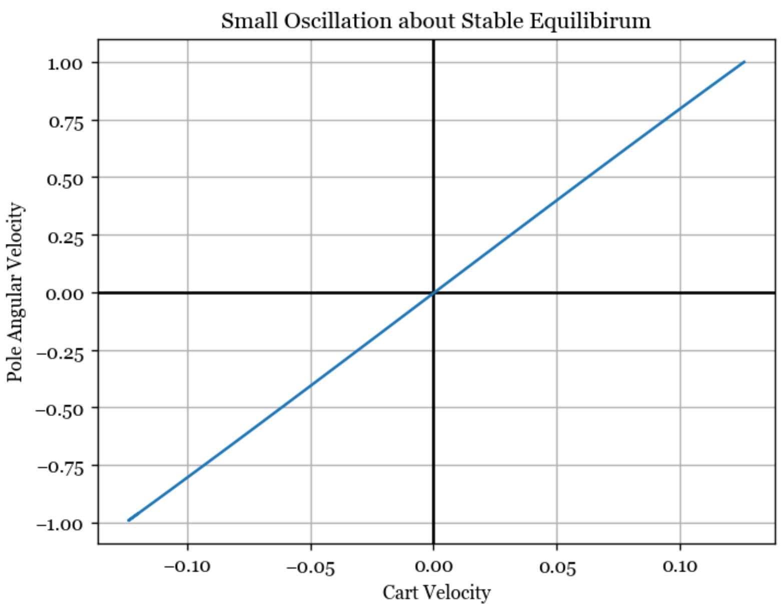
SF3: Machine Learning
Interim Report

Oscar Saharoy

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

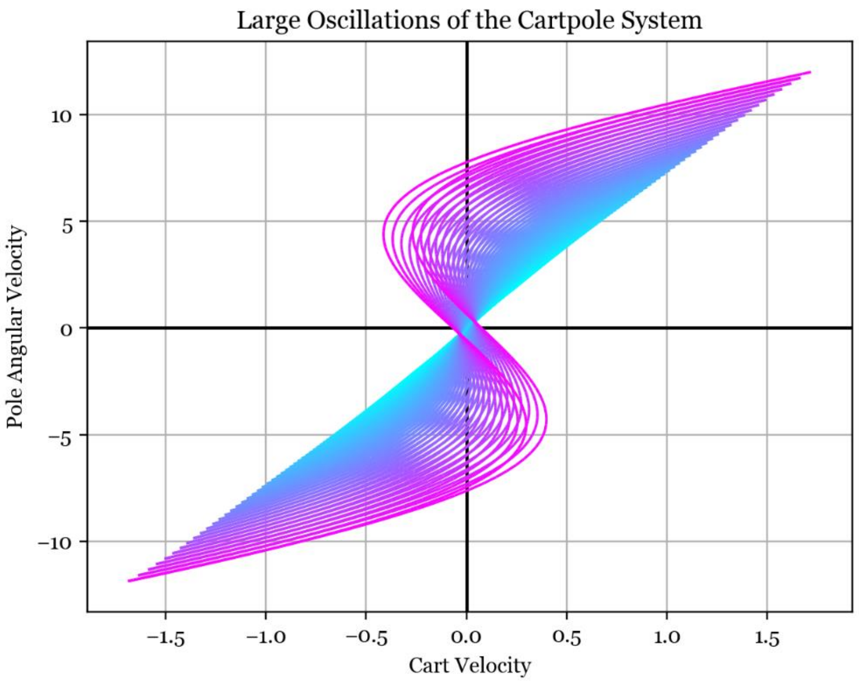
Task 1.1 - Simulation of Rollout

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

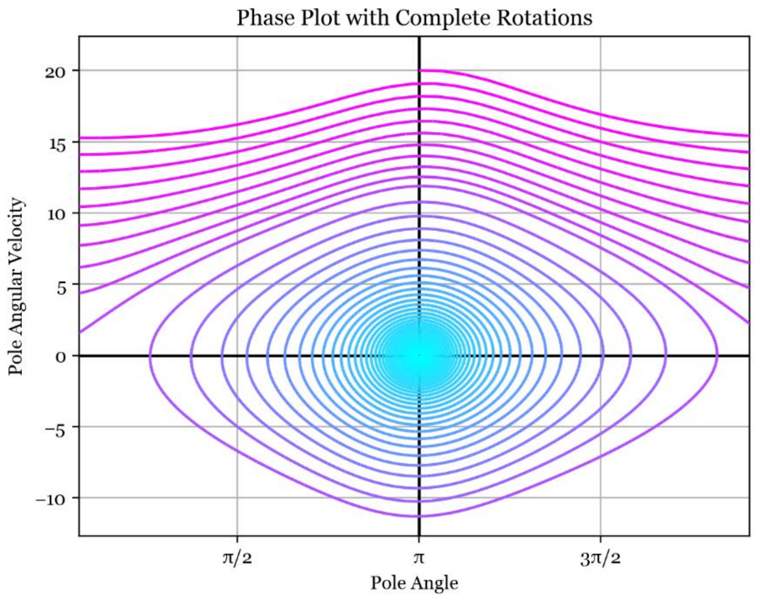


Small Oscillation about Stable Equilibrium. The plot on the left shows pole angular velocity plotted against cart velocity as time progresses through small oscillations - the system traverses up and down the blue line as the pole swings. The linear trend reflects the small angle assumption - the restoring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

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



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

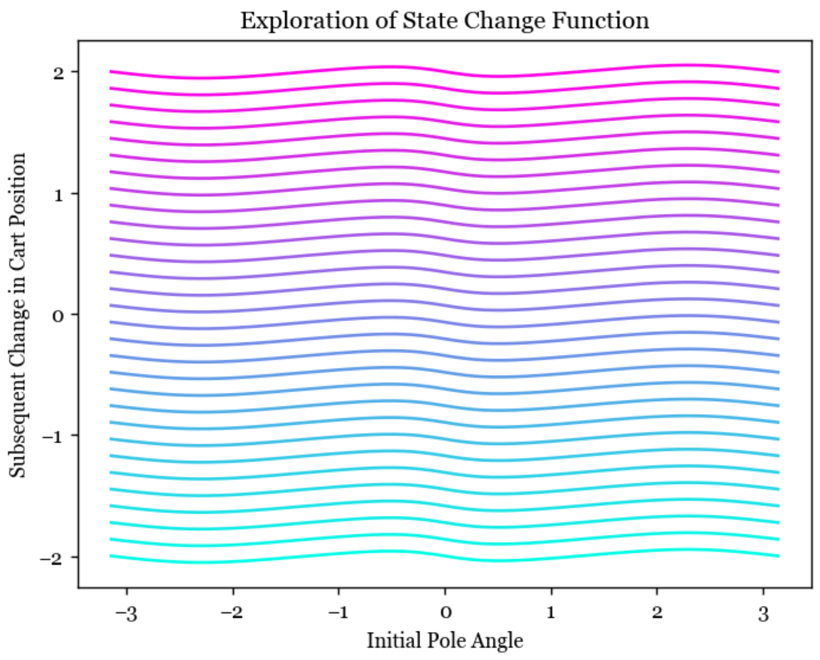


Task 1.2 - Changes in State

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

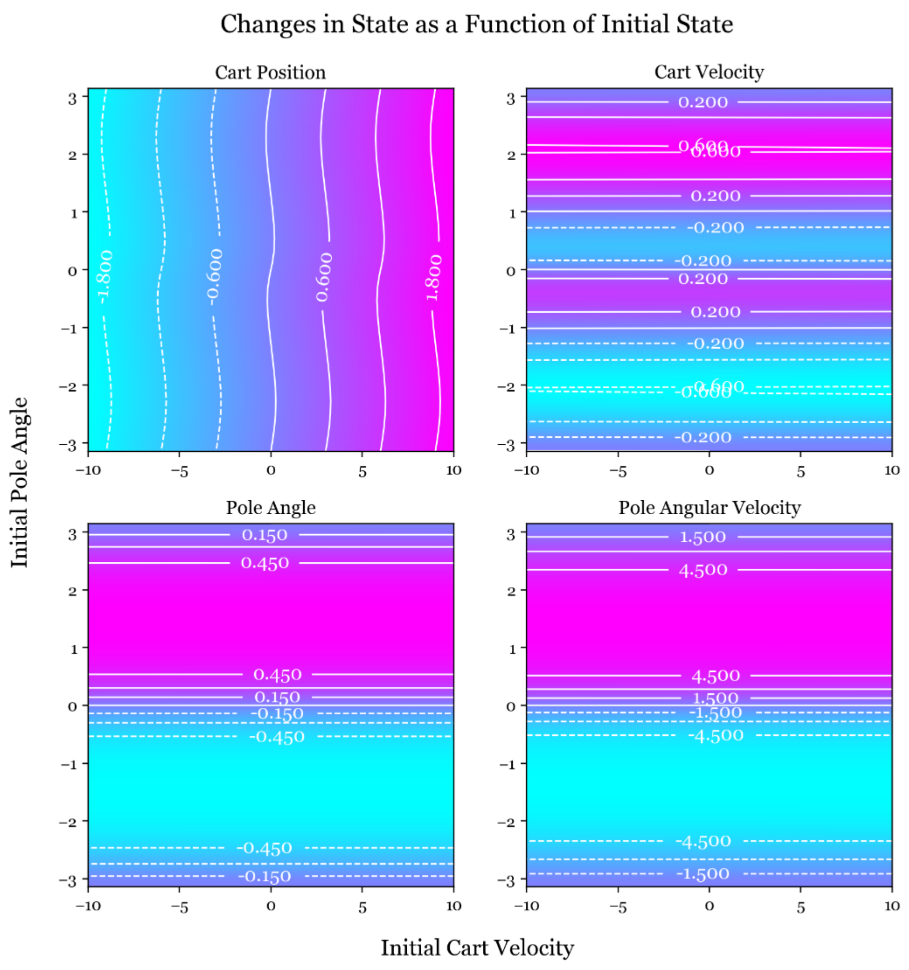
It is more helpful to investigate the change in state as a function of the initial state rather than the subsequent state as a function of the current state. This is because the subsequent state will be quite similar (almost linearly related) to the initial state, assuming a small timestep. We can capture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

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



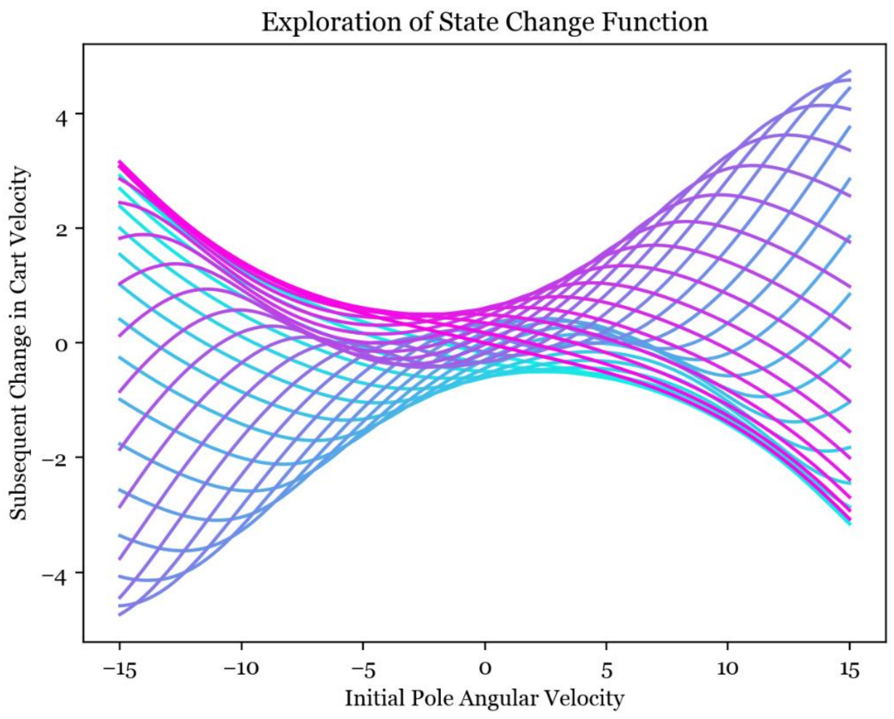
However, the change in cart position is a slightly nonlinear function of the pole angle since there is a wobble in the lines. Plotting the changes in other state variables in this way yields coincident lines which are better represented in the contour plot below.

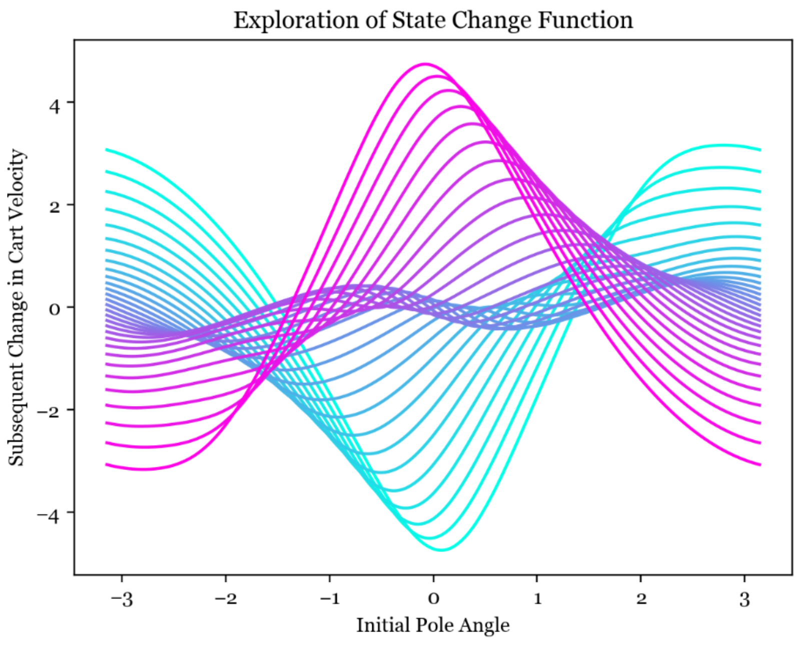
In this plot, the initial cart velocity varies in the x direction and the initial pole angle in the y direction. The subsequent changes in the state vector are shown as contour plots. The changes in cart velocity, pole angle and pole angular velocity are not functions of the initial cart velocity. This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



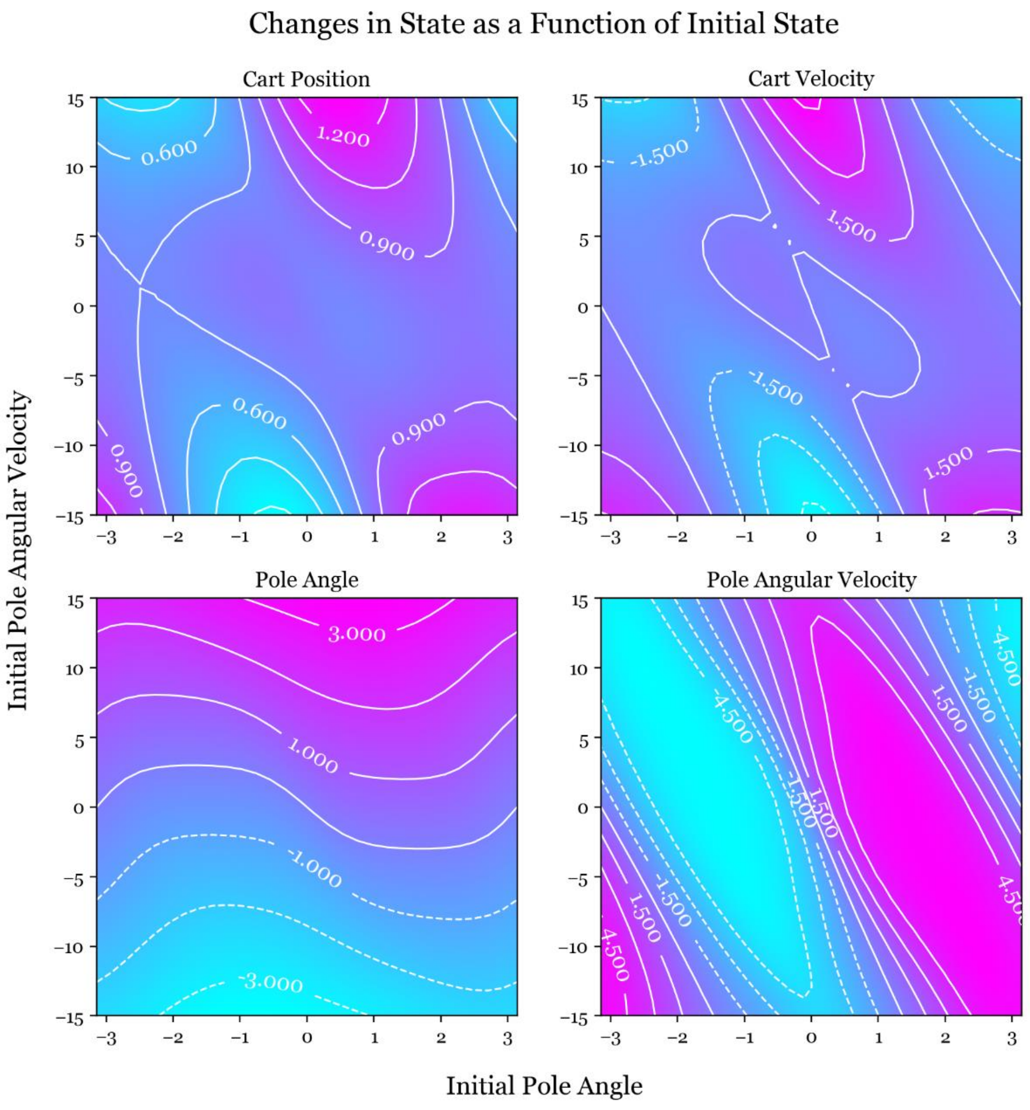
The same reasoning leads to the contour plots for varying cart velocity and pole angular velocity consisting of straight lines so I will move to the more interesting combination of varying pole angle and angular velocity.

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





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



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

Task 1.3 – Linear Model

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

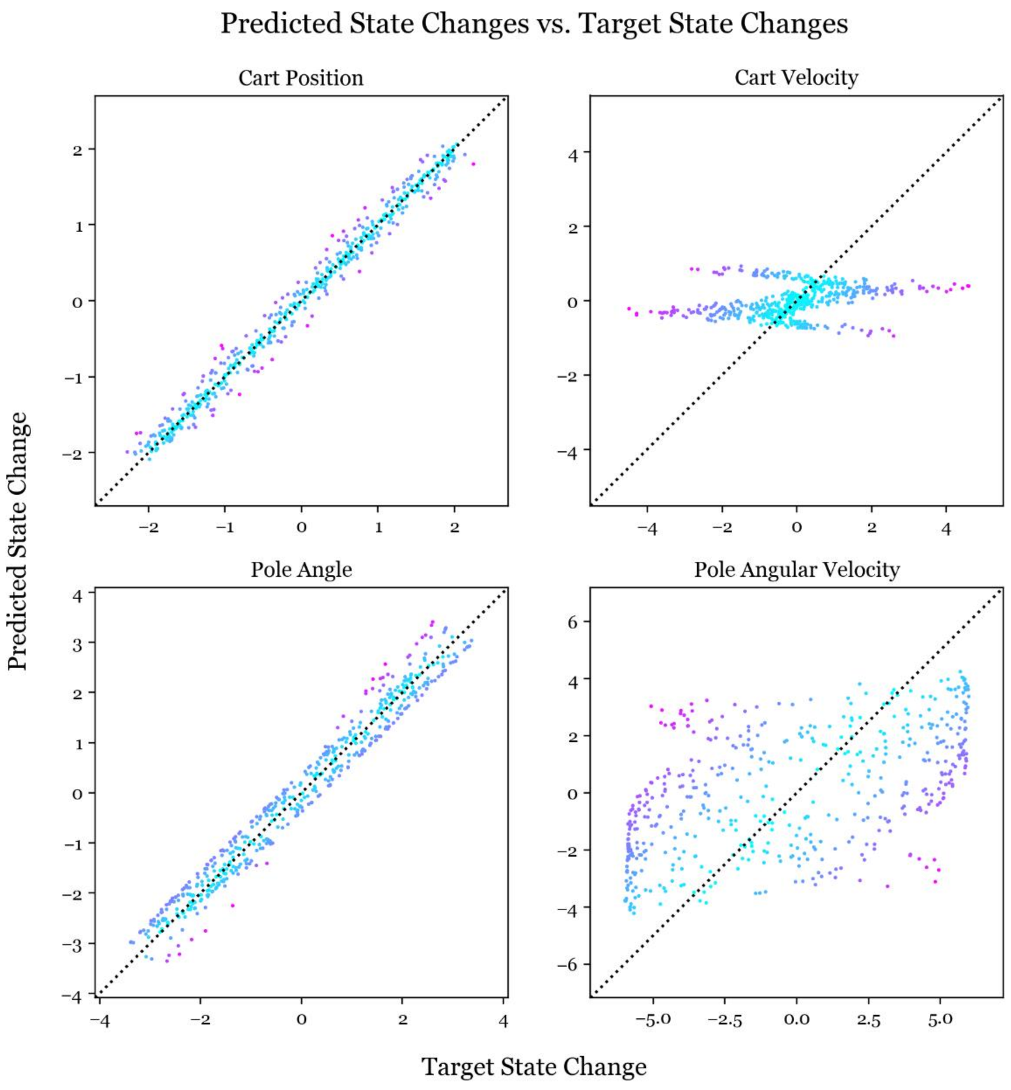
$$\boldsymbol{Y} = \boldsymbol{f}(\boldsymbol{X}) = \boldsymbol{XC}$$

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

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

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

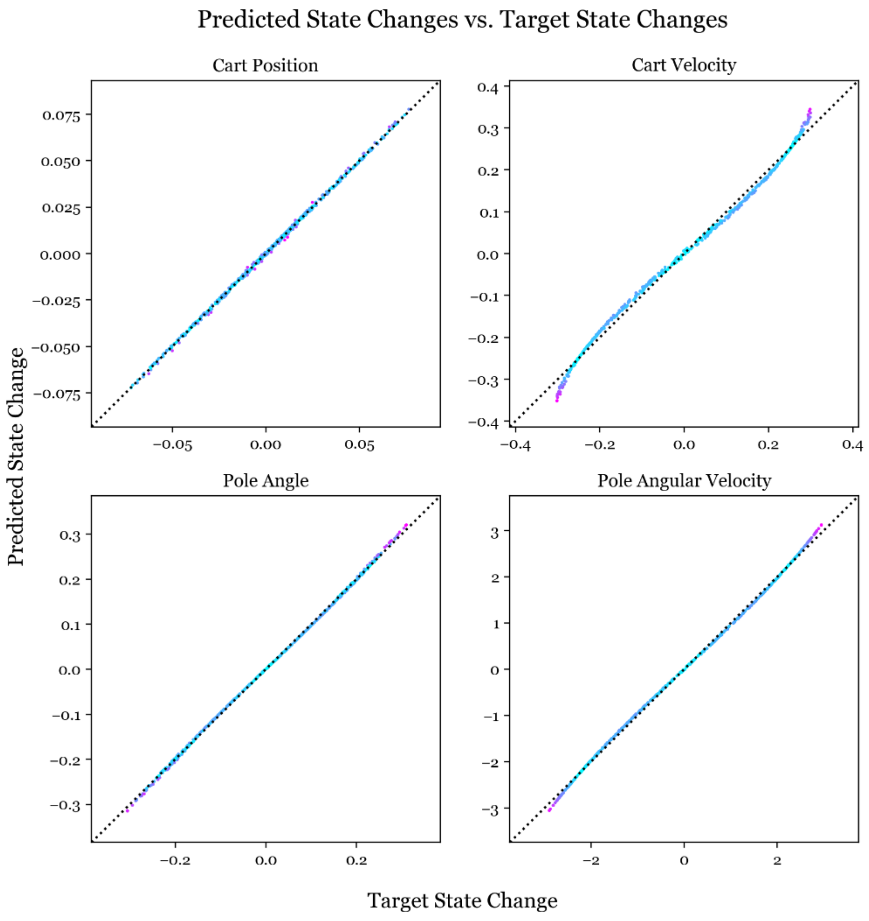
Predicted State Change vs. Target State Change. Here I have plotted one point per training data element on a graph for each element in the state change vector. For each point, the x coordinate is the target change in state as returned from `perform_action` and the y coordinate is the predicted change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



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

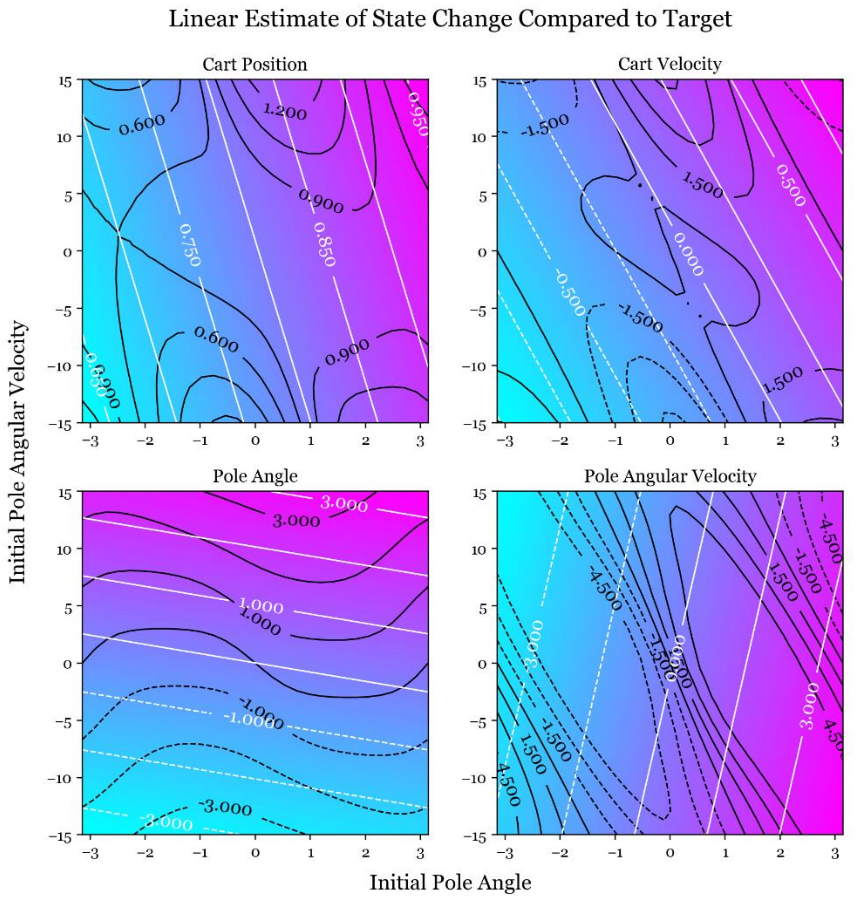
Linear Model for Small Deflections Only. I repeated the above procedure but limited the training data to small deflections from equilibrium (initial states between -0.25 and 0.25). Thinking back to the Taylor series approximation for a function I expected this scheme to yield better predictions near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

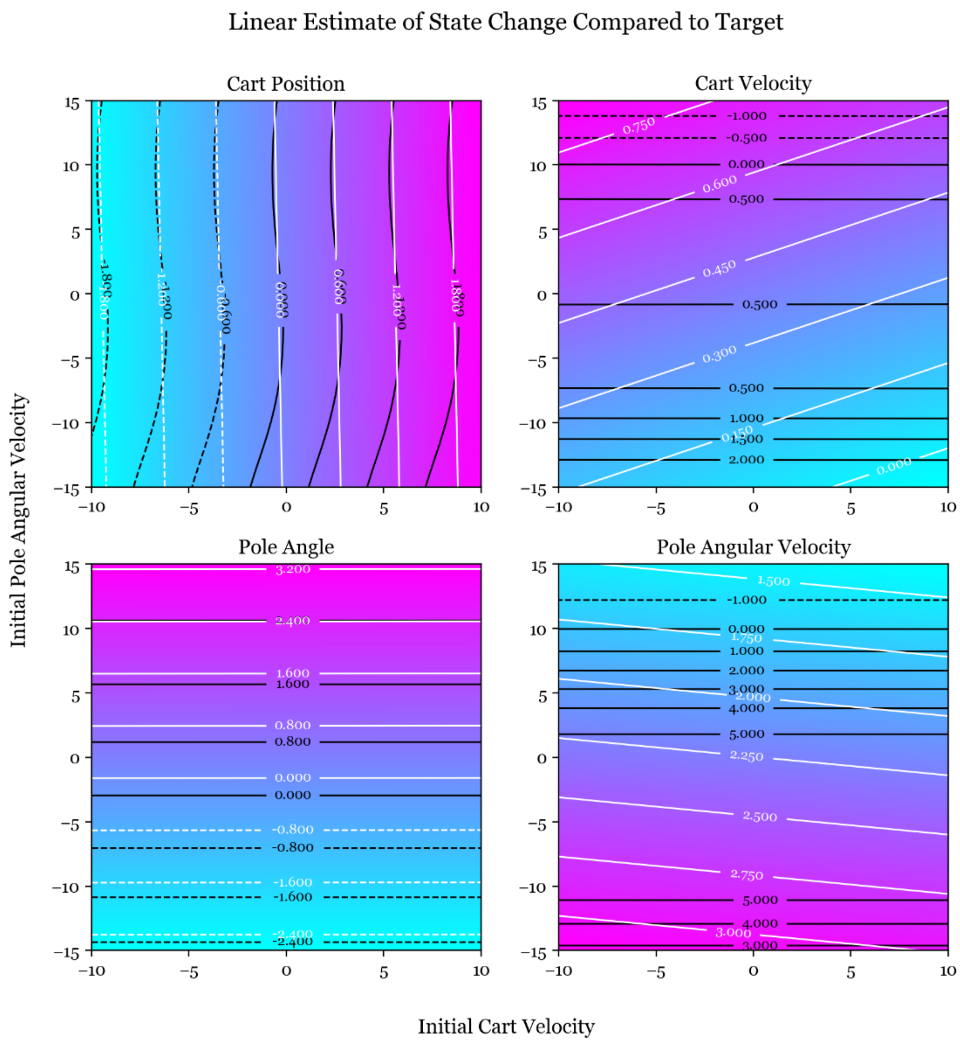
The linear model works nicely for small deflections as shown opposite, with all the points near to the ideal lines. Perhaps we could build a good model by using many linear models centred around different points, which provides motivation for moving to a nonlinear model.



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

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





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

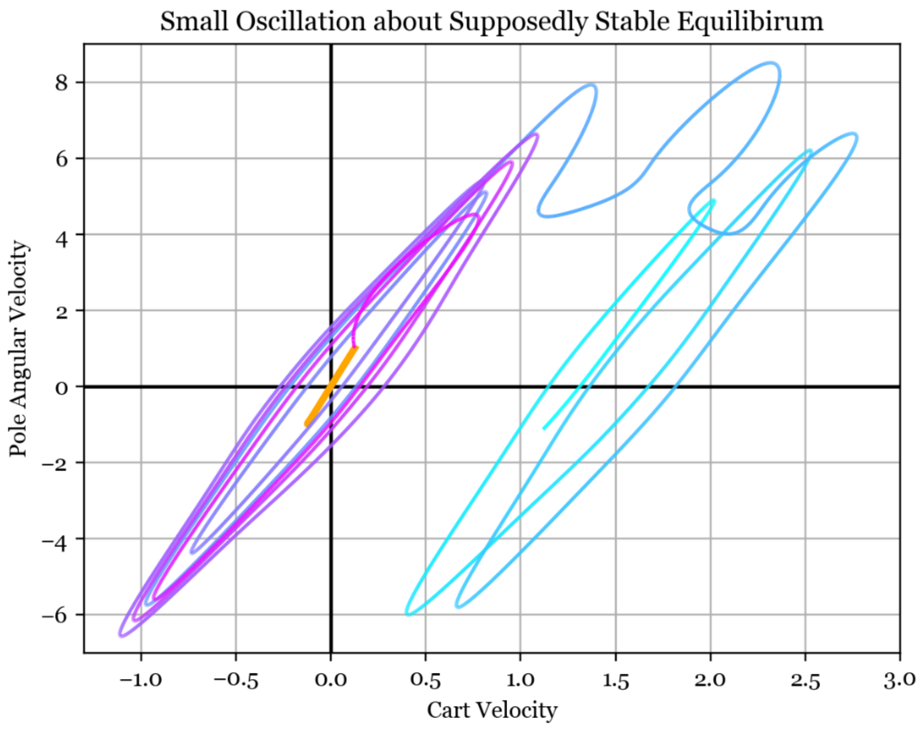
Prediction Accuracy. The elements that are well predicted by the linear model are the change in cart location and pole angle. The change in cart location will be quite similar to the velocity of the cart times the timestep (neglecting acceleration during the timestep), so it is expected that we could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

Task 1.4 - Prediction of System Evolution

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

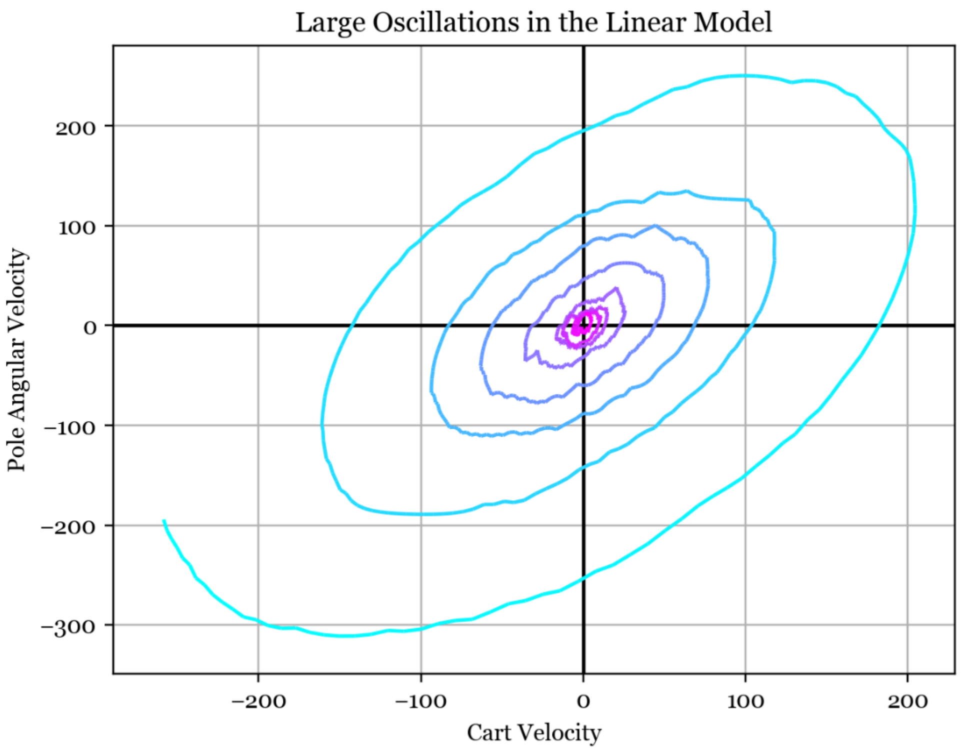
Remapping. It is vital to remap the pole angle to the range $-\pi$ to π during this process. If this is not done, the angle grows beyond anything seen in the training data and the result is an abnormally large input of energy to the system which causes divergence. If we don't remap, the state after $\boldsymbol{k}$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

Small Oscillations. The plot below and right is initialised the same as the small oscillations case from the first plot in this report, and shows the time evolution of the pole angular velocity plotted against cart velocity. The target behaviour is shown in orange, which is the data from the small oscillation plot using the analytic dynamics. The linear estimation time evolution is the curve that starts in magenta at initialisation and turns more cyan as time goes on.



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

Large Oscillations and Complete Revolutions. This plot is the same as the previous but starts with a larger pole angular velocity so that the oscillations start larger and grow to full revolutions.



As time progresses the line goes from magenta to cyan, and both the pole angular velocity and cart velocity diverge, reaching large values. The outward spiral is indicative of an unstable system: the linear model can't enforce conservation of energy. However, the spiral structure conveys some understanding that a larger cart velocity implies a smaller pole angular velocity.

Appendix A - CartPole.py

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

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

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

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


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

def loss(state):
    return 1-np.exp(-np.dot(state,state)/(2.0 * 0.5**2))


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

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

        self.reset()

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

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

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

        if self.visual:
            self.drawPlot( fig_num )


    # reset the state vector to the initial state (down-hanging pole)
    def reset(self):
        self.cart_position = 0.0
        self.cart_velocity = 0.0
        self.pole_angle    = np.pi
        self.pole_angvel   = 0.0


    def set_state(self, state):

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


    def get_state(self):

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


    def remap_angle(self):

        self.pole_angle = remap_angle( self.pole_angle )


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


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

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

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

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

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

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

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

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

        if self.visual:
            self._render()


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

        plt.ion()
        self.fig = plt.figure( fig_num, figsize=(9.5,2) )

        # draw cart
        self.axes = self.fig.add_subplot(111, aspect='equal')
        self.box = plt.Rectangle(xy=(self.cart_position - self.cartwidth / 2.0, -self.cartheight / 2.0),
                                 width=self.cartwidth, height=self.cartheight)
        self.axes.add_artist(self.box)
        self.box.set_clip_box(self.axes.bbox)

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

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


    def _render(self):

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

        self.fig.canvas.draw()

class Object(object):
    pass

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

    self = Object()

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

    self.pole_length = 0.5
    self.pole_mass = 0.5

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

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

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

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

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

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

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

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

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

Appendix B – sf3.ipynb

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

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

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

# store results for later
cache = {}


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


# # Task 1.1 - Simulation of Rollout
# instantiate a cartpole and use a small timestep to get smooth lines
rollout_cartpole = CartPole.CartPole()

rollout_cartpole.sim_steps = 1
rollout_cartpole.delta_time = 0.01


# ## Small Oscillation about Stable Equilibirum
rollout_cartpole.reset()

rollout_cartpole.cart_velocity = 0.126
rollout_cartpole.pole_angvel = 1

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

for i in range(50):

    rollout_cartpole.perform_action()

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

    states.append( rollout_cartpole.get_state() )
```

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

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

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


# ## Large Amplitude Oscillations

# large oscillations about stable equilibrium

rollout_cartpole.reset()

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

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

for i in range(10000):

    rollout_cartpole.perform_action()

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

    states.append( rollout_cartpole.get_state() )


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

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

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

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

ax.add_collection( linecollection )

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

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


# ## Swinging Over the Top

# even larger oscillations

rollout_cartpole.reset()

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

x, y = [], []

for i in range( 5700 ):

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

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

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

    rollout_cartpole.perform_action()
```

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

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

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

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

ax.add_collection( linecollection )

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

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


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

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

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

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

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

# create array of initial state vectors

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

# get 2d array of subsquent state vectors

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

state_changes = subsequent_states - initial_states


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

# create array of interpolated colours

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


for i, row in enumerate(state_changes):

    # convert to arrays and extract certain components from vectors

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

    # code to smooth plot lines

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

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

    # plot then move onto next line

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

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

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

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

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

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

    ax.set_title( titles[i] )

fig.text(0.5, 0.95, 'Changes in State as a Function of Initial State', ha='center', va='center', fontsize=14)
fig.text(0.5, 0.04, 'Initial Cart Velocity', ha='center', va='center', fontsize=12)
fig.text(0.06, 0.5, 'Initial Pole Angle', ha='center', va='center', rotation='vertical', fontsize=12)


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

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

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

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

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

# create array of initial state vectors

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

# get 2d array of subsquent state vectors

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

# cache data for later

cache["state_changes_varying_angle_and_angvel"] = state_changes


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

# create array of interpolated colours

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


for i, row in enumerate(state_changes):

    # convert to arrays and extract certain components from vectors

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

    # code to smooth plot lines

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

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

    # plot then move onto next line

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

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


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

```python
# create array of interpolated colours

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


for i, row in enumerate(state_changes):

    # convert to arrays and extract certain components from vectors

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

    # code to smooth plot lines

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

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

    # plot then move onto next line

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

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


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

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

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

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

fig.text(0.5, 0.95, 'Changes in State as a Function of Initial State', ha='center', va='center', fontsize=14)
fig.text(0.5, 0.04, 'Initial Pole Angle', ha='center', va='center', fontsize=12)
fig.text(0.06, 0.5, 'Initial Pole Angular Velocity', ha='center', va='center', rotation='vertical', fontsize=12)


# # Task 1.3: Linear Model
# ## Generating Training Data

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

# generate random arrays of 500 values

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

# stack random values into 500 state vectors

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

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


# ## Finding the Least-Squares Fit

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

cache["C_large_deviations"] = C


# ## Evaluating the Linear Estimator
# ## Plotting Predicted State Change Against Target State Change

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

XC = X @ C
```

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

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

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

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

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

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

    ax.set_title( titles[i] )

fig.text(0.5, 0.95, 'Predicted State Changes vs. Target State Changes', ha='center', va='center', fontsize=14)
fig.text(0.5, 0.04, 'Target State Change', ha='center', va='center', fontsize=12)
fig.text(0.06, 0.5, 'Predicted State Change', ha='center', va='center', rotation='vertical', fontsize=12)



# ## Linear Model for Small Deflections Only
# generate random arrays of 500 values

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

# stack random values into 500 state vectors

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

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


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


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

XC = X @ C

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

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

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

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

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

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

    ax.set_title( titles[i] )

fig.text(0.5, 0.95, 'Predicted State Changes vs. Target State Changes', ha='center', va='center', fontsize=14)
fig.text(0.5, 0.04, 'Target State Change', ha='center', va='center', fontsize=12)
fig.text(0.06, 0.5, 'Predicted State Change', ha='center', va='center', rotation='vertical', fontsize=12)


# ## Comparing Prediction Contour Plots to Target Contour Plots

C = cache["C_large_deviations"]

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

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

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

initial_cart_positions  = np.array( [2] )
```

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

# create array of initial state vectors

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

# get 2d array of subsquent state vectors

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

cache["linear_predicted_changes_varying_angle_and_angvel"] = predicted_changes


state_changes = cache["state_changes_varying_angle_and_angvel"]

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

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

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

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

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

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

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


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

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

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

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

# create array of initial state vectors

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

# get 2d array of subsquent state vectors

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

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


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

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

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

    ax.imshow( predicted_changes[:,:,i], interpolation="bicubic", extent=(-10, 10, -15, 15), aspect='auto',
cmap="cool", origin='lower' )
```

```
    target_contour = ax.contour( initial_states[0,:,1], initial_states[:,0,3], state_changes[:,:,i],
colors="black", linewidths=1 )

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

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

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


# # Task 1.4: Prediction of System Evolution

# ## Small Oscillations

actual_states = cache["states_small_oscillations"]

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

# small oscillations about stable equilibrium

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

prediction_states = []

for i in range(80):

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


prediction_states = np.array( prediction_states )

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

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

ax.set_xlim( -1.3, 3 )
ax.set_ylim( -7, 9 )

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

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

ax.add_collection( linecollection )

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

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

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


actual_states = cache["states_large_oscillations"]

# large oscillations about stable equilibrium

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

x, y = [], []

for i in range(1000):

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

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


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

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

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

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

ax.add_collection( linecollection )

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

states = np.array( states )


# even larger oscillations

rollout_cartpole.reset()

state = np.array( [0, 4, np.pi, 20] )

x, y = [], []

for i in range( 300 ):

    pole_angle = state[2] % (2*np.pi)


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

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


fig, ax = plt.subplots(1, 1, num=24)
sf3utility.setup_phase_portrait( ax )
ax.axvline( x=np.pi, color="black" )

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

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

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

ax.add_collection( linecollection )

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

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

Appendix C – sf3utility.py

```
import matplotlib.pyplot as plt


def setup_phase_portrait( ax ):

    # show the grid
    ax.grid( visible = True, zorder=1 )

    ax.axhline( color="black", zorder=2 )
    ax.axvline( color="black", zorder=2 )
```