SF3 Interim Report

Chengyuan Li
cl854
Queens' College

May 2023

Introduction

In this report, we will be presenting the initial results obtained from simulating a system consisting of a pendulum attached to a cart. We will also investigate the usefulness of a linear model in predicting the behaviour of the system. The report will be divided into sections as per the tasks in the project handout.

Task 1.1

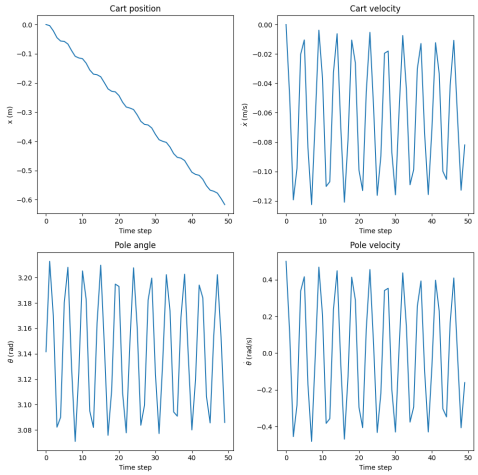

Figure 1: Time evolution of the system when it is set to oscillate around the stable equilibrium

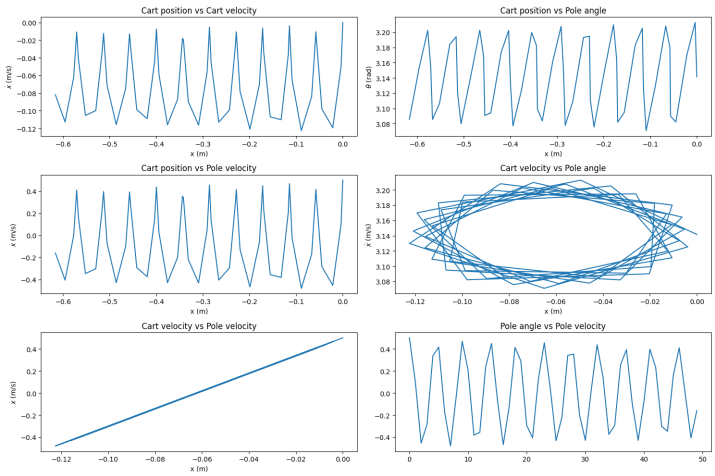

Figure 2: Different state variables of the system plotted against each other, when it is set to oscillate around the stable equilibrium

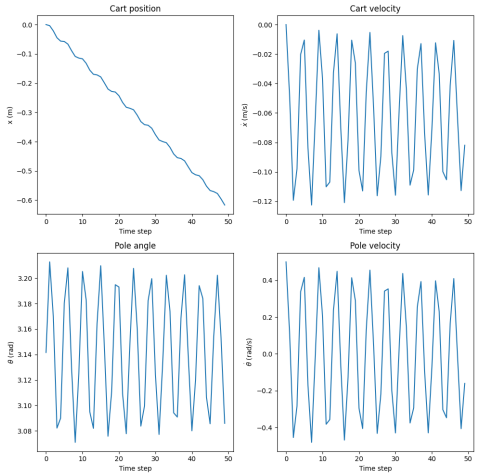

Figure 3: Time evolution of the system when the pendulum rotates completely

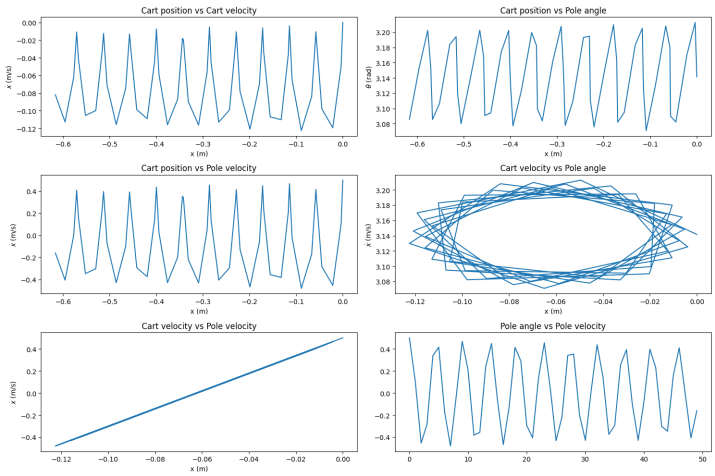

Figure 4: Different state variables of the system plotted against each other, when the pendulum rotates completely

The state of the system can be characterised by the vector $[x, \dot{x}, \theta, \dot{\theta}]$, where x is the position of the cart and θ is the angle that the pendulum makes with respect to the upright position. By specifying the initial conditions of the system, we can make the pendulum undergo different types of behaviours. The two behaviours we are interested in here are oscillation around the stable equilibrium (initial state = [0,0,π,0.5]), and complete rotation of the pendulum (initial state = [0,0,π,15]).

The top left graph on figures 1 and 3 shows that even though there is no initial cart velocity, the movement of the pendulum causes a net force on the system, causing the cart to move with net momentum one direction as well (average velocity equal to the offset of the cart velocity plot). In the absence of external forces, all three of the other state variables oscillates about some equilibrium.

Task 1.2

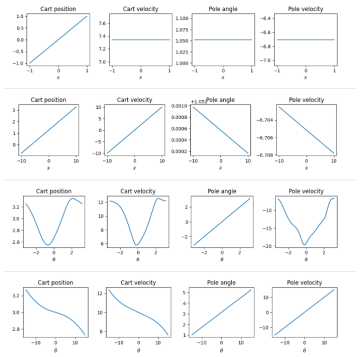

Figure 5: Sweep across a suitable range for each of the 4 state variables, plotting the value in one variable after a single call to performAction(), keeping the other variables constant

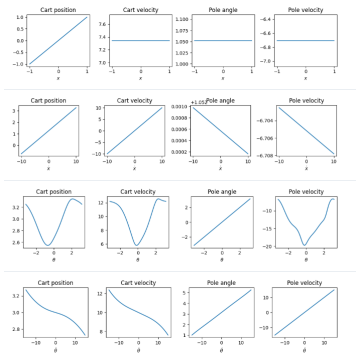

Figure 6: Sweep across a suitable range for each of the 4 state variables, plotting the change in one state variable after a single call to performAction(), while keeping the other variables constant

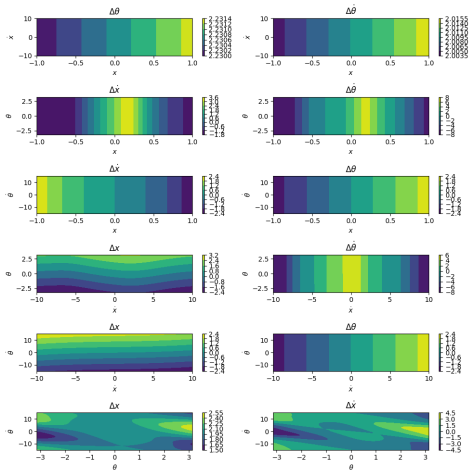

Figure 7: A series of contour plots, generated by keeping two of the state variables constant, while measuring the change in another.

The series of plots in figure 5 give the value of each of the state variables after sweeping a single call to the update function `performAction()` - the function which models the evolution of the system in time. The data is collected by sweeping each state variable across a corresponding suitable range, while keeping all other variables constant. As expected, the relationship of each variable with itself after evolving in time is linear. Looking at the different rows, we can clearly see, that the relationship between the cart location have no effect on the other state variables, and all 4 variables share a linear relationship with cart velocity. When the pole angle and the pole angular velocity are varied, however, the plots begin to exhibit varying degrees of non-linearity.

Figure 6 mirrored the approach of the first, but with a critical difference in the data being collected. Instead of focusing on the final state of each variable, we shifted our attention to the changes each state variable underwent after the invocation of the `performAction()` function. This resulted in another 16 plots illustrating the dynamics of the system. We can see that the shape of the plots are the same as those of figure 6, but the scale is reduced, as can be seen by the values on the y-axis.

Figure 7 shows a series of contour plots, the results of which further reinforces the linearity of the relationships between the variables shown previously. In each plot, two of the variables are being varied, while third variable gets evaluated while the final variable remains constant. The contour

formed by the pole angle and angular velocity shows the highest degree of irregularity - showing multiple local maximums and minimums in the given range, while all other plots show a consistent form, in the sense that there are easily identifiable trends in the values of the measured variable on the contour.

Task 1.3

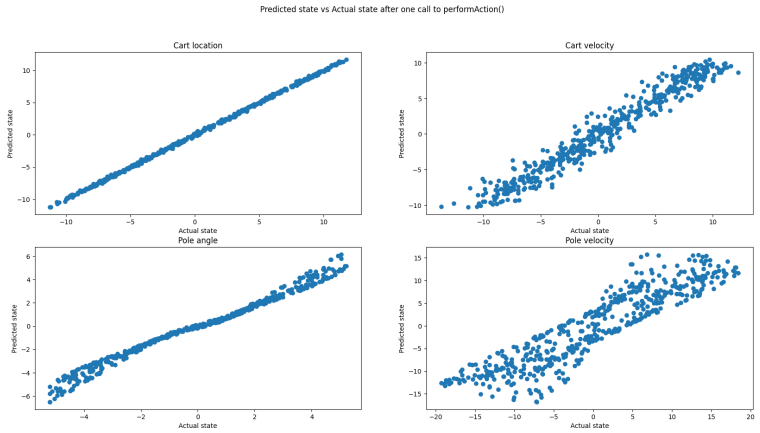

Figure 8: The actual values plotted against the predicted values using the linear model

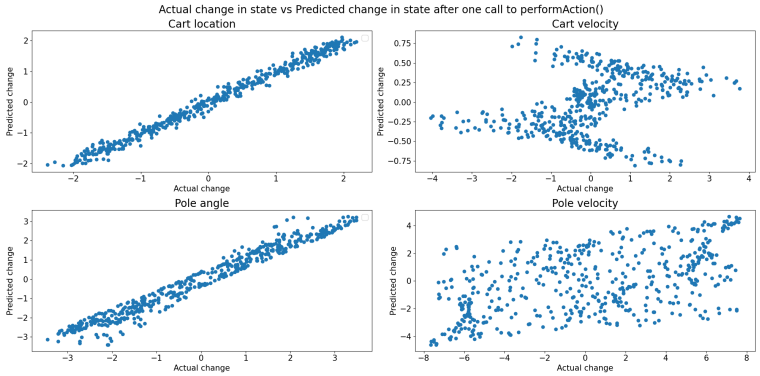

Figure 9: The actual changes in values plotted against the predicted values using the linear model

In this section, we investigated the performance of the linear model for modelling the state of the system over time. The target Y is assumed to be a linear function of the current state X: $f(X) = CX$, where C is a 4×4 matrix of coefficients.

First we generated an array of 500 random states, denoted with the vector X, and computed the target vector Y, which is equal to the input after running `performAction()` once. Next, using the `numpy.linalg.lstsq` function, we were able to calculate the optimal coefficient matrix, which we then used to compute predictions for a large number of different X values within the suitable range. The actual Y for these X variables are also computed and stored in an array. The actual Y of the four state variables are then plotted against their corresponding predictions using the linear model, and shown in figure 8. The better the prediction, the more straight the line. We can see that the linear model best predicts cart location, and do a decent job for the pole angle. But the plots for the cart

velocity and and pole velocities are a lot noisier.

Next, we change the target variable Y to equal the change in the state vector after calling `performAction()`. The resulting plot is given in figure 9. We see that as before, the model predicts positions much better than velocities, but it is noisier for all four state variables, especially the velocities, which has hardly any semblance of a linear relationship.

We draw the conclusion that the linear model is better suited for linear models, which is a zeroth order derivative of time. The higher the order of the derivative, the more non-linear the relationship, which makes the linear model a poor fit. This is also the reason why the model performed worse across the board for the difference in state variables (figure 9), as taking the difference between time steps has a similar effect to taking the derivative. The model is also less suitable for the state of the pole, as there are more factors which govern its motion such as gravity, complicating the relationship.

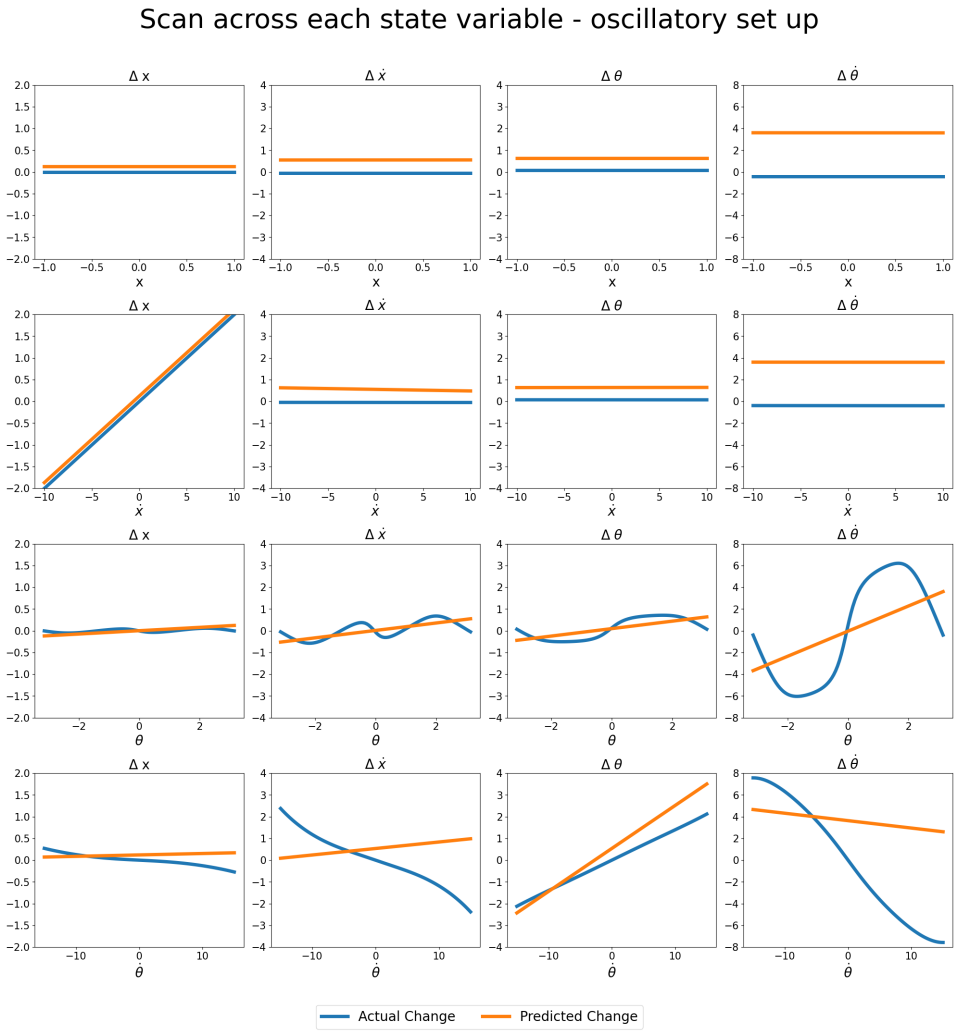

Figure 10: Repeat of the one variable sweep in section 1.2, with the system oscillating about its stable equilibrium

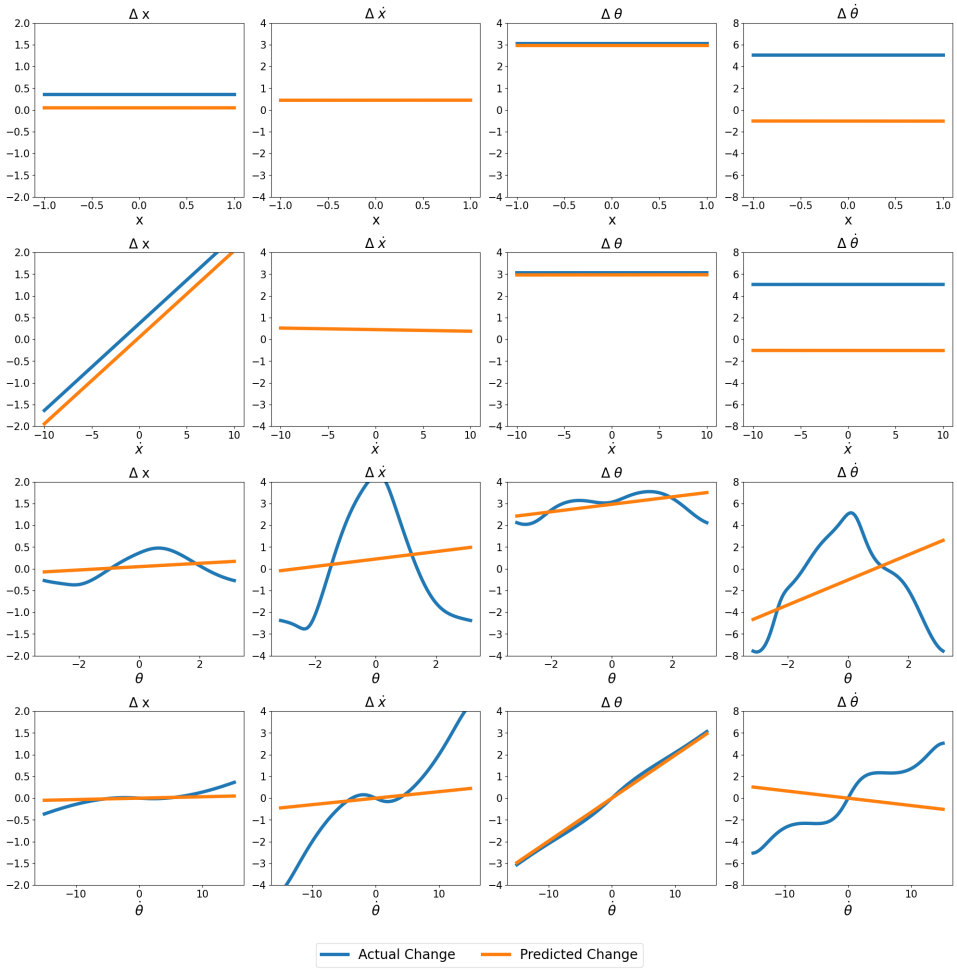

Figure 11: Repeat of the variable sweep in section 1.2, with the pendulum undergoing complete rotations

In figure 10 and 11, we repeated the single variable scan which we conducted in Task 1.2, producing two more grids of 16 plots. Figure 10 uses initial conditions that lead to the system oscillating, whereas figure 11 uses initial conditions that lead to the pendulum undergoing complete rotations. All plots have the prediction of the linear model overlayed on top of it. We can see that for scans of cart position and velocity, which takes a linear form, the model is generally a good fit, though still prone to error as is the case when $\dot{\theta}$ or $\Delta\dot{\theta}$ is being measured. For the plots where θ or $\dot{\theta}$ is across the horizontal axis, the plots are generally curved, and the linear model proves inadequate.

Task 1.4

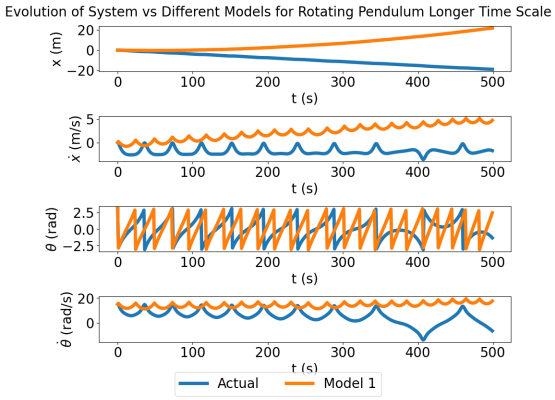

Figure 12: Prediction of time evolution of the system state vs prediction of time evolution using the linear model

Here, we test the model's ability at performing its true task - how it performs in predicting the time evolution of the system, rather than how well it matches with data already collected. The time series is mapped out over 2500 different data-points, and is shown in figure 12 for each state variable. Note that the linear model is applied at each time step, hence the overall evolution of the system is not constrained to a mere straight line.

We note that as time goes on, the time prediction for cart position and velocity diverges from the data collected from the actual simulation. For pole angle, and model is able to track the the roughly periodic trajectory of the collected data, though being noticeably out of sync. The performance for predicting pole velocity was poor, with the period of the model being different to that of the actual simulation.

Appendix

.1 Task 1.1 code

```
    import numpy as np
import matplotlib.pyplot as plt
from CartPole import CartPole

cp = CartPole()

n = 50

initial_state_simple_osc = [0, 0, np.pi, 0.5]
initial_state_complete_rot = [0, 0, np.pi, 15]

# Oscillation around the equilibrium

def rollout(initial_state):
    cp.setState(initial_state_simple_osc)
    pos = []
    velo = []
    pole_pos = []
    pole_velo = []
```

```
for _ in range(n):
    pos.append(cp.cart_location)
    velo.append(cp.cart_velocity)
    pole_pos.append(cp.pole_angle)
    pole_velo.append(cp.pole_velocity)
    cp.performAction()

times = np.arange(0,len(pos),1)

plt.figure(figsize=(10,10))
plt.subplot(2,2,1)
plt.plot(times, pos)
plt.xlabel('Time step')
plt.ylabel('x (m)')
plt.title("Cart position")
plt.subplot(2,2,2)
plt.plot(times, velo)
plt.xlabel('Time step')
plt.ylabel(r'$\dot{x}$ (m/s)')
plt.title("Cart velocity")
plt.subplot(2,2,3)
plt.plot(times, pole_pos)
plt.xlabel('Time step')
plt.ylabel(r'$\theta$ (rad)')
plt.title("Pole angle")
plt.subplot(2,2,4)
plt.plot(times, pole_velo)
plt.xlabel('Time step')
plt.ylabel(r'$\dot{\theta}$ (rad/s)')
plt.title("Pole velocity")
plt.tight_layout()
plt.show()

plt.figure(figsize=(15,10))
plt.subplot(3,2,1)
plt.plot(pos, velo)
plt.title("Cart position vs Cart velocity")
plt.xlabel("x (m)")
plt.ylabel(r"$\dot{x}$ (m/s)")
plt.subplot(3,2,2)
plt.plot(pos, pole_pos)
plt.title("Cart position vs Pole angle")
plt.xlabel("x (m)")
plt.ylabel(r"$\theta$ (rad)")
plt.subplot(3,2,3)
plt.plot(pos, pole_velo)
plt.title("Cart position vs Pole velocity")
plt.xlabel("x (m)")
plt.ylabel(r"$\dot{x}$ (m/s)")
plt.subplot(3,2,4)
plt.plot(velo, pole_pos)
plt.title("Cart velocity vs Pole angle")
plt.xlabel("x (m)")
```

```
    plt.ylabel(r"$\dot{x}$ (m/s)")
    plt.subplot(3,2,5)
    plt.plot(velo, pole_velo)
    plt.title("Cart velocity vs Pole velocity")
    plt.xlabel("x (m)")
    plt.ylabel(r"$\dot{x}$ (m/s)")
    plt.subplot(3,2,6)
    plt.plot(times, pole_velo)
    plt.title("Pole angle vs Pole velocity")
    plt.tight_layout()
    plt.show()


rollout(initial_state_simple_osc)
rollout(initial_state_complete_rot)
```

.2 Task 1.2 code

```
    import random

# Y defined here as the output of PerformAction with X as the input

# Using [-10, 10] as suitable range for cart position

cp = CartPole()
n = 50

c_location = random.uniform(-10,10)
c_velocity = random.uniform(-10,10)
p_angle = random.uniform(-np.pi, np.pi)
p_velocity = random.uniform(-15,15)
state = [c_location, c_velocity, p_angle, p_velocity]


def sweep1(var):
    l_list = []
    v_list = []
    pa_list = []
    pv_list = []
    cp.setState(state)
    # sweep across cart position

    label = 0

    if var == 1:
        x_range = np.linspace(-1,1,n)
        for x in x_range:
            state[0] = x
            cp.setState(state)
            cp.performAction()
            l_list.append(state[0])
            v_list.append(cp.cart_velocity)
            pa_list.append(cp.pole_angle)
            pv_list.append(cp.pole_velocity)
```

```
        label = r'$x$'
# sweep across cart velocity
if var == 2:
    x_range = np.linspace(-10,10,n)
    for x in x_range:
        state[1] = x
        cp.setState(state)
        cp.performAction()
        l_list.append(cp.cart_location)
        v_list.append(state[1])
        pa_list.append(cp.pole_angle)
        pv_list.append(cp.pole_velocity)
        label = r'$\dot{x}$'
# sweep across pole angle
if var == 3:
    x_range = np.linspace(-np.pi,np.pi,n)
    for x in x_range:
        state[2] = x
        cp.setState(state)
        cp.performAction()
        l_list.append(cp.cart_location)
        v_list.append(cp.cart_velocity)
        pa_list.append(state[2])
        pv_list.append(cp.pole_velocity)
        label = r'$\theta$'
# sweep across pole velocity
if var == 4:
    x_range = np.linspace(-15,15,n)
    for x in x_range:
        state[3] = x
        cp.setState(state)
        cp.performAction()
        l_list.append(cp.cart_location)
        v_list.append(cp.cart_velocity)
        pa_list.append(cp.pole_angle)
        pv_list.append(state[3])
        label = r'$\dot{\theta}$'
plt.figure(figsize=(10,2.5))
plt.subplot(1,4,1)
plt.plot(x_range, l_list)
plt.xlabel(label)
plt.title("Cart position")
plt.subplot(1,4,2)
plt.plot(x_range,v_list)
plt.xlabel(label)
plt.title("Cart velocity")
plt.subplot(1,4,3)
plt.plot(x_range,pa_list)
plt.xlabel(label)
plt.title("Pole angle")
plt.subplot(1,4,4)
plt.plot(x_range, pv_list)
plt.xlabel(label)
plt.title("Pole velocity")
```

```
    plt.tight_layout()
    plt.show()


sweep1(1)
sweep1(2)
sweep1(3)
sweep1(4)

c_location = random.uniform(-10,10)
c_velocity = random.uniform(-10,10)
p_angle = random.uniform(-np.pi, np.pi)
p_velocity = random.uniform(-15,15)
state = [c_location, c_velocity, p_angle, p_velocity]

def sweep2(var):
    l_list = []
    v_list = []
    pa_list = []
    pv_list = []
    cp.setState(state)
    # sweep across cart position
    if var == 1:
        x_range = np.linspace(-1,1,n)
        for x in x_range:
            state[0] = x
            cp.setState(state)
            prev_cl = state[0]
            prev_cv = state[1]
            prev_pl = state[2]
            prev_pv = state[3]
            cp.performAction()
            l_list.append(cp.cart_location - x)
            v_list.append(cp.cart_velocity - prev_cv)
            pa_list.append(cp.pole_angle - prev_pl)
            pv_list.append(cp.pole_velocity - prev_pv)
    # sweep across cart velocity
    if var == 2:
        x_range = np.linspace(-10,10,n)
        for x in x_range:
            state[1] = x
            cp.setState(state)
            prev_cl = state[0]
            prev_cv = state[1]
            prev_pl = state[2]
            prev_pv = state[3]
            cp.performAction()
            l_list.append(cp.cart_location - prev_cl)
            v_list.append(cp.cart_velocity - x)
            pa_list.append(cp.pole_angle - prev_pl)
            pv_list.append(cp.pole_velocity - prev_pv)
    # sweep across pole angle
    if var == 3:
        x_range = np.linspace(-np.pi,np.pi,n)
```

```
        for x in x_range:
            state[2] = x
            cp.setState(state)
            prev_cl = state[0]
            prev_cv = state[1]
            prev_pl = state[2]
            prev_pv = state[3]
            cp.performAction()
            l_list.append(cp.cart_location - prev_cl)
            v_list.append(cp.cart_velocity - prev_cv)
            pa_list.append(cp.pole_angle - x)
            pv_list.append(cp.pole_velocity - prev_pv)
    # sweep across pole velocity
    if var == 4:
        x_range = np.linspace(-15,15,n)
        for x in x_range:
            state[3] = x
            cp.setState(state)
            prev_cl = state[0]
            prev_cv = state[1]
            prev_pl = state[2]
            prev_pv = state[3]
            cp.performAction()
            l_list.append(cp.cart_location - prev_cl)
            v_list.append(cp.cart_velocity - prev_cv)
            pa_list.append(cp.pole_angle - prev_pl)
            pv_list.append(cp.pole_velocity - x)

    plt.figure(figsize=(10,2.5))
    plt.subplot(1,4,1)
    plt.plot(x_range, l_list)
    plt.title("Cart position", pad=20)
    plt.subplot(1,4,2)
    plt.plot(x_range,v_list)
    plt.title("Cart velocity",pad=20)
    plt.subplot(1,4,3)
    plt.plot(x_range,pa_list)
    plt.title("Pole angle", pad=20)
    plt.subplot(1,4,4)
    plt.plot(x_range, pv_list)
    plt.title("Pole velocity", pad=20)
    plt.tight_layout()
    plt.show()

sweep2(1)
sweep2(2)
sweep2(3)
sweep2(4)

import matplotlib.tri as tri

# List of pairs of meshgrids:
# cl - cv
# cl - pa
```

```
# cl - pv
# cv - pa
# cv - pv
# pa - pv

cp = CartPole()
n = 50

c_location = random.uniform(-10,10)
c_velocity = random.uniform(-10,10)
p_angle = random.uniform(-np.pi, np.pi)
p_velocity = random.uniform(-15,15)
state = [c_location, c_velocity, p_angle, p_velocity]


cl_range = np.linspace(-1,1,50)
cv_range = np.linspace(-10,10,50)
pa_range = np.linspace(-np.pi,np.pi,50)
pv_range = np.linspace(-15,15,50)


Y_cl_range = np.zeros((50,50))
Y_cv_range = np.zeros((50,50))
Y_pa_range = np.zeros((50,50))
Y_pv_range = np.zeros((50,50))

plt.figure(figsize=(10,10))


# Pair 1: cl - cv

for i, cl in enumerate(cl_range):
    for j, cv in enumerate(cv_range):
        state[0] = cl
        state[1] = cv
        cp.setState(state)
        cp.performAction()

        new_state = cp.getState()
        Y_cl_range[i][j] = new_state[0] - state[0]
        Y_cv_range[i][j] = new_state[1] - state[1]
        Y_pa_range[i][j] = new_state[2] - state[2]
        Y_pv_range[i][j] = new_state[3] - state[3]



meshed_cl, meshed_cv = np.meshgrid(cl_range,cv_range)
meshed_cl = meshed_cl.flatten()
meshed_cv = meshed_cv.flatten()
Y_cl_range = Y_cl_range.flatten()
Y_cv_range = Y_cv_range.flatten()
Y_pa_range = Y_pa_range.flatten()
Y_pv_range = Y_pv_range.flatten()
triang_1 = tri.Triangulation(meshed_cl, meshed_cv)
```

```
plt.subplot(6,2,1)
plt.tricontourf(triang_1, Y_pa_range)
plt.xlabel(r'$x$')
plt.ylabel(r'$\dot{x}$')
plt.title(r'$\Delta\theta$')
plt.colorbar()

plt.subplot(6,2,2)
plt.tricontourf(triang_1, Y_pv_range)
plt.xlabel(r'$x$')
plt.ylabel(r'$\dot{x}$')
plt.title(r'$\Delta \dot{\theta}$')
plt.colorbar()


# Pair 2: cl - pa

Y_cl_range = np.zeros((50,50))
Y_cv_range = np.zeros((50,50))
Y_pa_range = np.zeros((50,50))
Y_pv_range = np.zeros((50,50))
cl_range = np.linspace(-1,1,50)
cv_range = np.linspace(-10,10,50)
pa_range = np.linspace(-np.pi,np.pi,50)
pv_range = np.linspace(-15,15,50)

for i, cl in enumerate(cl_range):
    for j, pa in enumerate(pa_range):
        state[0] = cl
        state[2] = pa
        cp.setState(state)
        cp.performAction()

        new_state = cp.getState()
        Y_cl_range[i][j] = new_state[0] - state[0]
        Y_cv_range[i][j] = new_state[1] - state[1]
        Y_pa_range[i][j] = new_state[2] - state[2]
        Y_pv_range[i][j] = new_state[3] - state[3]


meshed_cl, meshed_pa = np.meshgrid(cl_range,pa_range)
meshed_cl = meshed_cl.flatten()
meshed_pa = meshed_pa.flatten()
Y_cl_range = Y_cl_range.flatten()
Y_cv_range = Y_cv_range.flatten()
Y_pa_range = Y_pa_range.flatten()
Y_pv_range = Y_pv_range.flatten()
triang_2 = tri.Triangulation(meshed_cl, meshed_pa)

plt.subplot(6,2,3)
plt.tricontourf(triang_2, Y_cv_range)
plt.xlabel(r'$x$')
```

```
plt.ylabel(r'$\theta$')
plt.title(r'$\Delta \dot{x}$')
plt.colorbar()

plt.subplot(6,2,4)
plt.tricontourf(triang_2, Y_pv_range)
plt.xlabel(r'$x$')
plt.ylabel(r'$\theta$')
plt.title(r'$\Delta \dot{\theta}$')
plt.colorbar()

# Pair 3: cl - pv

Y_cl_range = np.zeros((50,50))
Y_cv_range = np.zeros((50,50))
Y_pa_range = np.zeros((50,50))
Y_pv_range = np.zeros((50,50))
cl_range = np.linspace(-1,1,50)
cv_range = np.linspace(-10,10,50)
pa_range = np.linspace(-np.pi,np.pi,50)
pv_range = np.linspace(-15,15,50)

for i, cl in enumerate(cl_range):
    for j, pv in enumerate(pv_range):
        state[0] = cl
        state[3] = pv
        cp.setState(state)
        cp.performAction()

        new_state = cp.getState()
        Y_cl_range[i][j] = new_state[0] - state[0]
        Y_cv_range[i][j] = new_state[1] - state[1]
        Y_pa_range[i][j] = new_state[2] - state[2]
        Y_pv_range[i][j] = new_state[3] - state[3]


meshed_cl, meshed_pv = np.meshgrid(cl_range,pv_range)
meshed_cl = meshed_cl.flatten()
meshed_pv = meshed_pv.flatten()
Y_cl_range = Y_cl_range.flatten()
Y_cv_range = Y_cv_range.flatten()
Y_pa_range = Y_pa_range.flatten()
Y_pv_range = Y_pv_range.flatten()
triang_3 = tri.Triangulation(meshed_cl, meshed_pv)

plt.subplot(6,2,5)
plt.tricontourf(triang_3, Y_cv_range)
plt.xlabel(r'$x$')
plt.ylabel(r'$\dot{\theta}$')
plt.title(r'$\Delta \dot{x}$')
plt.colorbar()

plt.subplot(6,2,6)
plt.tricontourf(triang_3, Y_pa_range)
```

```
plt.xlabel(r'$x$')
plt.ylabel(r'$\dot{\theta}$')
plt.title(r'$\Delta \theta$')
plt.colorbar()

# Pair 4: cv - pa

Y_cl_range = np.zeros((50,50))
Y_cv_range = np.zeros((50,50))
Y_pa_range = np.zeros((50,50))
Y_pv_range = np.zeros((50,50))
cl_range = np.linspace(-1,1,50)
cv_range = np.linspace(-10,10,50)
pa_range = np.linspace(-np.pi,np.pi,50)
pv_range = np.linspace(-15,15,50)

for i, cv in enumerate(cv_range):
    for j, pa in enumerate(pa_range):
        state[1] = cv
        state[2] = pa
        cp.setState(state)
        cp.performAction()

        new_state = cp.getState()
        Y_cl_range[i][j] = new_state[0] - state[0]
        Y_cv_range[i][j] = new_state[1] - state[1]
        Y_pa_range[i][j] = new_state[2] - state[2]
        Y_pv_range[i][j] = new_state[3] - state[3]


meshed_cv, meshed_pa = np.meshgrid(cv_range,pa_range)
meshed_cv = meshed_cv.flatten()
meshed_pa = meshed_pa.flatten()
Y_cl_range = Y_cl_range.flatten()
Y_cv_range = Y_cv_range.flatten()
Y_pa_range = Y_pa_range.flatten()
Y_pv_range = Y_pv_range.flatten()
triang_4 = tri.Triangulation(meshed_cv, meshed_pa)

plt.subplot(6,2,7)
plt.tricontourf(triang_4, Y_cl_range)
plt.xlabel(r'$\dot{x}$')
plt.ylabel(r'$\theta$')
plt.title(r'$\Delta x$')
plt.colorbar()

plt.subplot(6,2,8)
plt.tricontourf(triang_4, Y_pv_range)
plt.xlabel(r'$\dot{x}$')
plt.ylabel(r'$\theta$')
plt.title(r'$\Delta \dot{\theta}$')
plt.colorbar()

# Pair 5: cv - pv
```

```
Y_cl_range = np.zeros((50,50))
Y_cv_range = np.zeros((50,50))
Y_pa_range = np.zeros((50,50))
Y_pv_range = np.zeros((50,50))
cl_range = np.linspace(-1,1,50)
cv_range = np.linspace(-10,10,50)
pa_range = np.linspace(-np.pi,np.pi,50)
pv_range = np.linspace(-15,15,50)

for i, cv in enumerate(cv_range):
    for j, pv in enumerate(pv_range):
        state[1] = cv
        state[3] = pv
        cp.setState(state)
        cp.performAction()

        new_state = cp.getState()
        Y_cl_range[i][j] = new_state[0] - state[0]
        Y_cv_range[i][j] = new_state[1] - state[1]
        Y_pa_range[i][j] = new_state[2] - state[2]
        Y_pv_range[i][j] = new_state[3] - state[3]


meshed_cv, meshed_pv = np.meshgrid(cv_range,pv_range)
meshed_cv = meshed_cv.flatten()
meshed_pv = meshed_pv.flatten()
Y_cl_range = Y_cl_range.flatten()
Y_cv_range = Y_cv_range.flatten()
Y_pa_range = Y_pa_range.flatten()
Y_pv_range = Y_pv_range.flatten()
triang_5 = tri.Triangulation(meshed_cv, meshed_pv)

plt.subplot(6,2,9)
plt.tricontourf(triang_5, Y_cl_range)
plt.xlabel(r'$\dot{x}$')
plt.ylabel(r'$\dot{\theta}$')
plt.title(r'$\Delta x$')
plt.colorbar()

plt.subplot(6,2,10)
plt.tricontourf(triang_5, Y_pa_range)
plt.xlabel(r'$\dot{x}$')
plt.ylabel(r'$\dot{\theta}$')
plt.title(r'$\Delta \theta$')
plt.colorbar()

# Pair 6: pa - pv

Y_cl_range = np.zeros((50,50))
Y_cv_range = np.zeros((50,50))
Y_pa_range = np.zeros((50,50))
Y_pv_range = np.zeros((50,50))
cl_range = np.linspace(-1,1,50)
```

```
cv_range = np.linspace(-10,10,50)
pa_range = np.linspace(-np.pi,np.pi,50)
pv_range = np.linspace(-15,15,50)

for i, pa in enumerate(pa_range):
    for j, pv in enumerate(pv_range):
        state[2] = pa
        state[3] = pv
        cp.setState(state)
        cp.performAction()

        new_state = cp.getState()
        Y_cl_range[i][j] = new_state[0] - state[0]
        Y_cv_range[i][j] = new_state[1] - state[1]
        Y_pa_range[i][j] = new_state[2] - state[2]
        Y_pv_range[i][j] = new_state[3] - state[3]


meshed_pa, meshed_pv = np.meshgrid(pa_range,pv_range)
meshed_pa = meshed_pa.flatten()
meshed_pv = meshed_pv.flatten()
Y_cl_range = Y_cl_range.flatten()
Y_cv_range = Y_cv_range.flatten()
Y_pa_range = Y_pa_range.flatten()
Y_pv_range = Y_pv_range.flatten()
triang_6 = tri.Triangulation(meshed_pa, meshed_pv)

plt.subplot(6,2,11)
plt.tricontourf(triang_6, Y_cl_range)
plt.xlabel(r'$\theta$')
plt.ylabel(r'$\dot{\theta}$')
plt.title(r'$\Delta x$')
plt.colorbar()

plt.subplot(6,2,12)
plt.tricontourf(triang_6, Y_cv_range)
plt.xlabel(r'$\theta$')
plt.ylabel(r'$\dot{\theta}$')
plt.title(r'$\Delta\dot{x}$')
plt.colorbar()


plt.tight_layout()
plt.show()
```

.3 Task 1.3 code

```
import numpy as np
import matplotlib.pyplot as plt
from cartpole import CartPole

# Define all functions here
def generate_random_state(x_range, x_dot_range, theta_range, theta_dot_range):
    return np.array([np.random.uniform(*x_range), np.random.uniform(*x_dot_range),
```

```
                    np.random.uniform(*theta_range), np.random.uniform(*theta_dot_range)])

def generate_data(number_of_points, delta_time, sim_steps, generate_next_state_randomly, start_state,
                  x_range, x_dot_range, theta_range, theta_dot_range, remap_angle):

    past_state = np.zeros((number_of_points, 4))
    future_state = np.zeros((number_of_points, 4))
    cartpole = CartPole()
    cartpole.delta_time = delta_time
    cartpole.sim_steps = sim_steps

    if generate_next_state_randomly:
        np.random.seed(0)
        for i in range(number_of_points):
            cartpole.setState(generate_random_state(x_range, x_dot_range, theta_range, theta_dot_rang
            current_state = cartpole.getState()
            if remap_angle:
                current_state[2] = remap_angle(current_state[2])
            past_state[i] = current_state
            cartpole.performAction(0)
            new_state = cartpole.getState()
            if remap_angle:
                new_state[2] = remap_angle(new_state[2])
            future_state[i] = new_state
    else:
        for i in range(number_of_points):
            current_state = start_state if i == 0 else future_state[i-1]
            if remap_angle:
                current_state[2] = remap_angle(current_state[2])
            past_state[i] = current_state
            cartpole.setState(past_state[i])
            cartpole.performAction(0)
            new_state = cartpole.getState()
            if remap_angle:
                new_state[2] = remap_angle(new_state[2])
            future_state[i] = new_state

    change = future_state - past_state
    return past_state, change

def get_model(past_state, change):
    return np.linalg.lstsq(past_state, change, rcond=None)[0]

def predict_change(state, model):
    return state @ model

def predict_state(state, model, remap_theta=True):
    change = predict_change(state, model)
    new_state = state + change
    if remap_theta:
        new_state[2] = remap_angle(new_state[2])
    return new_state

# Main code starts here
```

```
number_of_points = 500
delta_time = 0.2
sim_steps = 50
generate_next_state_randomly = True
x_range = (-10, 10)
x_dot_range = (-10, 10)
theta_range = (-np.pi, np.pi)
theta_dot_range = (-15, 15)
remap_angle = False
start_state = None

past_state, change = generate_data(number_of_points, delta_time, sim_steps, generate_next_state_rando
                  x_range, x_dot_range, theta_range, theta_dot_range, remap_angle)

small_time_step_model = get_model(past_state, change)
print(small_time_step_model)

# Continue from previous part

future = past_state + change
model_predictions = predict_change(past_state, small_time_step_model)
predictions = model_predictions.T
future_predictions = past_state + model_predictions

def generate_plot(ax, future, future_predictions, label, xlabel, ylabel, title):
    ax.scatter(future, future_predictions, label=label)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.title.set_text(title)

fig, ax = plt.subplots(2, 2, figsize=(20, 10))

generate_plot(ax[0, 0], future[:, 0], future_predictions[:, 0], 'x', 'Actual state', 'Predicted state
generate_plot(ax[0, 1], future[:, 1], future_predictions[:, 1], 'x_dot', 'Actual state', 'Predicted s
generate_plot(ax[1, 0], future[:, 2], future_predictions[:, 2], 'theta', 'Actual state', 'Predicted s
generate_plot(ax[1, 1], future[:, 3], future_predictions[:, 3], 'theta_dot', 'Actual state', 'Predict

fig.suptitle('Predicted state vs Actual state after one call to performAction()')
plt.show()

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
fig.suptitle("Actual change in state vs Predicted change in state after one call to performAction()",

generate_plot(ax1, change[:, 0], predictions[0], None, 'Actual change', 'Predicted change', 'Cart loc
generate_plot(ax2, change[:, 1], predictions[1], None, 'Actual change', 'Predicted change', 'Cart vel
generate_plot(ax3, change[:, 2], predictions[2], None, 'Actual change', 'Predicted change', 'Pole ang
generate_plot(ax4, change[:, 3], predictions[3], None, 'Actual change', 'Predicted change', 'Pole vel

for ax in (ax1, ax2, ax3, ax4):
    ax.tick_params(axis='both', which='major', labelsize=15)
    ax.set_xlabel("Actual change", fontsize=15)
    ax.set_ylabel("Predicted change", fontsize=15)

plt.tight_layout()
```

```
plt.show()

def initialize_system(number_of_search_points, state_constant, model,
                      state_range=((-1, 1), (-10, 10), (-np.pi, np.pi), (-15, 15)), delta_time=0.2, s

    initial_state_case_1 = np.full((4, number_of_search_points), state_constant)
    initial_state = np.stack([initial_state_case_1] * 4, axis=0)

    for idx in range(4):
        initial_state[idx, idx, :] = np.linspace(state_range[idx][0], state_range[idx][1], number_of_

    change_in_state = np.zeros_like(initial_state)
    model_change_in_state = np.zeros_like(initial_state)

    return initial_state, change_in_state, model_change_in_state


def compute_change(number_of_search_points, model, initial_state, change_in_state, model_change_in_st
    cartpole = CartPole()
    cartpole.delta_time = delta_time
    cartpole.sim_steps = sim_steps

    for i in range(4):
        for j in range(number_of_search_points):
            cartpole.setState(initial_state[i, :, j])
            cartpole.performAction(0)
            change_in_state[i, :, j] = cartpole.getState() - initial_state[i, :, j]
            model_change_in_state[i, :, j] = model.predict_change(initial_state[i, :, j])

    return change_in_state, model_change_in_state


def single_variable_scan(number_of_search_points, state_constant, model,
                         state_range=((-1, 1), (-10, 10), (-np.pi, np.pi), (-15, 15)), delta_time=0.2
    initial_state, change_in_state, model_change_in_state = initialize_system(number_of_search_points
    change_in_state, model_change_in_state = compute_change(number_of_search_points, model, initial_s

    return initial_state, change_in_state, model_change_in_state

# Plot functions
def plot_change(initial_state, change_in_state, model_change_in_state, fig=None, axs=None, case="", l
    if axs is None:
        fig, axs = plt.subplots(4, 4)
    fig.suptitle('Change in state variables when one state variable is varied', fontsize=30, y = 1.5)
    state_variables = ["x", "$\\dot{x}$", "$\\theta$", "$\\dot{\\theta}$"]
    for i in range(4):
        x_axis = initial_state[i, i, :]
        for j in range(4):
            y_actual = change_in_state[i, j, :]
            y_predicted = model_change_in_state[i, j, :]
            axs[i, j].plot(x_axis, y_actual, label="Actual Change", linewidth=5, linestyle=linestyle1
            axs[i, j].plot(x_axis, y_predicted, label="Predicted Change", linewidth=5, linestyle=line
            axs[i, j].set_title(f"$\\Delta$ {state_variables[j]}", fontsize=20)
            axs[i, j].set_xlabel(state_variables[i], fontsize=20)
```

```
            axs[i, j].set_ylabel(None, fontsize=20)
            axs[i, j].tick_params(axis='both', which='major', labelsize=15)
    return axs

def plot_difference_in_change(initial_state, change_in_state, model_change_in_state, fig=None, axs=No
    if axs is None:
        fig, axs = plt.subplots(4, 4)
    fig.suptitle('Difference between Actual Change and Pricted Change', fontsize=30)
    state_variables = ["x", "$\\dot{x}$", "$\\theta$", "$\\dot{\\theta}$"]
    for i in range(4):
        x_axis = initial_state[i, i, :]
        for j in range(4):
            y_actual = change_in_state[i, j, :]
            y_predicted = model_change_in_state[i, j, :]
            difference = y_actual - y_predicted
            axs[i, j].plot(x_axis, difference, label="Difference" + case, linewidth=5, linestyle=line
            axs[i, j].set_title(f"$\\Delta$ {state_variables[j]}", fontsize=20)
            axs[i, j].set_xlabel(state_variables[i], fontsize=20)
            axs[i, j].set_ylabel(None, fontsize=20)
            axs[i, j].tick_params(axis='both', which='major', labelsize=15)
    return axs

number_of_train_points = 1000
number_of_points = 100
state_constant = (0, 0, np.pi, 0)
delta_time = 0.2
sim_steps = 50

model = LinearModel(number_of_points=number_of_train_points, delta_time=delta_time, sim_steps=sim_ste

linear_initial_state, linear_change_in_state, linear_model_change_in_state = single_variable_scan(
    number_of_points, state_constant, model, delta_time=delta_time, sim_steps=sim_steps)

oscillatory_initial_state, oscillatory_change_in_state, oscillatory_model_change_in_state = single_va
    number_of_points, state_constant, model, delta_time=delta_time, sim_steps=sim_steps)

rotation_initial_state, rotation_change_in_state, rotation_model_change_in_state = single_variable_sc
    number_of_points, state_constant, model, delta_time=delta_time, sim_steps=sim_steps)

fig, axs = plt.subplots(4, 4, figsize=(20, 20), layout="constrained")
plot_change(oscillatory_initial_state, oscillatory_change_in_state, oscillatory_model_change_in_state

# Have to choose the range of ticks for the y axis to be the same for each column
for ax in axs:
    ax[0].set_ylim([-2, 2])
    ax[1].set_ylim([-4, 4])
    ax[2].set_ylim([-4, 4])
    ax[3].set_ylim([-8, 8])

handles, labels = plt.gca().get_legend_handles_labels()
fig.legend(handles, labels, loc='lower center', bbox_to_anchor=(0.5, -0.05), fontsize=20, ncol=3)
fig.suptitle('Scan across each state variable - oscillation about stable equilibirum', fontsize=40,y=
plt.tight_layout()
plt.show()
```

```
fig, axs = plt.subplots(4, 4, figsize=(20, 20), layout="constrained")
plot_change(rotation_initial_state, rotation_change_in_state, rotation_model_change_in_state, fig, ax

for ax in axs:
    ax[0].set_ylim([-2, 2])
    ax[1].set_ylim([-4, 4])
    ax[2].set_ylim([-4, 4])
    ax[3].set_ylim([-8, 8])

handles, labels = plt.gca().get_legend_handles_labels()
fig.legend(handles, labels, loc='lower center', bbox_to_anchor=(0.5, -0.05), fontsize=20, ncol=3)
fig.suptitle('Scan across each state variable - complete rotation of pendulum', fontsize=40,y=1.02)
plt.tight_layout()
plt.show()
```

```
    class CartPoleControl:
    def __init__(self, initial_state, force, time_steps, remap_angle=False, delta_time=0.2, sim_steps
        self.initial_state = initial_state
        self.force = force
        self.time_steps = time_steps
        self.remap_angle = remap_angle
        self.t = np.arange(0, time_steps, 1)
        self.state = np.zeros((time_steps, 4))
        self.state[0] = initial_state
        self.time_multiplier = delta_time * sim_steps
        self.remapped = np.zeros(time_steps)
        self.delta_time = delta_time
        self.sim_steps = sim_steps

    # The rest of the class remains unchanged

# Initialize
number_of_train_points = 2500
delta_time = 0.02
sim_steps = 50
oscillatory_cartpole_initial_state = np.array([0, 0, np.pi, 15])
number_of_points= 300

def zero_force(state):
    return 0

# Create model
random_model = LinearModel()

# Train model
random_model.train_model(train_initial_state, train_change_in_state)

# Create control and do simulation
oscillating_cartpole_pole_control = CartPoleControl(initial_state=oscillatory_cartpole_initial_state,
                                                    force=zero_force,
                                                    remap_angle=True,
                                                    time_steps=number_of_points,
                                                    delta_time=delta_time,
                                                    sim_steps=sim_steps)
oscillating_cartpole_pole_control.do_simulation()
```

```
# Perform prediction
past_state, change = get_data_iteratively(oscillatory_cartpole_initial_state, random_model, number_of

# Duplicate past state and add noise
repeated_past_state = np.repeat(past_state, 100, axis=0)
noise_mean = 0
noise_std = 0.001
noise = np.random.normal(noise_mean, noise_std, repeated_past_state.shape)
new_past_state = repeated_past_state + noise
new_change = random_model.predict_change(new_past_state, add_noise=True, noise_mean=noise_mean, noise

# Create and train new model
new_model = LinearModel()
new_model.train_model(new_past_state, new_change)

# Predict states
oscillating_cartpole_states = predict_states(random_model, oscillatory_cartpole_initial_state, number
oscillating_cartpole_iterative_control = predict_states(oscillatory_cartpole_iterative_model, oscilla
new_control = predict_states(new_model, oscillatory_cartpole_initial_state, number_of_points, remap_t

# Predict changes
predicted_change = random_model.predict_change(past_state)
new_predicted_change = new_model.predict_change(past_state)

# Plot
fig, axs = plt.subplots(4, 1, figsize=(10, 8))
oscillating_cartpole_pole_control.plot_four_quantities(axs, label='Actual')
time = oscillating_cartpole_pole_control.t * oscillating_cartpole_pole_control.time_multiplier
for i in range(4):
    axs[i].plot(time, oscillating_cartpole_states[:, i], label='Model 1', linewidth=5, linestyle='sol
    handles, labels = axs[i].get_legend_handles_labels()
fig.legend(handles, labels, loc='lower center', bbox_to_anchor=(0.5, -0.05),  ncol=4, fontsize=20)
fig.suptitle("Linear Model for predicting", fontsize=20)
plt.tight_layout()
```