ES3J1 Advanced Systems and Software Engineering

Python Assignment for 2023-2024

Question 1

Deriving Second-Order ODE

Using Newton's Second Law of Motion: $\sum F = ma$.

Find sum of forces on the drone, comprising of ocean flow plus control force: $\sum F = \gamma(v_b - v) + u$. Acceleration: $a = \frac{dq^2}{dt^2}$. Mass of drone = m

Substitute into Newton's Second Law of Motion, we get:

$$\gamma(v_b - v) + u = m\frac{dq^2}{dt^2} \tag{1}$$

Substitue in values for v_b & v and converting into term of x:

$$\gamma(\beta R_{\frac{3\pi}{4}} q - v) + u = m\frac{dq^2}{dt^2} \tag{2}$$

Reformulating into Linear First-Order ODE (state-space model)

Starting with the form:

$$\dot{\mathrm{x}} = Ax + Bu \tag{3}$$

Adding rotation matrix to †, $f = \gamma(\beta Rq - v)$, where R =

$$\begin{bmatrix} \frac{-\sqrt{2}}{2} & \frac{-\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{-\sqrt{2}}{2} \end{bmatrix} \tag{2-2}$$

Bottom left element of matrix A is then $\gamma\beta R$ which gives:

$$\begin{bmatrix} \frac{-5\sqrt{2}}{2} & \frac{-5\sqrt{2}}{2} \\ \frac{5\sqrt{2}}{2} & \frac{-5\sqrt{2}}{2} \end{bmatrix} \tag{2-2}$$

Bottom right element of matrix A is just the top right element given in assignment brief multiplied by $-\gamma$, so $\gamma m^{-1} I_{2x2}$ which gives:

$$\begin{bmatrix} \frac{-2.5}{m} & 0 \\ 0 & \frac{-2.5}{m} \end{bmatrix} \tag{2-2}$$

Both top row elements are given for matrix A in brief, so A is:

$$\begin{bmatrix} 0 & 0 & \frac{1}{m} & 0 \\ 0 & 0 & 0 & \frac{1}{m} \\ \frac{-5\sqrt{2}}{2} & \frac{-5\sqrt{2}}{2} & \frac{-2.5}{m} & 0 \\ \frac{5\sqrt{2}}{2} & \frac{-5\sqrt{2}}{2} & 0 & \frac{-2.5}{m} \end{bmatrix} \tag{2-2}$$

Seeing as the control force, u is only relevant to the force component of this state space model, matrix B is simply:

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{2-2}$$

So the overall state space model would look like this:

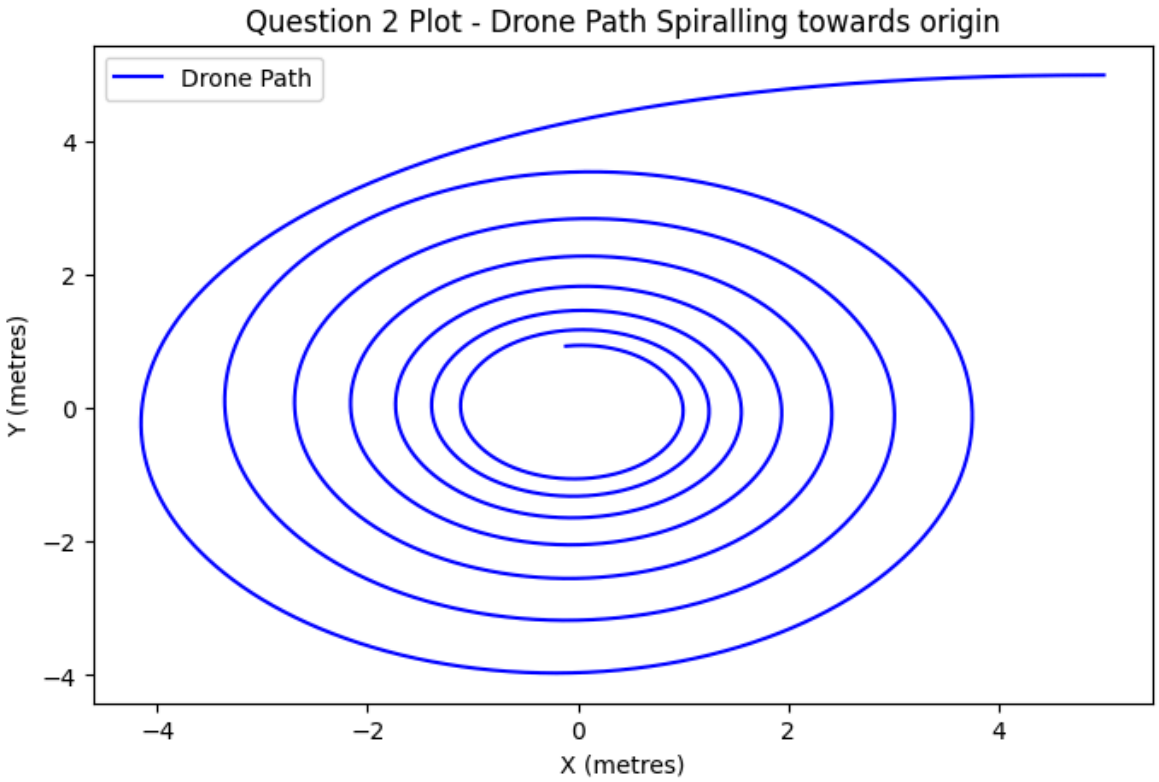
$$\begin{bmatrix} \dot{q} \\ \dot{p} \end{bmatrix} = \begin{bmatrix} 0 & 0 & \frac{1}{m} & 0 \\ 0 & 0 & 0 & \frac{1}{m} \\ \frac{-5\sqrt{2}}{2} & \frac{-5\sqrt{2}}{2} & \frac{-2.5}{m} & 0 \\ \frac{5\sqrt{2}}{2} & \frac{-5\sqrt{2}}{2} & 0 & \frac{-2.5}{m} \end{bmatrix} \begin{bmatrix} q \\ p \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} u \tag{2-2}$$

Question 2

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

def solve_ivp_be(A, b, x0, T, u):
    #implicit euler function
    X = [None for ti in T]
    X[0] = x0
    d = len(x0)
    for i in range(len(T))[1:]:
        Delta_t = T[i] - T[i-1]
        X[i] = np.linalg.solve(np.eye(d) - Delta_t * A(T[i]), X[i-1] - Delta_t *
    return T, X

def decay_A(t):
    return A

def decay_b(t):
    return b
    #b matrix could also be substituted for a zero matrix of same size, seeing a

u = np.zeros(2)

A = [[0.0, 0.0, 2.0/3.0, 0.0],
    [0.0, 0.0, 0.0, 2.0/3.0],
    [(-5.0*np.sqrt(2.0))/2.0,(-5.0*np.sqrt(2.0))/2.0, -2.5/1.5, 0.0],
    [(5.0*np.sqrt(2.0))/2.0,(-5.0*np.sqrt(2.0))/2.0, 0.0, -2.5/1.5]]
A = np.array(A) #defining A matrix as used in previous question

b = [[0,0], [0,0], [1,0], [0,1]]
```

```
b  = np.array(b) #defining b as used in previous question

x0 = np.array([5.0, 5.0, 0.0, 0.0]) #starting position for drone
t_min = 0.0
t_max = 30.0 #time range
dt = 0.001 #time step
T = np.arange(t_min, t_max + dt, dt)
T, X = solve_ivp_be(decay_A, decay_b, x0, T, u) #takes inputs and computes the i

column1 = np.array([row[0] for row in X]) #extracting the first two columns of '
column2 = np.array([row[1] for row in X])
plt.figure(figsize=(8, 5))
plt.plot(column1, column2, label = 'Drone Path', color = 'b')
plt.title("Question 2 Plot - Drone Path Spiralling towards origin")
plt.xlabel("X (metres)")
plt.ylabel("Y (metres)")
plt.legend()
```

Out[ ]: <matplotlib.legend.Legend at 0x291eb336350>



Implicit Euler scheme has been implemented for the numerical solution of the ODE with constant time step, dt = 0.001. Matrices A & B are passed into the function from the previous question's state space model, along with the drone starting position, the time series vector and motor force (which is zero at this stage). The resulting plot shows the drone does indeed follow a path spiralling inwards towards origin and drifts within 2m of the origin within around 15s.

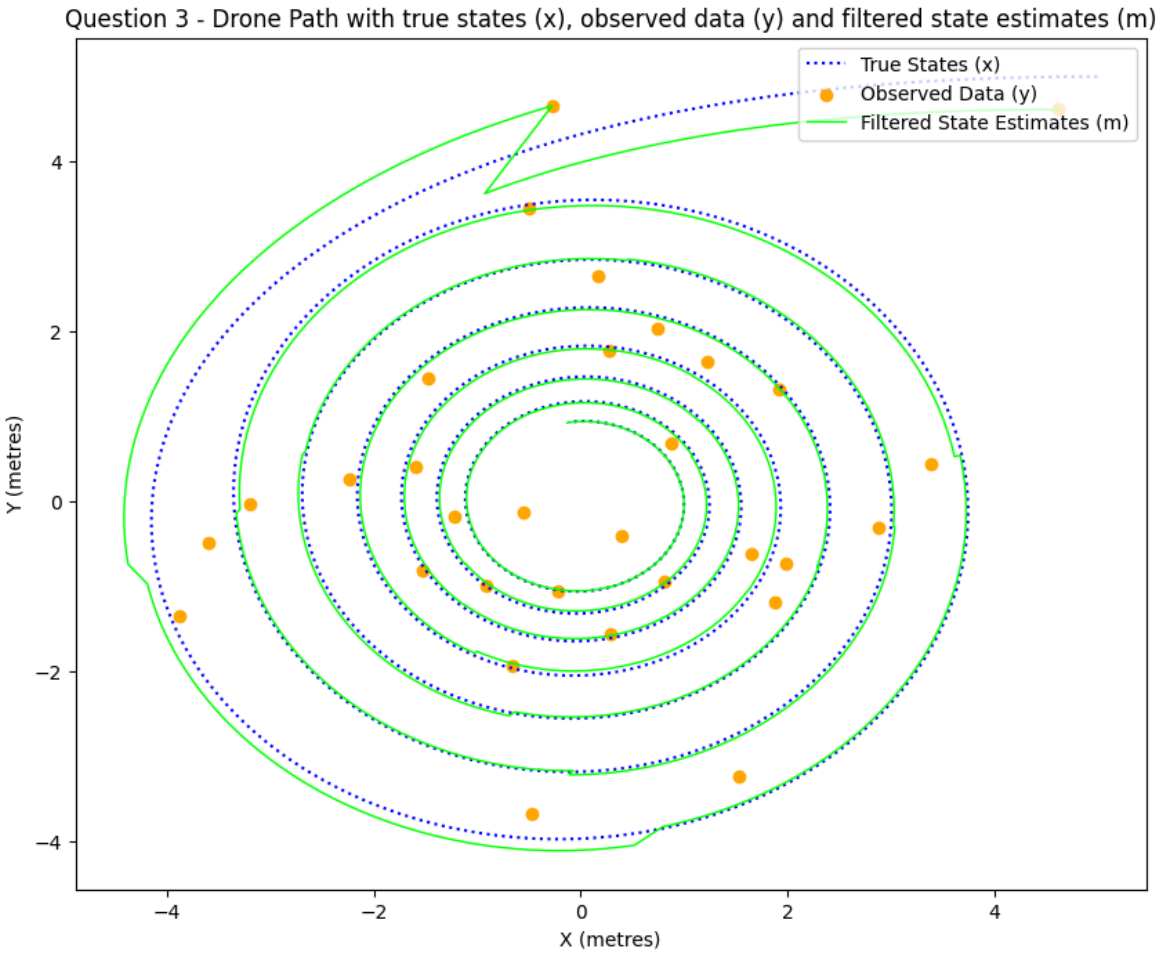
Question 3

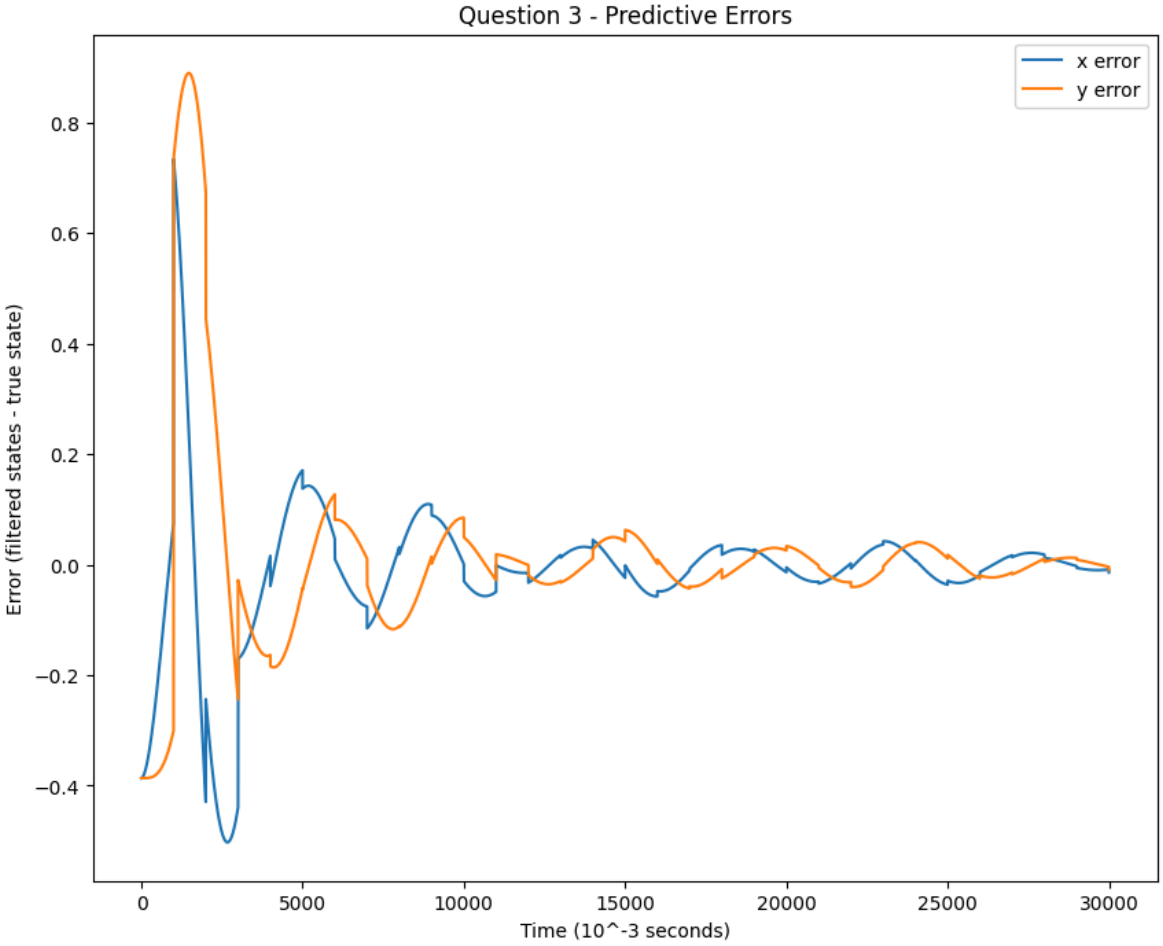
In [ ]:
```
import scipy.linalg as splin
import matplotlib.pyplot as plt
import random
np.random.seed(2128009)
```

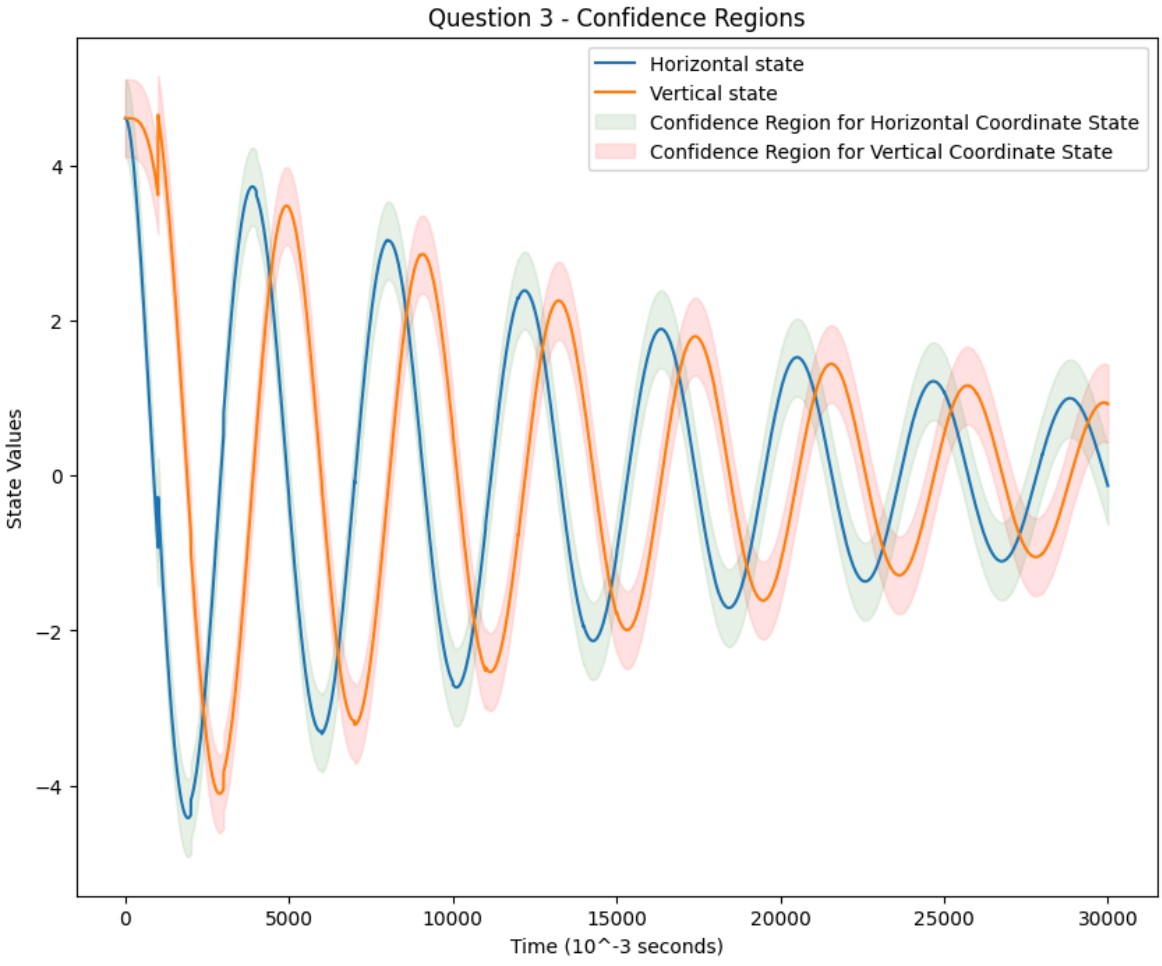
```
#x_true - reorganising drone path from question 2
x_true = np.zeros((30001, 4))
x_true_int = np.zeros((31,4)) #integer states

for i in range(0, 30001): # for all time states
    x_true[i] = X[i]
x_true = x_true.T

j = 0
for i in range(0, 30001, 1000): #for time integer states
    x_true_int[j] = X[i]
    j = j + 1
x_true_int = x_true_int.T

H = np.array([[1.0, 0.0, 0.0, 0.0],[0.0, 1.0, 0.0, 0.0]]) #H - observation matri
sigma = 0.5 #standard deviation of 0.5
C = 10**6 * np.eye(4) #highly agnostic state covariance
m = np.zeros((4,30001))
F = splin.expm(dt*A) # discretising A
Gamma = sigma * sigma * np.eye(2)
y = H @ x_true_int + sigma * np.random.normal(size=(31)) #noisy observation

j = 1
K = C @ H.T @ np.linalg.inv(H @ C @ H.T + Gamma) #Kalman Gain
m[:,0] = m[:,0] + K @ (y[:,0] - H @ m[:,0]) #initial value for mean
C = C - K @ H @ C #initial value for covariance

for i in range (1,30001):

    m[:,i] = F @ m[:,i-1]
    C = F @ C @ F.T
    # +fi and +Qi not needed at this stage because u = 0

    if i % 1000 == 0: #if an integer time has been reach a new kalman gain will
        K = C @ H.T @ np.linalg.inv(H @ C @ H.T + Gamma) #Kalman Gain
        m[:,i] = m[:,i] + K @ (y[:,j] - H @ m[:,i])
        C = C - K @ H @ C
        j = j + 1


columnxtrue1 = np.array([row[0] for row in X]) #true state
columnxtrue2 = np.array([row[1] for row in X])
columnxtrue3 = np.array([row[2] for row in X])
columnxtrue4 = np.array([row[3] for row in X])
plt.figure(figsize=(10, 8))
plt.plot(columnxtrue1,columnxtrue2, label = 'True States (x)', linestyle = 'dott

columny1 = y[0,:] #observed data
columny2 = y[1,:]
plt.scatter(columny1, columny2, label = 'Observed Data (y)', color = 'orange')

columnM1 = m[0,:] #state estimates
columnM2 = m[1,:]
columnM3 = m[2,:]
columnM4 = m[3,:]
plt.plot(columnM1, columnM2, label = 'Filtered State Estimates (m)', linewidth =


plt.title("Question 3 - Drone Path with true states (x), observed data (y) and f
```

```
plt.xlabel("X (metres)")
plt.ylabel("Y (metres)")
plt.legend(loc = 'upper right')

err1 = columnM1 - columnxtrue1 #calculating and plotting predictive error
err2 = columnM2 - columnxtrue2
err3 = columnM3 - columnxtrue3
err4 = columnM4 - columnxtrue4
plt.figure(figsize=(10, 8))
plt.plot(err1, label = 'x error' )
plt.plot(err2, label = 'y error')
plt.title("Question 3 - Predictive Errors")
plt.xlabel("Time (10^-3 seconds)")
plt.ylabel("Error (filtered states - true state)")
plt.legend(loc = 'upper right')

Txaxis = np.arange(30001)  #plotting confidence regions
plt.figure(figsize=(10, 8))
plt.plot(columnM1, label = 'Horizontal state')
plt.plot(columnM2, label = 'Vertical state')
plt.fill_between(Txaxis, columnM1 - sigma, columnM1 + sigma, alpha=0.1, color="f
plt.fill_between(Txaxis, columnM2 - sigma, columnM2 + sigma, alpha=0.1, color="r
plt.title("Question 3 - Confidence Regions")
plt.xlabel("Time (10^-3 seconds)")
plt.ylabel("State Values")
plt.legend(loc = 'upper right')
```

Out[ ]: <matplotlib.legend.Legend at 0x291910a4390>




Question 3 - Predictive Errors
x error
y error
Error (filtered states - true state)
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
0
5000
10000
15000
20000
25000
30000
Time (10^-3 seconds)


Question 3 - Confidence Regions
Horizontal state
Vertical state
Confidence Region for Horizontal Coordinate State
Confidence Region for Vertical Coordinate State
State Values
4
2
0
−2
−4
0
5000
10000
15000
20000
25000
30000
Time (10^-3 seconds)

The Kalman filter has been implemented with the given inputs, which only updates at each integer time. This means the system works with whatever estimate is current until a new estimate is made in between the integer time intervals. As time progresses the filtered state estimate become increasingly more accurate until is almost replicates the true state. The confidence region accounts for the predictive error, as the predictive error values never exceed ±1 standard deviation, gradually the predictive error also tends towards zero as time goes on.

Question 4

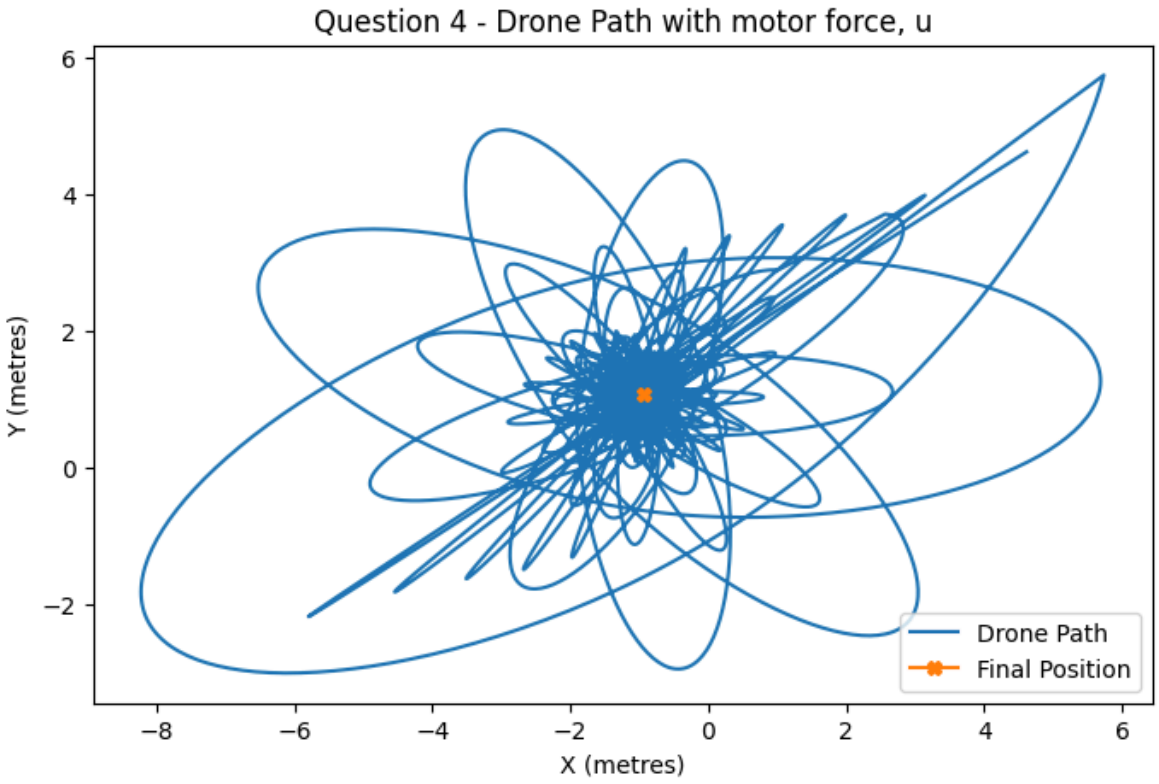
In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg as splin
import random
np.random.seed(2128009)

def solve_ivp_be(A, b, x0, T, u):
    #implicit euler function
    X = [None for ti in T]
    X[0] = x0
    d = len(x0)
    for i in range(len(T))[1:]:
        Delta_t = T[i] - T[i-1]
        X[i] = np.linalg.solve(np.eye(d) - Delta_t * A(T[i]), X[i-1] - Delta_t *
    return T, X

def decay_A(t):
    return A

def decay_b(t):
    return b

u = np.zeros(2)

A = [[0.0, 0.0, 2.0/3.0, 0.0],
    [0.0, 0.0, 0.0, 2.0/3.0],
    [(-5.0*np.sqrt(2.0))/2.0,(-5.0*np.sqrt(2.0))/2.0, -2.5/1.5, 0.0],
    [(5.0*np.sqrt(2.0))/2.0,(-5.0*np.sqrt(2.0))/2.0, 0.0, -2.5/1.5]]
A = np.array(A) #defining A matrix as used in previous question

b = [[0,0], [0,0], [1,0], [0,1]]
b  = np.array(b) #defining b

x0 = np.array([5.0, 5.0, 0.0, 0.0]) #starting position for drone
t_min = 0.0
t_max = 30.0 #time range
dt = 0.001 #time step
T = np.arange(t_min, t_max + dt, dt)
T, X = solve_ivp_be(decay_A, decay_b, x0, T, u)

#x_true
x_true = np.zeros((30001, 4))
x_true_int = np.zeros((31,4)) #integer states

for i in range(0, 30001): # for all time states
    x_true[i] = X[i]
```

```
x_true = x_true.T

j = 0
for i in range(0, 30001, 1000): #for time integer states
    x_true_int[j] = X[i]
    j = j + 1
x_true_int = x_true_int.T

H = np.array([[1.0, 0.0, 0.0, 0.0],[0.0, 1.0, 0.0, 0.0]]) #H - observation matri
sigma = 0.5 #standard deviation of 0.5
C = 10**6 * np.eye(4) #highly agnostic state covariance
m = np.zeros((4,30001))
F = splin.expm(dt*A)  # discretising A matrix
B = ((F - np.eye(4)) ** -1) @ b  # discretising B matrix
u = np.zeros((2,30001))
Q = 0
error = np.zeros((30001,2))
y = H @ x_true + sigma * np.random.normal(size=(30001))

j = 1
K = C @ H.T @ np.linalg.inv(H @ C @ H.T + Gamma) #Kalman Gain
m[:,0] = m[:,0] + K @ (y[:,0] - H @ m[:,0]) #initial value for mean
C = C - K @ H @ C #initial value for covariance

X = np.zeros((4,len(T)))
X[:,0] = np.array([5, 5, 0, 0])

for i in range (1,30001):

    m[:,i] = F @ m[:,i-1] + b @ u[:,i-1]
    C = F @ C @ F.T + Q

    if i % 1000 == 0: #if an integer time has been reach a new kalman gain will
        K = C @ H.T @ np.linalg.inv(H @ C @ H.T + Gamma) #Kalman Gain
        m[:,i] = m[:,i] + K @ (y[:,j] - H @ m[:,i])
        C = C - K @ H @ C
        j = j + 1

    # pd controller
    currx = m[0,i]
    curry = m[1,i]
    error[i,:] = np.array([-1, 1]) - [currx, curry]
    Kp = 500
    Kd = 0.5
    p = Kp * error[i,:]
    d = Kd * (error[i,:]- error[i-1,:])/dt
    u[:,i] = p + d

    #maximum magnitude constraint
    u_mag = np.sqrt(u[0,i]**2 + u[1,i]**2)
    if u_mag > 5:
       u = 5/u_mag * u

columnx1 = m[0,:]
columnx2 = m[1,:]
print("Final X position:")
print(currx)
print("Final Y position:")
print(curry)
```

```
plt.figure(figsize=(8, 5))
plt.plot(columnx1, columnx2, label = 'Drone Path')
plt.plot(currx, curry, label = 'Final Position', marker = 'X')
plt.title("Question 4 - Drone Path with motor force, u")
plt.xlabel("X (metres)")
plt.ylabel("Y (metres)")
plt.legend(loc = 'lower right')
```

```
Final X position:
-0.9423146580828206
Final Y position:
1.0861950560371512
```

Out[ ]: <matplotlib.legend.Legend at 0x291919a6a10>



At each time step the drone's current position is compared to the desired coordinates and an error is calculated. This is then used to calculate a new drone force using Kp and Kd values. If the magnitude of applied control force u is greater than the maximum constraint then it is limited by calculating a new u from $u = 5/u_{mag} * u$. The drone lands within 1 decimal place of desired coordinates, (-1,1), at the end of the simulation, so 1 d.p. could be an appropriate error tolerance. This also is the same error that the standard deviation gives, 0.5 either side of the dersired coordinates, so is the same as ±1 standard deviation.