ANA670 Applied Optimization
Assignment Week Three (125 points)

First Name	Randall
Last Name	Crawford
ID#	901012194
Email Address	rcc_0149@yahoo.com

Table of Contents

How to submit your Assignment

After filling all the parts in this file, please follow the following steps.

1) Add your name and ID to the first page.
2) Save the file in the original format (Docx or Doc)

   (please do not convert to other file formats e.g. PDF, ZIP, RAR, …).

3) Rename the file as

   *YOUR **First** Name* - *YOUR **Last** Name* - *ID* – ANA670 – ***HW3****.docx*

   Example: John - Smith - ID - ANA670 – HW3.docx

4) Upload the file and submit it (only using BrightSpace)

P1 [30 points]

Python Program

Write a program to simulate a two dimensional basic random walk on a two dimensional x, y coordinate plane, starting at the origin where step size is simply 1 in a random direction but with the following special constraint: **do not allow the next step to simply reverse the previous one**. In other words, if the previous step went from (3,2) to (3,3), the next step cannot go back to (3,2). Run it 100 times, each walk being 350 steps. *Although you do not need to turn any plots for this problem, plotting some random walks is a good way to confirm that your program works.*

Outline

1. Save max x travelled: Each step within a walk, check to see if the current step exceeded the all-time maximum absolute value of x distance travelled within this and all other paths. If so, keep this information.

2. Save max y travelled: Each step within a walk, check to see if the current step exceeded the all-time maximum absolute value of y distance travelled within this and all other paths. If so, keep this information.

3. **Output** (should be visible in your output report below): the coordinates of the final destination point that the walk happened to arrive at (there should be 100 of these outputs seen in your output report).

4. (Keep whatever information you will need to compute the mean and standard deviation of the destinations in the x dimension.)

5. (Also keep whatever information you will need to compute the mean and standard deviation of the destinations in the y dimension.

Output (I): absolute max x travelled in all random walks.

Output (II): absolute max y travelled in all random walks.

Output (III): The mean and standard deviation (here is an explanation with example) of all 100 of the destinations in the x dimension.

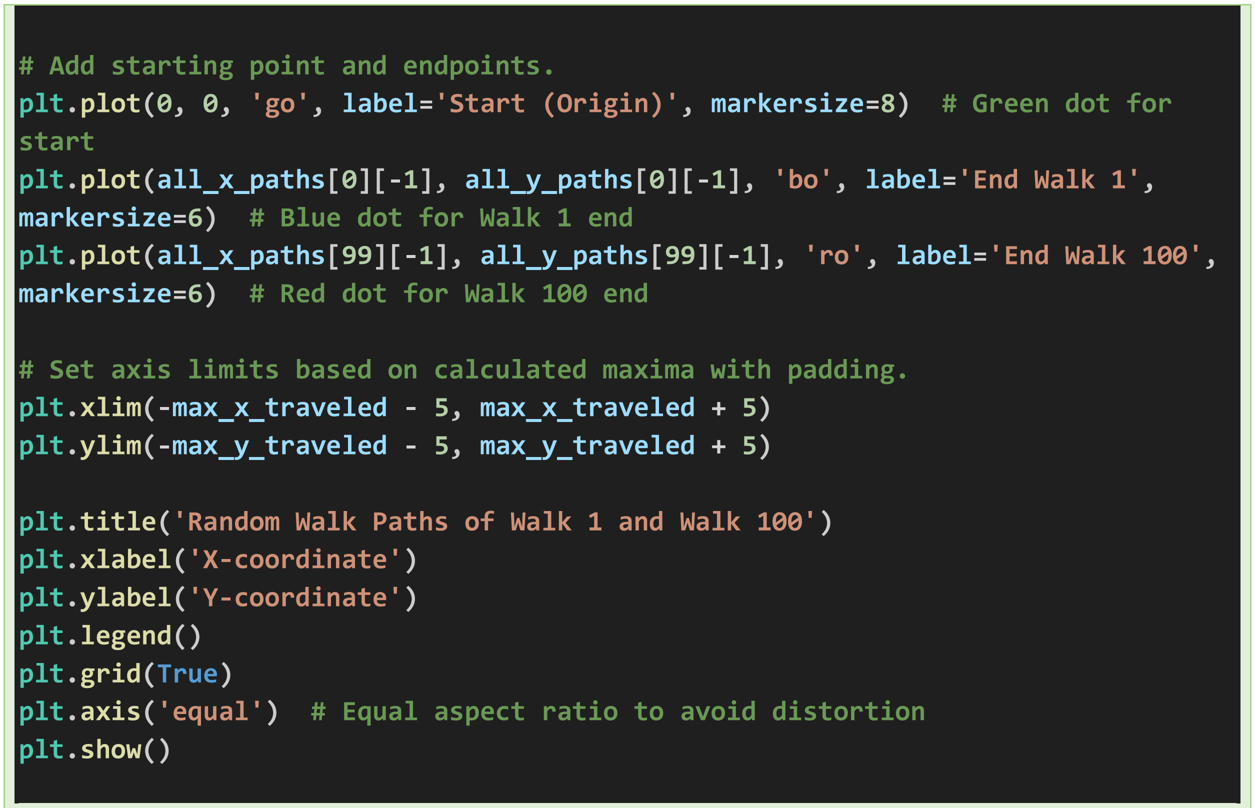
Output (IV): The mean and standard deviation of all 100 destinations in the y dimension.

(The usual rules apply: if it is unclear from the context [import statements] cite any code re-used from another source.)

Your Code

```python
import random
import math
import matplotlib.pyplot as plt

# Establish number of walks and steps per walk.
num_walks = 100
steps_per_walk = 350

# Create lists to store final destinations and paths.
x_destinations = []
y_destinations = []
all_x_paths = []  # List to hold all x-paths (one per walk)
all_y_paths = []  # List to hold all y-paths (one per walk)

# Possible moves (up, down, left, right).
moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]

for walk in range(num_walks):
    x_path = [0]  # Start at origin for this walk
    y_path = [0]
    x, y = 0, 0
    prev_x, prev_y = 0, 0

    for _ in range(steps_per_walk):
        # Get valid moves (exclude the reverse of the last move).
        valid_moves = [move for move in moves if not (move[0] == -(x - prev_x) and move[1] == -(y - prev_y))]
        dx, dy = random.choice(valid_moves)

        # Update position.
        prev_x, prev_y = x, y
        x += dx
        y += dy

        # Store path.
        x_path.append(x)
        y_path.append(y)

    # Store final destination and entire path for this walk.
    x_destinations.append(x)
    y_destinations.append(y)
    all_x_paths.append(x_path)  # Append the full x-path list
    all_y_paths.append(y_path)  # Append the full y-path list
```

```python
# Calculate the true maximum distances from all paths.
max_x_traveled = 0
max_y_traveled = 0
for i, (x_path, y_path) in enumerate(zip(all_x_paths, all_y_paths)):
    path_max_x = max(abs(x) for x in x_path)
    path_max_y = max(abs(y) for y in y_path)
    max_x_traveled = max(max_x_traveled, path_max_x)
    max_y_traveled = max(max_y_traveled, path_max_y)
    print(f"Walk {i + 1} - Max |x|: {path_max_x}, Max |y|: {path_max_y}, Final Point: ({x_path[-1]}, {y_path[-1]})")

# Calculate mean and standard deviation.
def mean(data):
    return sum(data) / len(data)

def std_dev(data, mean_val):
    variance = sum((x - mean_val) ** 2 for x in data) / len(data)
    return math.sqrt(variance)

x_mean = mean(x_destinations)
x_std = std_dev(x_destinations, x_mean)
y_mean = mean(y_destinations)
y_std = std_dev(y_destinations, y_mean)

# Output results.
print(f"Output (I): absolute max x traveled in all random walks: {max_x_traveled}")
print(f"Output (II): absolute max y traveled in all random walks: {max_y_traveled}")
print(f"Output (III): The mean and standard deviation of all 100 of the destinations in the x dimension: mean = {x_mean:.2f}, std dev = {x_std:.2f}")
print(f"Output (IV): The mean and standard deviation of all 100 destinations in the y dimension: mean = {y_mean:.2f}, std dev = {y_std:.2f}")

# Plotting the paths of Walk 1 and Walk 100.
plt.figure(figsize=(10, 6))
plt.plot(all_x_paths[0], all_y_paths[0], label='Walk 1', color='teal', linewidth=1)
plt.plot(all_x_paths[99], all_y_paths[99], label='Walk 100', color='red', linewidth=1)
```



```
# Add starting point and endpoints.
plt.plot(0, 0, 'go', label='Start (Origin)', markersize=8)  # Green dot for start
plt.plot(all_x_paths[0][-1], all_y_paths[0][-1], 'bo', label='End Walk 1', markersize=6)  # Blue dot for Walk 1 end
plt.plot(all_x_paths[99][-1], all_y_paths[99][-1], 'ro', label='End Walk 100', markersize=6)  # Red dot for Walk 100 end

# Set axis limits based on calculated maxima with padding.
plt.xlim(-max_x_traveled - 5, max_x_traveled + 5)
plt.ylim(-max_y_traveled - 5, max_y_traveled + 5)

plt.title('Random Walk Paths of Walk 1 and Walk 100')
plt.xlabel('X-coordinate')
plt.ylabel('Y-coordinate')
plt.legend()
plt.grid(True)
plt.axis('equal')  # Equal aspect ratio to avoid distortion
plt.show()
```

Stats report (fill in the blanks, should match program output)

Mean of the destinations in the x dimension: **-4.10**

Standard Deviation of the destinations in the x dimension: **18.01**

Mean of the destinations in the y dimension: **-1.60**

Standard Deviation of the destinations in the y dimension: **19.37**

Absolute max x traveled in all random walks: **54**

Absolute max y traveled in all random walks: **52**

Analysis: Comment on max distance traveled in each dimension, also the means and standard deviations (minimum 100 words). Did any of this surprise you, or was all as expected?

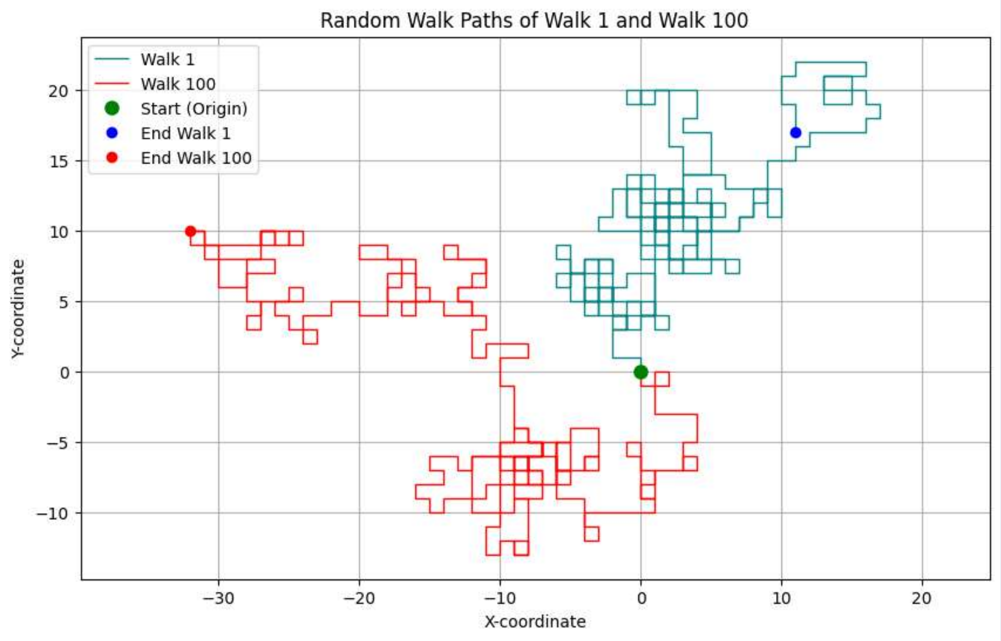
The maximum distances traveled (54 for x, 52 for y) indicate that the random walks explored significant portions of the plane, with some walks reaching far from the origin due to the 350 steps and the no-reverse constraint. This constraint likely prevented backtracking, allowing greater net movement and explaining the high maxima. The means (-4.10 for x, -1.60 for y) are close to zero, suggesting no strong directional bias, though the slight negativity might reflect random fluctuations or the constraint's subtle

influence. The standard deviations (18.01 for x, 19.37 for y) are consistent with a √350 (~18.7) diffusion model, indicating considerable variability in final positions. I was somewhat surprised that the standard deviations varied less than the means across runs, as I expected more consistency in spread given the fixed step count.

However, with 350 steps and the no-reverse rule, the destination points varied considerably, which aligns with the high standard deviations. The process's random nature, modified by the constraint, explains the balance between mean proximity to zero and the wide dispersion, making the results largely expected yet intriguing in their variability.

Program output report

```
Walk 1 - Max |x|: 17, Max |y|: 22, Final Point: (11, 17)
Walk 2 - Max |x|: 29, Max |y|: 17, Final Point: (-28, 10)
Walk 3 - Max |x|: 29, Max |y|: 38, Final Point: (-26, -36)
Walk 4 - Max |x|: 31, Max |y|: 16, Final Point: (-21, 15)
Walk 5 - Max |x|: 21, Max |y|: 37, Final Point: (21, 33)
Walk 6 - Max |x|: 29, Max |y|: 13, Final Point: (22, 12)
Walk 7 - Max |x|: 23, Max |y|: 28, Final Point: (-8, -14)
Walk 8 - Max |x|: 15, Max |y|: 37, Final Point: (11, -13)
Walk 9 - Max |x|: 17, Max |y|: 9, Final Point: (-6, 0)
Walk 10 - Max |x|: 19, Max |y|: 14, Final Point: (2, -10)
Walk 11 - Max |x|: 20, Max |y|: 9, Final Point: (18, -6)
Walk 12 - Max |x|: 21, Max |y|: 16, Final Point: (-12, -14)
Walk 13 - Max |x|: 13, Max |y|: 13, Final Point: (0, -8)
Walk 14 - Max |x|: 17, Max |y|: 19, Final Point: (-3, 11)
Walk 15 - Max |x|: 29, Max |y|: 28, Final Point: (-23, -25)
Walk 16 - Max |x|: 24, Max |y|: 18, Final Point: (-16, 16)
Walk 17 - Max |x|: 37, Max |y|: 35, Final Point: (27, -35)
Walk 18 - Max |x|: 31, Max |y|: 23, Final Point: (-22, 0)
Walk 19 - Max |x|: 20, Max |y|: 12, Final Point: (5, 9)
Walk 20 - Max |x|: 33, Max |y|: 40, Final Point: (-33, -39)
Walk 21 - Max |x|: 29, Max |y|: 14, Final Point: (5, 5)
Walk 22 - Max |x|: 15, Max |y|: 32, Final Point: (0, -22)
Walk 23 - Max |x|: 10, Max |y|: 14, Final Point: (6, 12)
Walk 24 - Max |x|: 21, Max |y|: 16, Final Point: (-14, -6)
Walk 25 - Max |x|: 15, Max |y|: 33, Final Point: (15, -27)
Walk 26 - Max |x|: 31, Max |y|: 20, Final Point: (23, -5)
Walk 27 - Max |x|: 16, Max |y|: 13, Final Point: (-6, 4)
Walk 28 - Max |x|: 22, Max |y|: 21, Final Point: (-21, -21)
Walk 29 - Max |x|: 27, Max |y|: 47, Final Point: (-17, 45)
Walk 30 - Max |x|: 16, Max |y|: 16, Final Point: (-14, -8)
Walk 31 - Max |x|: 17, Max |y|: 27, Final Point: (-7, 27)
Walk 32 - Max |x|: 19, Max |y|: 17, Final Point: (-16, -6)
Walk 33 - Max |x|: 24, Max |y|: 14, Final Point: (-20, 14)
Walk 34 - Max |x|: 24, Max |y|: 14, Final Point: (-24, 4)
Walk 35 - Max |x|: 21, Max |y|: 39, Final Point: (16, -38)
Walk 36 - Max |x|: 35, Max |y|: 29, Final Point: (-23, 19)
Walk 37 - Max |x|: 11, Max |y|: 16, Final Point: (-3, -15)
Walk 38 - Max |x|: 44, Max |y|: 21, Final Point: (-35, -11)
Walk 39 - Max |x|: 26, Max |y|: 52, Final Point: (-19, -47)
Walk 40 - Max |x|: 25, Max |y|: 16, Final Point: (-20, -6)
Walk 41 - Max |x|: 36, Max |y|: 26, Final Point: (-36, -14)
Walk 42 - Max |x|: 16, Max |y|: 12, Final Point: (-5, -7)
Walk 43 - Max |x|: 54, Max |y|: 19, Final Point: (-48, -12)
Walk 44 - Max |x|: 24, Max |y|: 15, Final Point: (4, 8)
Walk 45 - Max |x|: 17, Max |y|: 9, Final Point: (11, -7)
Walk 46 - Max |x|: 20, Max |y|: 10, Final Point: (12, 6)
Walk 47 - Max |x|: 12, Max |y|: 45, Final Point: (3, 41)
Walk 48 - Max |x|: 12, Max |y|: 26, Final Point: (-1, 7)
Walk 49 - Max |x|: 32, Max |y|: 31, Final Point: (22, -20)
Walk 50 - Max |x|: 20, Max |y|: 27, Final Point: (2, 26)
Walk 51 - Max |x|: 12, Max |y|: 12, Final Point: (-3, 1)
Walk 52 - Max |x|: 24, Max |y|: 9, Final Point: (-21, -5)
Walk 53 - Max |x|: 17, Max |y|: 29, Final Point: (15, -27)
Walk 54 - Max |x|: 17, Max |y|: 7, Final Point: (-14, 0)
Walk 55 - Max |x|: 13, Max |y|: 40, Final Point: (9, 37)
Walk 56 - Max |x|: 19, Max |y|: 28, Final Point: (16, -28)
Walk 57 - Max |x|: 17, Max |y|: 22, Final Point: (0, 4)
Walk 58 - Max |x|: 18, Max |y|: 27, Final Point: (18, 16)
Walk 59 - Max |x|: 13, Max |y|: 19, Final Point: (7, 3)
Walk 60 - Max |x|: 19, Max |y|: 21, Final Point: (-9, -21)
Walk 61 - Max |x|: 23, Max |y|: 18, Final Point: (-17, -3)
Walk 62 - Max |x|: 11, Max |y|: 27, Final Point: (-10, 8)
Walk 63 - Max |x|: 28, Max |y|: 12, Final Point: (-24, -2)
Walk 64 - Max |x|: 18, Max |y|: 36, Final Point: (-7, -33)
Walk 65 - Max |x|: 19, Max |y|: 21, Final Point: (4, 20)
Walk 66 - Max |x|: 20, Max |y|: 38, Final Point: (4, 34)
Walk 67 - Max |x|: 19, Max |y|: 35, Final Point: (19, 27)
Walk 68 - Max |x|: 15, Max |y|: 33, Final Point: (-12, -28)
Walk 69 - Max |x|: 20, Max |y|: 17, Final Point: (-17, -5)
Walk 70 - Max |x|: 19, Max |y|: 30, Final Point: (-14, -20)
Walk 71 - Max |x|: 20, Max |y|: 14, Final Point: (10, -4)
Walk 72 - Max |x|: 15, Max |y|: 30, Final Point: (5, -21)
Walk 73 - Max |x|: 31, Max |y|: 14, Final Point: (-30, 0)
Walk 74 - Max |x|: 13, Max |y|: 41, Final Point: (-6, 38)
Walk 75 - Max |x|: 27, Max |y|: 17, Final Point: (-19, 1)
Walk 76 - Max |x|: 36, Max |y|: 30, Final Point: (-21, -17)
Walk 77 - Max |x|: 28, Max |y|: 23, Final Point: (-26, -14)
Walk 78 - Max |x|: 27, Max |y|: 17, Final Point: (25, 7)
Walk 79 - Max |x|: 29, Max |y|: 22, Final Point: (29, 21)
Walk 80 - Max |x|: 32, Max |y|: 24, Final Point: (-30, -22)
Walk 81 - Max |x|: 18, Max |y|: 25, Final Point: (9, -5)
Walk 82 - Max |x|: 19, Max |y|: 32, Final Point: (18, -26)
Walk 83 - Max |x|: 28, Max |y|: 21, Final Point: (28, -12)
Walk 84 - Max |x|: 34, Max |y|: 36, Final Point: (29, 21)
Walk 85 - Max |x|: 11, Max |y|: 22, Final Point: (5, 3)
Walk 86 - Max |x|: 21, Max |y|: 15, Final Point: (1, 9)
Walk 87 - Max |x|: 28, Max |y|: 25, Final Point: (-24, -12)
Walk 88 - Max |x|: 32, Max |y|: 32, Final Point: (-6, -26)
Walk 89 - Max |x|: 18, Max |y|: 18, Final Point: (3, 3)
Walk 90 - Max |x|: 14, Max |y|: 17, Final Point: (-4, -6)
Walk 91 - Max |x|: 17, Max |y|: 21, Final Point: (-14, 4)
Walk 92 - Max |x|: 37, Max |y|: 13, Final Point: (34, 6)
Walk 93 - Max |x|: 22, Max |y|: 17, Final Point: (-1, -7)
Walk 94 - Max |x|: 26, Max |y|: 32, Final Point: (22, 28)
Walk 95 - Max |x|: 13, Max |y|: 13, Final Point: (-4, 6)
Walk 96 - Max |x|: 47, Max |y|: 31, Final Point: (-45, -27)
Walk 97 - Max |x|: 15, Max |y|: 19, Final Point: (-9, -9)
Walk 98 - Max |x|: 24, Max |y|: 43, Final Point: (-22, 40)
Walk 99 - Max |x|: 19, Max |y|: 18, Final Point: (12, 4)
Walk 100 - Max |x|: 32, Max |y|: 13, Final Point: (-32, 10)

Output (I): absolute max x traveled in all random walks: 54
Output (II): absolute max y traveled in all random walks: 52
Output (III): The mean and standard deviation of all 100 of the destinations in the x dimension:
mean = -4.10, std dev = 18.01
Output (IV): The mean and standard deviation of all 100 destinations in the y dimension: mean =
-1.60, std dev = 19.37
```



P2 [40 points]

Levy flights simulate behavior of foraging animals. Write a simple program to carry out some Levy flights in a 3-d continuous space and demonstrate how varying both the Alpha and Beta parameters affects the distance traveled.

> **Step Direction:** Each step's **direction** angles in each plane should be drawn from a continuous distribution of [0, 2pi], which will be needed to compute x and y components of the next point from the Euclidean step distance.
>
> **Step distance**: Euclidean distance of each step is to be drawn from Levy distribution.

Run least 100 Levy flights with at least 75 steps carried out on each flight. First try it with **alpha=1**, **beta=0**, then vary both parameters, noting the effects.

Scipy has the Levy distribution coded. Scroll down to the Green "Your code" box to see some starter code. It is easy to install Scipy, for example how to on a Mac.[1] More detail on the Levy characteristic function (online book chapter from LibreTexts).

Your report should include the following:

1) **Narrative** describing the effects of varying both parameters Alpha and Beta.
2) **Plots:** minimum 8, maximum 20 plots with different Alpha and Beta that make your points. Label your plots somehow with the Alpha and Beta you used so that I can tell which Alpha and Beta apply to which plot.

Report (follow instructions above)

1) **Narrative**

The alpha parameter controls the stability index of the Levy stable distribution, which influences the step lengths, while beta parameter governs the skewness of the distribution, affecting the direction and asymmetry of the flight paths.

Alpha (ranging from 0.5 to 1.9 in the plots) determines the heaviness of the tails in the Levy stable distribution, which affects the frequency and magnitude of large step lengths. Lower alpha values indicate heavier tails, leading to more frequent large jumps, while higher alpha values approach a Gaussian-like distribution with smaller, more frequent steps.

Beta (ranging from -1.0 to 1.0 in the plots) controls the asymmetry of the Levy stable distribution. A negative beta skews the distribution left (toward negative values), a positive beta skews it right (toward positive values), and beta = 0 results in a symmetric distribution. This skewness influences the directionality of the flight paths along the Z-axis and other dimensions.

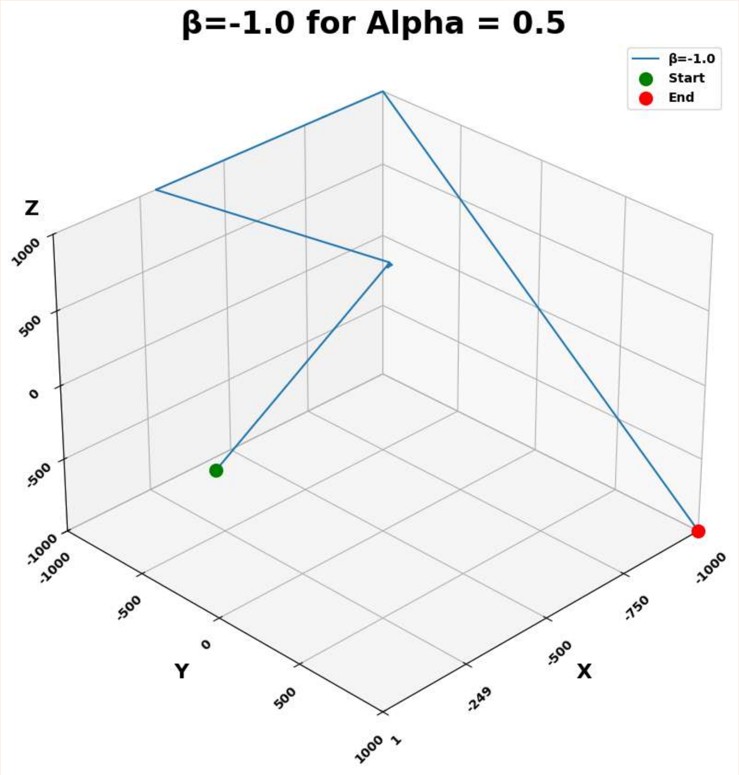
[1] Note: In general, IDE paths are by default partial, broken and not automatically updated when you install a package. If you want to run your Python interpreter within your IDE with scipy, add the location where you installed Scipy to your IDE's path. Your IDE's documentation should explain how to complete this task. Alternatively, run from your system's command prompt.

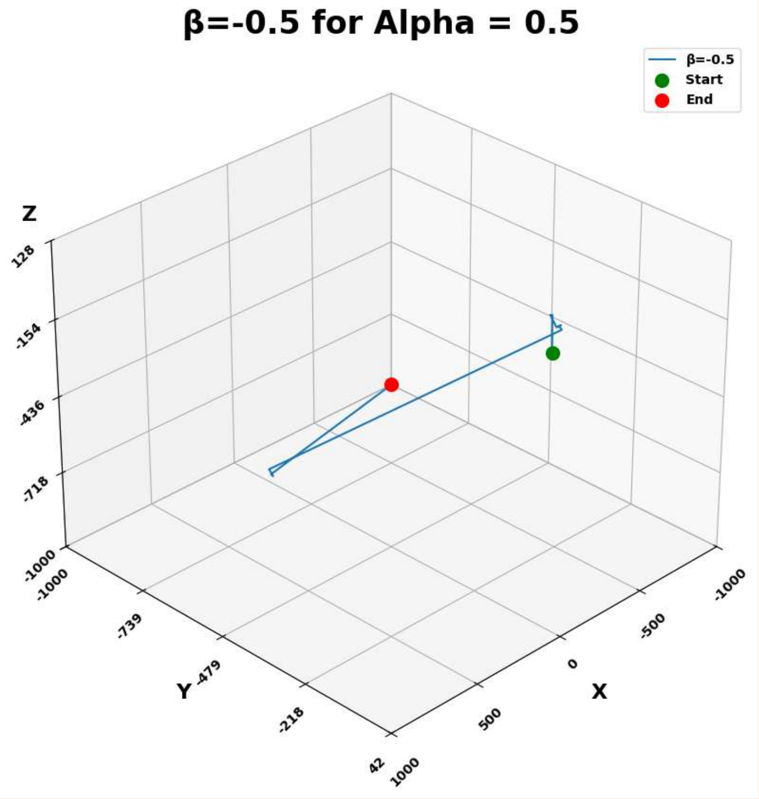
```
Average Distance Table (rounded to 3 decimal places):
Alpha      0.5          1.0          1.5        1.9
Beta
-1.0    2906.182  314.347  122.173  83.543
-0.5    3234.055  605.700  101.776  87.874
 0.0    2918.192  324.543  161.494  84.535
 0.5    2999.595  388.524  110.479  88.860
 1.0    3166.573  864.734  161.478  91.249
```

Alpha Trend: Average distances decrease as alpha increases (0.5 to 1.0 to 1.5 to 1.9), due to the shift from heavy-tailed to Gaussian behavior.

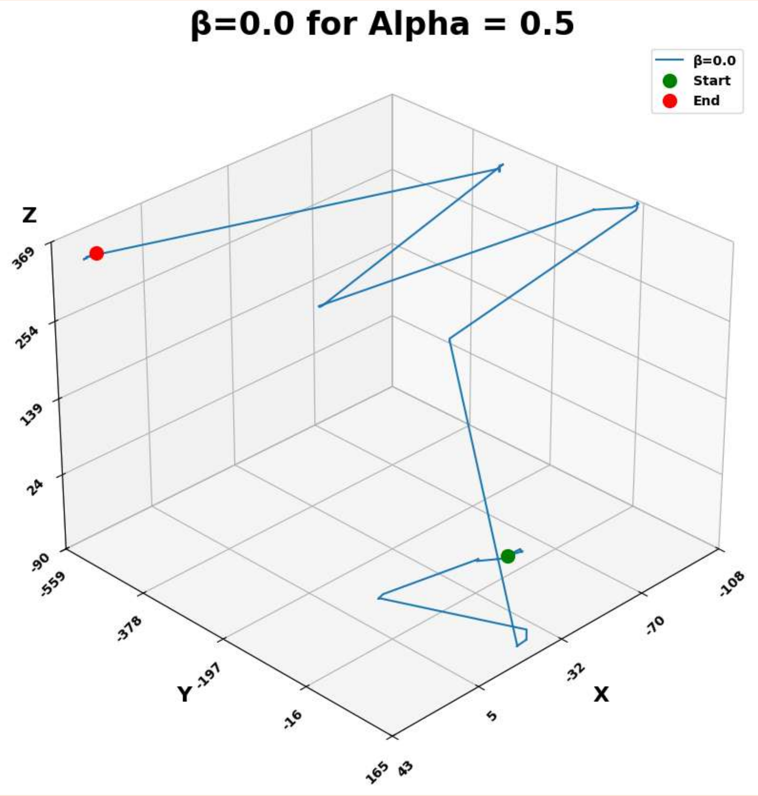
Beta Trend: For a fixed alpha, the average distance may vary with beta, with higher positive beta values potentially increasing distance due to upward skewness, and negative beta values possibly decreasing it due to downward bias. However, the random nature might smooth these effects unless aggregated over many flights.

2) Plots

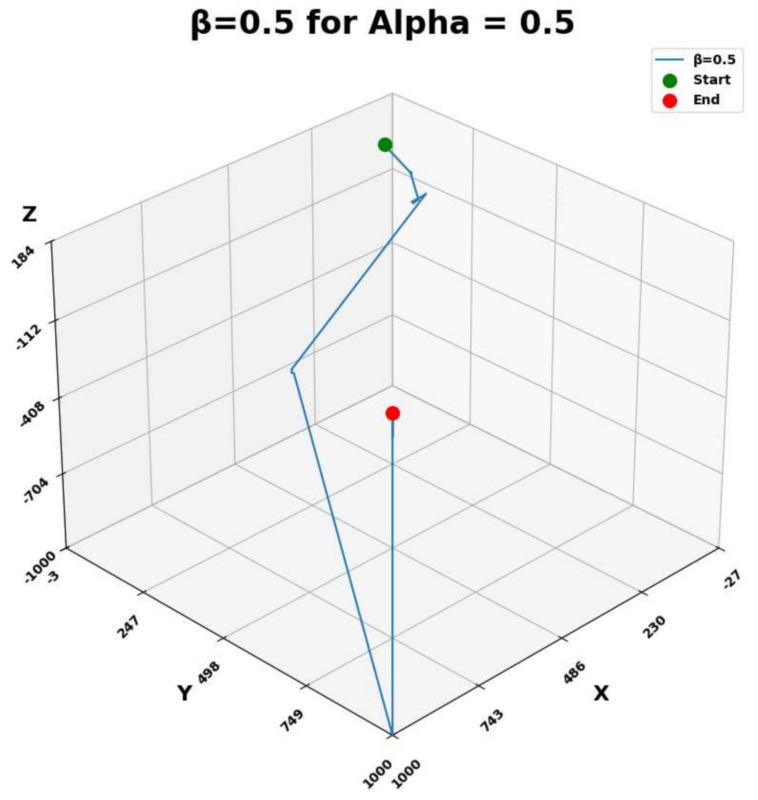


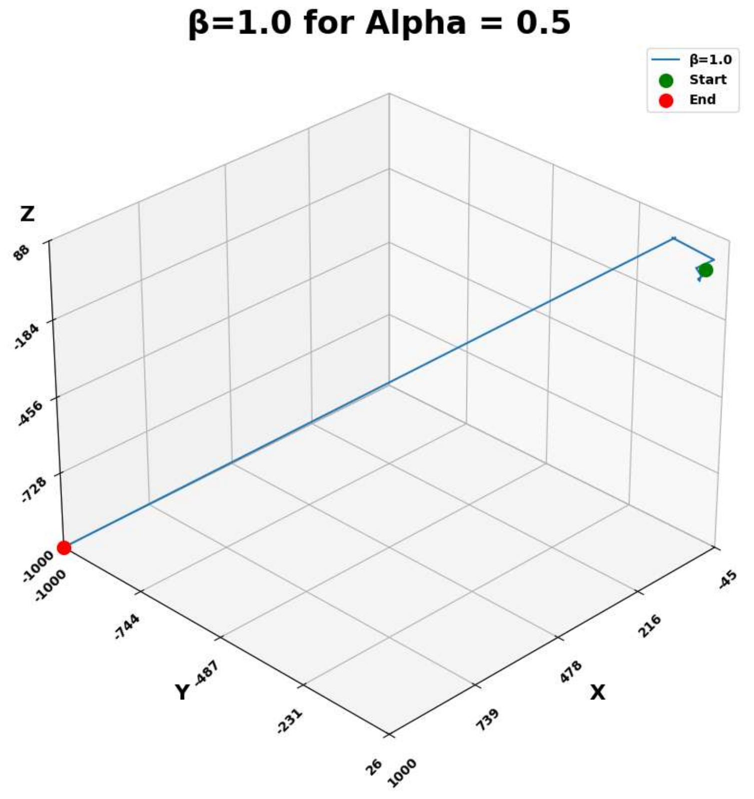


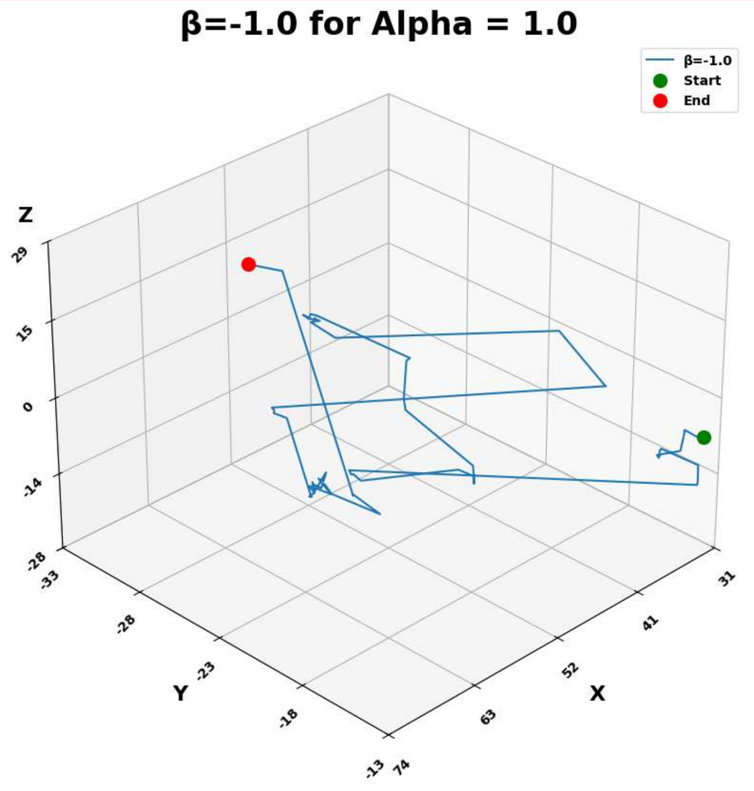
β=-0.5 for Alpha = 0.5

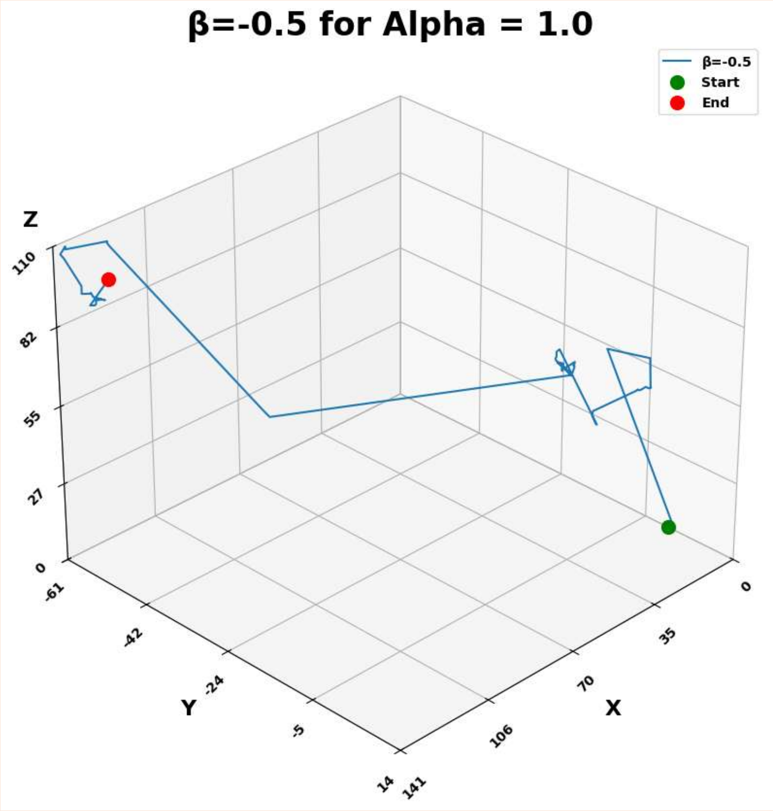


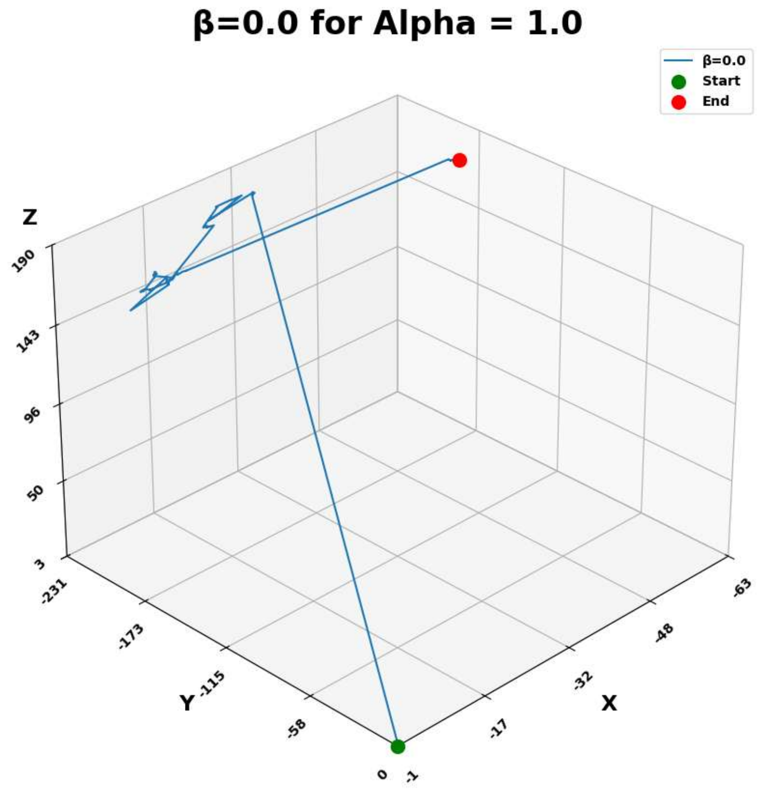
β=0.0 for Alpha = 0.5

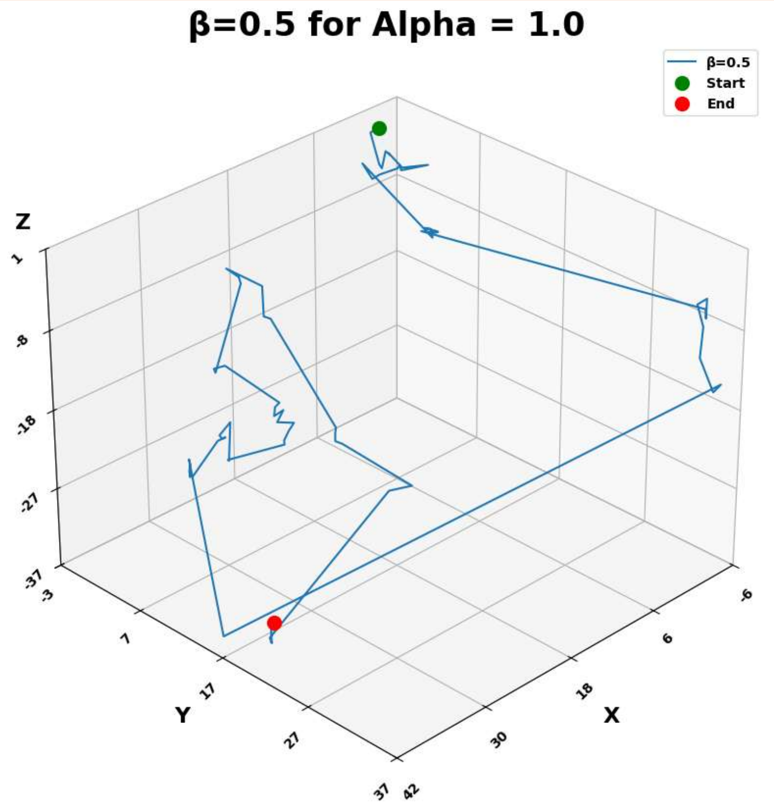

β=0.5 for Alpha = 0.5
β=0.5
Start
End
Z
184
-112
-408
-704
-1000
Y
-3
247
498
749
1000
X
1000
743
486
230
-27

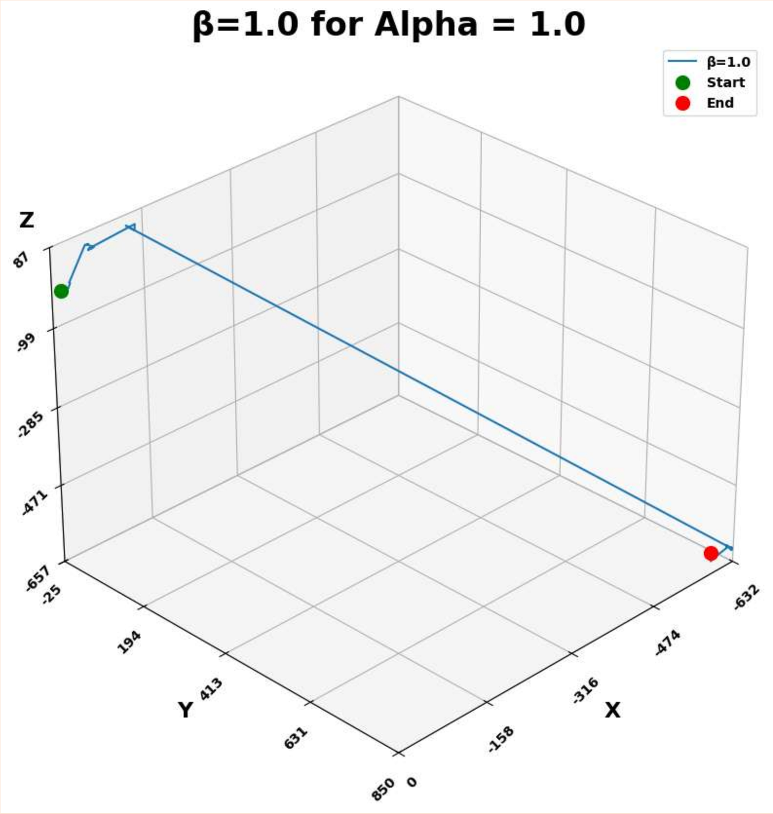

β=1.0 for Alpha = 0.5
β=1.0
Start
End
Z
88
-184
-456
-728
-1000
Y
-1000
-744
-487
-231
26
X
1000
739
478
216
-45

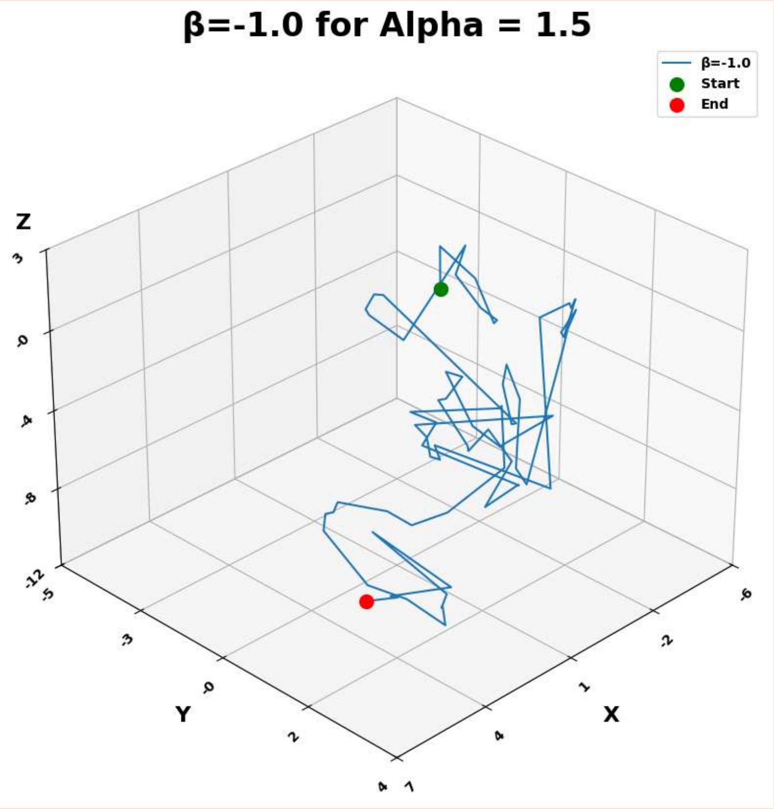

β=-1.0 for Alpha = 1.0
β=-1.0
Start
End
Z
29
15
0
-14
-28
Y
-33
-28
-23
-18
-13
X
74
63
52
41
31

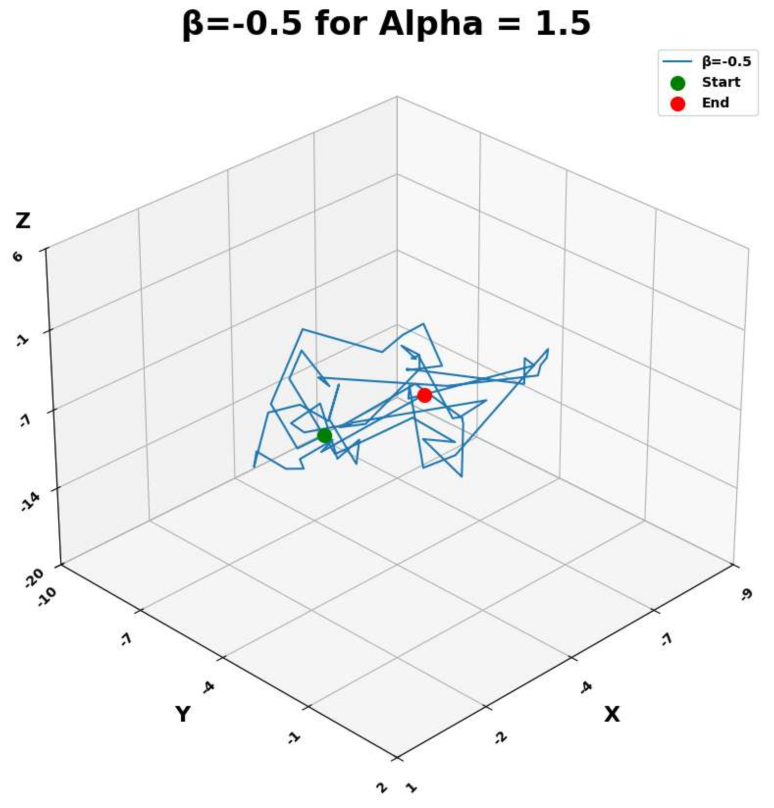

β=-0.5 for Alpha = 1.0
β=-0.5
Start
End
Z
110
82
55
27
0
Y
-61
-42
-24
-5
14
X
141
106
70
35
0


β=0.0 for Alpha = 1.0
β=0.0
Start
End
Z
190
143
96
50
3
Y
-231
-173
-115
-58
0
X
-1
-17
-32
-48
-63

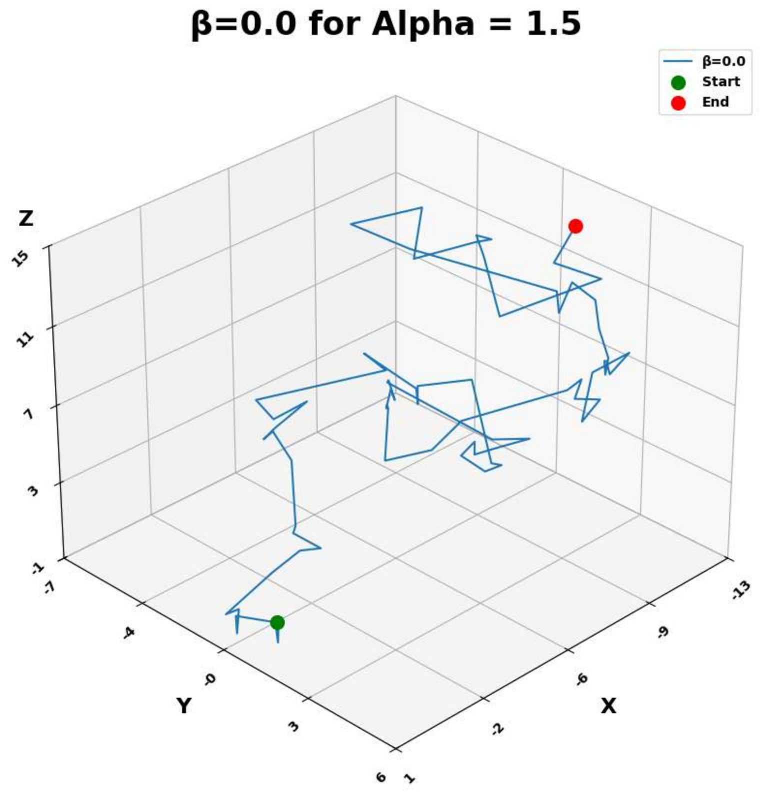

β=0.5 for Alpha = 1.0
β=0.5
Start
End
Z
1
-8
-18
-27
-37
Y
-3
7
17
27
37
X
42
30
18
6
-6


β=1.0 for Alpha = 1.0
β=1.0
Start
End
Z
87
-99
-285
-471
-657
Y
-25
194
413
631
850
X
0
-158
-316
-474
-632

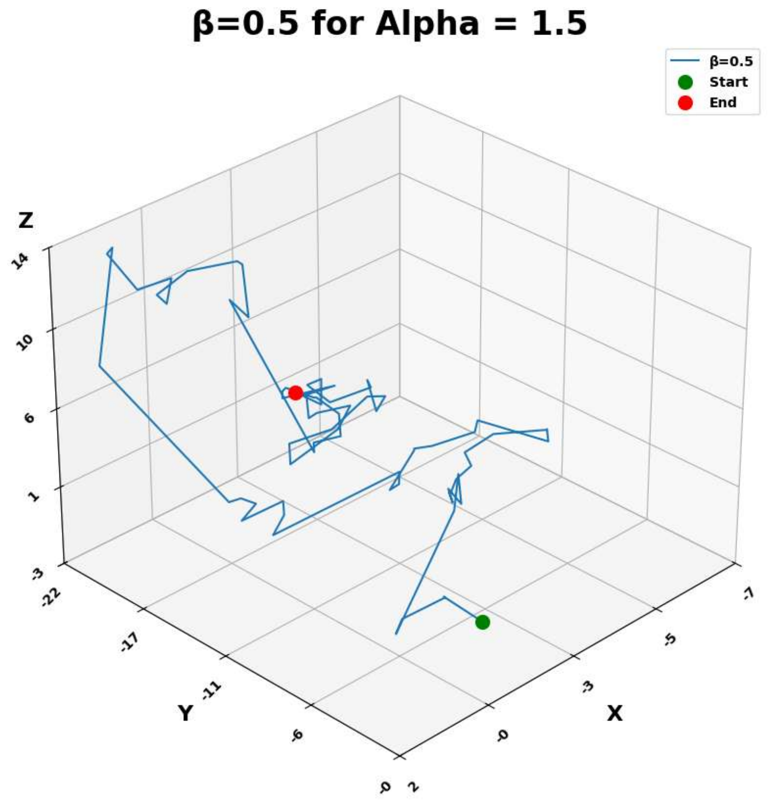

β=-1.0 for Alpha = 1.5
β=-1.0
Start
End
Z
3
-0
-4
-8
-12
Y
-5
-3
-0
2
4
X
7
4
1
-2
-6

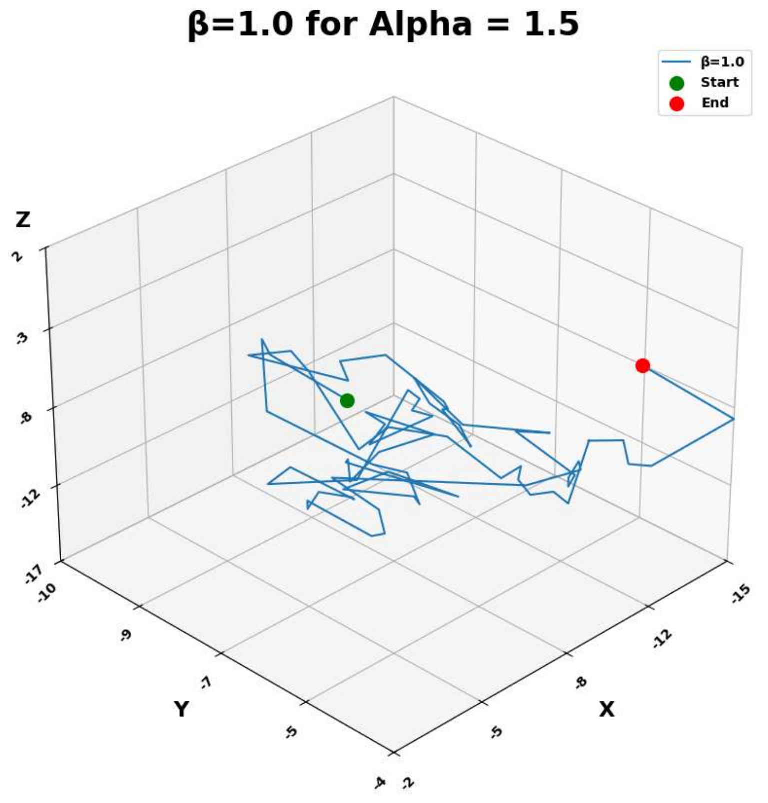


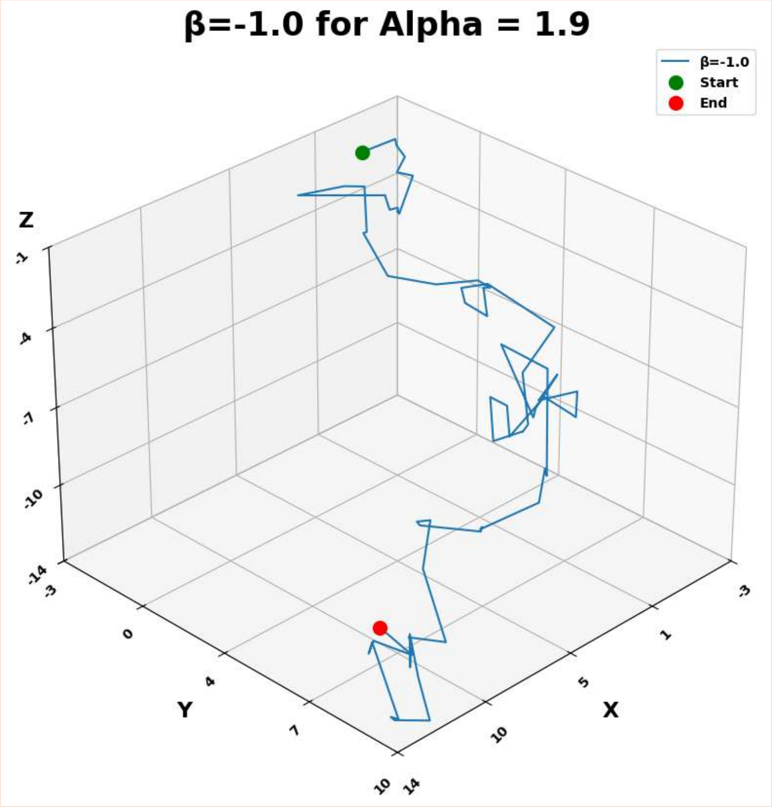
β=-0.5 for Alpha = 1.5

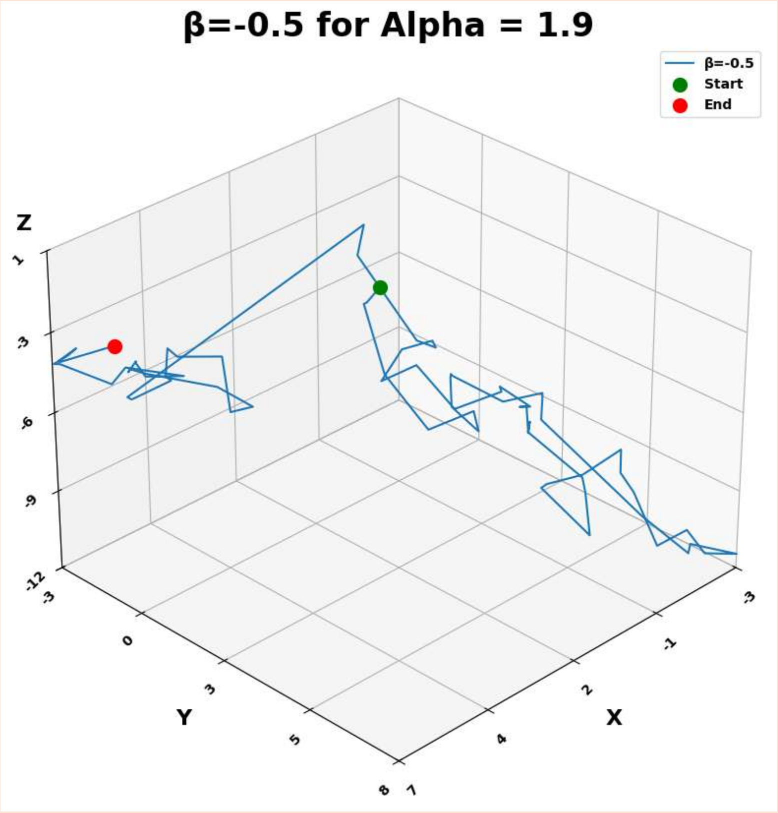


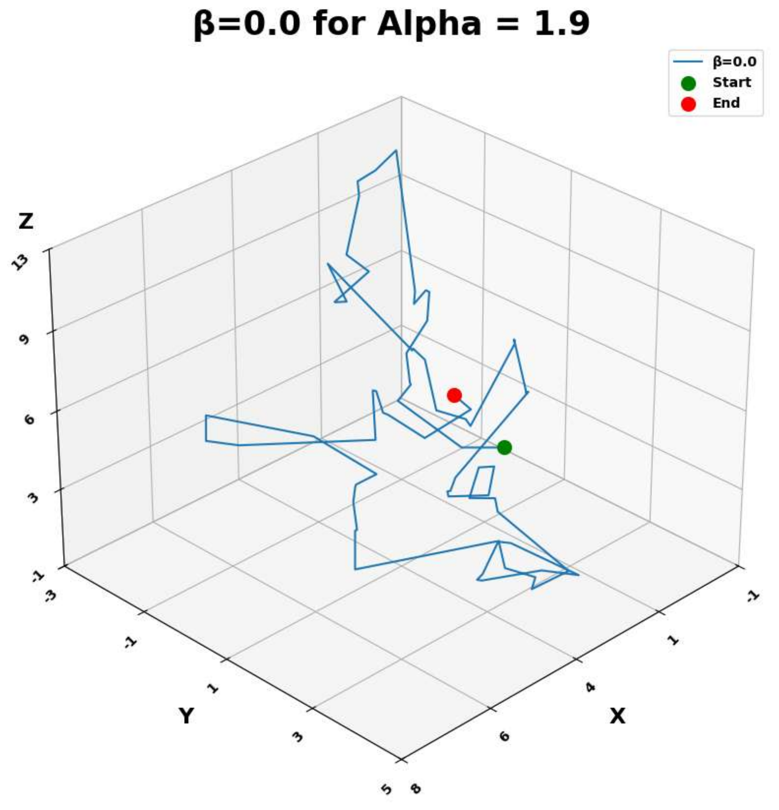
β=0.0 for Alpha = 1.5


β=0.5 for Alpha = 1.5
β=0.5
Start
End
Z
14
10
6
1
-3
Y
-22
-17
-11
-6
-0
X
2
-0
-3
-5
-7

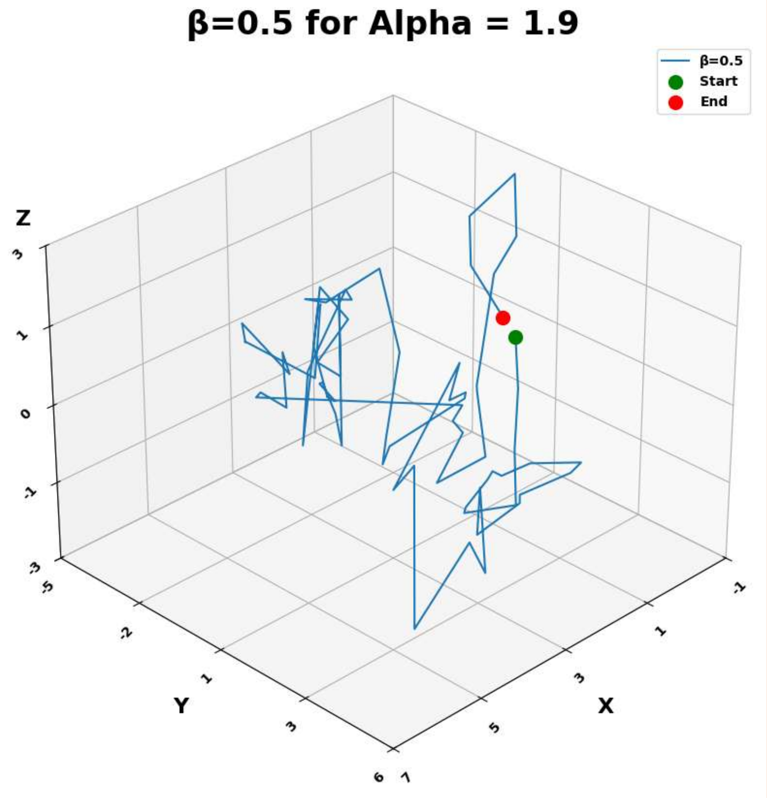

β=1.0 for Alpha = 1.5
β=1.0
Start
End
Z
2
-3
-8
-12
-17
Y
-10
-9
-7
-5
-4
X
-2
-5
-8
-12
-15


β=-1.0 for Alpha = 1.9
β=-1.0
Start
End
Z
-1
-4
-7
-10
-14
Y
-3
0
4
7
10
X
-3
1
5
10
14

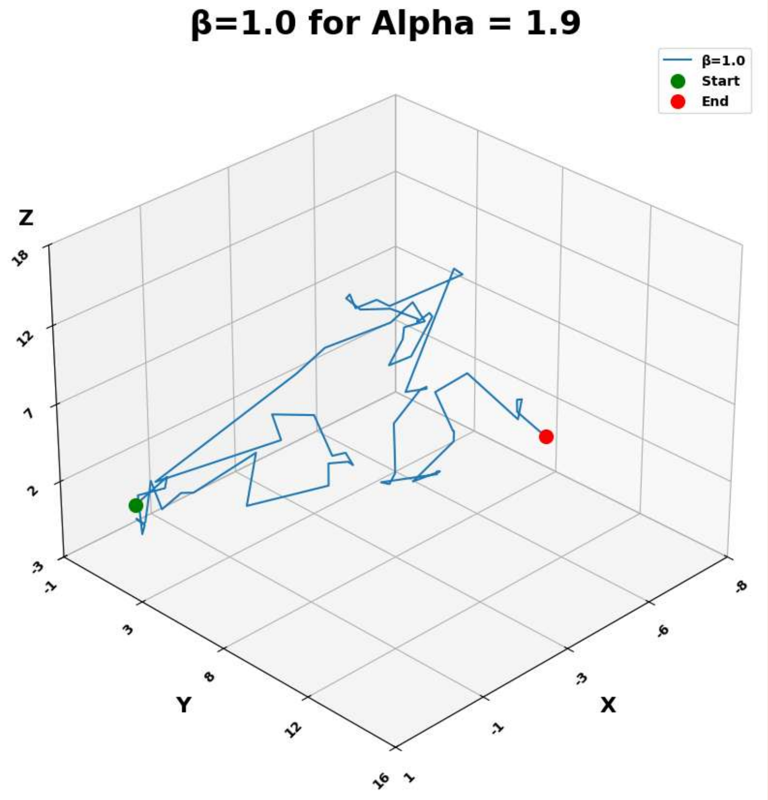

β=-0.5 for Alpha = 1.9
β=-0.5
Start
End
Z
1
-3
-6
-9
-12
Y
-3
0
3
5
8
X
-3
-1
2
4
7



β=0.0 for Alpha = 1.9



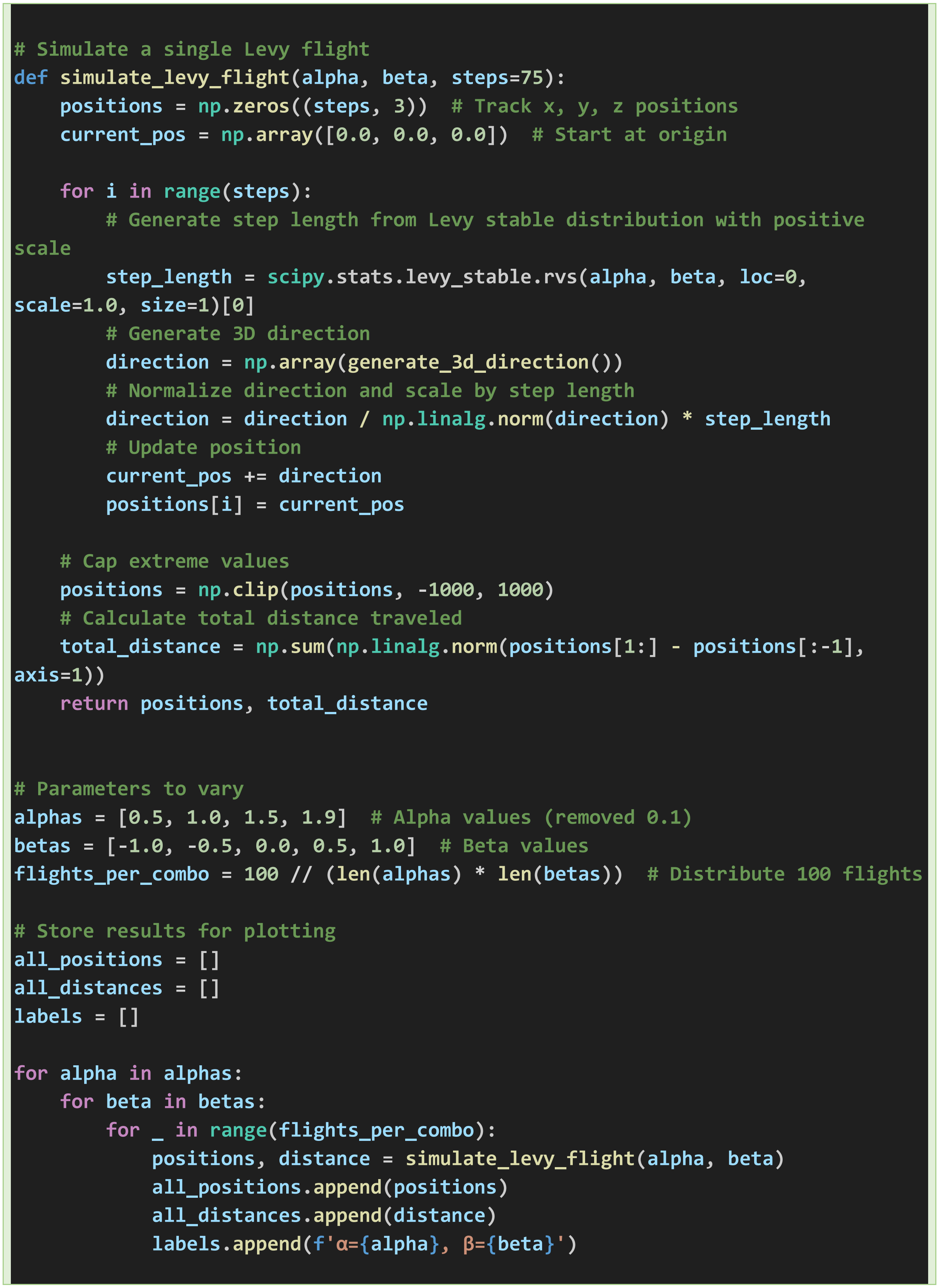
β=0.5 for Alpha = 1.9



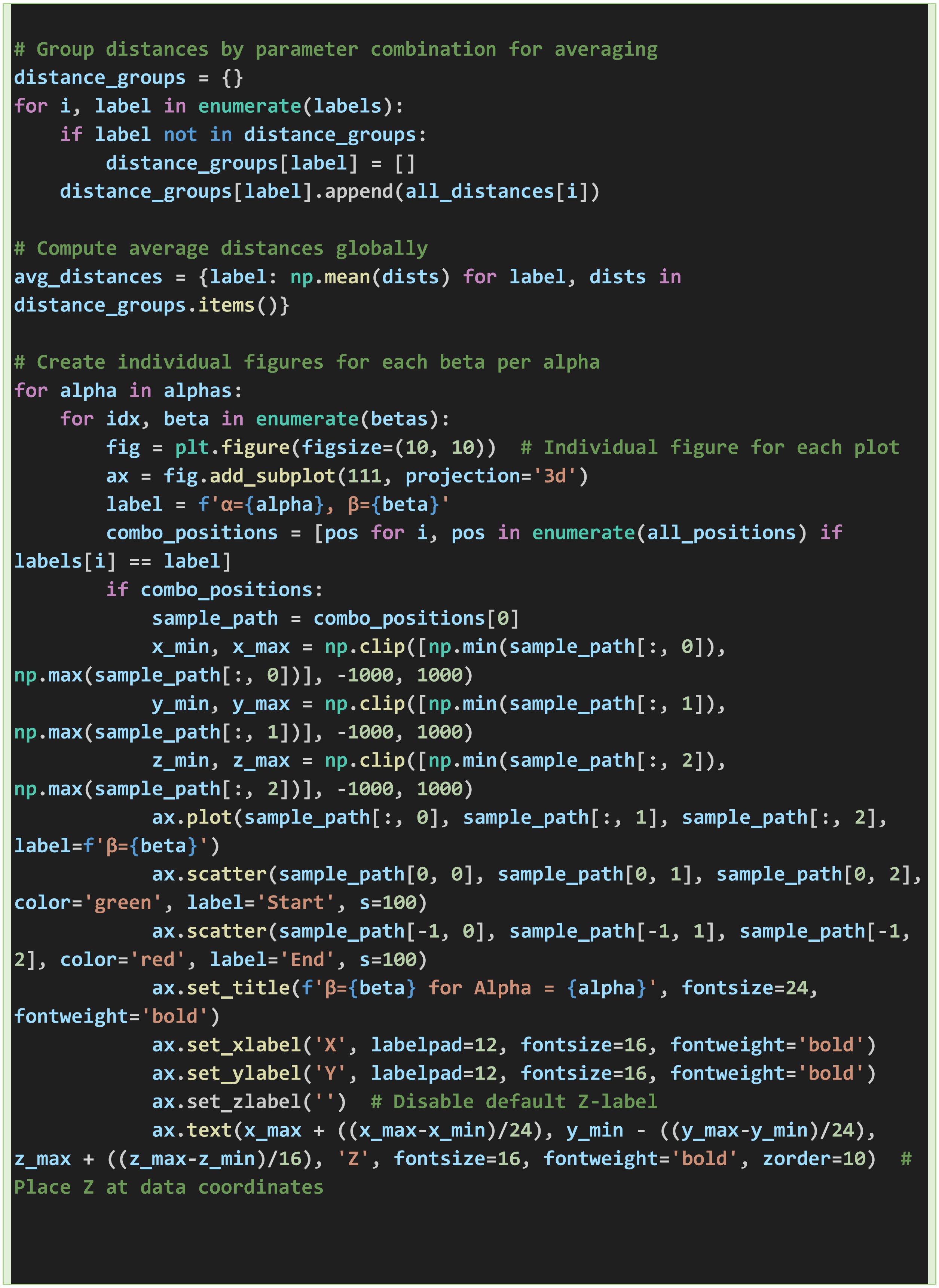
The elevation of 30 degrees and azimuth of 45 degrees were chosen to better identify the effects of alpha and beta settings.

Your Code

```
# AUTHOR: Randall Crawford
# DATE: June 19, 2025
import numpy as np
import scipy
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import FormatStrFormatter

# Generates a random angle for the next step in 3D
def generate_3d_direction():
    theta = np.random.uniform(0, np.pi)  # Polar angle [0, π]
    phi = np.random.uniform(0, 2 * np.pi)  # Azimuthal angle [0, 2π]
    x = np.sin(theta) * np.cos(phi)
    y = np.sin(theta) * np.sin(phi)
    z = np.cos(theta)
    return x, y, z
```



```
# Simulate a single Levy flight
def simulate_levy_flight(alpha, beta, steps=75):
    positions = np.zeros((steps, 3))  # Track x, y, z positions
    current_pos = np.array([0.0, 0.0, 0.0])  # Start at origin

    for i in range(steps):
        # Generate step length from Levy stable distribution with positive
scale
        step_length = scipy.stats.levy_stable.rvs(alpha, beta, loc=0,
scale=1.0, size=1)[0]
        # Generate 3D direction
        direction = np.array(generate_3d_direction())
        # Normalize direction and scale by step length
        direction = direction / np.linalg.norm(direction) * step_length
        # Update position
        current_pos += direction
        positions[i] = current_pos

    # Cap extreme values
    positions = np.clip(positions, -1000, 1000)
    # Calculate total distance traveled
    total_distance = np.sum(np.linalg.norm(positions[1:] - positions[:-1],
axis=1))
    return positions, total_distance


# Parameters to vary
alphas = [0.5, 1.0, 1.5, 1.9]  # Alpha values (removed 0.1)
betas = [-1.0, -0.5, 0.0, 0.5, 1.0]  # Beta values
flights_per_combo = 100 // (len(alphas) * len(betas))  # Distribute 100 flights

# Store results for plotting
all_positions = []
all_distances = []
labels = []

for alpha in alphas:
    for beta in betas:
        for _ in range(flights_per_combo):
            positions, distance = simulate_levy_flight(alpha, beta)
            all_positions.append(positions)
            all_distances.append(distance)
            labels.append(f'α={alpha}, β={beta}')
```



```
# Group distances by parameter combination for averaging
distance_groups = {}
for i, label in enumerate(labels):
    if label not in distance_groups:
        distance_groups[label] = []
    distance_groups[label].append(all_distances[i])

# Compute average distances globally
avg_distances = {label: np.mean(dists) for label, dists in
distance_groups.items()}

# Create individual figures for each beta per alpha
for alpha in alphas:
    for idx, beta in enumerate(betas):
        fig = plt.figure(figsize=(10, 10))  # Individual figure for each plot
        ax = fig.add_subplot(111, projection='3d')
        label = f'α={alpha}, β={beta}'
        combo_positions = [pos for i, pos in enumerate(all_positions) if
labels[i] == label]
        if combo_positions:
            sample_path = combo_positions[0]
            x_min, x_max = np.clip([np.min(sample_path[:, 0]),
np.max(sample_path[:, 0])], -1000, 1000)
            y_min, y_max = np.clip([np.min(sample_path[:, 1]),
np.max(sample_path[:, 1])], -1000, 1000)
            z_min, z_max = np.clip([np.min(sample_path[:, 2]),
np.max(sample_path[:, 2])], -1000, 1000)
            ax.plot(sample_path[:, 0], sample_path[:, 1], sample_path[:, 2],
label=f'β={beta}')
            ax.scatter(sample_path[0, 0], sample_path[0, 1], sample_path[0, 2],
color='green', label='Start', s=100)
            ax.scatter(sample_path[-1, 0], sample_path[-1, 1], sample_path[-1,
2], color='red', label='End', s=100)
            ax.set_title(f'β={beta} for Alpha = {alpha}', fontsize=24,
fontweight='bold')
            ax.set_xlabel('X', labelpad=12, fontsize=16, fontweight='bold')
            ax.set_ylabel('Y', labelpad=12, fontsize=16, fontweight='bold')
            ax.set_zlabel('')  # Disable default Z-label
            ax.text(x_max + ((x_max-x_min)/24), y_min - ((y_max-y_min)/24),
z_max + ((z_max-z_min)/16), 'Z', fontsize=16, fontweight='bold', zorder=10)  #
Place Z at data coordinates
```

```
            ax.set_xlim([x_min, x_max])
            ax.set_ylim([y_min, y_max])
            ax.set_zlim([z_min, z_max])
            ax.set_xticks(np.linspace(x_min, x_max, 5))
            ax.set_yticks(np.linspace(y_min, y_max, 5))
            ax.set_zticks(np.linspace(z_min, z_max, 5))
            ax.xaxis.set_major_formatter(FormatStrFormatter('%.0f'))
            ax.yaxis.set_major_formatter(FormatStrFormatter('%.0f'))
            ax.zaxis.set_major_formatter(FormatStrFormatter('%.0f'))
            for label in ax.get_xticklabels() + ax.get_yticklabels() +
ax.get_zticklabels():
                label.set_fontweight('bold')
                label.set_fontsize(10)
                label.set_rotation(45)
            ax.legend(fontsize=16, prop={'weight': 'bold'})
            ax.view_init(elev=30, azim=45)
        # fig.subplots_adjust(top=0.9, left=0.05, right=0.95, bottom=0.1)
        plt.tight_layout()
        plt.show()

# Create table for average distances with 3 decimal place precision
import pandas as pd
table_data = {alpha: [] for alpha in alphas}
for alpha in alphas:
    for beta in betas:
        label = f'α={alpha}, β={beta}'
        avg_dist = avg_distances.get(label, 0)
        table_data[alpha].append(round(avg_dist, 3))

df = pd.DataFrame(table_data, index=betas)
df.index.name = 'Beta'
df.columns.name = 'Alpha'

print("Average Distance Table (rounded to 3 decimal places):")
print(df)
```

P3 [15 points]

The method of least squares works well for most data fitting problems; however, for polynomials

$$y = f(x) = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \ldots + \alpha_p x^p,$$

it tends to result in high-order polynomials to best-fit the given data set (x_i, y_i) where $i = 1, 2, \ldots, n$, even though the higher-order polynomials are not necessarily the best model (potentially overfitting). A way to improve this is to use some penalty in coefficients to balance the goodness of the fit and the choice of reasonably simple model. Design an optimization algorithm to achieve this by limiting the sum of the coefficients α_k where $k = 0, 1, \ldots, p$.

Solution

In designing an optimization algorithm by extending the traditional least squares method with a penalty term, Lasso seemed like the best option when trying to balance the goodness of fit and choose a reasonably simple model. It can potentially eliminate needless coefficients and effectively utilize a hard limit constraint (i.e. LARS).

The objective function to minimize:

$$J(\alpha) = \sum_{i=1}^{n} (y_i - f(x_i))^2 + \lambda \sum_{k=0}^{p} |\alpha_k|$$

Where:

- y_i **is the observed value.**
- $f(x_i) = \alpha_0 + \alpha_1 x_i + \alpha_2 x_i^2 + \ldots + \alpha_p x_i^p$ **is the polynomial model.**
- $\sum_{i=1}^{n} (y_i - f(x_i))^2$ **is the least squares error (goodness of fit).**
- $\lambda \sum_{k=0}^{p} |\alpha_k|$ **is the penalty term, with λ as a tuning parameter that controls the trade-off between fit and simplicity.**
- α_k **are the polynomial coefficients to be optimized.**

Algorithm Design:

1. Initialize: Start with an initial guess for the coefficients α_k (e.g., all zeros or a low-order polynomial fit).

2. Iterative Optimization: Use a gradient-based method or a specialized solver to minimize $J(\alpha)$.

- **Compute the gradient of $J(\alpha)$ with respect to each α_k, which includes the derivative of the penalty term $\lambda \text{sign}(\alpha_k)$.**
- **Update α_k iteratively until convergence.**

3. Constraint Enforcement: Optionally, a hard constraint $\sum_{k=0}^{p} |\alpha_k| \leq C$ (where C is a user-defined constant) could be imposed, and the constrained optimization problem could be solved using methods like quadratic programming.

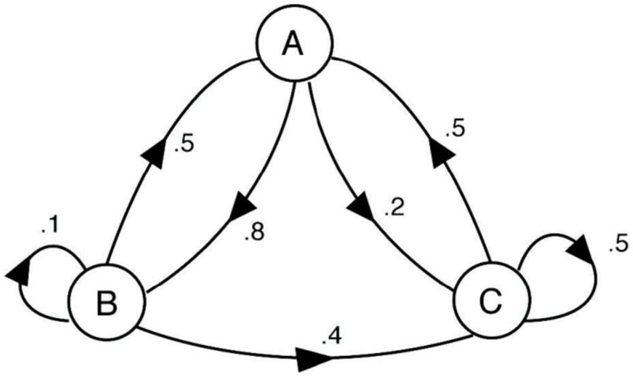
4. Tune λ and/or C: Use cross-validation on the dataset to select an appropriate value for λ and/or C that balances fit and model complexity.

This approach ensures that the model does not overfit by penalizing large coefficients, aligning with the goal of achieving a reasonably simple model. The choice between a soft penalty (via λ), a hard constraint, or both depends on the specific requirements of your application, but the penalty method is more commonly used in practice for its flexibility.

P4 [40 points]

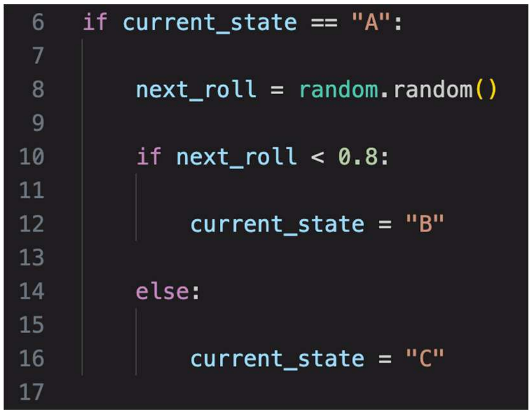
Recall that Markov processes are distinctive in that they describe the likelihood of the next state only in terms of the currently known state; as such they are described as memoryless.[2] Create a simple program that generates a Markov chain relying on knowledge of only the current state to generate the next.

First create the transition matrix for the Markov graph of transition probabilities shown below (scroll down to see the template "Your Transition Matrix" and fill it in)



[2] Incorporating some memory information from prior states is a new research area, cf. Wu, S. J., & Chu, M. T. (2017). Markov chains with memory, tensor formulation, and the dynamics of power iteration. *Applied Mathematics and Computation*, *303*, 226-239.

Have your Python program simulate an agent that generates 100 markov chains starting at State C, each time visiting the next node with the probability given in the graph (use a random number generator for this). For example, if it's arrived at state A, it will need to then do something like this within your main loop:



```
if current_state == "A":

    next_roll = random.random()

    if next_roll < 0.8:

        current_state = "B"

    else:

        current_state = "C"

```

Terminate each of your 100 Markov chain generations when their overall relative probability of the occurrence of the chain (given we start at state C) is less than 1/10 of 1% (0.001 of the product of all the chain's transition probabilities). For example once we have generated **CCABABACAB**, P(CCABABACAB) = .0008 < 0.001, so we would stop there and go to generate the next Markov chain, not adding more states after B.

Print each Markov Chain's sequence of states and probability in the format

```
CCABABACAB, 0.008
```

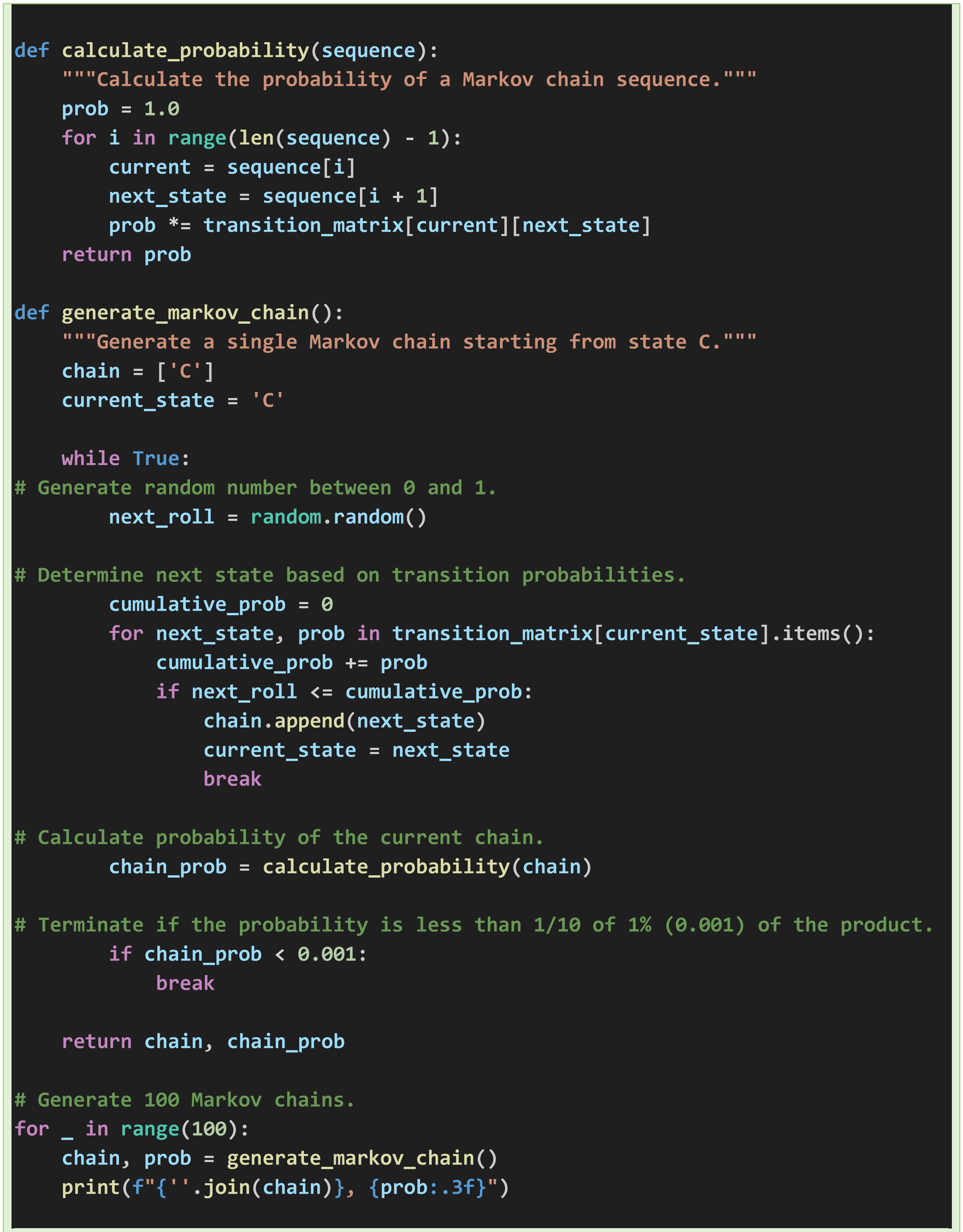
(there should be 100 outputs like this)

Your Transition Matrix

	to A	to B	to C
From A	**0**	**0.8**	**0.2**
From B	**0.5**	**0.1**	**0.4**
From C	**0.5**	**0**	**0.5**

Your Code

```
import random

# Transition matrix
transition_matrix = {
    'A': {'A': 0, 'B': 0.8, 'C': 0.2},
    'B': {'A': 0.5, 'B': 0.1, 'C': 0.4},
    'C': {'A': 0.5, 'B': 0, 'C': 0.5}
}
```



```
def calculate_probability(sequence):
    """Calculate the probability of a Markov chain sequence."""
    prob = 1.0
    for i in range(len(sequence) - 1):
        current = sequence[i]
        next_state = sequence[i + 1]
        prob *= transition_matrix[current][next_state]
    return prob

def generate_markov_chain():
    """Generate a single Markov chain starting from state C."""
    chain = ['C']
    current_state = 'C'

    while True:
# Generate random number between 0 and 1.
        next_roll = random.random()

# Determine next state based on transition probabilities.
        cumulative_prob = 0
        for next_state, prob in transition_matrix[current_state].items():
            cumulative_prob += prob
            if next_roll <= cumulative_prob:
                chain.append(next_state)
                current_state = next_state
                break

# Calculate probability of the current chain.
        chain_prob = calculate_probability(chain)

# Terminate if the probability is less than 1/10 of 1% (0.001) of the product.
        if chain_prob < 0.001:
            break

    return chain, chain_prob

# Generate 100 Markov chains.
for _ in range(100):
    chain, prob = generate_markov_chain()
    print(f"{''.join(chain)}, {prob:.3f}")
```

Program output report

```
CABBABCABABC, 0.000
CABCABABCABCC, 0.001
CABABCABACCC, 0.001
CABCCABABCABAC, 0.000
CABBABCAC, 0.001
CCCABABCCCCCA, 0.001
CCACCCCABCC, 0.001
CABCACCABCC, 0.001
CCABACACCC, 0.001
CABCABBCCC, 0.001
CABCCCCACAC, 0.000
CACACCCABC, 0.000
CCCCACCCCC, 0.001
CCABCABABCCCC, 0.001
CABCABACABB, 0.000
CCABCABCCABABC, 0.000
CABCABABACCA, 0.001
CABCCABABCABB, 0.000
CCCABCCCCABB, 0.000
CCCABCABABCAC, 0.000
CCABABABABCABC, 0.001
CABABBABCCA, 0.001
CCABBCABABA, 0.001
CCABABCCABABABA, 0.001
CABACABABABB, 0.000
CCABABCABACC, 0.001
CCABCACCCAB, 0.001
CABACCABABABC, 0.001
CCCABCCCACC, 0.001
CABABCCCABABABA, 0.001
CABCABBCCC, 0.001
CABABCABABCABA, 0.001
CABACABABB, 0.001
CABABABCCCABCC, 0.001
CABABCCABABCCC, 0.001
CCACABABCCC, 0.001
CCCCCCACCC, 0.001
CCCCABCABB, 0.001
CABCABCABBA, 0.001
CCABCACCABB, 0.000
CCABCABCCAC, 0.001
CCCABABCABCABA, 0.001
CABCACCABCA, 0.001
CABCABABCCAC, 0.001
CABABABACCABC, 0.001
CACABCABCCA, 0.001
CABABACABCABA, 0.001
CCCABCCABCCC, 0.001
CCACCABBC, 0.000
CCABABACCAC, 0.000
```

```
CABCABCACAC, 0.000
CABCCCABCAC, 0.001
CCABABCCCCABB, 0.000
CCCACABCABC, 0.001
CCABABCABCCABB, 0.000
CACCCCCABAC, 0.000
CABABCABCCCCA, 0.001
CABCCCCCCCCA, 0.001
CCABBCCCCC, 0.001
CABABABABCACA, 0.001
CCCCCCABABABAB, 0.001
CABCABABABABABA, 0.001
CCABACCCCCA, 0.001
CCABABACABAC, 0.000
CABABCCABCABAB, 0.001
CABABABCCABAC, 0.001
CCABBABABCA, 0.001
CABACABCABABA, 0.001
CCCACCCCCA, 0.001
CCABCABCABABAC, 0.000
CABCABCCABCAB, 0.001
CACCABABCABC, 0.001
CCCABABABABABABA, 0.001
CACCCABCABC, 0.001
CABABABCCABB, 0.001
CCCABABACABC, 0.001
CCABABCABCABB, 0.000
CABCCABCCCAC, 0.000
CCACABABABAC, 0.000
CACCCCCCCA, 0.001
CCCABABABCCCC, 0.001
CABCCABABABABAC, 0.000
CACABABABAC, 0.001
CABCCCCABABCA, 0.001
CABCCCABCABAC, 0.000
CACCABACAB, 0.001
CABCCABACABA, 0.001
CCABABABBCA, 0.001
CCCCABCABAC, 0.001
CACCACABC, 0.001
CABCABCCACC, 0.001
CCCABACABABC, 0.001
CABABABCCCCCC, 0.001
CCABCABCCABB, 0.000
CCCABABCCCABC, 0.001
CABABABABABABABC, 0.001
CABABCACCCA, 0.001
CCCABCABBA, 0.001
CCACCABBA, 0.001
CCCCCABACABC, 0.000
```

The end