

# Homework Assignment 5.2

Max Green TIF155

December 22, 2024

## Exercise a

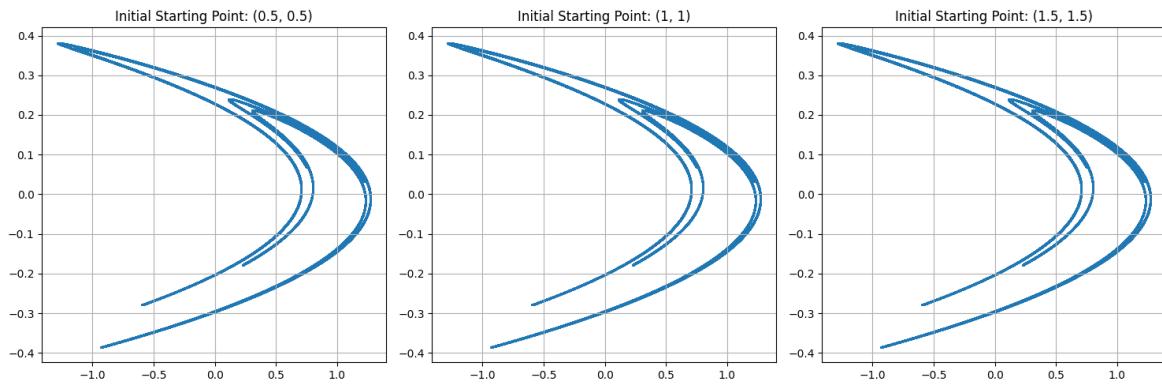


Figure 1: Approximation of the fractal attractor.

## b

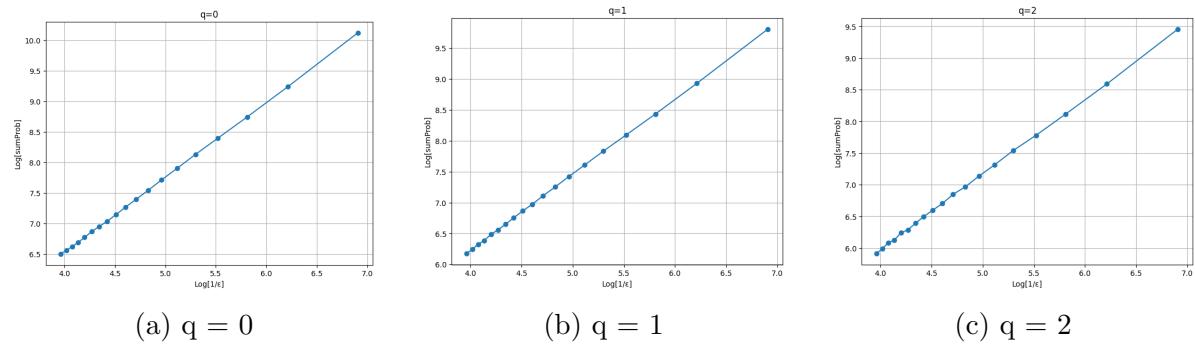
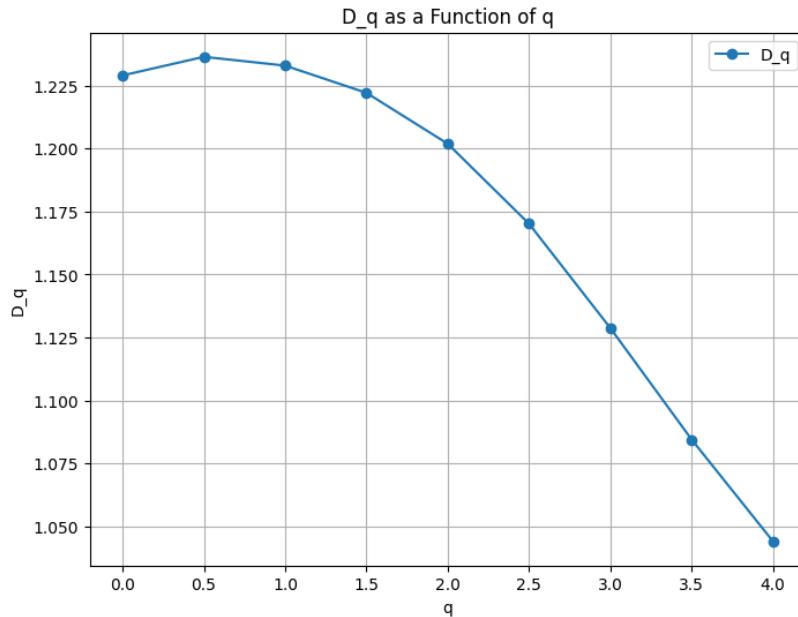


Figure 2: Three plots with different values for  $q$ .

## c

See code for calculation.

**d**Figure 3:  $D_q$  as a function of  $q$ .**e**

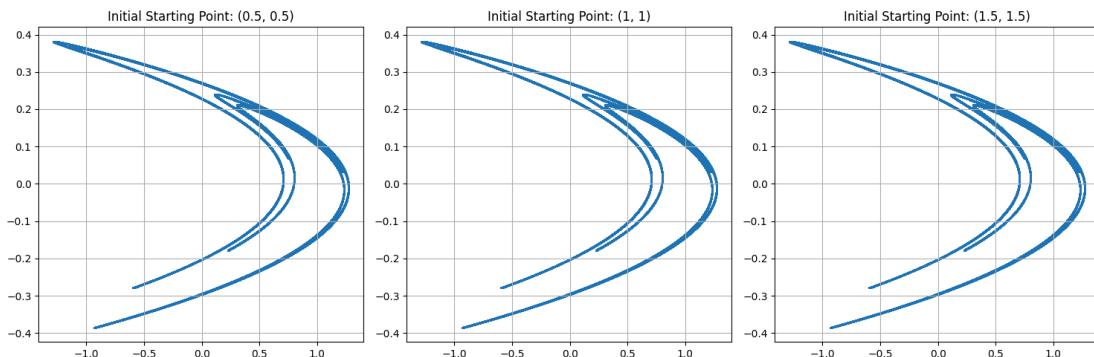
See code.

**f**

See code.

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

```
In [ ]: a = 1.4  
b = 0.3  
n = 1000000  
  
def generate_data(start, a, b, n):  
    x, y = start  
    data = []  
    for _ in range(n):  
        new_x = y + 1 - a * x**2  
        new_y = b * x  
        data.append((new_x, new_y))  
        x, y = new_x, new_y  
    return np.array(data)  
  
# Generate data for different starting points  
data1 = generate_data((0.5, 0.5), a, b, n)[4:]  
data2 = generate_data((1, 1), a, b, n)[4:]  
data3 = generate_data((1.5, 1.5), a, b, n)[4:]  
  
plt.figure(figsize=(15, 5))  
plt.subplot(1, 3, 1)  
plt.scatter(data1[:, 0], data1[:, 1], s=0.1)  
plt.title("Initial Starting Point: (0.5, 0.5)")  
plt.grid(True)  
plt.subplot(1, 3, 2)  
plt.scatter(data2[:, 0], data2[:, 1], s=0.1)  
plt.title("Initial Starting Point: (1, 1)")  
plt.grid(True)  
plt.subplot(1, 3, 3)  
plt.scatter(data3[:, 0], data3[:, 1], s=0.1)  
plt.title("Initial Starting Point: (1.5, 1.5)")  
plt.grid(True)  
plt.tight_layout()  
plt.show()
```



```
In [ ]: iterations = 2 * 10**6  
epsilons = np.arange(0.001, 0.02, 0.001)  
  
def henon_map(state, a, b):  
    x, y = state  
    return y + 1 - a * x**2, b * x  
  
data = np.zeros((iterations, 2))
```

```

data[0] = [0.1, 0.1]
for i in range(1, iterations):
    data[i] = henon_map(data[i - 1], a, b)

# Function to compute slope and plot for different q values
def compute_and_plot(q, title):
    bins_list = []
    probabilities = []
    for epsilon in epsilons:
        x_bins = np.arange(np.min(data[:, 0]), np.max(data[:, 0]) + epsilon, epsilon)
        y_bins = np.arange(np.min(data[:, 1]), np.max(data[:, 1]) + epsilon, epsilon)
        hist, _, _ = np.histogram2d(data[:, 0], data[:, 1], bins=(x_bins, y_bins))
        bins_list.append(hist.flatten())
        probabilities.append(hist.flatten() / (iterations - 1))

    if q == 1:
        sum_prob = [np.sum(p[p > 0] * np.log(1 / p[p > 0])) for p in probabilities]
        y_values = sum_prob
    else:
        sum_prob = [np.sum(p[p > 0] ** q) for p in probabilities]
        y_values = [np.log(sp) / (1 - q) for sp in sum_prob]

    x_values = np.log(1 / epsilon)

    nominator = y_values[-1] - y_values[0]
    denominator = x_values[-1] - x_values[0]
    slope = nominator / denominator

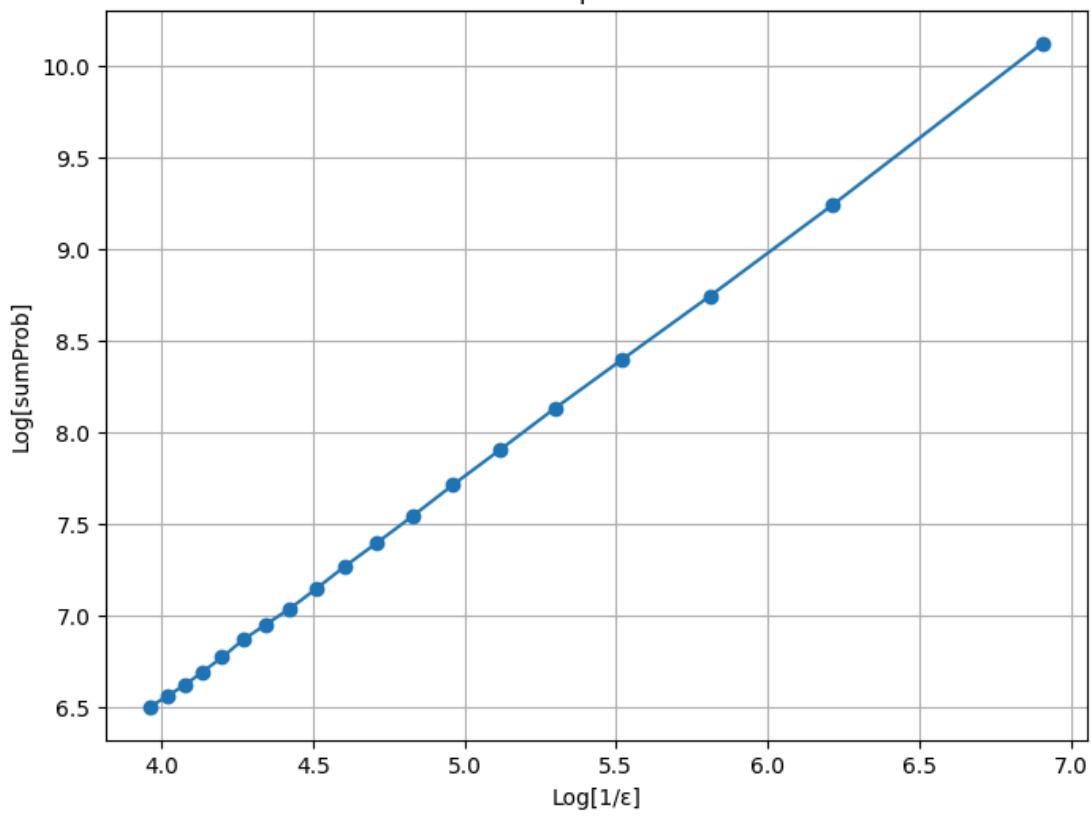
    plt.figure(figsize=(8, 6))
    plt.plot(x_values, y_values, 'o-')
    plt.xlabel('Log[1/ε]')
    plt.ylabel('Log[sumProb]')
    plt.title(title)
    plt.grid(True)
    plt.show()

    print(f"Slope for {title}: {slope}")

compute_and_plot(0, "q=0")
compute_and_plot(1, "q=1")
compute_and_plot(2, "q=2")

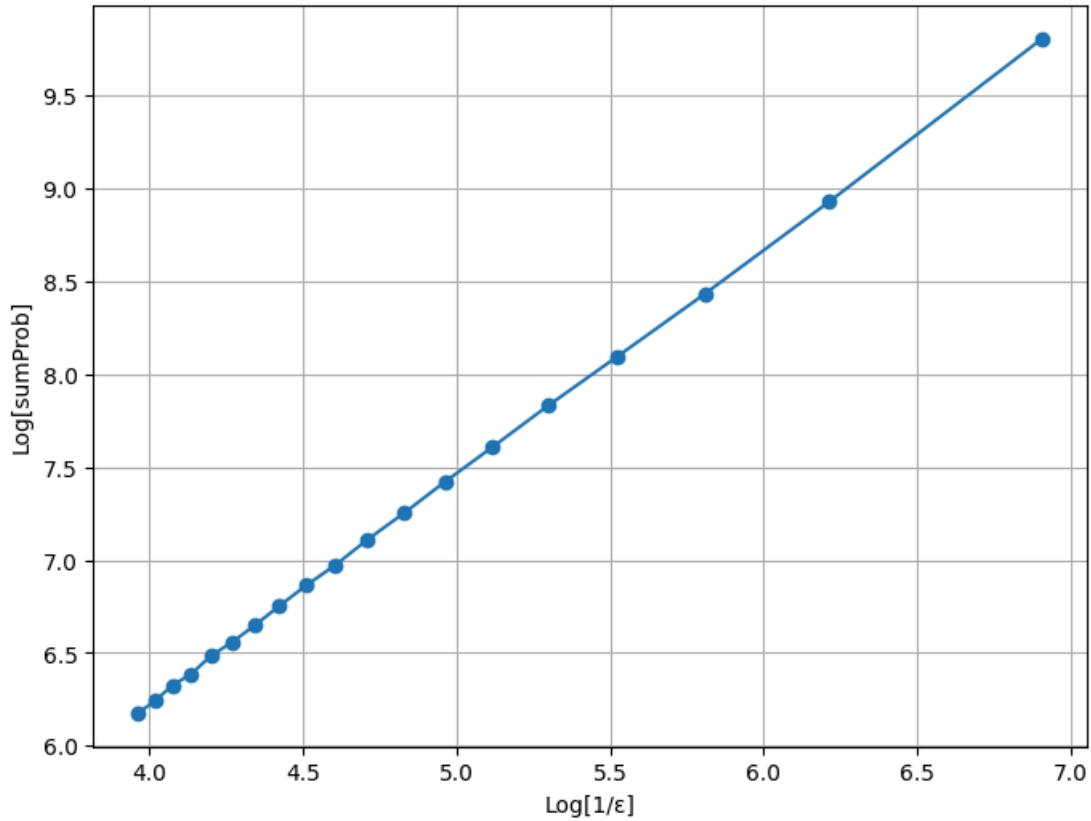
```

$q=0$

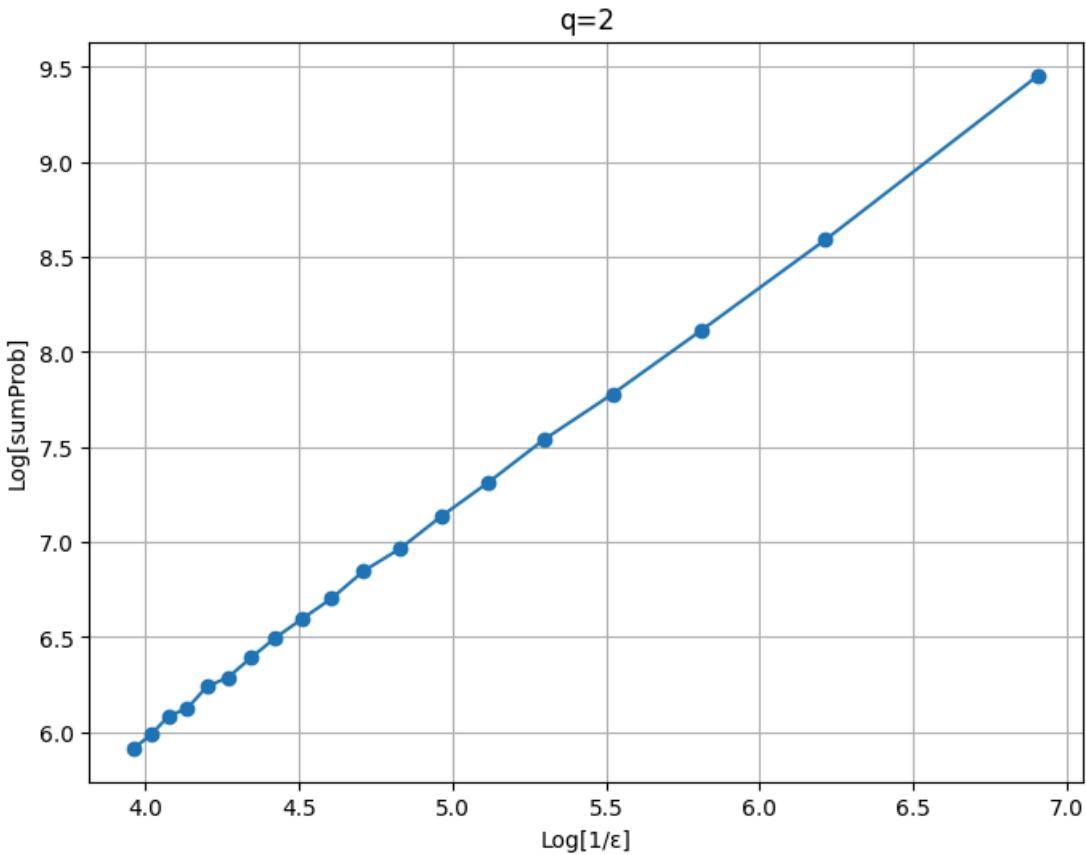


Slope for  $q=0$ : 1.2290383810689962

$q=1$



Slope for  $q=1$ : 1.2329288099159232



Slope for q=2: 1.2018917070332482

```
In [ ]: data = np.zeros((iterations, 2))
data[0] = [0.1, 0.1]
for i in range(1, iterations):
    data[i] = henon_map(data[i - 1], a, b)

# Function to compute slope for a given q
def compute_slope(q):
    bins_list = []
    probabilities = []
    for epsilon in epsilons:
        x_bins = np.arange(np.min(data[:, 0]), np.max(data[:, 0]) + epsilon, epsilon)
        y_bins = np.arange(np.min(data[:, 1]), np.max(data[:, 1]) + epsilon, epsilon)
        hist, _, _ = np.histogram2d(data[:, 0], data[:, 1], bins=(x_bins, y_bins))
        bins_list.append(hist.flatten())
        probabilities.append(hist.flatten() / (iterations - 1))

    if q == 1:
        sum_prob = [np.sum(p[p > 0] * np.log(1 / p[p > 0])) for p in probabilities]
        y_values = sum_prob
    else:
        sum_prob = [np.sum(p[p > 0] ** q) for p in probabilities]
        y_values = [np.log(sp) / (1 - q) for sp in sum_prob]

    x_values = np.log(1 / epsilon)

    nominator = y_values[-1] - y_values[0]
    denominator = x_values[-1] - x_values[0]
    return nominator / denominator

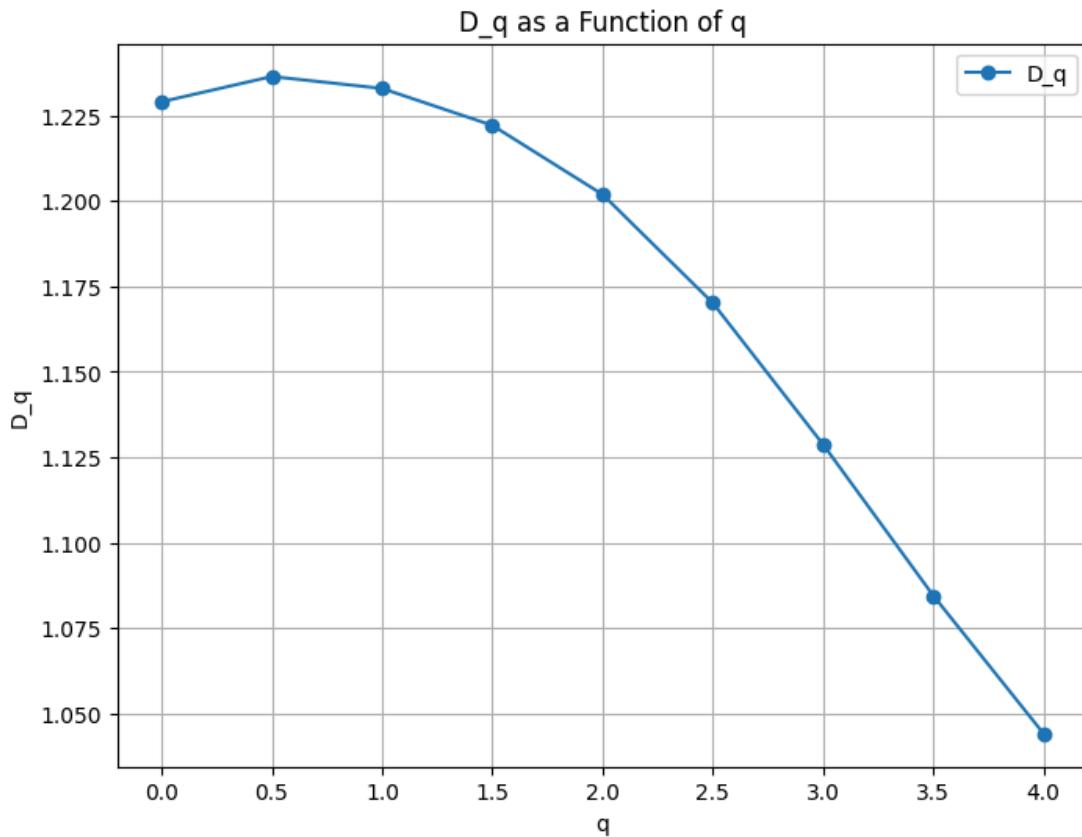
q_values = np.linspace(0, 4, 9) # 9 evenly spaced values between 0 and 4
```

```

slopes = [compute_slope(q) for q in q_values]

# Plot D_q as a function of q
plt.figure(figsize=(8, 6))
plt.plot(q_values, slopes, 'o-', label="D_q")
plt.xlabel("q")
plt.ylabel("D_q")
plt.title("D_q as a Function of q")
plt.grid(True)
plt.legend()
plt.show()

```



In [ ]:

```

tMax = 10000

data = np.zeros((tMax + 1, 2))
data[0] = [1, 1] # Initial condition
for t in range(1, tMax + 1):
    data[t] = henon_map(data[t - 1], a, b)

# Define the Jacobian function
def jacobian_func(x):
    return np.array([[ -2 * a * x, 1], [b, 0]])

jacobians = np.array([jacobian_func(x) for x in data[:, 0]])

Q_old = np.eye(2)
lambda1 = 0
lambda2 = 0
lambda_values = np.zeros((tMax, 3))

# QR Decomposition Loop
for t in range(tMax):

```

```

M_old = jacobians[t]
Q, R = np.linalg.qr(M_old @ Q_old)
Q_old = Q.T # Transpose Q for the next iteration
lambda1 += np.log(abs(R[0, 0]))
lambda2 += np.log(abs(R[1, 1]))
lambda_values[t] = [t + 1, lambda1 / (t + 1), lambda2 / (t + 1)]

# Final Lyapunov exponents
a = lambda1 / tMax
b = lambda2 / tMax

print(f'Largest Lyapunov Exponent (\lambda1): {a}')
print(f'Second Lyapunov Exponent (\lambda2): {b}')

```

Largest Lyapunov Exponent ( $\lambda_1$ ): 0.419751864790073  
Second Lyapunov Exponent ( $\lambda_2$ ): -1.6237246691160057

In [ ]: D\_L = 1 - a/b  
print(f'D\_L: {D\_L}')

D\_L: 1.2585117247855795

~\Documents\Skola dokue\Master Year 1\DynamicalStuff master 1\HW5\5.2\RenyiDim.py

```
1 %% Import libraries
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 %% a)
6 a = 1.4
7 b = 0.3
8 n = 1000000
9
10 def generate_data(start, a, b, n):
11     x, y = start
12     data = []
13     for _ in range(n):
14         new_x = y + 1 - a * x**2
15         new_y = b * x
16         data.append((new_x, new_y))
17         x, y = new_x, new_y
18     return np.array(data)
19
20 # Generate data for different starting points
21 data1 = generate_data((0.5, 0.5), a, b, n)[4:]
22 data2 = generate_data((1, 1), a, b, n)[4:]
23 data3 = generate_data((1.5, 1.5), a, b, n)[4:]
24
25 plt.figure(figsize=(15, 5))
26 plt.subplot(1, 3, 1)
27 plt.scatter(data1[:, 0], data1[:, 1], s=0.1)
28 plt.title("Initial Starting Point: (0.5, 0.5)")
29 plt.grid(True)
30 plt.subplot(1, 3, 2)
31 plt.scatter(data2[:, 0], data2[:, 1], s=0.1)
32 plt.title("Initial Starting Point: (1, 1)")
33 plt.grid(True)
34 plt.subplot(1, 3, 3)
35 plt.scatter(data3[:, 0], data3[:, 1], s=0.1)
36 plt.title("Initial Starting Point: (1.5, 1.5)")
37 plt.grid(True)
38 plt.tight_layout()
39 plt.show()
40
41 # %% b) and c)
42 iterations = 2 * 10**6
43 epsilons = np.arange(0.001, 0.02, 0.001)
44
45 def henon_map(state, a, b):
46     x, y = state
47     return y + 1 - a * x**2, b * x
48
49 data = np.zeros((iterations, 2))
50 data[0] = [0.1, 0.1]
51 for i in range(1, iterations):
```

```

52     data[i] = henon_map(data[i - 1], a, b)
53
54 # Function to compute slope and plot for different q values
55 def compute_and_plot(q, title):
56     bins_list = []
57     probabilities = []
58     for epsilon in epsilons:
59         x_bins = np.arange(np.min(data[:, 0]), np.max(data[:, 0]) + epsilon, epsilon)
60         y_bins = np.arange(np.min(data[:, 1]), np.max(data[:, 1]) + epsilon, epsilon)
61         hist, _, _ = np.histogram2d(data[:, 0], data[:, 1], bins=(x_bins, y_bins))
62         bins_list.append(hist.flatten())
63         probabilities.append(hist.flatten() / (iterations - 1))
64
65     if q == 1:
66         sum_prob = [np.sum(p[p > 0] * np.log(1 / p[p > 0])) for p in probabilities]
67         y_values = sum_prob
68     else:
69         sum_prob = [np.sum(p[p > 0] ** q) for p in probabilities]
70         y_values = [np.log(sp) / (1 - q) for sp in sum_prob]
71
72     x_values = np.log(1 / epsilons)
73
74     nominator = y_values[-1] - y_values[0]
75     denominator = x_values[-1] - x_values[0]
76     slope = nominator / denominator
77
78     plt.figure(figsize=(8, 6))
79     plt.plot(x_values, y_values, 'o-')
80     plt.xlabel('Log[1/ε]')
81     plt.ylabel('Log[sumProb]')
82     plt.title(title)
83     plt.grid(True)
84     plt.show()
85
86     print(f"Slope for {title}: {slope}")
87
88 compute_and_plot(0, "q=0")
89 compute_and_plot(1, "q=1")
90 compute_and_plot(2, "q=2")
91
92 # %% d)
93
94 data = np.zeros((iterations, 2))
95 data[0] = [0.1, 0.1]
96 for i in range(1, iterations):
97     data[i] = henon_map(data[i - 1], a, b)
98
99 # Function to compute slope for a given q
100 def compute_slope(q):
101     bins_list = []
102     probabilities = []
103     for epsilon in epsilons:
104         x_bins = np.arange(np.min(data[:, 0]), np.max(data[:, 0]) + epsilon, epsilon)
105         y_bins = np.arange(np.min(data[:, 1]), np.max(data[:, 1]) + epsilon, epsilon)

```

```

106         hist, _, _ = np.histogram2d(data[:, 0], data[:, 1], bins=(x_bins, y_bins))
107         bins_list.append(hist.flatten())
108         probabilities.append(hist.flatten() / (iterations - 1))
109
110     if q == 1:
111         sum_prob = [np.sum(p[p > 0] * np.log(1 / p[p > 0])) for p in probabilities]
112         y_values = sum_prob
113     else:
114         sum_prob = [np.sum(p[p > 0] ** q) for p in probabilities]
115         y_values = [np.log(sp) / (1 - q) for sp in sum_prob]
116
117     x_values = np.log(1 / epsilons)
118
119     nominator = y_values[-1] - y_values[0]
120     denominator = x_values[-1] - x_values[0]
121     return nominator / denominator
122
123 q_values = np.linspace(0, 4, 9) # 9 evenly spaced values between 0 and 4
124 slopes = [compute_slope(q) for q in q_values]
125
126 # Plot D_q as a function of q
127 plt.figure(figsize=(8, 6))
128 plt.plot(q_values, slopes, 'o-', label="D_q")
129 plt.xlabel("q")
130 plt.ylabel("D_q")
131 plt.title("D_q as a Function of q")
132 plt.grid(True)
133 plt.legend()
134 plt.show()
135
136 # %% e)
137 tMax = 10000
138
139 data = np.zeros((tMax + 1, 2))
140 data[0] = [1, 1] # Initial condition
141 for t in range(1, tMax + 1):
142     data[t] = henon_map(data[t - 1], a, b)
143
144 # Define the Jacobian function
145 def jacobian_func(x):
146     return np.array([[ -2 * a * x, 1], [b, 0]])
147
148 jacobians = np.array([jacobian_func(x) for x in data[:, 0]])
149
150 Q_old = np.eye(2)
151 lambda1 = 0
152 lambda2 = 0
153 lambda_values = np.zeros((tMax, 3))
154
155 # QR Decomposition Loop
156 for t in range(tMax):
157     M_old = jacobians[t]
158     Q, R = np.linalg.qr(M_old @ Q_old)
159     Q_old = Q.T # Transpose Q for the next iteration

```

```
160     lambda1 += np.log(np.abs(R[0, 0]))
161     lambda2 += np.log(np.abs(R[1, 1]))
162     lambda_values[t] = [t + 1, lambda1 / (t + 1), lambda2 / (t + 1)]
163
164 # Final Lyapunov exponents
165 a = lambda1 / tMax
166 b = lambda2 / tMax
167
168 print(f'Largest Lyapunov Exponent ( $\lambda_1$ ): {a}')
169 print(f'Second Lyapunov Exponent ( $\lambda_2$ ): {b}')
170
171
172 # %% f)
173 D_L = 1 - a/b
174 print(f'D_L: {D_L}')
175
176 # %%
177
```