Homework Assignment 5.2

Max Green TIF155

December 22, 2024

Exercise a

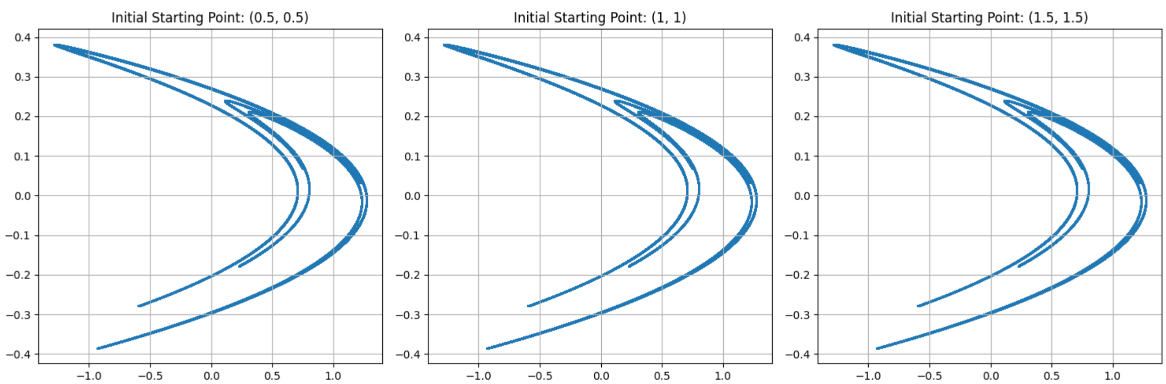
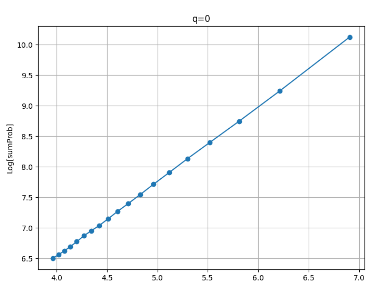
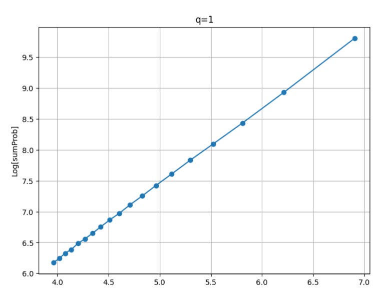


Figure 1: Approximation of the fractal attractor.

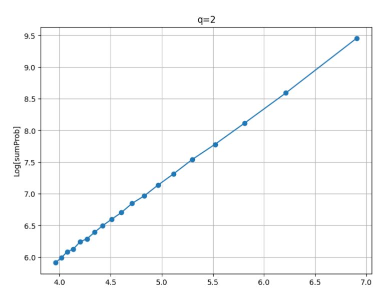
b



(a) q = 0



(b) q = 1



(c) q = 2

Figure 2: Three plots with different values for q.

c

See code for calculation.

d

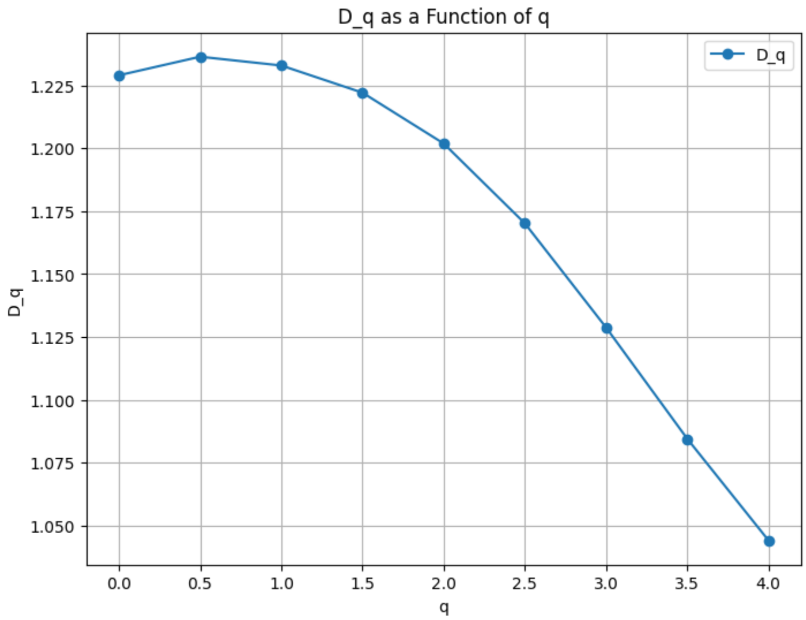


Figure 3: Dq as a function of q.

e

See code.

f

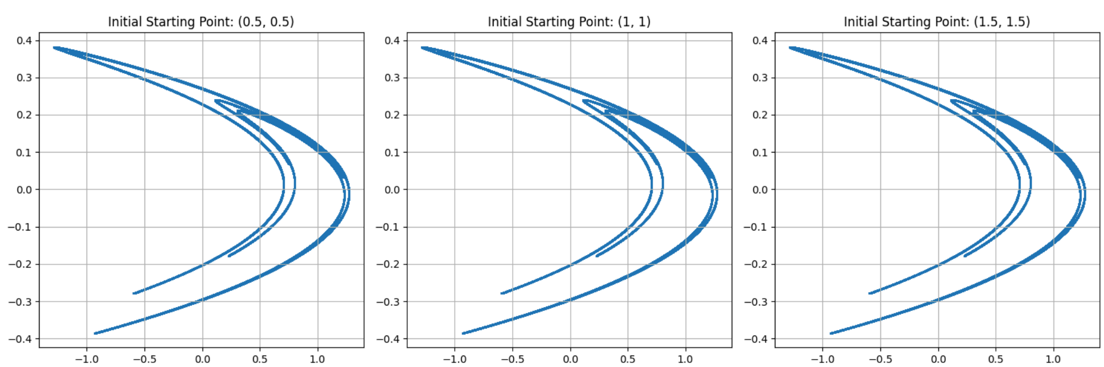
See code.

In [ ]:

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

In [ ]:

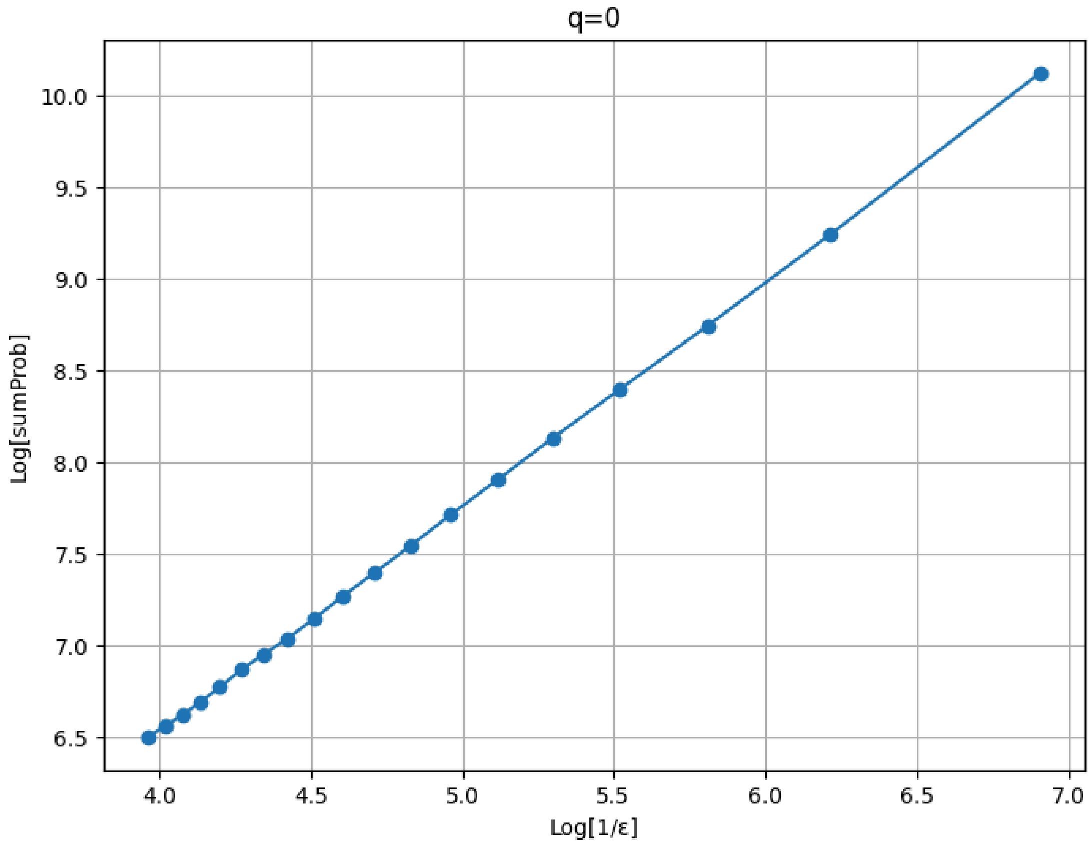
```
a = 1.4
b = 0.3
n = 1000000

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()
```



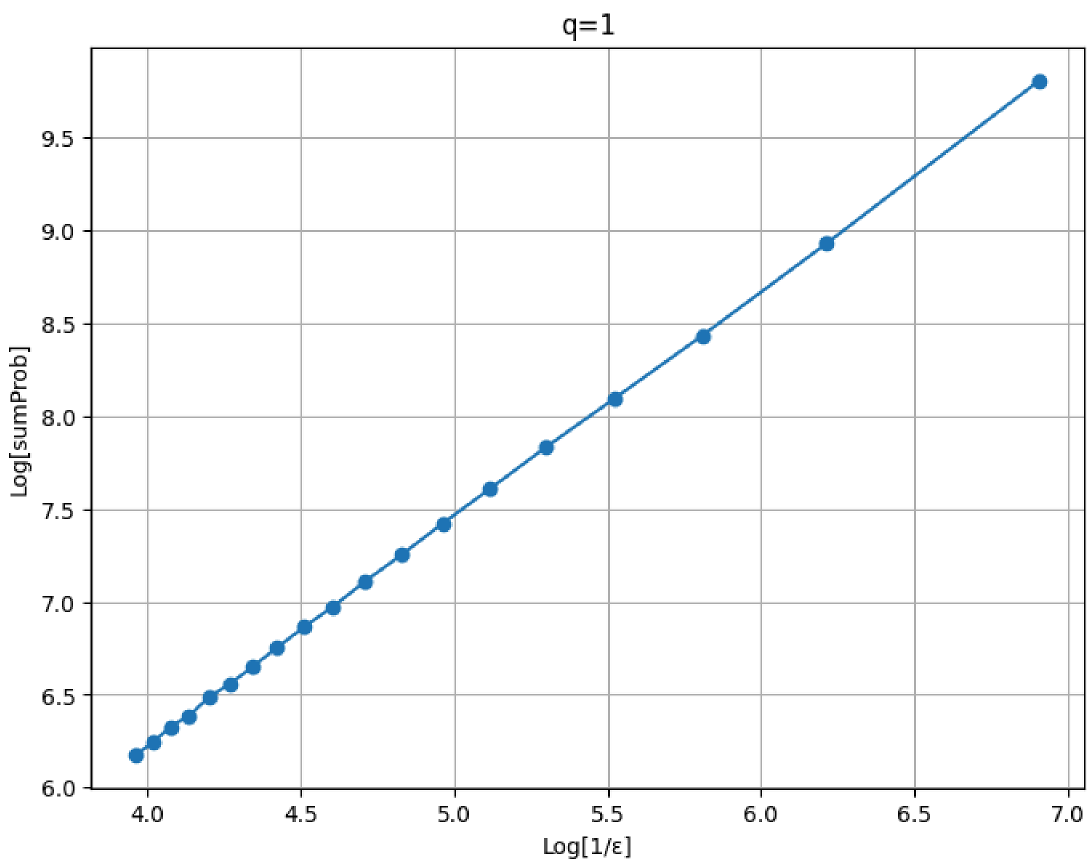
In [ ]:

```
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))
```

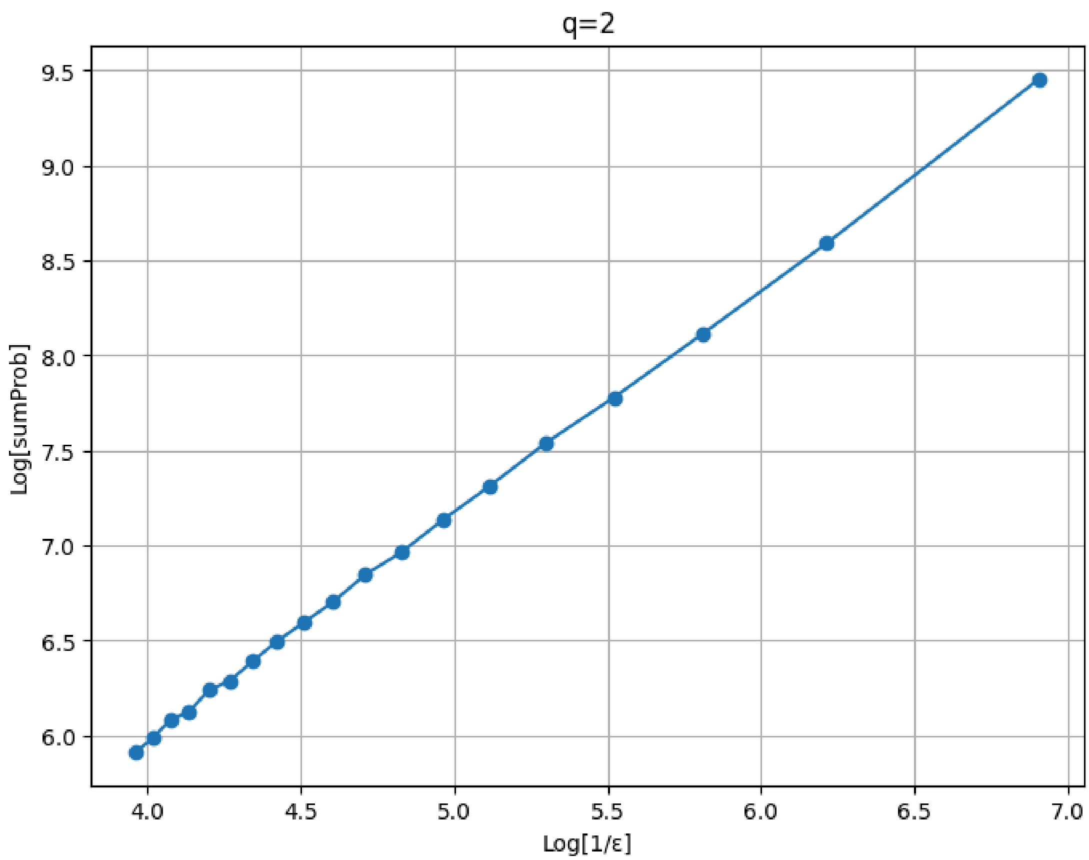
```
data[0] = [0.1, 0.1]
for i in range(1, iterations):
    data[i] = henon_map(data[i - 1], a, b)

# 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, eps
        y_bins = np.arange(np.min(data[:, 1]), np.max(data[:, 1]) + epsilon, eps
        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 probabiliti
        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 / epsilons)

    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")
```




```
Slope for q=0: 1.2290383810689962
```



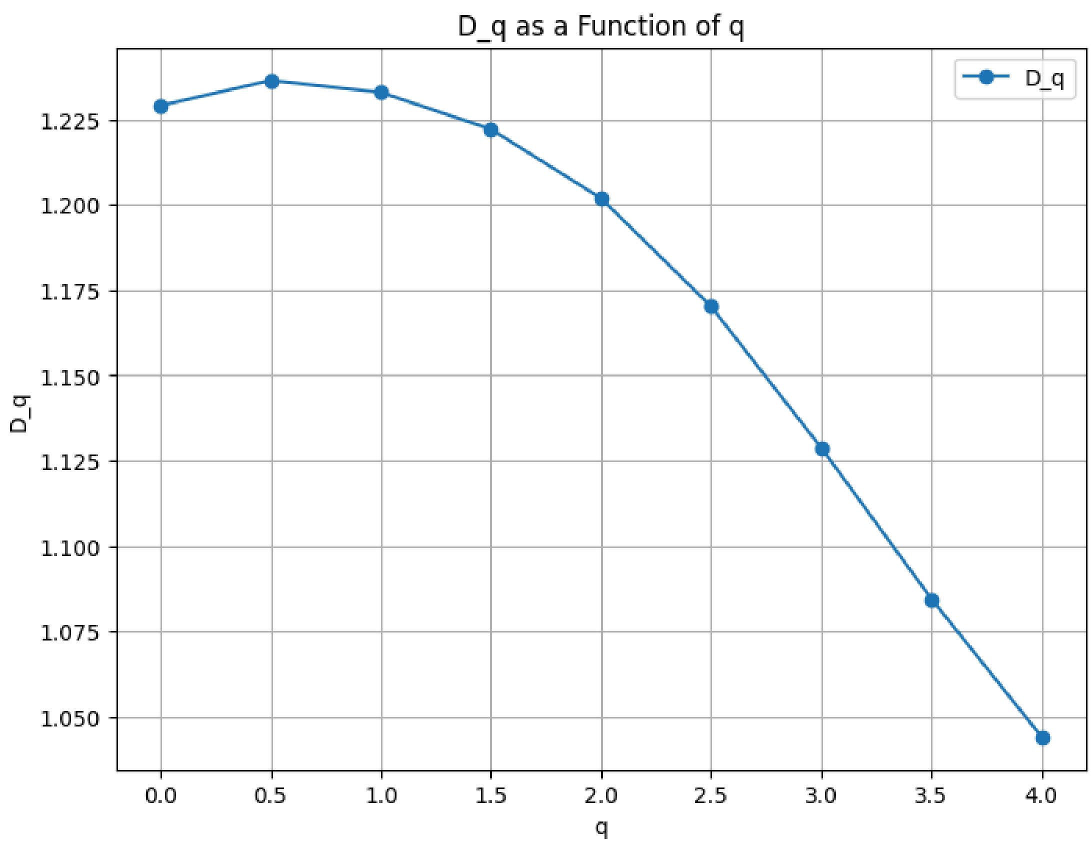

```
Slope for q=1: 1.2329288099159232
```




```
Slope for q=2: 1.2018917070332482
```

In [ ]:
```python
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, eps
        y_bins = np.arange(np.min(data[:, 1]), np.max(data[:, 1]) + epsilon, eps
        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 probabiliti
        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 / epsilons)

    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):
```

```python
    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 (λ1): {a}")
print(f"Second Lyapunov Exponent (λ2): {b}")
```

```
Largest Lyapunov Exponent (λ1): 0.419751864790073
Second Lyapunov Exponent (λ2): -1.6237246691160057
```

In [ ]:

```python
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

```python
#%% Import libraries
import numpy as np
import matplotlib.pyplot as plt

#%% a)
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()

# %% b) and c)
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):
```

```python
    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 / epsilons)

    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")

# %% d)

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)
```

```python
        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 / epsilons)

    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()

# %% e)
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 (λ1): {a}")
print(f"Second Lyapunov Exponent (λ2): {b}")


# %% f)
D_L = 1 - a/b
print(f'D_L: {D_L}')

# %%

```