Assignment

Double Pendulum

Name TN 1: Tim Peinkofer

tim.peinkofer@student.uni-tuebingen.de

Name TN 2: Fabian Kostow

fabian.kostow@student.uni-tuebingen.de

Tutor: Jose Carlos Olvera Meneses

Date: 18. Januar 2025

Inhaltsverzeichnis

1 Question 1 and 2

We first want to plot the the behavior of the double pendulum based on the following initial conditions:

$$\theta_1' = 0$$
$$\theta_2' = 0$$
$$\theta_1 = \frac{\pi}{2}$$
$$\theta_2 = \frac{\pi}{2}$$

We will use the code below to plot the solution:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Constants
g = 9.8
L1 = 1
L2 = 1
m1 = 1
m2 = 1

def double_pendulum(t, state): # Definition of the function
    theta1, theta2, omega1, omega2 = state

    # Definition of the ODEs
    dtheta1_dt = omega1
    dtheta2_dt = omega2

    denom1 = L1 * (2 * m1 + m2 - m2 * np.cos(2 * theta1 - 2 *
                theta2))
    denom2 = L2 * (2 * m1 + m2 - m2 * np.cos(2 * theta1 - 2 *
                theta2))
    omega1_dot = (-g * (2 * m1 + m2) * np.sin(theta1) - m2 *
                               g * np.sin(theta1 - 2 *
                               theta2)
                  - 2 * np.sin(theta1 - theta2) * m2 * (
                                             omega2 ** 2
                                             * L2 +
                                             omega1 ** 2
                                             * L1 * np.
                                             cos(theta1 -
                                              theta2))
                  ) / denom1

    omega2_dot = (2 * np.sin(theta1 - theta2) * (omega1 ** 2
                               * L1 * (m1 + m2) + g * (m1
                                + m2) * np.cos(theta1)
                  + omega2 ** 2 * L2 * m2 * np.cos(theta1 -
                                             theta2))) /
                                             denom2

    return np.array([dtheta1_dt, dtheta2_dt, omega1_dot,
                               omega2_dot])

# 4th Order Runge Kutta
def runge_kutta(f, y_0, t_start, t_end, n):
    h = (t_end - t_start) / n  # Stepsize
    t_values = np.linspace(t_start, t_end, n + 1)
    y_values = np.zeros((n + 1, len(y_0)))
    y_values[0] = y_0

    for i in range(n):
        t_i = t_values[i]
        y_i = y_values[i]

        k1 = h * f(t_i, y_i)
        k2 = h * f(t_i + 0.5 * h, y_i + 0.5 * k1)
        k3 = h * f(t_i + 0.5 * h, y_i + 0.5 * k2)
        k4 = h * f(t_i + h, y_i + k3)

        y_values[i + 1] = y_i + (1 / 6) * (k1 + 2 * k2 + 2 *
                                   k3 + k4)

    return t_values, y_values
```

```python
# Boundary conditions
theta1_0 = np.pi / 2
theta2_0 = np.pi / 2
omega1_0 = 0
omega2_0 = 0
y_0 = [theta1_0, theta2_0, omega1_0, omega2_0]


t_start = 0
t_end = 40
n = 1800
dt = (t_end - t_start) / n


t, result = runge_kutta(double_pendulum, y_0, t_start, t_end,
                                n)

# Positions
theta1 = result[:, 0]
theta2 = result[:, 1]
x1 = L1 * np.sin(theta1)
y1 = -L1 * np.cos(theta1)
x2 = x1 + L2 * np.sin(theta2)
y2 = y1 - L2 * np.cos(theta2)

# Plot
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(t, theta1, label="Theta1 (Angle 1)")
plt.plot(t, theta2, label="Theta2 (Angle 2)")
plt.xlabel("Time (s)")
plt.ylabel("Angle (rad)")
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(t, result[:, 2], label="Omega1 (Velocity 1)")
plt.plot(t, result[:, 3], label="Omega2 (Velocity 2)")
plt.xlabel("Time (s)")
plt.ylabel("Velocity (rad/s)")
plt.legend()

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

```
# Plot und Animation
fig, ax = plt.subplots()
ax.set_aspect('equal', adjustable='box')
ax.set_xlim(-2 * L1 - 0.1, 2 * L1 + 0.1)
ax.set_ylim(-2 * L1 - 0.1, 2 * L1 + 0.1)

line, = ax.plot([], [], 'o-', lw=2, label="Pendelum")
path, = ax.plot([], [], 'r-', lw=1, label="Path")
trail_x, trail_y = [], []


def init():
    line.set_data([], [])
    path.set_data([], [])
    trail_x.clear()
    trail_y.clear()
    return line, path

# Animationsfunktion
def update(frame):
    x_positions = [0, x1[frame], x2[frame]]
    y_positions = [0, y1[frame], y2[frame]]

    line.set_data(x_positions, y_positions)
    trail_x.append(x2[frame])
    trail_y.append(y2[frame])
    path.set_data(trail_x, trail_y)

    return line, path

# Generate animation
ani = FuncAnimation(fig, update, frames=len(t), init_func=
                               init, blit=True, interval=50)
                                # Intervall angepasst

plt.legend()
plt.title("Douple pendulum")
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.show()
```

As a result we get the following plots:

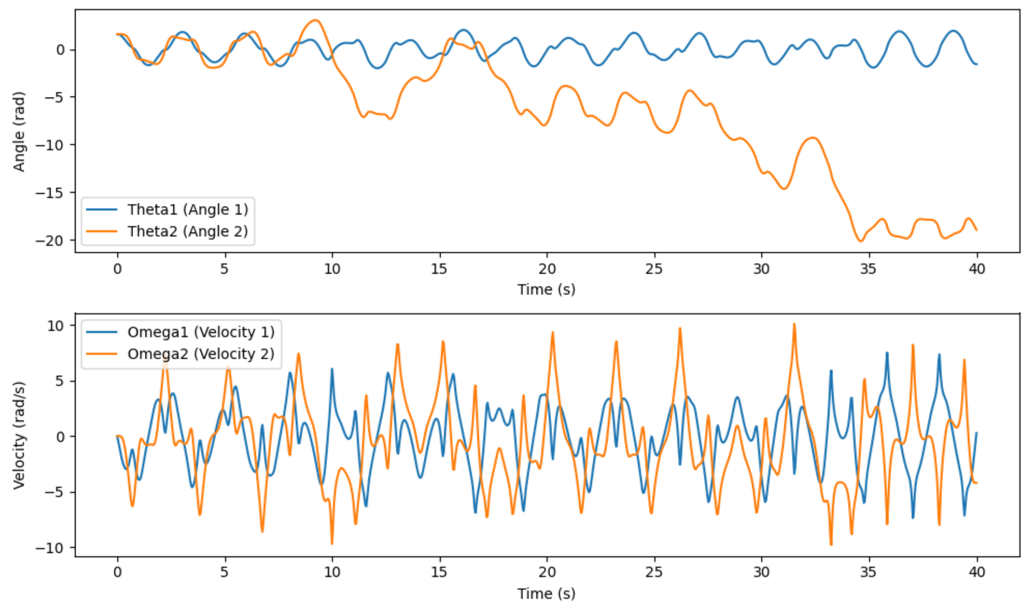

Abbildung 1: θ_1 and θ_2 behavior

The animation of the pendulum can be found on GitHub. The image of a fixed point in time is shown below.

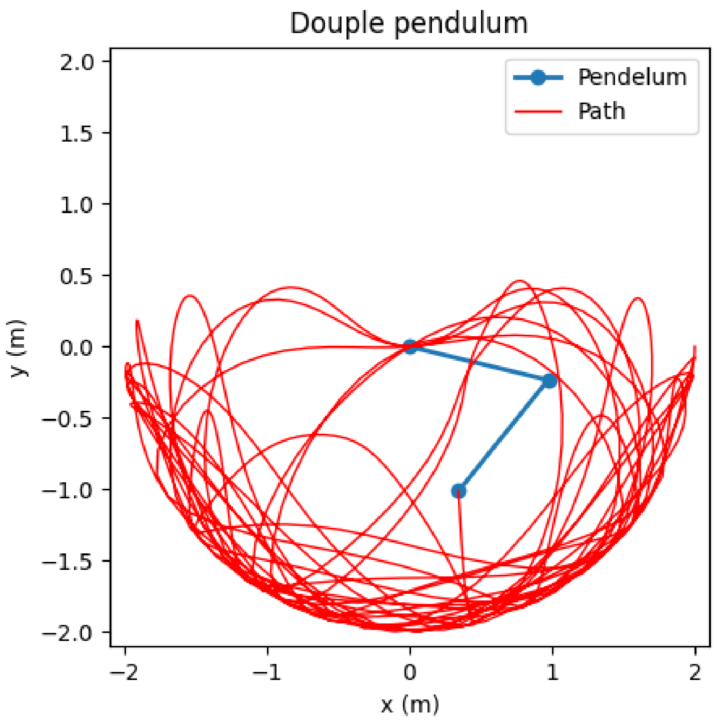

Abbildung 2: Animation of the pendulum

2 Question 3

As we see above we can verify the chaotic behavior of our double pendulum. We can also see it in the animation part. We also get different results if we change the initial condition by a little constant ϵ which is an indicator of a chaotic behavior.

3 Question 4

In the next step we want to calculate the Lyapunov exponent of our problem. The code is at follows:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Constants
g = 9.8
L1 = 1
L2 = 1
m1 = 1
m2 = 1

# Function to calculate Lyapunov exponent
def lyapunov_exponent(t, sol, epsilon_0):
    # Calculate the logarithmic growth of the separation
    return np.log(sol / epsilon_0) / t

def double_pendulum(t, state):
    theta1, theta2, omega1, omega2 = state

    # Definition of the ODEs
    dtheta1_dt = omega1
    dtheta2_dt = omega2

    denom1 = L1 * (2 * m1 + m2 - m2 * np.cos(2 * theta1 - 2 *
                                             theta2))
    denom2 = L2 * (2 * m1 + m2 - m2 * np.cos(2 * theta1 - 2 *
                                             theta2))

    omega1_dot = (-g * (2 * m1 + m2) * np.sin(theta1) - m2 *
                                    g * np.sin(theta1 - 2 *
                                    theta2)
                  - 2 * np.sin(theta1 - theta2) * m2 * (
                                                  omega2 ** 2
                                                  * L2 +
                                                  omega1 ** 2
                                                  * L1 * np.
                                                  cos(theta1 -
                                                   theta2))
                  ) / denom1
    omega2_dot = (2 * np.sin(theta1 - theta2) * (omega1 ** 2
                                   * L1 * (m1 + m2) + g * (m1
                                    + m2) * np.cos(theta1)
                  + omega2 ** 2 * L2 * m2 * np.cos(theta1 -
                                                  theta2))) /
                                                  denom2

    return np.array([dtheta1_dt, dtheta2_dt, omega1_dot,
                                   omega2_dot])

# 4th Order Runge Kutta
def runge_kutta(f, y_0, t_start, t_end, n):
    h = (t_end - t_start) / n  # Stepsize
    t_values = np.linspace(t_start, t_end, n + 1)
    y_values = np.zeros((n + 1, len(y_0)))
    y_values[0] = y_0

    for i in range(n):
        t_i = t_values[i]
        y_i = y_values[i]

        k1 = h * f(t_i, y_i)
        k2 = h * f(t_i + 0.5 * h, y_i + 0.5 * k1)
        k3 = h * f(t_i + 0.5 * h, y_i + 0.5 * k2)
        k4 = h * f(t_i + h, y_i + k3)

        y_values[i + 1] = y_i + (1 / 6) * (k1 + 2 * k2 + 2 *
                                     k3 + k4)

    return t_values, y_values

# Boundary conditions
theta1_0 = np.pi / 2
theta2_0 = np.pi / 2
omega1_0 = 0
omega2_0 = 0
y_0 = [theta1_0, theta2_0, omega1_0, omega2_0]

t_start = 0
t_end = 40
n = 1800
dt = (t_end - t_start) / n

# Solve for the first trajectory
t_1, result_1 = runge_kutta(double_pendulum, y_0, t_start,
                               t_end, n)
```

```python
# Perturbed initial conditions
theta1_0 = np.pi / 2 + 1e-3
theta2_0 = np.pi / 2
omega1_0 = 0
omega2_0 = 0
y_0 = [theta1_0, theta2_0, omega1_0, omega2_0]

# Solve for the second trajectory
t_2 , result_2 = runge_kutta(double_pendulum, y_0, t_start,
                               t_end, n)

# Positions for both trajectories
theta1_1 = result_1[:, 0]
theta2_1 = result_1[:, 1]
x1_1 = L1 * np.sin(theta1_1)
y1_1 = -L1 * np.cos(theta1_1)
x2_1 = x1_1 + L2 * np.sin(theta2_1)
y2_1 = y1_1 - L2 * np.cos(theta2_1)

theta1_2 = result_2[:, 0]
theta2_2 = result_2[:, 1]
x1_2 = L1 * np.sin(theta1_2)
y1_2 = -L1 * np.cos(theta1_2)
x2_2 = x1_2 + L2 * np.sin(theta2_2)
y2_2 = y1_2 - L2 * np.cos(theta2_2)

# Separation between the two trajectories
sol = np.sqrt((x2_1 - x2_2)**2 + (y2_1 - y2_2)**2 + (x1_1 -
                               x1_2)**2 + (y1_1 - y1_2)**2)

# Initial separation (epsilon_0)
epsilon_0 = np.sqrt((x2_1[0] - x2_2[0])**2 + (y2_1[0] - y2_2[
                               0])**2 + (x1_1[0] - x1_2[0])**
                               2 + (y1_1[0] - y1_2[0])**2)

# Calculate Lyapunov exponent over time
lyapunov_vals = lyapunov_exponent(t_1, sol, epsilon_0)

# Plot the Lyapunov exponent over time
print('Lyapunov Exponent:', lyapunov_vals[-1])
plt.plot(t_1, lyapunov_vals[-1]*np.ones(len(t_1)), 'r--')
plt.plot(t_1, lyapunov_vals)
plt.xlabel('Time (s)')
plt.ylabel('Lyapunov Exponent')
plt.title('Lyapunov Exponent for the Double Pendulum')
plt.show()
```

If we now plot the solution we can predict that our pendulum will have an chaotic behavior because the exponent is greater then zero.

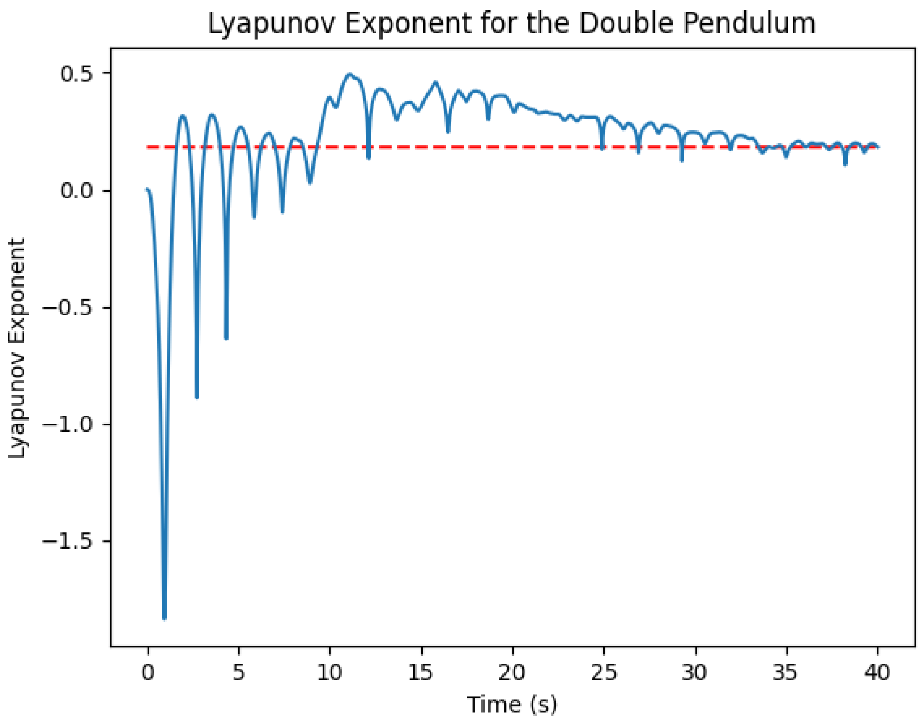

Abbildung 3: Convergence of the Lyapunov exponent

4 Question 5

The chaotic behavior can be generate in different ways. One possible way would be to increase the mass m_1 or use different lengths for the two pendulums. If we now increase the mass m_1 by 1 kg we will get the following result:

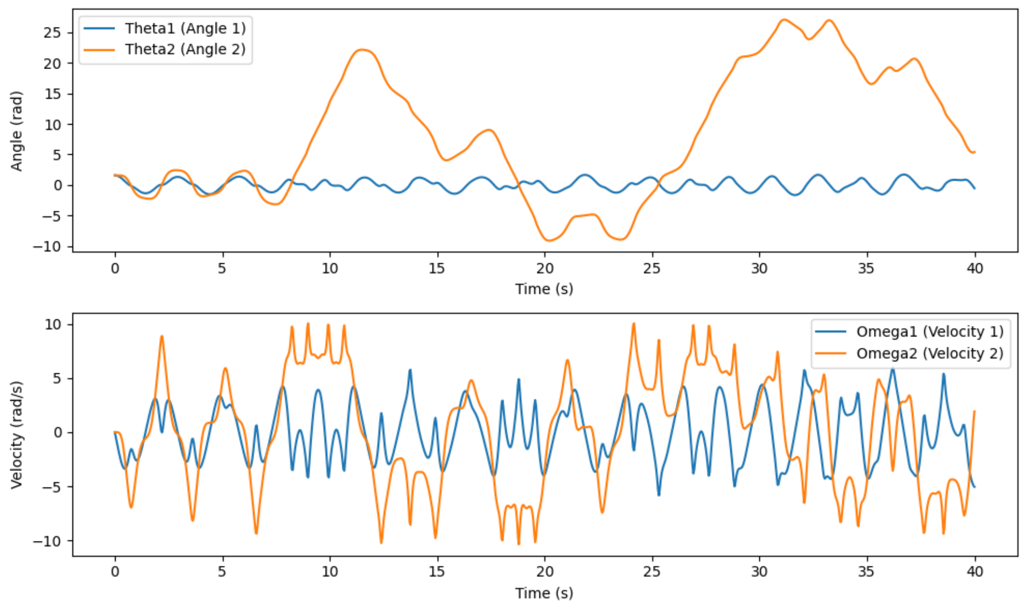

Abbildung 4: θ_1 and θ_2 behavior with bigger m_1

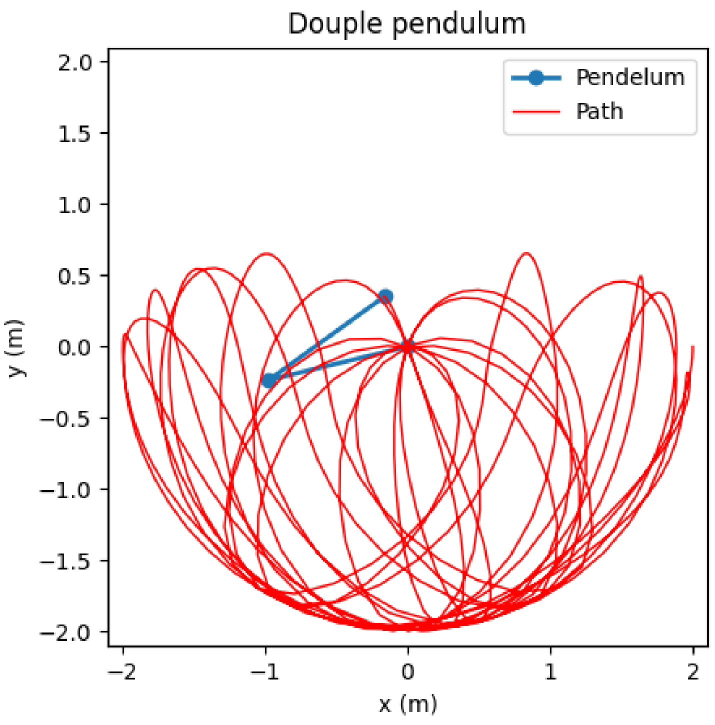

Abbildung 5: Animation of the pendulum with bigger m_1

As we see the number of rollovers increases directly. These rollovers are a important note of the chaotic behavior. As much rollovers the system has, as more chaotic it is.

5 Question 6

Let us now take a look, what happens, if we use different stepsizes.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Constants
g = 9.8
L1 = 1
L2 = 1
m1 = 1
m2 = 1

def double_pendulum(t, state):  # Definition of the function
    theta1, theta2, omega1, omega2 = state

    # Definition of the ODEs
    dtheta1_dt = omega1
    dtheta2_dt = omega2

    denom1 = L1 * (2 * m1 + m2 - m2 * np.cos(2 * theta1 - 2 *
                                 theta2))
    denom2 = L2 * (2 * m1 + m2 - m2 * np.cos(2 * theta1 - 2 *
                                 theta2))

    omega1_dot = (-g * (2 * m1 + m2) * np.sin(theta1) - m2 *
                                g * np.sin(theta1 - 2 *
                                theta2)
                  - 2 * np.sin(theta1 - theta2) * m2 * (
                                              omega2 ** 2
                                              * L2 +
                                              omega1 ** 2
                                              * L1 * np.
                                              cos(theta1 -
                                               theta2))
                  ) / denom1

    omega2_dot = (2 * np.sin(theta1 - theta2) * (omega1 ** 2
                                * L1 * (m1 + m2) + g * (m1
                                 + m2) * np.cos(theta1)
                  + omega2 ** 2 * L2 * m2 * np.cos(theta1 -
                                              theta2))) /
                                              denom2

    return np.array([dtheta1_dt, dtheta2_dt, omega1_dot,
                                omega2_dot])
```

```
# 4th Order Runge Kutta
def runge_kutta(f, y_0, t_start, t_end, n):
    h = (t_end - t_start) / n  # Stepsize
    t_values = np.linspace(t_start, t_end, n + 1)
    y_values = np.zeros((n + 1, len(y_0)))
    y_values[0] = y_0

    for i in range(n):
        t_i = t_values[i]
        y_i = y_values[i]

        k1 = h * f(t_i, y_i)
        k2 = h * f(t_i + 0.5 * h, y_i + 0.5 * k1)
        k3 = h * f(t_i + 0.5 * h, y_i + 0.5 * k2)
        k4 = h * f(t_i + h, y_i + k3)

        y_values[i + 1] = y_i + (1 / 6) * (k1 + 2 * k2 + 2 *
                                             k3 + k4)

    return t_values, y_values

# Boundary conditions
theta1_0 = np.pi / 2
theta2_0 = np.pi / 2
omega1_0 = 0
omega2_0 = 0
y_0 = [theta1_0, theta2_0, omega1_0, omega2_0]

t_start = 0
t_end = 40
n = 1800

# Plotting the trajectories for different step sizes
plt.figure(figsize=(6, 6))
for i in range(1, 10):
    n_steps = 1800 * i
    t, result = runge_kutta(double_pendulum, y_0, t_start,
                                  t_end, n_steps)
    theta1 = result[:, 0]
    theta2 = result[:, 1]
    dt = (t_end - t_start) / n_steps  # Calculate dt for
                                  current n_steps
    plt.plot(theta1, theta2, label=f'Stepsize: {dt:.4f}')
plt.xlabel(r'$\theta_1$')
plt.ylabel(r'$\theta_2$')
plt.title('Trajectory of the $(\\theta_1, \\theta_2)$-space')
plt.legend()
plt.grid(True)
plt.axis('equal')  # Um gleiche Skalierung f r beide Achsen
                             zu gew hrleisten
plt.show()
```

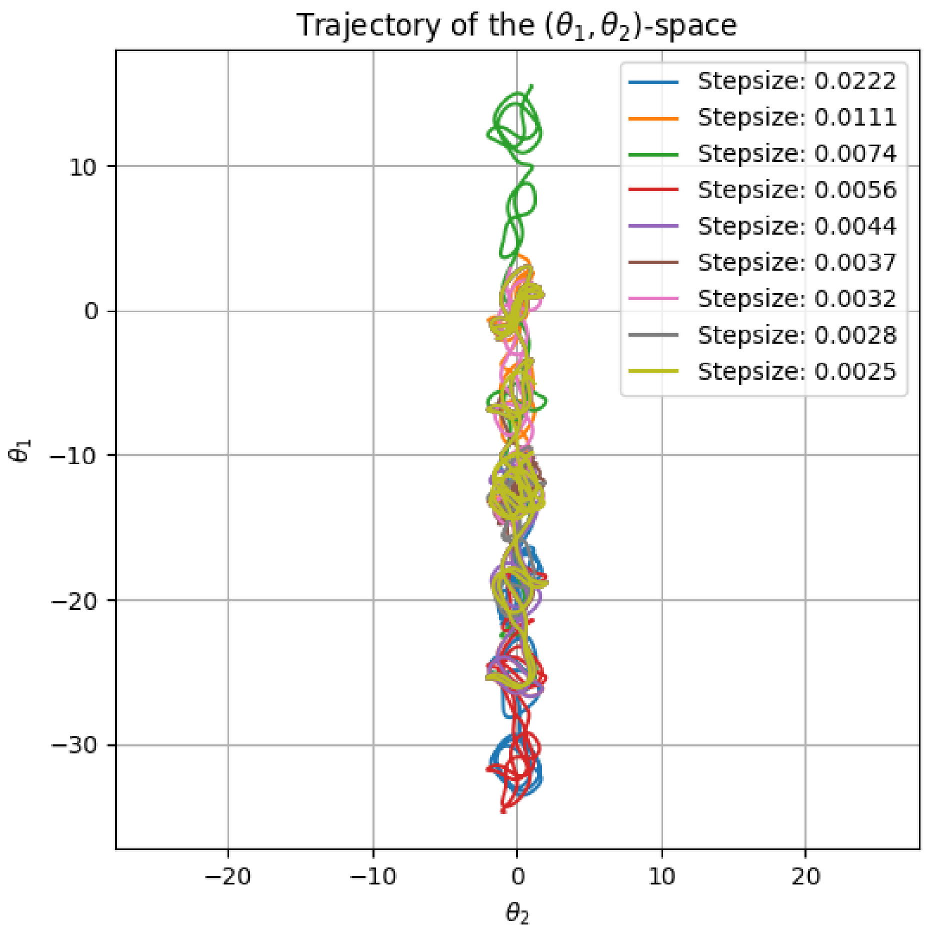

Abbildung 6: Tracetory of the θ_1-θ_2-space

The result is a different distribution based on the choice of our stepsize.