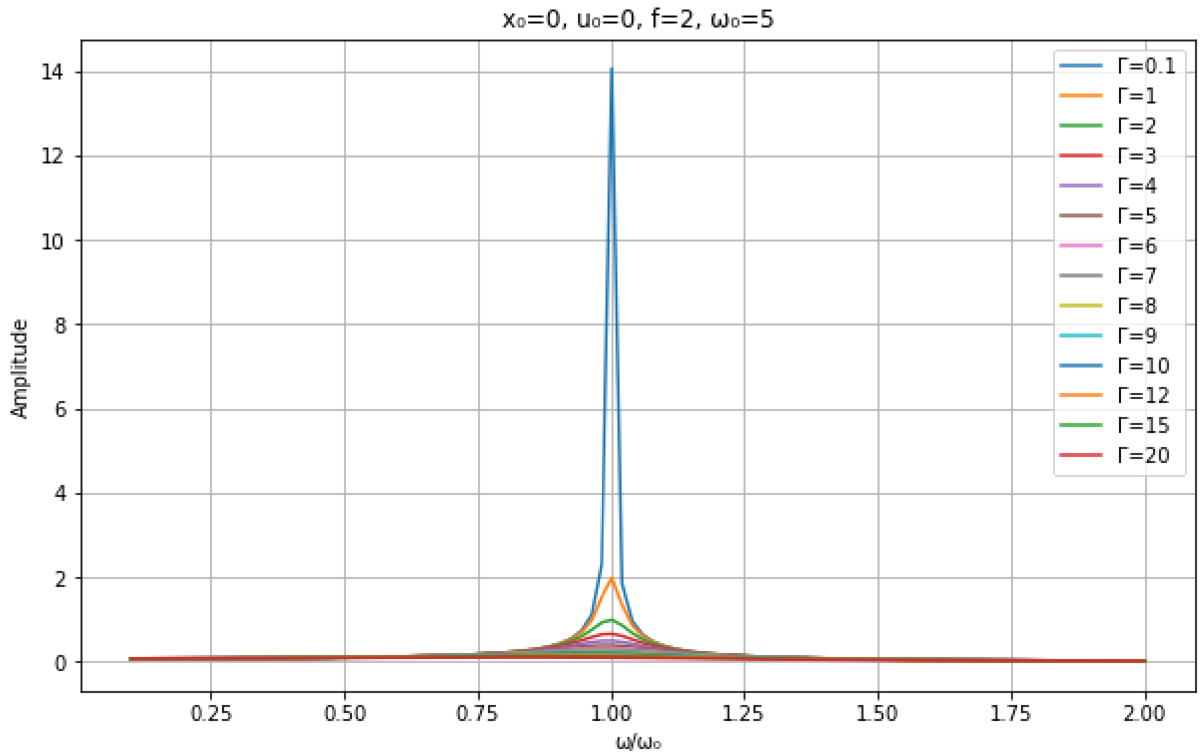
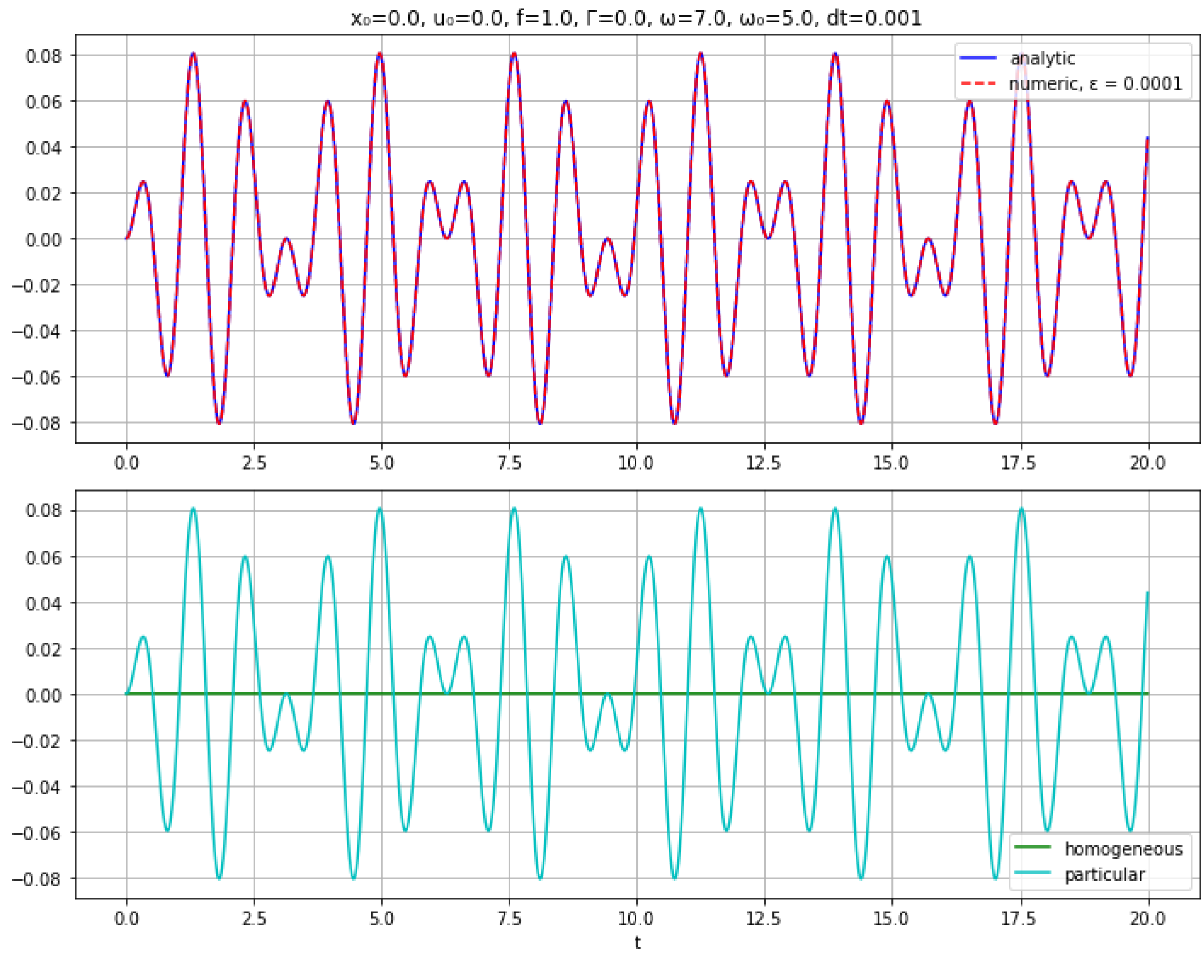
Paweł Mozgowiec sprawozdanie LAB3

ZAD.1

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

def euler_method(x0, u0, f, Gamma, omega, omega0, dt, tmax):
    # Initialize arrays
    t = np.arange(0, tmax, dt)
    n = len(t)
    x = np.zeros(n)
    u = np.zeros(n)

    # Set initial conditions
    x[0] = x0
    u[0] = u0

    # Euler method
    for i in range(1, n):
        x[i] = x[i-1] + dt * u[i-1]
        u[i] = u[i-1] + dt * (f * np.cos(omega * t[i]) - (Gamma/omega) * u[i-1] - omega0**2 * x[i])

    return t, x, u

def analytic_solution(x0, u0, f, Gamma, omega, omega0, t):
    # For the undamped case (Gamma = 0)
    if Gamma == 0:
        # Homogeneous solution
        xh = x0 * np.cos(omega0 * t) + (u0/omega0) * np.sin(omega0 * t)

        # Particular solution (steady-state)
        if abs(omega - omega0) < 1e-10:  # For resonance case
            xp = (f/(2*omega0)) * t * np.sin(omega0 * t)
        else:
            xp = f * (np.cos(omega * t) - np.cos(omega0 * t)) / (omega0**2 - omega**2)
    else:
        # For the damped case, formula becomes more complex
        # This is a simplified approximation
        gamma = Gamma/omega
        denom = (omega0**2 - omega**2)**2 + (gamma * omega)**2
        A = f / np.sqrt(denom)
        phi = np.arctan2(gamma * omega, omega0**2 - omega**2)

        # Homogeneous solution (decaying oscillation)
        xh = np.exp(-gamma*t/2) * (x0 * np.cos(np.sqrt(omega0**2 - (gamma/2)**2) * t) +
                                   (u0 + gamma*x0/2)/np.sqrt(omega0**2 - (gamma/2)**2) *
                                   np.sin(np.sqrt(omega0**2 - (gamma/2)**2) * t))

        # Particular solution
        xp = A * np.cos(omega * t - phi)

    return xh, xp, xh + xp

# Parameters
x0 = 0.0      # Initial position
u0 = 0.0      # Initial velocity
f = 1.0       # Force amplitude
Gamma = 0.0   # Damping coefficient
omega = 7.0   # Driving frequency
omega0 = 5.0  # Natural frequency
dt = 1.0e-3   # Time step
tmax = 20.0   # End time
```

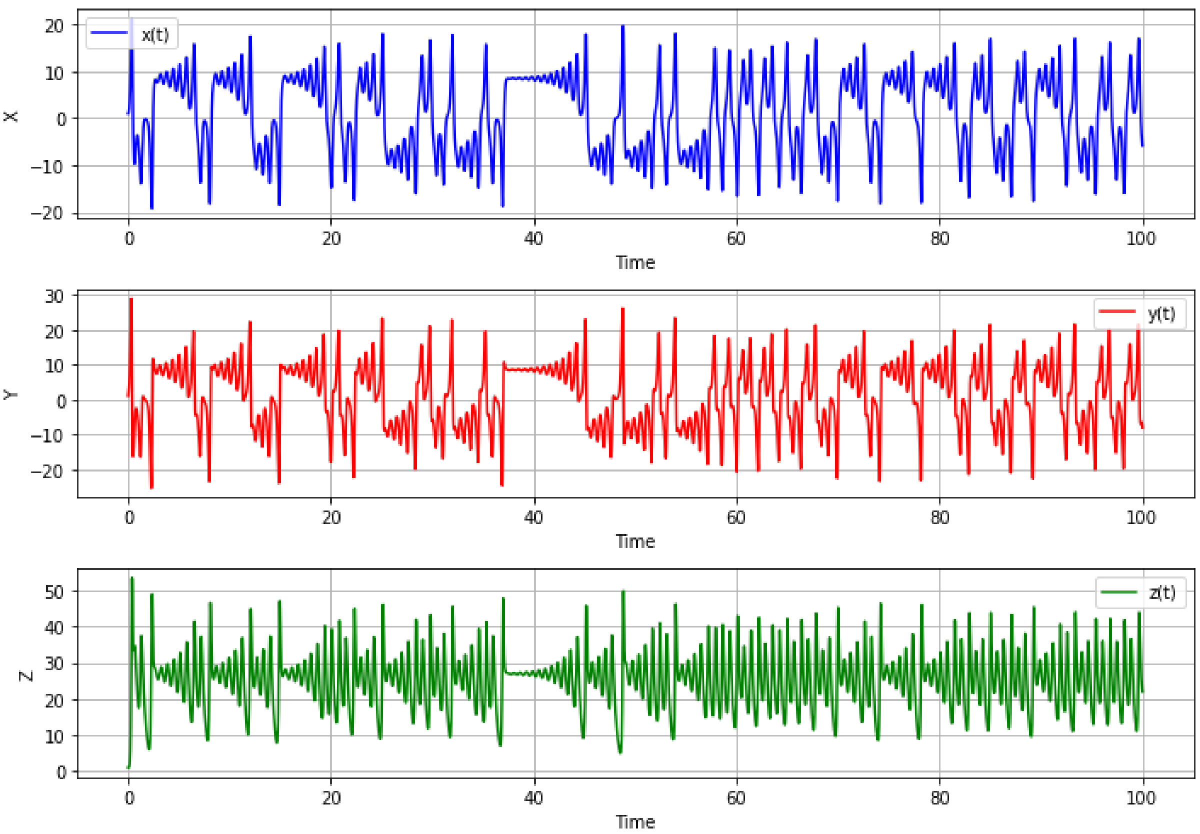
```python
# Numerical solution
t, x_numeric, u_numeric = euler_method(x0, u0, f, Gamma, omega, omega0, dt, tmax)

# Analytic solution
x_homogeneous, x_particular, x_analytic = analytic_solution(x0, u0, f, Gamma, omega, omega0, t)

# Error calculation
error = np.sum(np.abs(x_numeric - x_analytic)) / len(t)

# Plotting
plt.figure(figsize=(10, 8))

# Plot 1: Numerical vs Analytic Solution
plt.subplot(2, 1, 1)
plt.plot(t, x_analytic, 'b-', label='analytic')
plt.plot(t, x_numeric, 'r--', label=f'numeric, ε = {error:.4f}')
plt.title(f'x₀={x0}, u₀={u0}, f={f}, Γ={Gamma}, ω={omega}, ω₀={omega0}, dt={dt}')
plt.legend()
plt.grid(True)

# Plot 2: Homogeneous vs Particular Solution
plt.subplot(2, 1, 2)
plt.plot(t, x_homogeneous, 'g-', label='homogeneous')
plt.plot(t, x_particular, 'c-', label='particular')
plt.legend()
plt.grid(True)
plt.xlabel('t')

plt.tight_layout()
plt.show()

# Now let's study the amplitude response vs frequency ratio
def amplitude_response(f, Gamma_values, omega0, omega_ratio):
    """Calculate amplitude for various damping and frequency ratios"""
    amplitudes = {}

    for Gamma in Gamma_values:
        amp = []
        for ratio in omega_ratio:
            omega = ratio * omega0
            denom = (omega0**2 - omega**2)**2 + (Gamma * omega/omega0)**2
            A = f / np.sqrt(denom)
            amp.append(A)
        amplitudes[Gamma] = amp

    return amplitudes

# Calculate amplitude response for different Gamma values
omega_ratio = np.linspace(0.1, 2.0, 100)
Gamma_values = [0.1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 20]
amplitudes = amplitude_response(f=2.0, Gamma_values=Gamma_values,
                                omega0=5.0, omega_ratio=omega_ratio)

# Plot amplitude response
plt.figure(figsize=(10, 6))
for Gamma, amp in amplitudes.items():
    plt.plot(omega_ratio, amp, label=f'Γ={Gamma}')

plt.xlabel('ω/ω₀')
plt.ylabel('Amplitude')
plt.title('x₀=0, u₀=0, f=2, ω₀=5')
plt.legend()
plt.grid(True)
plt.show()
```


x₀=0, u₀=0, f=2, ω₀=5
Amplitude
ω/ω₀
Γ=0.1
Γ=1
Γ=2
Γ=3
Γ=4
Γ=5
Γ=6
Γ=7
Γ=8
Γ=9
Γ=10
Γ=12
Γ=15
Γ=20

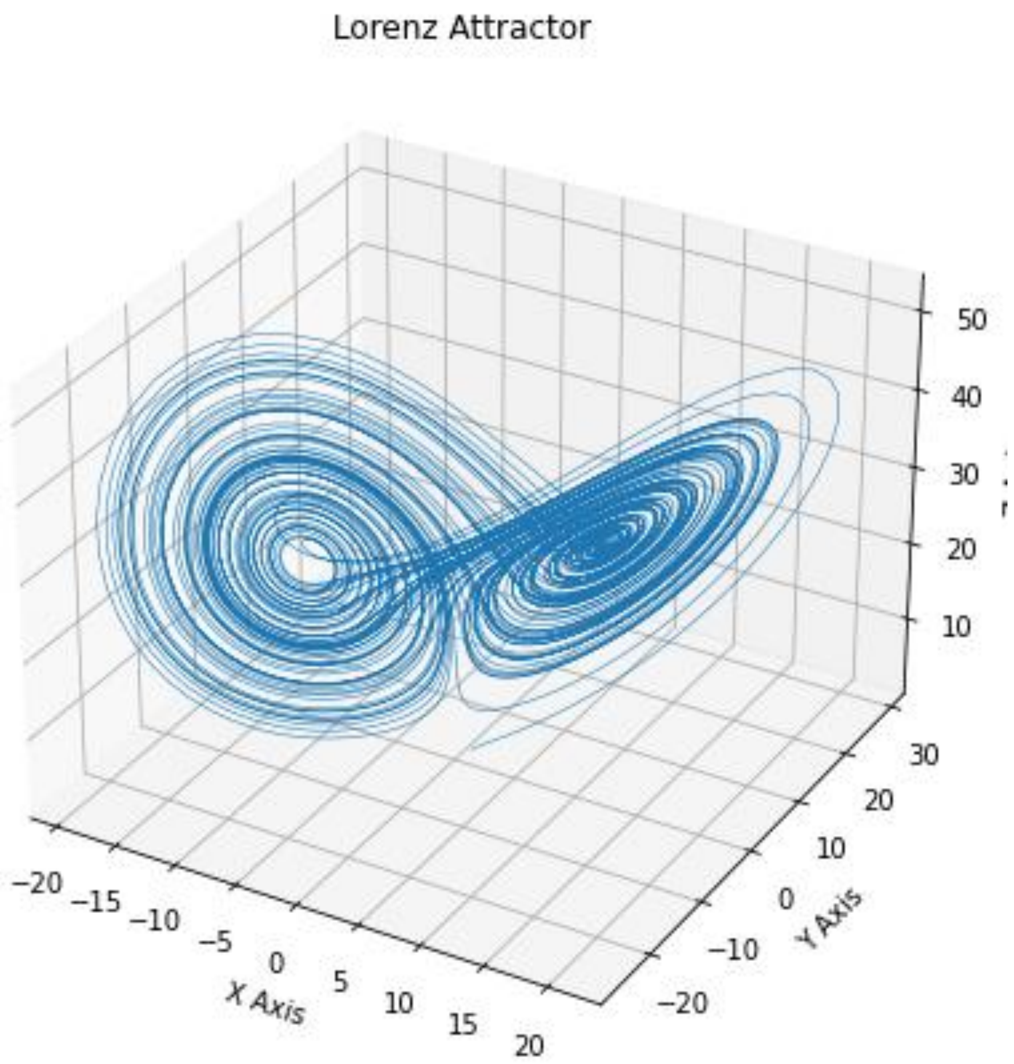

x₀=0.0, u₀=0.0, f=1.0, Γ=0.0, ω=7.0, ω₀=5.0, dt=0.001
analytic
numeric, ε = 0.0001
homogeneous
particular
t

ZAD.2

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

# Lorenz system parameters
sigma = 10.0
r = 28.0
b = 8.0 / 3.0
dt = 0.01  # Time step
num_steps = 10000  # Number of steps

# Initialize arrays
x = np.zeros(num_steps)
y = np.zeros(num_steps)
z = np.zeros(num_steps)

# Initial conditions
x[0], y[0], z[0] = 1.0, 1.0, 1.0

# Euler method for solving the Lorenz system
for i in range(num_steps - 1):
    x[i + 1] = x[i] + dt * sigma * (y[i] - x[i])
    y[i + 1] = y[i] + dt * (x[i] * (r - z[i]) - y[i])
    z[i + 1] = z[i] + dt * (x[i] * y[i] - b * z[i])

# Plot the Lorenz attractor
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')
ax.plot(x, y, z, lw=0.5)
ax.set_xlabel("X Axis")
ax.set_ylabel("Y Axis")
ax.set_zlabel("Z Axis")
ax.set_title("Lorenz Attractor")
plt.show()

# Plot x, y, z as functions of time
time = np.linspace(0, num_steps * dt, num_steps)
fig, axs = plt.subplots(3, 1, figsize=(10, 7))

axs[0].plot(time, x, label='x(t)', color='b')
axs[0].set_xlabel('Time')
axs[0].set_ylabel('X')
axs[0].legend()
axs[0].grid()

axs[1].plot(time, y, label='y(t)', color='r')
axs[1].set_xlabel('Time')
axs[1].set_ylabel('Y')
axs[1].legend()
axs[1].grid()

axs[2].plot(time, z, label='z(t)', color='g')
axs[2].set_xlabel('Time')
axs[2].set_ylabel('Z')
axs[2].legend()
axs[2].grid()

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



x(t)
X
Time
y(t)
Y
Time
z(t)
Z
Time


Lorenz Attractor
50
40
30
20
10
30
20
10
0
−10
−20
Y Axis
−20
−15
−10
−5
0
5
10
15
20
X Axis

ZAD.3

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy.special import jn

# Bessel equation rewritten as a system of first-order ODEs
def bessel_eqn(x, y, n):
    dy1 = y[1]
    dy2 = -(x * y[1] + (x**2 - n**2) * y[0]) / x**2
    return [dy1, dy2]

# Initial conditions for J0 and J1
def initial_conditions(n):
    if n == 0:
        return [1.0, 0.0]  # J0(0) = 1, J0'(0) = 0
    elif n == 1:
        return [0.0, 0.5]  # J1(0) = 0, J1'(0) = 0.5

# Solve for n=0 and n=1
x_span = (0.001, 10)  # Avoid division by zero at x=0
x_eval = np.linspace(0.001, 10, 1000)

fig, axs = plt.subplots(1, 2, figsize=(14, 5))

for i, n in enumerate([0, 1]):
    y0 = initial_conditions(n)
    sol = solve_ivp(bessel_eqn, x_span, y0, args=(n,), t_eval=x_eval, method='RK45')

    # Analytical solution using SciPy's jn function
    j_analytic = jn(n, x_eval)

    # Plot numerical and analytical solutions
    axs[i].plot(x_eval, j_analytic, 'o', label='analytic', markersize=3, markerfacecolor='none', markeredgecolor='blue')
    axs[i].plot(sol.t, sol.y[0], '-', label='numeric', color='orange')
    axs[i].set_xlabel('x')
    axs[i].set_ylabel('y(x)')
    axs[i].set_title(fr'$J_{n}(x)$, dx=0.001')
    axs[i].legend()
    axs[i].grid()

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