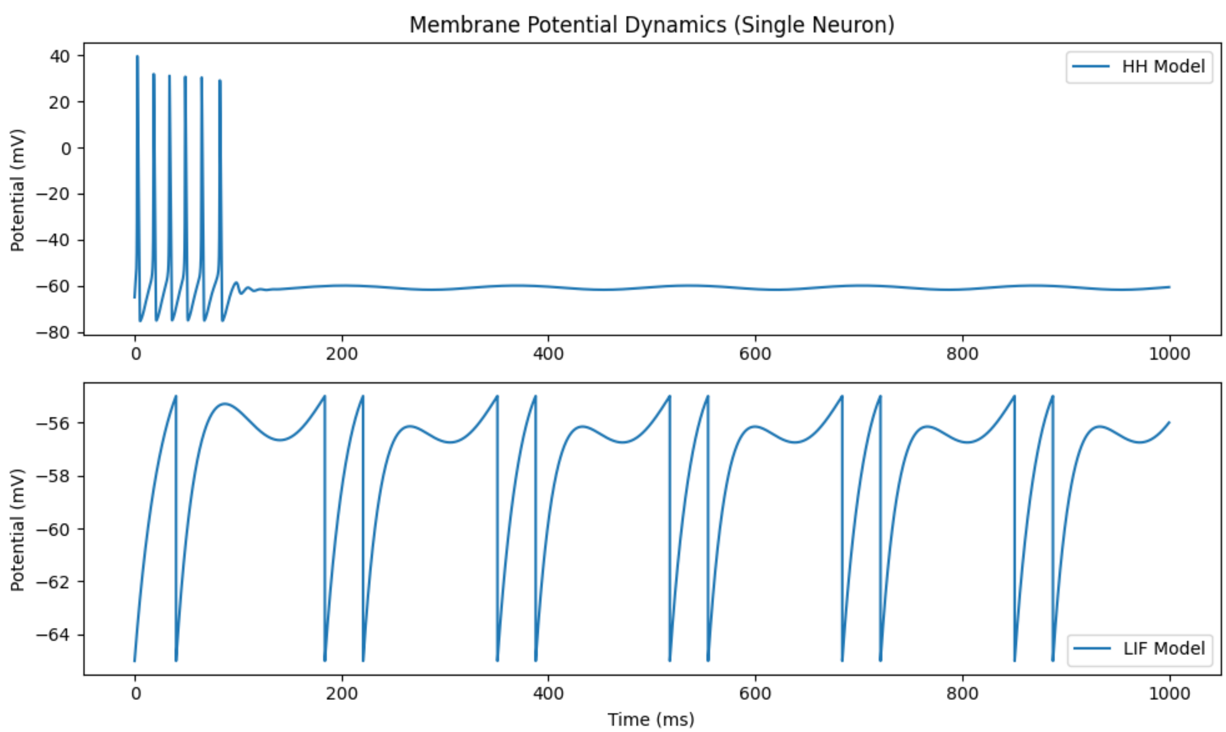
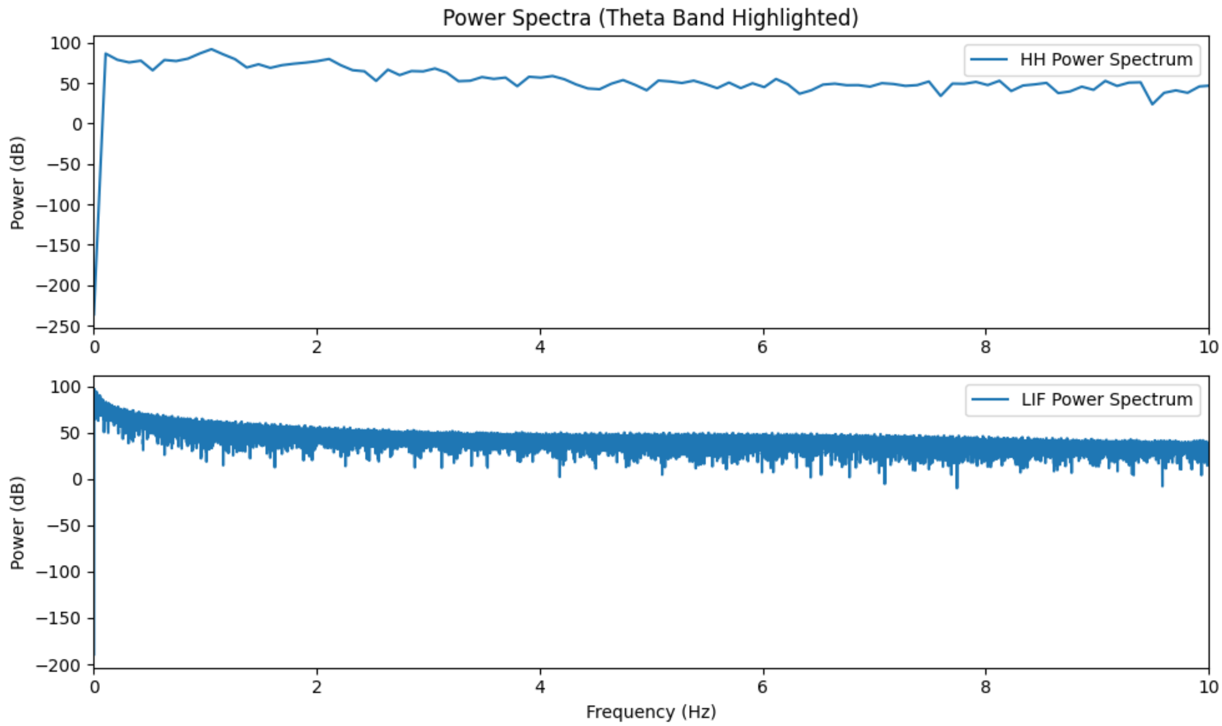
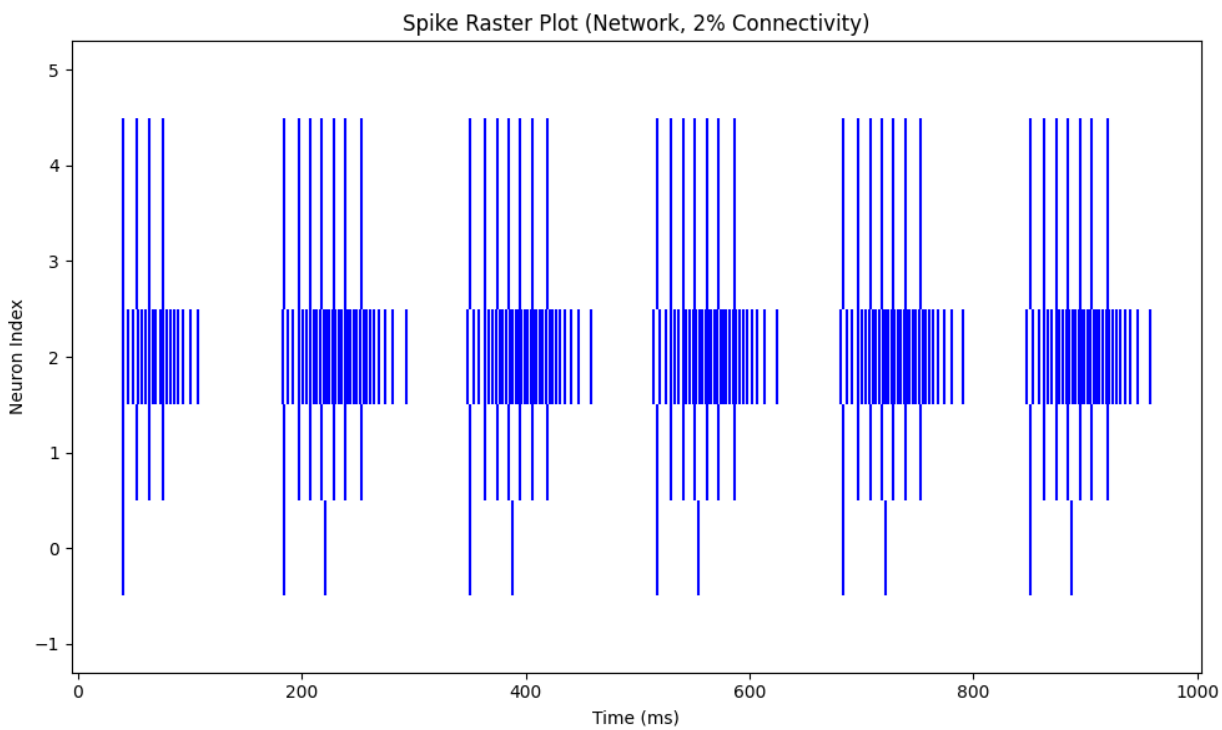
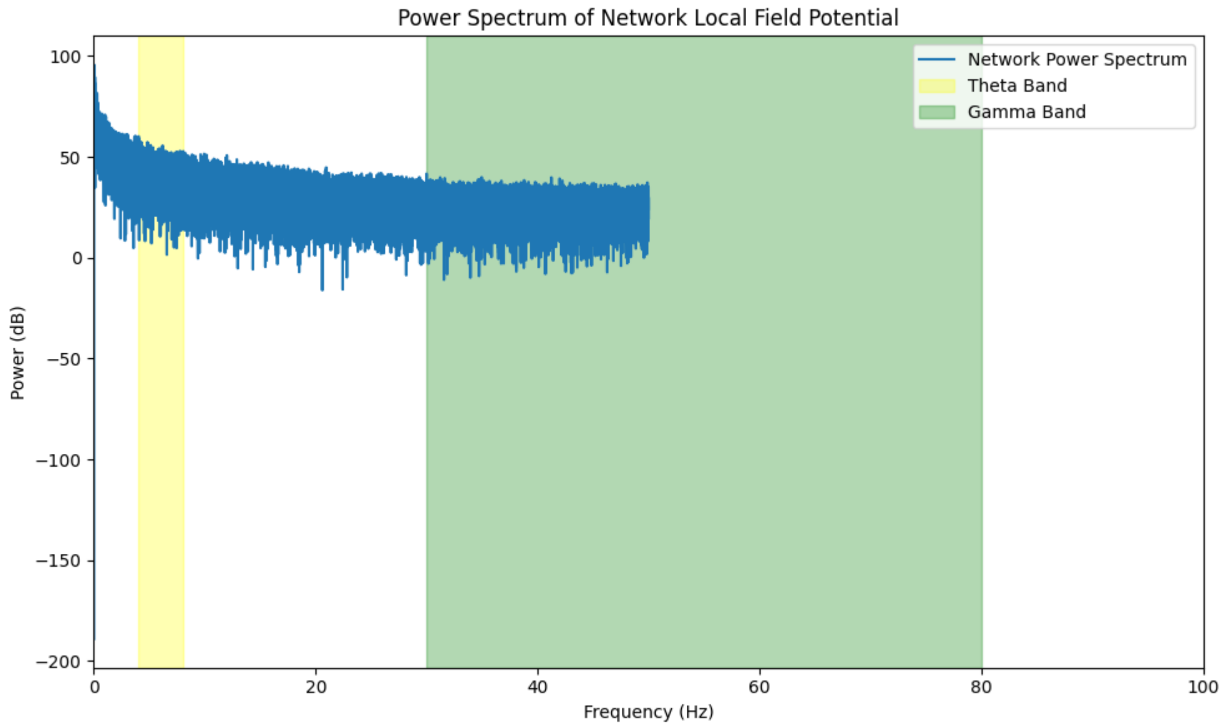
Model Type	Spiking Rate (Hz, mean ± SD)	Computation Time (s, mean ± SD)	Theta Power (dB)	Gamma Power (dB)
HH (Single)	6.0 ± 0.0	0.048 ± 0.003	58.7	31.9
LIF (Single, dt = 0.01)	11.0 ± 0.0	0.115 ± 0.001	50.3	36.1
LIF (Single, Optimized dt=0.02)	11.0 ± 0.0	0.057 ± 0.000	43.8	0.0
LIF (Network, 2%)	12.4 ± 2.0	1.808 ± 0.035	50.7	36.0
LIF Network (Sparsified Network , 1.4%)	13.8 ± 4.2	1.809 ± 0.010	50.8	36.0




Power Spectra (Theta Band Highlighted)
HH Power Spectrum
LIF Power Spectrum
Power (dB)
Frequency (Hz)
100
50
0
−50
−100
−150
−200
−250
0
2
4
6
8
10


Spike Raster Plot (Network, 2% Connectivity)
Neuron Index
Time (ms)
5
4
3
2
1
0
−1
0
200
400
600
800
1000




```
import numpy as np  # Library for numerical arrays and math functions
from scipy.integrate import solve_ivp  # For solving differential equations
import time  # Standard library for timing execution
from scipy.signal import find_peaks  # For detecting spikes in voltage traces
from scipy.fft import fft, fftfreq  # For computing Fast Fourier Transform
import matplotlib.pyplot as plt  # For generating figures

# Define voltage-dependent gating functions for Hodgkin-Huxley model
def alpha_m(V): return 0.1 * (V + 40) / (1 - np.exp(-(V + 40) / 10))  # Activation rate for sodium m
gate
def beta_m(V): return 4 * np.exp(-(V + 65) / 18)  # Deactivation rate for sodium m gate
def alpha_h(V): return 0.07 * np.exp(-(V + 65) / 20)  # Activation rate for sodium h gate
def beta_h(V): return 1 / (1 + np.exp(-(V + 35) / 10))  # Deactivation rate for sodium h gate
def alpha_n(V): return 0.01 * (V + 55) / (1 - np.exp(-(V + 55) / 10))  # Activation rate for
potassium n gate
def beta_n(V): return 0.125 * np.exp(-(V + 65) / 80)  # Deactivation rate for potassium n gate

# ODE function for Hodgkin-Huxley neuron with time-dependent external current
def hh_ode(t, y, I_mean, g_Na=120, g_K=36, g_L=0.3, E_Na=50, E_K=-77, E_L=-54.4, C_m=1):
    V, m, h, n = y  # Unpack state variables: membrane potential and gating variables
    I_ext = I_mean + 2 * np.sin(2 * np.pi * 6 * t / 1000)  # Sinusoidal theta drive at 6 Hz
    dV = (-g_Na * m**3 * h * (V - E_Na) - g_K * n**4 * (V - E_K) - g_L * (V - E_L) + I_ext) / C_m  #
Voltage derivative
```

```
    dm = alpha_m(V) * (1 - m) - beta_m(V) * m  # m gate derivative
    dh = alpha_h(V) * (1 - h) - beta_h(V) * h  # h gate derivative
    dn = alpha_n(V) * (1 - n) - beta_n(V) * n  # n gate derivative
    return [dV, dm, dh, dn]  # Return derivatives

# Simulation function for single Leaky Integrate-and-Fire neuron with time-dependent current
def lif_simulation(t_span, dt, tau=20, E_L=-65, R=10, V_th=-55, V_reset=-65, I_mean=0):
    t = np.arange(t_span[0], t_span[1], dt)  # Generate time array
    V = np.zeros(len(t))  # Initialize voltage array
    V[0] = E_L  # Set initial resting potential
    spikes = []  # List to store spike times
    for i in range(1, len(t)):  # Iterate over time steps
        I_ext = I_mean + 0.2 * np.sin(2 * np.pi * 6 * t[i] / 1000)  # Time-varying input (adjusted
amplitude)
        V[i] = V[i-1] + dt * (-(V[i-1] - E_L) + R * I_ext) / tau  # Update voltage using Euler method
        if V[i] > V_th:  # Check for spike threshold
            V[i] = V_reset  # Reset voltage
            spikes.append(t[i])  # Record spike time
    return t, V, spikes  # Return time, voltage, and spikes

# Optimized network simulation for Leaky Integrate-and-Fire neurons (modified to return spike
times)
def lif_network_simulation(t_span, dt, N=5, p_conn=0.2, tau=20, E_L=-65, R=10, V_th=-55,
V_reset=-65, I_mean=1.0, tau_syn=5, g=0.05, E_syn=0):
    t = np.arange(t_span[0], t_span[1], dt)  # Generate time array
    V = np.zeros((len(t), N))  # Initialize voltage matrix
    V[0] = E_L  # Set initial resting potentials
    s = np.zeros((len(t), N, N))  # Synaptic activation matrix (pre, post)
    conn = (np.random.rand(N, N) < p_conn).astype(float)  # Connectivity matrix
    w = g * conn  # Synaptic weights
    spikes_count = np.zeros(N)  # Spike counts per neuron
    spike_times = [[] for _ in range(N)]  # List of lists to store spike times per neuron
    for i in range(1, len(t)):  # Iterate over time steps
        I_ext = I_mean + 0.2 * np.sin(2 * np.pi * 6 * t[i] / 1000) * np.ones(N)  # External input
        gs = np.sum(w * s[i-1], axis=0)  # Summed synaptic conductances
        I_syn = gs * (E_syn - V[i-1])  # Synaptic currents
        V[i] = V[i-1] + dt * (-(V[i-1] - E_L) + R * I_ext + R * I_syn) / tau  # Update voltages
        spiked = V[i] > V_th  # Detect spikes
        for neuron in range(N):  # Record spike times
            if spiked[neuron]:
                spike_times[neuron].append(t[i])
        V[i][spiked] = V_reset  # Reset spiked neurons
        spikes_count[spiked] += 1  # Count spikes
        s[i] = s[i-1] - dt * s[i-1] / tau_syn  # Decay synaptic activations
        s[i] += np.outer(spiked, np.ones(N)) * conn  # Add pulses for spiked neurons
```

```
    return t, V, spikes_count, spike_times  # Return time, voltages, spike counts, and spike times

# Set simulation parameters
t_span = [0, 1000]  # Time span in ms
y0 = [-65, 0.05, 0.6, 0.32]  # Initial conditions for HH
I_mean_hh = 7  # Mean input for HH
I_mean_lif = 1.0  # Mean input for LIF
num_runs = 10  # Number of runs for statistics

# Run multiple simulations for statistics

# HH (Single)
spike_rates_hh = []
comp_times_hh = []
theta_dbs_hh = []
gamma_dbs_hh = []
for _ in range(num_runs):
    start = time.time()
    sol = solve_ivp(hh_ode, t_span, y0, args=(I_mean_hh,), method='LSODA', rtol=1e-6)
    comp_time = time.time() - start
    t, V = sol.t, sol.y[0]
    peaks, _ = find_peaks(V, height=0)
    rate = len(peaks) / (t_span[1] / 1000)
    N = len(t)
    yf = fft(V - np.mean(V))
    xf = fftfreq(N, t[1] - t[0])
    power = np.abs(yf[:N//2])**2
    mask_theta = (xf[:N//2] >= 4) & (xf[:N//2] <= 8)
    theta_power = np.max(power[mask_theta]) if np.any(mask_theta) else 0
    theta_db = 10 * np.log10(theta_power) if theta_power > 0 else 0
    mask_gamma = (xf[:N//2] >= 30) & (xf[:N//2] <= 80)
    gamma_power = np.max(power[mask_gamma]) if np.any(mask_gamma) else 0
    gamma_db = 10 * np.log10(gamma_power) if gamma_power > 0 else 0
    spike_rates_hh.append(rate)
    comp_times_hh.append(comp_time)
    theta_dbs_hh.append(theta_db)
    gamma_dbs_hh.append(gamma_db)
mean_rate_hh = np.mean(spike_rates_hh)
sd_rate_hh = np.std(spike_rates_hh)
mean_time_hh = np.mean(comp_times_hh)
sd_time_hh = np.std(comp_times_hh)
mean_theta_hh = np.mean(theta_dbs_hh)
mean_gamma_hh = np.mean(gamma_dbs_hh)

# LIF (Single, dt=0.01)
```

```
spike_rates_lif = []
comp_times_lif = []
theta_dbs_lif = []
gamma_dbs_lif = []
dt_lif = 0.01
for _ in range(num_runs):
    start = time.time()
    t, V, spikes = lif_simulation(t_span, dt_lif, I_mean=I_mean_lif)
    comp_time = time.time() - start
    rate = len(spikes) / (t_span[1] / 1000)
    N = len(t)
    yf = fft(V - np.mean(V))
    xf = fftfreq(N, dt_lif)
    power = np.abs(yf[:N//2])**2
    mask_theta = (xf[:N//2] >= 4) & (xf[:N//2] <= 8)
    theta_power = np.max(power[mask_theta]) if np.any(mask_theta) else 0
    theta_db = 10 * np.log10(theta_power) if theta_power > 0 else 0
    mask_gamma = (xf[:N//2] >= 30) & (xf[:N//2] <= 80)
    gamma_power = np.max(power[mask_gamma]) if np.any(mask_gamma) else 0
    gamma_db = 10 * np.log10(gamma_power) if gamma_power > 0 else 0
    spike_rates_lif.append(rate)
    comp_times_lif.append(comp_time)
    theta_dbs_lif.append(theta_db)
    gamma_dbs_lif.append(gamma_db)
mean_rate_lif = np.mean(spike_rates_lif)
sd_rate_lif = np.std(spike_rates_lif)
mean_time_lif = np.mean(comp_times_lif)
sd_time_lif = np.std(comp_times_lif)
mean_theta_lif = np.mean(theta_dbs_lif)
mean_gamma_lif = np.mean(gamma_dbs_lif)

# LIF (Single, Optimized dt=0.02)
spike_rates_lif_opt = []
comp_times_lif_opt = []
theta_dbs_lif_opt = []
gamma_dbs_lif_opt = []
dt_opt = 0.02
for _ in range(num_runs):
    start = time.time()
    t, V, spikes = lif_simulation(t_span, dt_opt, I_mean=I_mean_lif)
    comp_time = time.time() - start
    rate = len(spikes) / (t_span[1] / 1000)
    N = len(t)
    yf = fft(V - np.mean(V))
    xf = fftfreq(N, dt_opt)
```

```
    power = np.abs(yf[:N//2])**2
    mask_theta = (xf[:N//2] >= 4) & (xf[:N//2] <= 8)
    theta_power = np.max(power[mask_theta]) if np.any(mask_theta) else 0
    theta_db = 10 * np.log10(theta_power) if theta_power > 0 else 0
    mask_gamma = (xf[:N//2] >= 30) & (xf[:N//2] <= 80)
    gamma_power = np.max(power[mask_gamma]) if np.any(mask_gamma) else 0
    gamma_db = 10 * np.log10(gamma_power) if gamma_power > 0 else 0
    spike_rates_lif_opt.append(rate)
    comp_times_lif_opt.append(comp_time)
    theta_dbs_lif_opt.append(theta_db)
    gamma_dbs_lif_opt.append(gamma_db)
mean_rate_lif_opt = np.mean(spike_rates_lif_opt)
sd_rate_lif_opt = np.std(spike_rates_lif_opt)
mean_time_lif_opt = np.mean(comp_times_lif_opt)
sd_time_lif_opt = np.std(comp_times_lif_opt)
mean_theta_lif_opt = np.mean(theta_dbs_lif_opt)
mean_gamma_lif_opt = np.mean(gamma_dbs_lif_opt)

# LIF Network (2%)
spike_rates_net20 = []
comp_times_net20 = []
theta_dbs_net20 = []
gamma_dbs_net20 = []
p_conn20 = 0.02 #2%
for _ in range(num_runs):
    start = time.time()
    t, V, spikes_count, spike_times = lif_network_simulation(t_span, dt_lif, p_conn=p_conn20,
I_mean=I_mean_lif)
    comp_time = time.time() - start
    mean_rate = np.mean(spikes_count) / (t_span[1] / 1000)
    LFP = np.mean(V, axis=1)
    N = len(t)
    yf = fft(LFP - np.mean(LFP))
    xf = fftfreq(N, dt_lif)
    power = np.abs(yf[:N//2])**2
    mask_theta = (xf[:N//2] >= 4) & (xf[:N//2] <= 8)
    theta_power = np.max(power[mask_theta]) if np.any(mask_theta) else 0
    theta_db = 10 * np.log10(theta_power) if theta_power > 0 else 0
    mask_gamma = (xf[:N//2] >= 30) & (xf[:N//2] <= 80)
    gamma_power = np.max(power[mask_gamma]) if np.any(mask_gamma) else 0
    gamma_db = 10 * np.log10(gamma_power) if gamma_power > 0 else 0
    spike_rates_net20.append(mean_rate)
    comp_times_net20.append(comp_time)
    theta_dbs_net20.append(theta_db)
    gamma_dbs_net20.append(gamma_db)
```

```
mean_rate_net20 = np.mean(spike_rates_net20)
sd_rate_net20 = np.std(spike_rates_net20)
mean_time_net20 = np.mean(comp_times_net20)
sd_time_net20 = np.std(comp_times_net20)
mean_theta_net20 = np.mean(theta_dbs_net20)
mean_gamma_net20 = np.mean(gamma_dbs_net20)

# LIF Network (Sparsified, 1.4%)
spike_rates_net14 = []
comp_times_net14 = []
theta_dbs_net14 = []
gamma_dbs_net14 = []
p_conn14 = 0.014
for _ in range(num_runs):
    start = time.time()
    t, V, spikes_count, spike_times = lif_network_simulation(t_span, dt_lif, p_conn=p_conn14,
I_mean=I_mean_lif)
    comp_time = time.time() - start
    mean_rate = np.mean(spikes_count) / (t_span[1] / 1000)
    LFP = np.mean(V, axis=1)
    N = len(t)
    yf = fft(LFP - np.mean(LFP))
    xf = fftfreq(N, dt_lif)
    power = np.abs(yf[:N//2])**2
    mask_theta = (xf[:N//2] >= 4) & (xf[:N//2] <= 8)
    theta_power = np.max(power[mask_theta]) if np.any(mask_theta) else 0
    theta_db = 10 * np.log10(theta_power) if theta_power > 0 else 0
    mask_gamma = (xf[:N//2] >= 30) & (xf[:N//2] <= 80)
    gamma_power = np.max(power[mask_gamma]) if np.any(mask_gamma) else 0
    gamma_db = 10 * np.log10(gamma_power) if gamma_power > 0 else 0
    spike_rates_net14.append(mean_rate)
    comp_times_net14.append(comp_time)
    theta_dbs_net14.append(theta_db)
    gamma_dbs_net14.append(gamma_db)
mean_rate_net14 = np.mean(spike_rates_net14)
sd_rate_net14 = np.std(spike_rates_net14)
mean_time_net14 = np.mean(comp_times_net14)
sd_time_net14 = np.std(comp_times_net14)
mean_theta_net14 = np.mean(theta_dbs_net14)
mean_gamma_net14 = np.mean(gamma_dbs_net14)

# Generate figures using single runs
sol_hh = solve_ivp(hh_ode, t_span, y0, args=(I_mean_hh,), method='LSODA', rtol=1e-6)
t_hh, V_hh = sol_hh.t, sol_hh.y[0]
N_hh = len(t_hh)
```

```
yf_hh = fft(V_hh - np.mean(V_hh))
xf_hh = fftfreq(N_hh, t_hh[1] - t_hh[0])
power_hh = np.abs(yf_hh[:N_hh//2])**2

t_lif, V_lif, spikes_lif = lif_simulation(t_span, dt_lif, I_mean=I_mean_lif)
N_lif = len(t_lif)
yf_lif = fft(V_lif - np.mean(V_lif))
xf_lif = fftfreq(N_lif, dt_lif)
power_lif = np.abs(yf_lif[:N_lif//2])**2

t_net, V_net, spikes_net, spike_times_net = lif_network_simulation(t_span, dt_lif, p_conn=0.2,
I_mean=I_mean_lif)
LFP_net = np.mean(V_net, axis=1)
N_net = len(t_net)
yf_net = fft(LFP_net - np.mean(LFP_net))
xf_net = fftfreq(N_net, dt_lif)
power_net = np.abs(yf_net[:N_net//2])**2

# Generate Figure 1: Voltage traces
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(t_hh, V_hh, label='HH Model')
plt.title('Membrane Potential Dynamics (Single Neuron)')
plt.ylabel('Potential (mV)')
plt.legend()
plt.subplot(2, 1, 2)
plt.plot(t_lif, V_lif, label='LIF Model')
plt.xlabel('Time (ms)')
plt.ylabel('Potential (mV)')
plt.legend()
plt.tight_layout()
plt.savefig('figure1_voltage_traces.png')  # Save for inclusion

# Generate Figure 2: Power spectra
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(xf_hh[:N_hh//2], 10 * np.log10(power_hh), label='HH Power Spectrum')
plt.title('Power Spectra (Theta Band Highlighted)')
plt.xlim(0, 10)
plt.ylabel('Power (dB)')
plt.legend()
plt.subplot(2, 1, 2)
plt.plot(xf_lif[:N_lif//2], 10 * np.log10(power_lif), label='LIF Power Spectrum')
plt.xlabel('Frequency (Hz)')
plt.xlim(0, 10)
```

```
plt.ylabel('Power (dB)')
plt.legend()
plt.tight_layout()
plt.savefig('figure2_power_spectra.png')  # Save for inclusion

# Generate Figure 3: Spike Raster Plot for network (2% connectivity)
plt.figure(figsize=(10, 6))
plt.eventplot(spike_times_net, orientation='horizontal', colors='b')
plt.title('Spike Raster Plot (Network, 2% Connectivity)')
plt.xlabel('Time (ms)')
plt.ylabel('Neuron Index')
plt.tight_layout()
plt.savefig('figure3_raster.png')  # Save for inclusion

# Generate Figure 4: Network Power Spectrum
plt.figure(figsize=(10, 6))
plt.plot(xf_net[:N_net//2], 10 * np.log10(power_net), label='Network Power Spectrum')
plt.axvspan(4, 8, color='yellow', alpha=0.3, label='Theta Band')
plt.axvspan(30, 80, color='green', alpha=0.3, label='Gamma Band')
plt.title('Power Spectrum of Network Local Field Potential')
plt.xlim(0, 100)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Power (dB)')
plt.legend()
plt.tight_layout()
plt.savefig('figure4_network_power.png')  # Save for inclusion

# Output results table
print("Model Type\tSpiking Rate (Hz, mean ± SD)\tComputation Time (s, mean ± SD)\tTheta
Power (dB)\tGamma Power (dB)")
print(f"HH (Single)\t{mean_rate_hh:.1f} ± {sd_rate_hh:.1f}\t{mean_time_hh:.3f} ±
{sd_time_hh:.3f}\t{mean_theta_hh:.1f}\t{mean_gamma_hh:.1f}")
print(f"LIF (Single, dt = 0.01)\t{mean_rate_lif:.1f} ± {sd_rate_lif:.1f}\t{mean_time_lif:.3f} ±
{sd_time_lif:.3f}\t{mean_theta_lif:.1f}\t{mean_gamma_lif:.1f}")
print(f"LIF (Single, Optimized dt=0.02)\t{mean_rate_lif_opt:.1f} ± {sd_rate_lif_opt:.1f}
\t{mean_time_lif_opt:.3f} ± {sd_time_lif_opt:.3f}\t{mean_theta_lif_opt:.1f}
\t{mean_gamma_lif_opt:.1f}")
print(f"LIF Network (2%)\t{mean_rate_net20:.1f} ± {sd_rate_net20:.1f}\t{mean_time_net20:.3f} ±
{sd_time_net20:.3f}\t{mean_theta_net20:.1f}\t{mean_gamma_net20:.1f}")
print(f"LIF Network (Sparsified, 1.4%)\t{mean_rate_net14:.1f} ± {sd_rate_net14:.1f}
\t{mean_time_net14:.3f} ± {sd_time_net14:.3f}\t{mean_theta_net14:.1f}
\t{mean_gamma_net14:.1f}")
```