# Programming Examples and Simulation Code

## Neural Modeling and Biophysics

### Hodgkin-Huxley Model Simulation

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Define constants

C\_m = 1.0 # Membrane capacitance (uF/cm^2)

g\_Na = 120.0 # Sodium conductance (mS/cm^2)

g\_K = 36.0 # Potassium conductance (mS/cm^2)

g\_L = 0.3 # Leak conductance (mS/cm^2)

E\_Na = 50.0 # Sodium reversal potential (mV)

E\_K = -77.0 # Potassium reversal potential (mV)

E\_L = -54.387 # Leak reversal potential (mV)

# Define time parameters

dt = 0.01 # Time step (ms)

t\_max = 50.0 # Total simulation time (ms)

t = np.arange(0, t\_max, dt) # Time array

# Initialize variables

V = -65.0 # Membrane potential (mV)

m = 0.05 # Sodium activation gate

h = 0.6 # Sodium inactivation gate

n = 0.32 # Potassium activation gate

# Store results

V\_hist = np.zeros\_like(t)

I\_inj = np.zeros\_like(t)

I\_inj[int(10/dt):int(20/dt)] = 10.0 # Current injection (nA)

# Hodgkin-Huxley equations

def alpha\_m(V): return 0.1 \* (V + 40.0) / (1.0 - np.exp(-(V + 40.0) / 10.0))

def beta\_m(V): return 4.0 \* np.exp(-(V + 65.0) / 18.0)

def alpha\_h(V): return 0.07 \* np.exp(-(V + 65.0) / 20.0)

def beta\_h(V): return 1.0 / (1.0 + np.exp(-(V + 35.0) / 10.0))

def alpha\_n(V): return 0.01 \* (V + 55.0) / (1.0 - np.exp(-(V + 55.0) / 10.0))

def beta\_n(V): return 0.125 \* np.exp(-(V + 65.0) / 80.0)

# Simulation loop

for i in range(len(t)):

# Update gating variables

m += dt \* (alpha\_m(V) \* (1 - m) - beta\_m(V) \* m)

h += dt \* (alpha\_h(V) \* (1 - h) - beta\_h(V) \* h)

n += dt \* (alpha\_n(V) \* (1 - n) - beta\_n(V) \* n)

# Compute currents

I\_Na = g\_Na \* m\*\*3 \* h \* (V - E\_Na)

I\_K = g\_K \* n\*\*4 \* (V - E\_K)

I\_L = g\_L \* (V - E\_L)

# Update membrane potential

dV = (I\_inj[i] - I\_Na - I\_K - I\_L) / C\_m

V += dt \* dV

# Store results

V\_hist[i] = V

# Plot results

plt.plot(t, V\_hist, label='Membrane Potential (mV)')

plt.xlabel('Time (ms)')

plt.ylabel('V (mV)')

plt.title('Hodgkin-Huxley Model Simulation')

plt.legend()

plt.show()

\end{lstlisting}

### FitzHugh-Nagumo Model Implementation

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import solve\_ivp

# FitzHugh-Nagumo model parameters

a = 0.7

b = 0.8

tau = 12.5

I\_ext = 0.5 # External current input

# Define the FitzHugh-Nagumo equations

def fitzhugh\_nagumo(t, y):

v, w = y # v: membrane potential, w: recovery variable

dvdt = v - v\*\*3 / 3 - w + I\_ext # Voltage dynamics

dwdt = (v + a - b \* w) / tau # Recovery variable dynamics

return [dvdt, dwdt]

# Initial conditions

y0 = [0.0, 0.0] # Initial v and w values

# Time span for integration

t\_span = (0, 200) # Simulation time from 0 to 200 ms

t\_eval = np.linspace(t\_span[0], t\_span[1], 1000) # Time points for evaluation

# Solve the differential equations

sol = solve\_ivp(fitzhugh\_nagumo, t\_span, y0, t\_eval=t\_eval)

# Plot the results

plt.figure(figsize=(10, 6))

plt.plot(sol.t, sol.y[0], label='Membrane Potential (v)')

plt.plot(sol.t, sol.y[1], label='Recovery Variable (w)')

plt.xlabel('Time (ms)')

plt.ylabel('State Variables')

plt.title('FitzHugh-Nagumo Model Dynamics')

plt.legend()

plt.grid(True)

plt.show()

\end{lstlisting}

### Leaky Integrate-and-Fire Neuron Model

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Parameters

tau\_m = 20.0 # Membrane time constant (ms)

R\_m = 10.0 # Membrane resistance (MΩ)

V\_rest = -65.0 # Resting membrane potential (mV)

V\_thresh = -50.0 # Threshold potential (mV)

V\_reset = -70.0 # Reset potential after spike (mV)

I\_ext = 1.5 # External current input (nA)

dt = 0.1 # Time step (ms)

T = 200 # Total simulation time (ms)

# Initialize

time = np.arange(0, T, dt)

V = np.full\_like(time, V\_rest) # Membrane potential array

spikes = np.zeros\_like(time) # Spike train array

# Simulation loop

for i in range(1, len(time)):

dV = (-(V[i-1] - V\_rest) + R\_m \* I\_ext) \* (dt / tau\_m)

V[i] = V[i-1] + dV

# Check for spike

if V[i] >= V\_thresh:

V[i] = V\_reset # Reset membrane potential

spikes[i] = 1 # Record spike

# Plot results

plt.figure(figsize=(10, 6))

plt.plot(time, V, label="Membrane Potential (mV)")

plt.scatter(time[spikes == 1], np.full(np.sum(spikes), V\_thresh), color="red", label="Spikes")

plt.axhline(V\_thresh, color="gray", linestyle="--", label="Threshold")

plt.xlabel("Time (ms)")

plt.ylabel("Membrane Potential (mV)")

plt.title("Leaky Integrate-and-Fire Neuron Model")

plt.legend()

plt.grid(True)

plt.show()

\end{lstlisting}

### Izhikevich Neuron Model

[Izhikevich Neuron Model] \begin{lstlisting}

# Izhikevich Neuron Model Simulation

import numpy as np

import matplotlib.pyplot as plt

# Parameters for the Izhikevich model

a = 0.02 # Time scale of recovery variable u

b = 0.2 # Sensitivity of u to subthreshold fluctuations

c = -65 # After-spike reset value of v

d = 8 # After-spike reset value of u

# Simulation parameters

T = 1000 # Total simulation time (ms)

dt = 0.5 # Time step (ms)

time = np.arange(0, T, dt) # Time array

# Initial conditions

v = -65 # Membrane potential (mV)

u = b \* v # Recovery variable

v\_values = [] # To store membrane potential over time

# Input current (I) - can be modified for different scenarios

I = 10 # Constant input current (pA)

# Simulation loop

for t in time:

# Izhikevich model equations

dv = (0.04 \* v\*\*2 + 5 \* v + 140 - u + I) \* dt

du = (a \* (b \* v - u)) \* dt

# Update membrane potential and recovery variable

v += dv

u += du

# Spike condition

if v >= 30:

v\_values.append(30) # Spike peak

v = c # Reset membrane potential

u += d # Reset recovery variable

else:

v\_values.append(v)

# Plotting the results

plt.plot(time, v\_values, label="Membrane Potential (mV)")

plt.xlabel("Time (ms)")

plt.ylabel("Membrane Potential (mV)")

plt.title("Izhikevich Neuron Model Simulation")

plt.legend()

plt.show()

\end{lstlisting}

### Adaptive Exponential Integrate-and-Fire Model

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Parameters

C = 281 # Membrane capacitance (pF)

gL = 30 # Leak conductance (nS)

EL = -70.6 # Leak reversal potential (mV)

VT = -50.4 # Spike threshold (mV)

DeltaT = 2 # Slope factor (mV)

tauw = 144 # Adaptation time constant (ms)

a = 4 # Subthreshold adaptation (nS)

b = 80.5 # Spike-triggered adaptation (pA)

I = 400 # Input current (pA)

dt = 0.1 # Time step (ms)

T = 500 # Total simulation time (ms)

V0 = EL # Initial membrane potential (mV)

w0 = 0 # Initial adaptation variable (pA)

# Simulation

time = np.arange(0, T, dt) # Time array

V = np.zeros\_like(time) # Membrane potential array

w = np.zeros\_like(time) # Adaptation variable array

V[0] = V0 # Set initial membrane potential

w[0] = w0 # Set initial adaptation variable

for t in range(1, len(time)):

# Adaptive Exponential Integrate-and-Fire dynamics

dV = (gL \* (EL - V[t-1]) + gL \* DeltaT \* np.exp((V[t-1] - VT) / DeltaT) - w[t-1] + I) / C

dw = (a \* (V[t-1] - EL) - w[t-1]) / tauw

V[t] = V[t-1] + dV \* dt

w[t] = w[t-1] + dw \* dt

# Spike reset

if V[t] >= 0: # Spike condition

V[t] = EL # Reset membrane potential

w[t] += b # Update adaptation variable

# Plot results

plt.figure(figsize=(10, 6))

plt.plot(time, V, label='Membrane Potential (mV)')

plt.xlabel('Time (ms)')

plt.ylabel('Membrane Potential (mV)')

plt.title('Adaptive Exponential Integrate-and-Fire Model')

plt.legend()

plt.grid(True)

plt.show()

\end{lstlisting}

### Modeling Cable Theory for Dendritic Computation

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Define parameters for the cable model

length = 1000 # Length of the dendrite in micrometers (µm)

diameter = 2 # Diameter of the dendrite in micrometers (µm)

Ra = 100 # Axial resistance in ohm·cm

Rm = 20000 # Membrane resistance in ohm·cm²

Cm = 1 # Membrane capacitance in µF/cm²

dt = 0.1 # Time step in milliseconds (ms)

dx = 10 # Spatial step in micrometers (µm)

t\_max = 100 # Total simulation time in milliseconds (ms)

# Calculate derived parameters

lambda\_ = np.sqrt((Rm \* diameter) / (4 \* Ra)) # Space constant (µm)

tau = Rm \* Cm # Time constant (ms)

# Initialize voltage array

x = np.arange(0, length, dx) # Spatial grid

V = np.zeros\_like(x) # Voltage along the dendrite

V[0] = 10 # Initial voltage at the start of the dendrite (mV)

# Simulation loop

for t in np.arange(0, t\_max, dt):

# Update voltage using the cable equation

dV = (V[:-2] - 2 \* V[1:-1] + V[2:]) / (dx\*\*2) # Second spatial derivative

V[1:-1] += (dt / tau) \* (lambda\_\*\*2 \* dV - V[1:-1]) # Update voltage

# Plot the voltage distribution at specific time points

if t % 20 == 0:

plt.plot(x, V, label=f't = {t} ms')

# Finalize plot

plt.xlabel('Position along dendrite (µm)')

plt.ylabel('Voltage (mV)')

plt.title('Cable Theory Simulation for Dendritic Computation')

plt.legend()

plt.show()

\end{lstlisting}

### Voltage-Gated Ion Channel Kinetics Simulation

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Define parameters

V\_rest = -65 # Resting membrane potential (mV)

V\_thresh = -50 # Threshold potential (mV)

V\_max = 30 # Maximum membrane potential (mV)

dt = 0.01 # Time step (ms)

t\_max = 100 # Total simulation time (ms)

C\_m = 1 # Membrane capacitance (uF/cm^2)

g\_Na = 120 # Sodium conductance (mS/cm^2)

g\_K = 36 # Potassium conductance (mS/cm^2)

g\_L = 0.3 # Leak conductance (mS/cm^2)

E\_Na = 50 # Sodium reversal potential (mV)

E\_K = -77 # Potassium reversal potential (mV)

E\_L = -54.387 # Leak reversal potential (mV)

# Define voltage-gated ion channel kinetics

def alpha\_m(V): return 0.1 \* (V + 40) / (1 - np.exp(-(V + 40) / 10))

def beta\_m(V): return 4 \* np.exp(-(V + 65) / 18)

def alpha\_h(V): return 0.07 \* np.exp(-(V + 65) / 20)

def beta\_h(V): return 1 / (1 + np.exp(-(V + 35) / 10))

def alpha\_n(V): return 0.01 \* (V + 55) / (1 - np.exp(-(V + 55) / 10))

def beta\_n(V): return 0.125 \* np.exp(-(V + 65) / 80)

# Initialize variables

t = np.arange(0, t\_max, dt)

V = np.full\_like(t, V\_rest)

m = np.zeros\_like(t)

h = np.zeros\_like(t)

n = np.zeros\_like(t)

# Simulation loop

for i in range(1, len(t)):

# Update gating variables

m[i] = m[i-1] + dt \* (alpha\_m(V[i-1]) \* (1 - m[i-1]) - beta\_m(V[i-1]) \* m[i-1])

h[i] = h[i-1] + dt \* (alpha\_h(V[i-1]) \* (1 - h[i-1]) - beta\_h(V[i-1]) \* h[i-1])

n[i] = n[i-1] + dt \* (alpha\_n(V[i-1]) \* (1 - n[i-1]) - beta\_n(V[i-1]) \* n[i-1])

# Calculate ionic currents

I\_Na = g\_Na \* m[i]\*\*3 \* h[i] \* (V[i-1] - E\_Na)

I\_K = g\_K \* n[i]\*\*4 \* (V[i-1] - E\_K)

I\_L = g\_L \* (V[i-1] - E\_L)

# Update membrane potential

dV = (-I\_Na - I\_K - I\_L) / C\_m \* dt

V[i] = V[i-1] + dV

# Apply threshold for action potential

if V[i] >= V\_thresh:

V[i] = V\_max

# Plot results

plt.plot(t, V, label='Membrane Potential (mV)')

plt.xlabel('Time (ms)')

plt.ylabel('Voltage (mV)')

plt.title('Voltage-Gated Ion Channel Kinetics Simulation')

plt.legend()

plt.show()

\end{lstlisting}

### Calcium Signaling and Buffering in Neurons

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Parameters

dt = 0.01 # Time step (ms)

T = 1000 # Total simulation time (ms)

t = np.arange(0, T, dt) # Time array

Ca\_rest = 0.05 # Resting calcium concentration (µM)

tau\_Ca = 50 # Calcium decay time constant (ms)

beta = 0.1 # Calcium buffering factor

I\_Ca = 0.2 # Calcium current (µM/ms)

# Initialize calcium concentration

Ca = np.zeros\_like(t)

Ca[0] = Ca\_rest

# Simulate calcium dynamics

for i in range(1, len(t)):

dCa\_dt = (I\_Ca - (Ca[i-1] - Ca\_rest) / tau\_Ca) \* beta

Ca[i] = Ca[i-1] + dCa\_dt \* dt

# Plot results

plt.plot(t, Ca, label='Calcium Concentration')

plt.xlabel('Time (ms)')

plt.ylabel('[Ca²⁺] (µM)')

plt.title('Calcium Signaling and Buffering in Neurons')

plt.legend()

plt.grid(True)

plt.show()

\end{lstlisting}

### Simulating Synaptic Conductance-Based Models

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Define parameters

dt = 0.1 # Time step (ms)

T = 1000 # Total simulation time (ms)

t = np.arange(0, T, dt) # Time array

V\_rest = -65 # Resting membrane potential (mV)

V\_m = np.ones\_like(t) \* V\_rest # Membrane potential array

E\_syn = 0 # Synaptic reversal potential (mV)

g\_syn\_max = 0.5 # Maximum synaptic conductance (nS)

tau\_syn = 10 # Synaptic time constant (ms)

spike\_times = [200, 400, 600] # Presynaptic spike times (ms)

# Synaptic conductance function

def g\_syn(t, spike\_times, tau\_syn):

g = np.zeros\_like(t)

for spike in spike\_times:

g += np.exp(-(t - spike) / tau\_syn) \* (t >= spike)

return g\_syn\_max \* g

# Simulate membrane potential

g\_syn\_t = g\_syn(t, spike\_times, tau\_syn)

for i in range(1, len(t)):

I\_syn = g\_syn\_t[i] \* (E\_syn - V\_m[i-1]) # Synaptic current

V\_m[i] = V\_m[i-1] + (I\_syn) \* dt # Update membrane potential

# Plot results

plt.figure(figsize=(10, 6))

plt.plot(t, V\_m, label='Membrane Potential (mV)')

plt.plot(t, g\_syn\_t, label='Synaptic Conductance (nS)', linestyle='--')

plt.xlabel('Time (ms)')

plt.ylabel('Membrane Potential / Conductance')

plt.title('Synaptic Conductance-Based Model Simulation')

plt.legend()

plt.grid(True)

plt.show()

\end{lstlisting}

### Stochastic Hodgkin-Huxley Model for Variability

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Define constants

C\_m = 1.0 # Membrane capacitance (uF/cm^2)

g\_Na = 120.0 # Sodium conductance (mS/cm^2)

g\_K = 36.0 # Potassium conductance (mS/cm^2)

g\_L = 0.3 # Leak conductance (mS/cm^2)

E\_Na = 50.0 # Sodium reversal potential (mV)

E\_K = -77.0 # Potassium reversal potential (mV)

E\_L = -54.387 # Leak reversal potential (mV)

dt = 0.01 # Time step (ms)

t\_max = 50.0 # Total simulation time (ms)

I\_ext = 10.0 # External current (uA/cm^2)

noise\_amplitude = 0.5 # Noise amplitude for stochasticity

# Initialize variables

V = -65.0 # Membrane potential (mV)

m = 0.05 # Sodium activation gate

h = 0.6 # Sodium inactivation gate

n = 0.32 # Potassium activation gate

# Define rate functions

def alpha\_m(V): return 0.1 \* (V + 40.0) / (1.0 - np.exp(-(V + 40.0) / 10.0))

def beta\_m(V): return 4.0 \* np.exp(-(V + 65.0) / 18.0)

def alpha\_h(V): return 0.07 \* np.exp(-(V + 65.0) / 20.0)

def beta\_h(V): return 1.0 / (1.0 + np.exp(-(V + 35.0) / 10.0))

def alpha\_n(V): return 0.01 \* (V + 55.0) / (1.0 - np.exp(-(V + 55.0) / 10.0))

def beta\_n(V): return 0.125 \* np.exp(-(V + 65.0) / 80.0)

# Simulation loop

t = np.arange(0, t\_max, dt)

V\_trace = np.zeros\_like(t)

for i in range(len(t)):

# Update gating variables

m += dt \* (alpha\_m(V) \* (1 - m) - beta\_m(V) \* m)

h += dt \* (alpha\_h(V) \* (1 - h) - beta\_h(V) \* h)

n += dt \* (alpha\_n(V) \* (1 - n) - beta\_n(V) \* n)

# Compute currents

I\_Na = g\_Na \* m\*\*3 \* h \* (V - E\_Na)

I\_K = g\_K \* n\*\*4 \* (V - E\_K)

I\_L = g\_L \* (V - E\_L)

# Update membrane potential with stochastic noise

noise = noise\_amplitude \* np.random.normal()

dV = (I\_ext - I\_Na - I\_K - I\_L + noise) / C\_m \* dt

V += dV

# Store membrane potential

V\_trace[i] = V

# Plot results

plt.plot(t, V\_trace)

plt.xlabel('Time (ms)')

plt.ylabel('Membrane Potential (mV)')

plt.title('Stochastic Hodgkin-Huxley Model')

plt.show()

\end{lstlisting}

## Neural Dynamics and Synchronization

### Coupled Neural Oscillators (Kuramoto Model)

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Parameters

N = 100 # Number of oscillators

K = 1.0 # Coupling strength

omega = np.random.normal(0, 1, N) # Natural frequencies

theta = np.random.uniform(0, 2\*np.pi, N) # Initial phases

dt = 0.01 # Time step

T = 100 # Total time

steps = int(T / dt) # Number of steps

# Kuramoto model simulation

def kuramoto(theta, omega, K, N):

dtheta = np.zeros(N)

for i in range(N):

dtheta[i] = omega[i] + (K/N) \* np.sum(np.sin(theta - theta[i]))

return dtheta

# Time evolution

theta\_history = np.zeros((steps, N))

for t in range(steps):

theta += kuramoto(theta, omega, K, N) \* dt

theta\_history[t] = theta

# Plot results

plt.figure(figsize=(10, 6))

plt.plot(np.linspace(0, T, steps), np.sin(theta\_history))

plt.title("Coupled Neural Oscillators (Kuramoto Model)")

plt.xlabel("Time")

plt.ylabel("Phase (sin(theta))")

plt.show()

\end{lstlisting}

### Phase Space Analysis of Neural Excitability

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import solve\_ivp

# Define the FitzHugh-Nagumo model for neural excitability

def fitzhugh\_nagumo(t, y, a=0.7, b=0.8, tau=12.5, I\_ext=0.5):

v, w = y

dvdt = v - v\*\*3 / 3 - w + I\_ext

dwdt = (v + a - b \* w) / tau

return [dvdt, dwdt]

# Initial conditions and time span

y0 = [0.0, 0.0]

t\_span = (0, 100)

t\_eval = np.linspace(0, 100, 1000)

# Solve the differential equations

sol = solve\_ivp(fitzhugh\_nagumo, t\_span, y0, t\_eval=t\_eval)

# Extract the solution

v, w = sol.y

# Plot the phase space trajectory

plt.figure(figsize=(8, 6))

plt.plot(v, w, label='Phase Space Trajectory')

plt.xlabel('Membrane Potential (v)')

plt.ylabel('Recovery Variable (w)')

plt.title('Phase Space Analysis of Neural Excitability')

plt.legend()

plt.grid(True)

plt.show()

\end{lstlisting}

### Bifurcation Analysis in Neuronal Dynamics

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import solve\_ivp

# Define the neuronal dynamics model (Hodgkin-Huxley-like)

def neuronal\_model(t, state, I\_ext, g\_Na, g\_K, g\_L, E\_Na, E\_K, E\_L, C\_m):

V, m, h, n = state

I\_Na = g\_Na \* m\*\*3 \* h \* (V - E\_Na)

I\_K = g\_K \* n\*\*4 \* (V - E\_K)

I\_L = g\_L \* (V - E\_L)

dVdt = (I\_ext - I\_Na - I\_K - I\_L) / C\_m

dmdt = (alpha\_m(V) \* (1 - m) - beta\_m(V) \* m)

dhdt = (alpha\_h(V) \* (1 - h) - beta\_h(V) \* h)

dndt = (alpha\_n(V) \* (1 - n) - beta\_n(V) \* n)

return [dVdt, dmdt, dhdt, dndt]

# Define rate constants for gating variables

def alpha\_m(V): return 0.1 \* (V + 40) / (1 - np.exp(-(V + 40) / 10))

def beta\_m(V): return 4.0 \* np.exp(-(V + 65) / 18)

def alpha\_h(V): return 0.07 \* np.exp(-(V + 65) / 20)

def beta\_h(V): return 1 / (1 + np.exp(-(V + 35) / 10))

def alpha\_n(V): return 0.01 \* (V + 55) / (1 - np.exp(-(V + 55) / 10))

def beta\_n(V): return 0.125 \* np.exp(-(V + 65) / 80)

# Parameters

I\_ext\_values = np.linspace(0, 50, 100) # External current range for bifurcation

g\_Na, g\_K, g\_L = 120, 36, 0.3 # Conductances

E\_Na, E\_K, E\_L = 50, -77, -54.387 # Reversal potentials

C\_m = 1.0 # Membrane capacitance

initial\_state = [-65, 0.05, 0.6, 0.32] # Initial state [V, m, h, n]

# Perform bifurcation analysis

bifurcation\_data = []

for I\_ext in I\_ext\_values:

sol = solve\_ivp(neuronal\_model, [0, 100], initial\_state, args=(I\_ext, g\_Na, g\_K, g\_L, E\_Na, E\_K, E\_L, C\_m),

t\_eval=np.linspace(50, 100, 500)) # Simulate to steady state

V\_steady = sol.y[0, -100:] # Extract steady-state voltage

bifurcation\_data.append(V\_steady)

# Plot bifurcation diagram

plt.figure(figsize=(10, 6))

for I\_ext, V\_steady in zip(I\_ext\_values, bifurcation\_data):

plt.plot([I\_ext] \* len(V\_steady), V\_steady, 'k.', markersize=1)

plt.xlabel('External Current (I\_ext)')

plt.ylabel('Membrane Potential (V)')

plt.title('Bifurcation Diagram of Neuronal Dynamics')

plt.show()

\end{lstlisting}

### Modeling Neuronal Limit Cycles and Attractors

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import odeint

# Define the neuronal dynamics model

def neuronal\_model(state, t, alpha, beta, gamma):

x, y = state

dxdt = alpha \* x - beta \* y - x \* (x\*\*2 + y\*\*2)

dydt = beta \* x + alpha \* y - y \* (x\*\*2 + y\*\*2)

return [dxdt, dydt]

# Parameters for the model

alpha = 0.1 # Controls the stability of the limit cycle

beta = 1.0 # Controls the frequency of oscillations

gamma = 1.0 # Not used in this simple model, but can be extended

# Initial state of the system

initial\_state = [1.0, 0.0]

# Time points for simulation

t = np.linspace(0, 50, 1000)

# Solve the differential equations

states = odeint(neuronal\_model, initial\_state, t, args=(alpha, beta, gamma))

# Plot the limit cycle attractor

plt.figure(figsize=(8, 6))

plt.plot(states[:, 0], states[:, 1], label='Limit Cycle')

plt.xlabel('x (Neuronal Activity)')

plt.ylabel('y (Neuronal Activity)')

plt.title('Neuronal Limit Cycle and Attractor')

plt.legend()

plt.grid(True)

plt.show()

\end{lstlisting}

### Chaos in Neural Systems using Lyapunov Exponents

\begin{lstlisting}

import numpy as np

from scipy.integrate import solve\_ivp

# Define the neural system dynamics (e.g., a chaotic neuron model)

def neural\_system(t, y, a=1.0, b=3.0, c=1.0, d=5.0, r=0.006):

x, y = y

dxdt = y - a \* x\*\*3 + b \* x\*\*2 + r

dydt = c - d \* x\*\*2 - y

return [dxdt, dydt]

# Compute the Lyapunov exponent for the system

def lyapunov\_exponent(y0, t\_span, t\_eval, a=1.0, b=3.0, c=1.0, d=5.0, r=0.006):

# Solve the system of ODEs

sol = solve\_ivp(neural\_system, t\_span, y0, t\_eval=t\_eval, args=(a, b, c, d, r))

# Compute the Jacobian matrix for the system

def jacobian(t, y):

x, y = y

dfdx = -3 \* a \* x\*\*2 + 2 \* b \* x

dfdy = 1

dgdx = -2 \* d \* x

dgdy = -1

return np.array([[dfdx, dfdy], [dgdx, dgdy]])

# Compute the Lyapunov exponent using the Jacobian

lyapunov\_sum = 0.0

for i in range(len(sol.t) - 1):

J = jacobian(sol.t[i], sol.y[:, i])

eigenvalues = np.linalg.eigvals(J)

lyapunov\_sum += np.log(np.max(np.abs(eigenvalues)))

return lyapunov\_sum / (sol.t[-1] - sol.t[0])

# Parameters for the simulation

y0 = [0.1, 0.1] # Initial conditions

t\_span = [0, 100] # Time span

t\_eval = np.linspace(0, 100, 1000) # Time points for evaluation

# Compute the Lyapunov exponent

lyap\_exp = lyapunov\_exponent(y0, t\_span, t\_eval)

print(f"Lyapunov Exponent: {lyap\_exp}")

\end{lstlisting}

### Network Synchronization in Cortical Circuits

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import solve\_ivp

# Define the Kuramoto model for network synchronization

def kuramoto(t, theta, N, K, omega):

dtheta = np.zeros(N)

for i in range(N):

dtheta[i] = omega[i] + (K/N) \* np.sum(np.sin(theta - theta[i]))

return dtheta

# Parameters

N = 100 # Number of oscillators

K = 2.0 # Coupling strength

omega = np.random.normal(1.0, 0.1, N) # Natural frequencies

# Initial conditions

theta0 = np.random.uniform(0, 2\*np.pi, N)

# Time span for integration

t\_span = (0, 50)

t\_eval = np.linspace(t\_span[0], t\_span[1], 1000)

# Solve the ODE

sol = solve\_ivp(kuramoto, t\_span, theta0, args=(N, K, omega), t\_eval=t\_eval)

# Plot the results

plt.figure(figsize=(10, 6))

for i in range(N):

plt.plot(sol.t, np.sin(sol.y[i]), lw=0.5)

plt.title("Network Synchronization in Cortical Circuits")

plt.xlabel("Time")

plt.ylabel("Phase Sin(theta)")

plt.show()

\end{lstlisting}

### Gamma Oscillations and Cognitive Function

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import spectrogram

# Parameters for gamma oscillations simulation

sampling\_rate = 1000 # Sampling rate in Hz

duration = 2.0 # Duration of the signal in seconds

t = np.linspace(0, duration, int(sampling\_rate \* duration), endpoint=False)

gamma\_freq = 40 # Gamma frequency in Hz

amplitude = 1.0 # Amplitude of the gamma oscillation

# Generate gamma oscillations

gamma\_signal = amplitude \* np.sin(2 \* np.pi \* gamma\_freq \* t)

# Add noise to simulate neural activity

noise = 0.1 \* np.random.normal(size=len(t))

gamma\_signal += noise

# Compute spectrogram to analyze frequency content

frequencies, times, Sxx = spectrogram(gamma\_signal, fs=sampling\_rate, nperseg=256)

# Plot the gamma oscillations and spectrogram

plt.figure(figsize=(12, 6))

# Plot the time-domain signal

plt.subplot(2, 1, 1)

plt.plot(t, gamma\_signal)

plt.title('Gamma Oscillations in Time Domain')

plt.xlabel('Time (s)')

plt.ylabel('Amplitude')

# Plot the spectrogram

plt.subplot(2, 1, 2)

plt.pcolormesh(times, frequencies, 10 \* np.log10(Sxx), shading='gouraud')

plt.title('Spectrogram of Gamma Oscillations')

plt.xlabel('Time (s)')

plt.ylabel('Frequency (Hz)')

plt.colorbar(label='Intensity (dB)')

plt.tight\_layout()

plt.show()

\end{lstlisting}

### Simulating Neural Bursting Dynamics

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import solve\_ivp

# Define the Hindmarsh-Rose model for neural bursting dynamics

def hindmarsh\_rose(t, state, a, b, c, d, r, s, x\_r):

x, y, z = state

dxdt = y - a\*x\*\*3 + b\*x\*\*2 - z + I

dydt = c - d\*x\*\*2 - y

dzdt = r\*(s\*(x - x\_r) - z)

return [dxdt, dydt, dzdt]

# Parameters for the Hindmarsh-Rose model

a, b, c, d = 1.0, 3.0, 1.0, 5.0

r, s, x\_r = 0.006, 4.0, -1.6

I = 3.0 # External current input

# Initial state [x, y, z]

initial\_state = [0.0, 0.0, 0.0]

# Time span for the simulation

t\_span = (0, 200)

t\_eval = np.linspace(0, 200, 10000)

# Solve the differential equations

sol = solve\_ivp(hindmarsh\_rose, t\_span, initial\_state, args=(a, b, c, d, r, s, x\_r), t\_eval=t\_eval)

# Plot the results

plt.figure(figsize=(10, 6))

plt.plot(sol.t, sol.y[0], label='Membrane Potential (x)')

plt.xlabel('Time (ms)')

plt.ylabel('Membrane Potential')

plt.title('Neural Bursting Dynamics - Hindmarsh-Rose Model')

plt.legend()

plt.grid(True)

plt.show()

\end{lstlisting}

### Computational Modeling of Seizure Activity

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import solve\_ivp

# Define the neural mass model for seizure activity

def neural\_mass\_model(t, y, a, b, tau):

"""Model of neural dynamics with parameters a, b, and tau."""

V, W = y # V: membrane potential, W: recovery variable

dVdt = V - (V\*\*3)/3 - W + a # Membrane potential dynamics

dWdt = (V + b - W) / tau # Recovery variable dynamics

return [dVdt, dWdt]

# Parameters for seizure-like activity

a = 0.7 # External input

b = 0.8 # Recovery parameter

tau = 12.5 # Time constant

# Initial conditions

y0 = [0.0, 0.0] # Initial membrane potential and recovery variable

# Time span for simulation

t\_span = (0, 200) # Simulation time from 0 to 200 ms

t\_eval = np.linspace(0, 200, 1000) # Time points for evaluation

# Solve the differential equations

sol = solve\_ivp(neural\_mass\_model, t\_span, y0, args=(a, b, tau), t\_eval=t\_eval)

# Plot the results

plt.figure(figsize=(10, 6))

plt.plot(sol.t, sol.y[0], label="Membrane Potential (V)")

plt.plot(sol.t, sol.y[1], label="Recovery Variable (W)")

plt.xlabel("Time (ms)")

plt.ylabel("Amplitude")

plt.title("Computational Modeling of Seizure Activity")

plt.legend()

plt.grid(True)

plt.show()

\end{lstlisting}

### Wilson-Cowan Model for Excitatory-Inhibitory Networks

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Define Wilson-Cowan model parameters

tau\_e = 10 # Time constant for excitatory population

tau\_i = 10 # Time constant for inhibitory population

w\_ee = 12 # Excitatory to excitatory weight

w\_ei = 4 # Inhibitory to excitatory weight

w\_ie = 13 # Excitatory to inhibitory weight

w\_ii = 11 # Inhibitory to inhibitory weight

I\_e = 0.5 # External input to excitatory population

I\_i = 0.5 # External input to inhibitory population

# Define activation function (sigmoid)

def sigmoid(x):

return 1 / (1 + np.exp(-x))

# Define Wilson-Cowan equations

def wilson\_cowan(E, I):

dE\_dt = (-E + sigmoid(w\_ee \* E - w\_ei \* I + I\_e)) / tau\_e

dI\_dt = (-I + sigmoid(w\_ie \* E - w\_ii \* I + I\_i)) / tau\_i

return dE\_dt, dI\_dt

# Simulation parameters

dt = 0.1 # Time step

T = 100 # Total simulation time

steps = int(T / dt)

# Initialize arrays to store results

E = np.zeros(steps)

I = np.zeros(steps)

E[0] = 0.1 # Initial excitatory activity

I[0] = 0.1 # Initial inhibitory activity

# Run simulation

for t in range(1, steps):

dE\_dt, dI\_dt = wilson\_cowan(E[t-1], I[t-1])

E[t] = E[t-1] + dE\_dt \* dt

I[t] = I[t-1] + dI\_dt \* dt

# Plot results

plt.figure(figsize=(10, 6))

plt.plot(np.arange(0, T, dt), E, label='Excitatory Activity (E)')

plt.plot(np.arange(0, T, dt), I, label='Inhibitory Activity (I)')

plt.xlabel('Time')

plt.ylabel('Activity')

plt.title('Wilson-Cowan Model Dynamics')

plt.legend()

plt.grid(True)

plt.show()

\end{lstlisting}

## Neural Coding and Information Processing

### Poisson Spike Train Generation

\begin{lstlisting}

import numpy as np

def generate\_poisson\_spike\_train(rate, duration, dt=1e-3):

"""

Generate a Poisson spike train for a given firing rate and duration.

Parameters:

rate (float): Firing rate in spikes per second.

duration (float): Duration of the spike train in seconds.

dt (float): Time step in seconds (default is 1 ms).

Returns:

np.ndarray: Binary spike train where 1 indicates a spike and 0 indicates no spike.

"""

num\_steps = int(duration / dt) # Number of time steps

spike\_prob = rate \* dt # Probability of a spike in each time step

spike\_train = (np.random.rand(num\_steps) < spike\_prob).astype(int) # Generate spikes

return spike\_train

# Example usage

rate = 50 # 50 spikes per second

duration = 1.0 # 1 second

spike\_train = generate\_poisson\_spike\_train(rate, duration)

print("Spike Train:", spike\_train)

\end{lstlisting}

### Computing Fano Factor for Neuronal Variability

\begin{lstlisting}

import numpy as np

def compute\_fano\_factor(spike\_counts):

"""

Compute the Fano Factor for neuronal variability.

Parameters:

spike\_counts (list or numpy array): Array of spike counts across trials.

Returns:

float: Fano Factor value.

"""

mean\_spike\_count = np.mean(spike\_counts) # Compute mean spike count

variance\_spike\_count = np.var(spike\_counts) # Compute variance of spike counts

fano\_factor = variance\_spike\_count / mean\_spike\_count # Calculate Fano Factor

return fano\_factor

# Example usage

spike\_counts = [10, 12, 8, 11, 9, 13, 7, 10] # Example spike counts across trials

fano\_factor = compute\_fano\_factor(spike\_counts)

print(f"Fano Factor: {fano\_factor:.2f}")

\end{lstlisting}

### Information Theory: Mutual Information in Spike Trains

\begin{lstlisting}

import numpy as np

from scipy.stats import entropy

def mutual\_information(spike\_train1, spike\_train2, bin\_size=0.01):

"""

Calculate mutual information between two spike trains.

Parameters:

spike\_train1 (array): Spike times for the first neuron.

spike\_train2 (array): Spike times for the second neuron.

bin\_size (float): Time bin size for discretization.

Returns:

mi (float): Mutual information between the two spike trains.

"""

# Discretize spike trains into binary vectors

max\_time = max(max(spike\_train1), max(spike\_train2))

bins = np.arange(0, max\_time + bin\_size, bin\_size)

hist1, \_ = np.histogram(spike\_train1, bins=bins)

hist2, \_ = np.histogram(spike\_train2, bins=bins)

# Joint and marginal probabilities

joint\_prob = np.histogram2d(hist1, hist2, bins=(2, 2))[0] / len(bins)

prob1 = np.sum(joint\_prob, axis=1)

prob2 = np.sum(joint\_prob, axis=0)

# Calculate mutual information

mi = entropy(prob1) + entropy(prob2) - entropy(joint\_prob.flatten())

return mi

# Example usage

spike\_train1 = np.array([0.1, 0.3, 0.5, 0.7, 0.9])

spike\_train2 = np.array([0.2, 0.4, 0.6, 0.8, 1.0])

mi = mutual\_information(spike\_train1, spike\_train2)

print(f"Mutual Information: {mi:.4f}")

\end{lstlisting}

### Bayesian Decoding of Neural Signals

\begin{lstlisting}

import numpy as np

from scipy.stats import norm

# Simulate neural spike counts for a given stimulus

def simulate\_spike\_counts(stimulus, n\_neurons, tuning\_curves, noise\_std=1.0):

"""

Simulate spike counts based on tuning curves and Gaussian noise.

"""

spike\_counts = np.zeros(n\_neurons)

for i in range(n\_neurons):

mean\_rate = tuning\_curves[i](stimulus)

spike\_counts[i] = np.random.normal(mean\_rate, noise\_std)

return spike\_counts

# Define tuning curves for neurons (Gaussian tuning)

def gaussian\_tuning\_curve(peak, width):

"""

Return a Gaussian tuning curve function for a neuron.

"""

return lambda x: np.exp(-0.5 \* ((x - peak) / width) \*\* 2)

# Bayesian decoding of stimulus from spike counts

def bayesian\_decoding(spike\_counts, tuning\_curves, stimulus\_range, prior=None):

"""

Decode stimulus using Bayesian inference.

"""

if prior is None:

prior = np.ones\_like(stimulus\_range) / len(stimulus\_range) # Uniform prior

likelihood = np.ones\_like(stimulus\_range)

for i, curve in enumerate(tuning\_curves):

likelihood \*= norm.pdf(spike\_counts[i], loc=curve(stimulus\_range), scale=1.0)

posterior = likelihood \* prior

posterior /= np.sum(posterior) # Normalize posterior

return stimulus\_range[np.argmax(posterior)] # MAP estimate

# Example usage

stimulus\_range = np.linspace(0, 10, 100) # Possible stimulus values

n\_neurons = 5

tuning\_curves = [gaussian\_tuning\_curve(peak=2 + i \* 2, width=1) for i in range(n\_neurons)]

# Simulate spike counts for a true stimulus value

true\_stimulus = 5.0

spike\_counts = simulate\_spike\_counts(true\_stimulus, n\_neurons, tuning\_curves)

# Decode the stimulus using Bayesian inference

decoded\_stimulus = bayesian\_decoding(spike\_counts, tuning\_curves, stimulus\_range)

print(f"True Stimulus: {true\_stimulus}, Decoded Stimulus: {decoded\_stimulus}")

\end{lstlisting}

### Hidden Markov Model (HMM) for Neural State Estimation

\begin{lstlisting}

import numpy as np

from hmmlearn import hmm

# Simulate neural data: 100 trials, 50 time points, 2 states (e.g., active/inactive)

np.random.seed(42)

n\_trials = 100

n\_time\_points = 50

n\_states = 2

neural\_data = np.random.rand(n\_trials, n\_time\_points) # Random neural activity

# Define HMM model

model = hmm.GaussianHMM(n\_components=n\_states, covariance\_type="diag", n\_iter=100)

# Fit the model to the neural data

model.fit(neural\_data)

# Predict the most likely sequence of states

hidden\_states = model.predict(neural\_data)

# Output the transition matrix and state means

print("Transition Matrix:\n", model.transmat\_)

print("State Means:\n", model.means\_)

# Example: Visualize hidden states for the first trial

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 4))

plt.plot(hidden\_states[0], label="Hidden States")

plt.title("Hidden States for First Trial")

plt.xlabel("Time Points")

plt.ylabel("State")

plt.legend()

plt.show()

\end{lstlisting}

### Signal Filtering and Smoothing for Neural Data

\begin{lstlisting}

import numpy as np

import scipy.signal as signal

import matplotlib.pyplot as plt

# Simulate neural data (e.g., spike train convolved with a kernel)

np.random.seed(42)

time = np.linspace(0, 1, 1000)

neural\_signal = np.sin(2 \* np.pi \* 10 \* time) + 0.5 \* np.random.normal(size=len(time))

# Apply a low-pass Butterworth filter to remove high-frequency noise

b, a = signal.butter(4, 0.1, 'low') # 4th order, cutoff at 0.1 \* Nyquist frequency

filtered\_signal = signal.filtfilt(b, a, neural\_signal)

# Smooth the signal using a Savitzky-Golay filter

smoothed\_signal = signal.savgol\_filter(filtered\_signal, window\_length=51, polyorder=3)

# Plot the original, filtered, and smoothed signals

plt.figure(figsize=(10, 6))

plt.plot(time, neural\_signal, label='Original Signal', alpha=0.5)

plt.plot(time, filtered\_signal, label='Filtered Signal', linewidth=2)

plt.plot(time, smoothed\_signal, label='Smoothed Signal', linewidth=2)

plt.xlabel('Time (s)')

plt.ylabel('Amplitude')

plt.title('Signal Filtering and Smoothing for Neural Data')

plt.legend()

plt.grid(True)

plt.show()

\end{lstlisting}

### Kalman Filtering for Brain-Machine Interfaces

\begin{lstlisting}

import numpy as np

class KalmanFilter:

def \_\_init\_\_(self, A, H, Q, R, x0, P0):

"""

Initialize the Kalman Filter.

A: State transition matrix

H: Observation matrix

Q: Process noise covariance

R: Measurement noise covariance

x0: Initial state estimate

P0: Initial estimate covariance

"""

self.A = A

self.H = H

self.Q = Q

self.R = R

self.x = x0

self.P = P0

def predict(self):

"""Predict the next state and covariance."""

self.x = np.dot(self.A, self.x)

self.P = np.dot(np.dot(self.A, self.P), self.A.T) + self.Q

return self.x

def update(self, z):

"""

Update the state estimate with a new measurement.

z: Measurement vector

"""

y = z - np.dot(self.H, self.x) # Innovation

S = np.dot(np.dot(self.H, self.P), self.H.T) + self.R # Innovation covariance

K = np.dot(np.dot(self.P, self.H.T), np.linalg.inv(S)) # Kalman gain

self.x = self.x + np.dot(K, y) # Updated state estimate

self.P = self.P - np.dot(np.dot(K, self.H), self.P) # Updated estimate covariance

return self.x

# Example usage for Brain-Machine Interfaces

# Define matrices and initial conditions

A = np.array([[1, 0.1], [0, 1]]) # State transition matrix

H = np.array([[1, 0]]) # Observation matrix

Q = np.eye(2) \* 0.01 # Process noise covariance

R = np.array([[0.1]]) # Measurement noise covariance

x0 = np.array([0, 0]) # Initial state estimate

P0 = np.eye(2) # Initial estimate covariance

# Initialize Kalman Filter

kf = KalmanFilter(A, H, Q, R, x0, P0)

# Simulate a measurement and update

measurement = np.array([1.2]) # Simulated neural signal measurement

kf.predict()

updated\_state = kf.update(measurement)

print("Updated State Estimate:", updated\_state)

\end{lstlisting}

### Spike-Timing-Dependent Plasticity (STDP) Simulation

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Parameters

tau\_plus = 20.0 # Time constant for potentiation

tau\_minus = 20.0 # Time constant for depression

A\_plus = 0.01 # Potentiation amplitude

A\_minus = 0.01 # Depression amplitude

time\_steps = 1000 # Total simulation time steps

dt = 1.0 # Time step size

# Pre- and post-synaptic spike times

pre\_spikes = np.array([100, 300, 500]) # Pre-synaptic spike times

post\_spikes = np.array([150, 350, 550]) # Post-synaptic spike times

# Initialize weight change

delta\_w = 0.0

# STDP function

def stdp(dt):

if dt > 0:

return A\_plus \* np.exp(-dt / tau\_plus) # Potentiation

else:

return -A\_minus \* np.exp(dt / tau\_minus) # Depression

# Simulate STDP

for pre in pre\_spikes:

for post in post\_spikes:

delta\_t = (post - pre) \* dt

delta\_w += stdp(delta\_t)

# Plot STDP window

time\_diffs = np.linspace(-50, 50, 100)

stdp\_values = [stdp(t) for t in time\_diffs]

plt.plot(time\_diffs, stdp\_values, label="STDP Window")

plt.xlabel("Time Difference (ms)")

plt.ylabel("Weight Change")

plt.title("Spike-Timing-Dependent Plasticity (STDP)")

plt.axhline(0, color='black', linestyle='--')

plt.legend()

plt.show()

\end{lstlisting}

### Predictive Coding Model of Sensory Processing

\begin{lstlisting}

import numpy as np

class PredictiveCodingModel:

def \_\_init\_\_(self, input\_dim, hidden\_dim, learning\_rate=0.01):

# Initialize weights and biases for the predictive coding model

self.W = np.random.randn(hidden\_dim, input\_dim) # Weight matrix

self.b = np.zeros(hidden\_dim) # Bias vector

self.learning\_rate = learning\_rate

def predict(self, x):

# Generate predictions based on current model parameters

return np.dot(self.W, x) + self.b

def update\_parameters(self, x, prediction\_error):

# Update weights and biases using gradient descent

self.W -= self.learning\_rate \* np.outer(prediction\_error, x)

self.b -= self.learning\_rate \* prediction\_error

def train(self, x, y, epochs=100):

# Train the model using predictive coding principles

for \_ in range(epochs):

prediction = self.predict(x) # Generate prediction

prediction\_error = y - prediction # Compute prediction error

self.update\_parameters(x, prediction\_error) # Update parameters

# Example usage

input\_dim = 3

hidden\_dim = 2

model = PredictiveCodingModel(input\_dim, hidden\_dim)

# Sample input and target

x = np.array([0.5, 0.3, 0.2])

y = np.array([0.8, 0.6])

# Train the model

model.train(x, y, epochs=1000)

\end{lstlisting}

### Reinforcement Learning and Dopaminergic Reward Systems

\begin{lstlisting}

import numpy as np

class QLearningAgent:

def \_\_init\_\_(self, num\_states, num\_actions, alpha=0.1, gamma=0.9, epsilon=0.1):

self.num\_states = num\_states

self.num\_actions = num\_actions

self.alpha = alpha # Learning rate

self.gamma = gamma # Discount factor

self.epsilon = epsilon # Exploration rate

self.q\_table = np.zeros((num\_states, num\_actions)) # Q-value table

def choose\_action(self, state):

if np.random.rand() < self.epsilon: # Explore

return np.random.choice(self.num\_actions)

else: # Exploit

return np.argmax(self.q\_table[state])

def update\_q\_table(self, state, action, reward, next\_state):

# Q-learning update rule

best\_next\_action = np.argmax(self.q\_table[next\_state])

td\_target = reward + self.gamma \* self.q\_table[next\_state][best\_next\_action]

td\_error = td\_target - self.q\_table[state][action]

self.q\_table[state][action] += self.alpha \* td\_error

# Simulate a simple environment

num\_states = 5

num\_actions = 3

agent = QLearningAgent(num\_states, num\_actions)

# Example episode

state = 0

for \_ in range(10): # Simulate 10 steps

action = agent.choose\_action(state)

reward = np.random.randint(0, 10) # Random reward

next\_state = (state + 1) % num\_states # Simple state transition

agent.update\_q\_table(state, action, reward, next\_state)

state = next\_state

print("Q-table after simulation:")

print(agent.q\_table)

\end{lstlisting}

## Machine Learning for Neuroscience

### Principal Component Analysis (PCA) for Neural Data

\begin{lstlisting}

import numpy as np

from sklearn.decomposition import PCA

import matplotlib.pyplot as plt

# Simulate neural data: 100 neurons, 500 time points

np.random.seed(42)

neural\_data = np.random.randn(100, 500) # Random neural activity

# Perform PCA

pca = PCA(n\_components=3) # Reduce to 3 principal components

pca\_result = pca.fit\_transform(neural\_data.T) # Transpose for time x neurons

# Plot the first 3 principal components

plt.figure(figsize=(10, 6))

plt.plot(pca\_result[:, 0], label='PC1')

plt.plot(pca\_result[:, 1], label='PC2')

plt.plot(pca\_result[:, 2], label='PC3')

plt.xlabel('Time Points')

plt.ylabel('Component Value')

plt.title('Principal Components of Neural Data')

plt.legend()

plt.show()

\end{lstlisting}

### Independent Component Analysis (ICA) for EEG Processing

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from sklearn.decomposition import FastICA

from mne import Epochs, pick\_types, events\_from\_annotations

from mne.io import read\_raw\_fif

from mne.datasets import sample

# Load sample EEG data

data\_path = sample.data\_path()

raw\_fname = data\_path / 'MEG' / 'sample' / 'sample\_audvis\_filt-0-40\_raw.fif'

raw = read\_raw\_fif(raw\_fname, preload=True)

# Filter and epoch the data

raw.filter(1, 40, fir\_design='firwin')

events, \_ = events\_from\_annotations(raw, event\_id=1)

picks = pick\_types(raw.info, meg=False, eeg=True, eog=False, stim=False)

epochs = Epochs(raw, events, event\_id=1, tmin=-0.2, tmax=0.5, picks=picks, baseline=(None, 0))

# Extract EEG data for ICA

eeg\_data = epochs.get\_data() # Shape: (n\_epochs, n\_channels, n\_times)

eeg\_data = np.reshape(eeg\_data, (eeg\_data.shape[0], -1)) # Flatten time dimension

# Apply ICA to EEG data

ica = FastICA(n\_components=10, random\_state=42) # Reduce to 10 components

ica\_result = ica.fit\_transform(eeg\_data) # Transform data to ICA components

# Plot the ICA components

plt.figure(figsize=(12, 6))

for i in range(ica\_result.shape[1]):

plt.subplot(2, 5, i + 1)

plt.plot(ica\_result[:, i])

plt.title(f'Component {i + 1}')

plt.tight\_layout()

plt.show()

\end{lstlisting}

### t-SNE for Neural Population Analysis

\begin{lstlisting}

import numpy as np

from sklearn.manifold import TSNE

import matplotlib.pyplot as plt

# Simulated neural population data (100 neurons, 500 time points)

neural\_data = np.random.rand(100, 500) # Random data for illustration

# Apply t-SNE for dimensionality reduction

tsne = TSNE(n\_components=2, perplexity=30, n\_iter=300) # 2D embedding

embedded\_data = tsne.fit\_transform(neural\_data)

# Plot the t-SNE results

plt.figure(figsize=(8, 6))

plt.scatter(embedded\_data[:, 0], embedded\_data[:, 1], alpha=0.7)

plt.title('t-SNE of Neural Population Data')

plt.xlabel('t-SNE Component 1')

plt.ylabel('t-SNE Component 2')

plt.grid(True)

plt.show()

\end{lstlisting}

### Clustering Neural Activity using K-Means

[Clustering Neural Activity using K-Means] \begin{lstlisting}

import numpy as np

from sklearn.cluster import KMeans

import matplotlib.pyplot as plt

# Simulated neural activity data (e.g., spike counts or firing rates)

# Rows: neurons, Columns: time points

neural\_data = np.random.rand(50, 100) # 50 neurons, 100 time points

# Apply K-Means clustering to group neurons based on activity patterns

n\_clusters = 3 # Number of clusters (e.g., distinct activity patterns)

kmeans = KMeans(n\_clusters=n\_clusters, random\_state=42)

cluster\_labels = kmeans.fit\_predict(neural\_data)

# Visualize the clustering results

plt.figure(figsize=(10, 6))

for cluster in range(n\_clusters):

# Plot the average activity pattern for each cluster

cluster\_activity = neural\_data[cluster\_labels == cluster].mean(axis=0)

plt.plot(cluster\_activity, label=f'Cluster {cluster + 1}')

plt.title('Average Neural Activity Patterns by Cluster')

plt.xlabel('Time Points')

plt.ylabel('Normalized Activity')

plt.legend()

plt.show()

\end{lstlisting}

### Neural Network-Based Classification of EEG Data

\begin{lstlisting}

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout, Conv1D, MaxPooling1D, Flatten

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

# Load EEG data (example: X = features, y = labels)

# X = np.load('eeg\_data.npy') # Shape: (samples, time\_steps, channels)

# y = np.load('eeg\_labels.npy') # Shape: (samples,)

X, y = np.random.rand(1000, 128, 64), np.random.randint(0, 2, 1000) # Example data

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Standardize the data

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train.reshape(-1, X\_train.shape[-1])).reshape(X\_train.shape)

X\_test = scaler.transform(X\_test.reshape(-1, X\_test.shape[-1])).reshape(X\_test.shape)

# Build a 1D CNN model for EEG classification

model = Sequential([

Conv1D(filters=64, kernel\_size=3, activation='relu', input\_shape=(X\_train.shape[1], X\_train.shape[2])),

MaxPooling1D(pool\_size=2),

Dropout(0.5),

Flatten(),

Dense(128, activation='relu'),

Dropout(0.5),

Dense(1, activation='sigmoid') # Binary classification

])

# Compile the model

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Train the model

model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.2)

# Evaluate the model

loss, accuracy = model.evaluate(X\_test, y\_test)

print(f"Test Accuracy: {accuracy:.4f}")

\end{lstlisting}

### Deep Learning for Spike Sorting in Electrophysiology

\begin{lstlisting}

import numpy as np

import tensorflow as tf

from tensorflow.keras import layers, models

from sklearn.model\_selection import train\_test\_split

# Load preprocessed spike data (replace with actual data loading)

spike\_data = np.load('spike\_data.npy') # Shape: (n\_samples, n\_timepoints, n\_channels)

labels = np.load('labels.npy') # Shape: (n\_samples,)

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(spike\_data, labels, test\_size=0.2, random\_state=42)

# Define a deep learning model for spike sorting

model = models.Sequential([

layers.Conv1D(32, kernel\_size=5, activation='relu', input\_shape=(X\_train.shape[1], X\_train.shape[2])),

layers.MaxPooling1D(pool\_size=2),

layers.Conv1D(64, kernel\_size=3, activation='relu'),

layers.MaxPooling1D(pool\_size=2),

layers.Flatten(),

layers.Dense(128, activation='relu'),

layers.Dropout(0.5),

layers.Dense(len(np.unique(labels)), activation='softmax') # Output layer for classification

])

# Compile the model

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# Train the model

model.fit(X\_train, y\_train, epochs=20, batch\_size=32, validation\_split=0.1)

# Evaluate the model on the test set

test\_loss, test\_acc = model.evaluate(X\_test, y\_test)

print(f"Test Accuracy: {test\_acc:.4f}")

\end{lstlisting}

### Autoencoders for Feature Extraction from Neural Data

\begin{lstlisting}

import numpy as np

import tensorflow as tf

from tensorflow.keras import layers, models

# Load neural data (replace with actual data loading)

# Example: neural\_data = np.load('neural\_data.npy')

neural\_data = np.random.rand(1000, 50) # 1000 samples, 50 features

# Define the autoencoder model

input\_dim = neural\_data.shape[1]

encoding\_dim = 10 # Reduced feature dimension

# Encoder

input\_layer = layers.Input(shape=(input\_dim,))

encoded = layers.Dense(encoding\_dim, activation='relu')(input\_layer)

# Decoder

decoded = layers.Dense(input\_dim, activation='sigmoid')(encoded)

# Autoencoder model

autoencoder = models.Model(input\_layer, decoded)

# Compile the model

autoencoder.compile(optimizer='adam', loss='mean\_squared\_error')

# Train the autoencoder

autoencoder.fit(neural\_data, neural\_data, epochs=50, batch\_size=32, shuffle=True)

# Extract features using the encoder

encoder = models.Model(input\_layer, encoded)

encoded\_data = encoder.predict(neural\_data)

# encoded\_data now contains the extracted features

\end{lstlisting}

### Transfer Learning in Neuroscience Applications

\begin{lstlisting}

import torch

import torch.nn as nn

from torchvision import models

from torch.utils.data import DataLoader

from sklearn.model\_selection import train\_test\_split

import numpy as np

# Load pre-trained ResNet18 model

pretrained\_model = models.resnet18(pretrained=True)

# Modify the final layer for binary classification (e.g., cognitive task classification)

pretrained\_model.fc = nn.Linear(pretrained\_model.fc.in\_features, 1)

# Load neuroscience dataset (e.g., fMRI or EEG data)

# Assuming data is preprocessed and normalized

X = np.load('neuroscience\_data.npy') # Input features

y = np.load('labels.npy') # Binary labels

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Convert numpy arrays to PyTorch tensors

X\_train = torch.tensor(X\_train, dtype=torch.float32)

X\_test = torch.tensor(X\_test, dtype=torch.float32)

y\_train = torch.tensor(y\_train, dtype=torch.float32)

y\_test = torch.tensor(y\_test, dtype=torch.float32)

# Create DataLoader for batch processing

train\_dataset = torch.utils.data.TensorDataset(X\_train, y\_train)

train\_loader = DataLoader(train\_dataset, batch\_size=32, shuffle=True)

# Define loss function and optimizer

criterion = nn.BCEWithLogitsLoss()

optimizer = torch.optim.Adam(pretrained\_model.parameters(), lr=0.001)

# Training loop

for epoch in range(10): # Number of epochs

pretrained\_model.train()

for inputs, labels in train\_loader:

optimizer.zero\_grad()

outputs = pretrained\_model(inputs)

loss = criterion(outputs.squeeze(), labels)

loss.backward()

optimizer.step()

print(f'Epoch {epoch+1}, Loss: {loss.item()}')

# Evaluate the model on the test set

pretrained\_model.eval()

with torch.no\_grad():

test\_outputs = pretrained\_model(X\_test)

test\_predictions = (torch.sigmoid(test\_outputs) > 0.5).float()

accuracy = (test\_predictions.squeeze() == y\_test).float().mean()

print(f'Test Accuracy: {accuracy.item()}')

\end{lstlisting}

### Supervised vs. Unsupervised Learning in Neural Data

\begin{lstlisting}

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.cluster import KMeans

from sklearn.metrics import accuracy\_score, silhouette\_score

# Simulate neural data: 100 samples, 50 features, binary labels

np.random.seed(42)

X = np.random.randn(100, 50) # Neural activity data

y = np.random.randint(0, 2, 100) # Binary labels (e.g., stimulus presence)

# Supervised Learning: Logistic Regression

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

model = LogisticRegression()

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

print(f"Supervised Accuracy: {accuracy\_score(y\_test, y\_pred):.2f}")

# Unsupervised Learning: K-Means Clustering

kmeans = KMeans(n\_clusters=2, random\_state=42)

clusters = kmeans.fit\_predict(X)

silhouette\_avg = silhouette\_score(X, clusters)

print(f"Unsupervised Silhouette Score: {silhouette\_avg:.2f}")

\end{lstlisting}

### Graph Neural Networks (GNN) for Brain Connectivity

\begin{lstlisting}

import torch

import torch.nn.functional as F

from torch\_geometric.nn import GCNConv

from torch\_geometric.data import Data

# Define a simple GNN model for brain connectivity analysis

class BrainGNN(torch.nn.Module):

def \_\_init\_\_(self, num\_features, hidden\_channels, num\_classes):

super(BrainGNN, self).\_\_init\_\_()

self.conv1 = GCNConv(num\_features, hidden\_channels) # First GCN layer

self.conv2 = GCNConv(hidden\_channels, num\_classes) # Second GCN layer

def forward(self, x, edge\_index):

x = self.conv1(x, edge\_index) # Apply first convolution

x = F.relu(x) # Apply ReLU activation

x = self.conv2(x, edge\_index) # Apply second convolution

return F.log\_softmax(x, dim=1) # Apply log softmax for classification

# Example brain connectivity data (nodes and edges)

num\_nodes = 10

num\_features = 5

edge\_index = torch.tensor([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],

[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]], dtype=torch.long)

x = torch.randn(num\_nodes, num\_features) # Node features (e.g., brain regions)

# Initialize the GNN model

model = BrainGNN(num\_features=num\_features, hidden\_channels=16, num\_classes=2)

# Forward pass

output = model(x, edge\_index)

print(output) # Output predictions for each node

\end{lstlisting}

## Large-Scale Network Simulations

### Small-World and Scale-Free Brain Network Analysis

\begin{lstlisting}

import numpy as np

import networkx as nx

import matplotlib.pyplot as plt

# Generate a small-world network using Watts-Strogatz model

n = 100 # Number of nodes

k = 4 # Each node is connected to k nearest neighbors

p = 0.1 # Probability of rewiring

small\_world\_graph = nx.watts\_strogatz\_graph(n, k, p)

# Generate a scale-free network using Barabasi-Albert model

m = 2 # Number of edges to attach from a new node to existing nodes

scale\_free\_graph = nx.barabasi\_albert\_graph(n, m)

# Calculate small-world properties

clustering\_coefficient = nx.average\_clustering(small\_world\_graph)

shortest\_path\_length = nx.average\_shortest\_path\_length(small\_world\_graph)

# Calculate scale-free properties

degree\_sequence = sorted([d for n, d in scale\_free\_graph.degree()], reverse=True)

degree\_distribution = np.unique(degree\_sequence, return\_counts=True)

# Visualize the networks

plt.figure(figsize=(12, 6))

plt.subplot(121)

nx.draw(small\_world\_graph, node\_size=50, with\_labels=False)

plt.title("Small-World Network")

plt.subplot(122)

nx.draw(scale\_free\_graph, node\_size=50, with\_labels=False)

plt.title("Scale-Free Network")

plt.show()

# Output properties

print(f"Small-World Clustering Coefficient: {clustering\_coefficient}")

print(f"Small-World Shortest Path Length: {shortest\_path\_length}")

print(f"Scale-Free Degree Distribution: {degree\_distribution}")

\end{lstlisting}

### Simulating Large-Scale Cortical Networks in NEST

\begin{lstlisting}

import nest

import numpy as np

# Initialize NEST kernel

nest.ResetKernel()

# Set simulation parameters

sim\_time = 1000.0 # Simulation time in ms

num\_neurons = 10000 # Number of neurons in the network

connection\_prob = 0.1 # Connection probability

# Create neuron population

neuron\_params = {'tau\_m': 20.0, 'E\_L': -65.0, 'V\_th': -50.0, 'V\_reset': -65.0}

neurons = nest.Create('iaf\_psc\_alpha', num\_neurons, params=neuron\_params)

# Create spike detector

spike\_detector = nest.Create('spike\_detector')

# Connect neurons with random connectivity

nest.Connect(neurons, neurons, {'rule': 'pairwise\_bernoulli', 'p': connection\_prob})

# Connect neurons to spike detector

nest.Connect(neurons, spike\_detector)

# Simulate the network

nest.Simulate(sim\_time)

# Retrieve and analyze spike data

events = nest.GetStatus(spike\_detector, 'events')[0]

spike\_times = events['times']

spike\_senders = events['senders']

# Example: Calculate mean firing rate

mean\_firing\_rate = len(spike\_times) / (num\_neurons \* sim\_time / 1000.0)

print(f"Mean firing rate: {mean\_firing\_rate:.2f} Hz")

\end{lstlisting}

### Structural vs. Functional Connectivity in Brain Networks

\begin{lstlisting}

import numpy as np

import networkx as nx

import matplotlib.pyplot as plt

from scipy.stats import pearsonr

# Generate a structural connectivity matrix (random for demonstration)

num\_nodes = 10

structural\_connectivity = np.random.rand(num\_nodes, num\_nodes)

structural\_connectivity = (structural\_connectivity + structural\_connectivity.T) / 2 # Symmetrize

np.fill\_diagonal(structural\_connectivity, 0) # No self-connections

# Simulate functional connectivity using Pearson correlation

time\_series = np.random.randn(100, num\_nodes) # Simulated fMRI time series

functional\_connectivity = np.zeros((num\_nodes, num\_nodes))

for i in range(num\_nodes):

for j in range(num\_nodes):

functional\_connectivity[i, j], \_ = pearsonr(time\_series[:, i], time\_series[:, j])

# Visualize structural connectivity

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

G\_structural = nx.from\_numpy\_array(structural\_connectivity)

nx.draw(G\_structural, with\_labels=True, node\_color='lightblue', edge\_color='gray')

plt.title("Structural Connectivity")

# Visualize functional connectivity

plt.subplot(1, 2, 2)

G\_functional = nx.from\_numpy\_array(functional\_connectivity)

nx.draw(G\_functional, with\_labels=True, node\_color='lightgreen', edge\_color='gray')

plt.title("Functional Connectivity")

plt.show()

\end{lstlisting}

### Diffusion Tensor Imaging (DTI) and Tractography Simulation

\begin{lstlisting}

import numpy as np

import nibabel as nib

from dipy.data import get\_fnames

from dipy.io.gradients import read\_bvals\_bvecs

from dipy.core.gradients import gradient\_table

from dipy.reconst.dti import TensorModel

from dipy.reconst.csdeconv import auto\_response

from dipy.tracking import utils

from dipy.tracking.streamline import Streamlines

from dipy.tracking.local\_tracking import LocalTracking

from dipy.tracking.stopping\_criterion import ThresholdStoppingCriterion

from dipy.viz import window, actor

# Load example DTI data

hardi\_fname, bval\_fname, bvec\_fname = get\_fnames('stanford\_hardi')

data = nib.load(hardi\_fname).get\_fdata()

bvals, bvecs = read\_bvals\_bvecs(bval\_fname, bvec\_fname)

gtab = gradient\_table(bvals, bvecs)

# Fit the tensor model

tenmodel = TensorModel(gtab)

tenfit = tenmodel.fit(data)

# Generate FA and eigenvectors for tractography

FA = tenfit.fa

evecs = tenfit.evecs

# Create a stopping criterion for tracking

stopping\_criterion = ThresholdStoppingCriterion(FA, 0.2)

# Define seed points for tractography

seed\_mask = FA > 0.6

seeds = utils.seeds\_from\_mask(seed\_mask, density=1)

# Perform local tracking

streamline\_generator = LocalTracking(evecs, stopping\_criterion, seeds, step\_size=0.5)

streamlines = Streamlines(streamline\_generator)

# Visualize the streamlines

scene = window.Scene()

streamline\_actor = actor.line(streamlines)

scene.add(streamline\_actor)

window.show(scene)

\end{lstlisting}

### Eigenvector Centrality in Brain Network Graphs

\begin{lstlisting}

import numpy as np

import networkx as nx

# Generate a synthetic brain network graph (adjacency matrix)

num\_nodes = 50 # Number of brain regions

adj\_matrix = np.random.rand(num\_nodes, num\_nodes) # Random connectivity

adj\_matrix = (adj\_matrix + adj\_matrix.T) / 2 # Ensure symmetry

adj\_matrix[adj\_matrix < 0.7] = 0 # Threshold to create sparse connections

np.fill\_diagonal(adj\_matrix, 0) # Remove self-loops

# Create a graph from the adjacency matrix

brain\_graph = nx.from\_numpy\_array(adj\_matrix)

# Compute Eigenvector Centrality

eigenvector\_centrality = nx.eigenvector\_centrality(brain\_graph, max\_iter=1000, tol=1e-06)

# Display the centrality values for each node (brain region)

for node, centrality in eigenvector\_centrality.items():

print(f"Node {node}: Eigenvector Centrality = {centrality:.4f}")

\end{lstlisting}

### Functional Connectivity using Pearson Correlation

\begin{lstlisting}

import numpy as np

from scipy.stats import pearsonr

# Simulate fMRI time series data for multiple brain regions

num\_regions = 10 # Number of brain regions

time\_points = 100 # Number of time points

data = np.random.randn(num\_regions, time\_points) # Random fMRI data

# Initialize a matrix to store functional connectivity values

connectivity\_matrix = np.zeros((num\_regions, num\_regions))

# Compute Pearson correlation for each pair of regions

for i in range(num\_regions):

for j in range(num\_regions):

if i == j:

connectivity\_matrix[i, j] = 1.0 # Diagonal is self-correlation

else:

# Calculate Pearson correlation between regions i and j

connectivity\_matrix[i, j], \_ = pearsonr(data[i, :], data[j, :])

# Display the functional connectivity matrix

print("Functional Connectivity Matrix:")

print(connectivity\_matrix)

\end{lstlisting}

### Large-Scale Brain Simulations with The Virtual Brain (TVB)

\begin{lstlisting}

# Import necessary TVB modules

from tvb.simulator.lab import \*

from tvb.datatypes.connectivity import Connectivity

from tvb.simulator.models import ReducedWongWang

from tvb.simulator.integrators import HeunStochastic

from tvb.simulator.monitors import Raw

# Load a pre-defined connectivity matrix (e.g., default TVB connectivity)

conn = Connectivity.from\_file()

# Define the neural mass model (e.g., Reduced Wong-Wang model)

model = ReducedWongWang()

# Set up the stochastic integrator with a small noise level

integrator = HeunStochastic(dt=0.1, noise=noise.Additive(nsig=1e-4))

# Configure the monitor to record raw neural activity

monitor = Raw(period=1.0) # Record every 1ms

# Initialize the simulator with the defined components

sim = simulator.Simulator(

connectivity=conn,

model=model,

integrator=integrator,

monitors=[monitor]

).configure()

# Run the simulation for 1000ms

sim.run(simulation\_length=1000.0)

# Extract and print the recorded data

data, \_ = sim.history

print("Simulation data shape:", data.shape)

\end{lstlisting}

### Simulating Brain Waves and Neural Rhythms

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import solve\_ivp

# Define the neural mass model for simulating brain waves

def neural\_mass\_model(t, y, alpha, beta, gamma, I\_ext):

V, W = y # V: membrane potential, W: recovery variable

dVdt = V - (V\*\*3)/3 - W + I\_ext # FitzHugh-Nagumo model for neural dynamics

dWdt = alpha \* (V + beta - gamma \* W) # Recovery variable dynamics

return [dVdt, dWdt]

# Parameters for simulating alpha rhythms (8-12 Hz)

alpha = 0.08 # Recovery rate

beta = 0.7 # Membrane potential shift

gamma = 0.8 # Recovery variable scaling

I\_ext = 0.5 # External input current

# Initial conditions

y0 = [0.0, 0.0] # Initial membrane potential and recovery variable

# Time span for simulation

t\_span = (0, 100) # Simulate for 100 ms

t\_eval = np.linspace(t\_span[0], t\_span[1], 1000) # Time points for evaluation

# Solve the differential equations

sol = solve\_ivp(neural\_mass\_model, t\_span, y0, args=(alpha, beta, gamma, I\_ext), t\_eval=t\_eval)

# Plot the simulated brain waves

plt.figure(figsize=(10, 4))

plt.plot(sol.t, sol.y[0], label="Membrane Potential (V)")

plt.plot(sol.t, sol.y[1], label="Recovery Variable (W)", linestyle="--")

plt.title("Simulated Alpha Rhythms (8-12 Hz)")

plt.xlabel("Time (ms)")

plt.ylabel("Amplitude")

plt.legend()

plt.grid(True)

plt.show()

\end{lstlisting}

### Modeling Propagation Delays in Neural Networks

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Define neuron parameters

num\_neurons = 100 # Number of neurons in the network

time\_steps = 500 # Total simulation time steps

propagation\_delay = 5 # Fixed propagation delay between neurons

# Initialize neuron states and spike history

neuron\_states = np.zeros((num\_neurons, time\_steps))

spike\_history = np.zeros((num\_neurons, time\_steps))

# Define a simple spike generation function

def generate\_spikes(neuron\_id, time\_step):

if np.random.rand() < 0.02: # Random spike generation with 2% probability

return 1

return 0

# Simulate the network with propagation delays

for t in range(time\_steps):

for n in range(num\_neurons):

# Generate spikes for the current neuron

spike = generate\_spikes(n, t)

spike\_history[n, t] = spike

# Propagate spikes with delay

if spike and t + propagation\_delay < time\_steps:

neuron\_states[:, t + propagation\_delay] += 1 # Update downstream neurons

# Plot the spike raster and neuron states

plt.figure(figsize=(12, 6))

plt.subplot(2, 1, 1)

plt.title("Spike Raster Plot")

plt.imshow(spike\_history, aspect='auto', cmap='binary')

plt.xlabel("Time Steps")

plt.ylabel("Neuron ID")

plt.subplot(2, 1, 2)

plt.title("Neuron States Over Time")

plt.plot(np.sum(neuron\_states, axis=0))

plt.xlabel("Time Steps")

plt.ylabel("Total Active Neurons")

plt.tight\_layout()

plt.show()

\end{lstlisting}

### Neuromorphic Computing Simulation with Loihi

\begin{lstlisting}

import nxsdk

from nxsdk.api.n2a import Compiler

from nxsdk.api.n2a import N2A

# Initialize the Loihi board

board = nxsdk.board.Board()

# Define a simple spiking neural network

network = N2A.Network()

# Create input neurons

input\_neurons = network.createNeuronGroup(size=100, neuronType='excitatory')

# Create output neurons

output\_neurons = network.createNeuronGroup(size=50, neuronType='inhibitory')

# Define synaptic connections between input and output neurons

synapses = network.createSynapseGroup(

preGroup=input\_neurons,

postGroup=output\_neurons,

weight=0.5,

delay=1

)

# Compile the network for Loihi

compiler = Compiler()

compiled\_network = compiler.compile(network)

# Load the compiled network onto the Loihi board

board.load(compiled\_network)

# Run the simulation for 1000 time steps

board.run(1000)

# Retrieve and analyze the output spikes

output\_spikes = output\_neurons.getSpikes()

print("Output spikes:", output\_spikes)

# Reset the board for future simulations

board.reset()

\end{lstlisting}

## Computational Tools and Visualization

### Plotting Neuronal Spike Raster Plots

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Generate synthetic spike data for 10 neurons over 100 time steps

np.random.seed(42)

num\_neurons = 10

time\_steps = 100

spike\_data = np.random.rand(num\_neurons, time\_steps) < 0.1 # 10% spike probability

# Plotting the raster plot

plt.figure(figsize=(10, 6))

for neuron in range(num\_neurons):

spike\_times = np.where(spike\_data[neuron])[0] # Extract spike times

plt.vlines(spike\_times, neuron + 0.5, neuron + 1.5, color='black', linewidth=1)

# Customize the plot

plt.title('Neuronal Spike Raster Plot')

plt.xlabel('Time Steps')

plt.ylabel('Neuron ID')

plt.yticks(np.arange(1, num\_neurons + 1))

plt.grid(True, axis='y', linestyle='--', alpha=0.7)

plt.show()

\end{lstlisting}

### Visualizing Phase Portraits in Neural Dynamics

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import odeint

# Define the neural dynamics model (e.g., Wilson-Cowan model)

def neural\_dynamics(state, t, alpha, beta, gamma, delta):

x, y = state

dxdt = -alpha \* x + np.tanh(beta \* x - gamma \* y + delta)

dydt = -alpha \* y + np.tanh(beta \* y - gamma \* x + delta)

return [dxdt, dydt]

# Parameters for the model

alpha, beta, gamma, delta = 1.0, 2.0, 1.5, 0.5

# Time points for integration

t = np.linspace(0, 10, 100)

# Generate a grid of initial conditions

x = np.linspace(-2, 2, 20)

y = np.linspace(-2, 2, 20)

X, Y = np.meshgrid(x, y)

# Plot phase portrait

plt.figure(figsize=(8, 6))

for x0 in x:

for y0 in y:

# Integrate the system for each initial condition

states = odeint(neural\_dynamics, [x0, y0], t, args=(alpha, beta, gamma, delta))

plt.plot(states[:, 0], states[:, 1], 'b-', alpha=0.5) # Trajectories

plt.plot(states[0, 0], states[0, 1], 'ro') # Starting points

# Add labels and title

plt.xlabel('Neural Activity (x)')

plt.ylabel('Neural Activity (y)')

plt.title('Phase Portrait of Neural Dynamics')

plt.grid(True)

plt.show()

\end{lstlisting}

### Time-Frequency Analysis of Neural Signals

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import spectrogram

# Simulate a neural signal (e.g., EEG data)

fs = 1000 # Sampling frequency (Hz)

t = np.linspace(0, 2, 2 \* fs, endpoint=False) # Time vector (2 seconds)

signal = np.sin(2 \* np.pi \* 10 \* t) + 0.5 \* np.sin(2 \* np.pi \* 50 \* t) # 10 Hz + 50 Hz signal

# Compute the spectrogram

f, t\_spec, Sxx = spectrogram(signal, fs, nperseg=256, noverlap=128)

# Plot the spectrogram

plt.figure(figsize=(10, 6))

plt.pcolormesh(t\_spec, f, 10 \* np.log10(Sxx), shading='gouraud')

plt.colorbar(label='Power Spectral Density (dB/Hz)')

plt.ylabel('Frequency [Hz]')

plt.xlabel('Time [sec]')

plt.title('Time-Frequency Analysis of Neural Signal')

plt.show()

\end{lstlisting}

### Power Spectral Density Analysis of EEG Data

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import welch

# Simulated EEG data (replace with actual data)

fs = 256 # Sampling frequency in Hz

t = np.linspace(0, 10, 10 \* fs, endpoint=False) # Time vector

eeg\_data = np.sin(2 \* np.pi \* 10 \* t) + 0.5 \* np.sin(2 \* np.pi \* 20 \* t) # Example EEG signal

# Compute Power Spectral Density (PSD) using Welch's method

frequencies, psd = welch(eeg\_data, fs, nperseg=1024)

# Plot the PSD

plt.figure(figsize=(10, 6))

plt.plot(frequencies, psd, label='PSD')

plt.title('Power Spectral Density of EEG Data')

plt.xlabel('Frequency (Hz)')

plt.ylabel('Power/Frequency (dB/Hz)')

plt.grid(True)

plt.legend()

plt.show()

\end{lstlisting}

### Cross-Correlation Analysis of Neural Spiking Activity

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import correlate

# Simulate two spike trains for demonstration

np.random.seed(42)

spike\_train1 = np.random.binomial(1, 0.1, 1000) # Binary spike train 1

spike\_train2 = np.random.binomial(1, 0.1, 1000) # Binary spike train 2

# Compute cross-correlation

cross\_corr = correlate(spike\_train1, spike\_train2, mode='same')

# Normalize by the number of spikes in the reference train

cross\_corr = cross\_corr / np.sum(spike\_train1)

# Time lags for plotting

lags = np.arange(-len(spike\_train1)//2, len(spike\_train1)//2)

# Plot the cross-correlation

plt.figure(figsize=(10, 5))

plt.plot(lags, cross\_corr, label='Cross-Correlation')

plt.axvline(0, color='red', linestyle='--', label='Zero Lag')

plt.xlabel('Time Lag (ms)')

plt.ylabel('Cross-Correlation')

plt.title('Cross-Correlation Analysis of Neural Spiking Activity')

plt.legend()

plt.grid(True)

plt.show()

\end{lstlisting}

### Wavelet Transform for Multi-Scale Signal Analysis

\begin{lstlisting}

import numpy as np

import pywt

import matplotlib.pyplot as plt

# Generate a sample signal (e.g., EEG-like data)

fs = 1000 # Sampling frequency

t = np.linspace(0, 1, fs, endpoint=False)

signal = np.sin(2 \* np.pi \* 50 \* t) + 0.5 \* np.sin(2 \* np.pi \* 120 \* t)

# Perform Continuous Wavelet Transform (CWT)

scales = np.arange(1, 128) # Define scales for multi-resolution analysis

coefficients, frequencies = pywt.cwt(signal, scales, 'cmor', sampling\_period=1/fs)

# Visualize the wavelet transform

plt.figure(figsize=(10, 6))

plt.imshow(np.abs(coefficients), extent=[0, 1, 1, 128], cmap='jet', aspect='auto',

vmax=abs(coefficients).max(), vmin=0)

plt.colorbar(label='Magnitude')

plt.title('Wavelet Transform for Multi-Scale Signal Analysis')

plt.xlabel('Time [s]')

plt.ylabel('Scale')

plt.show()

\end{lstlisting}

### Generating Synthetic Neural Data for Model Testing

\begin{lstlisting}

import numpy as np

import matplotlib.pyplot as plt

# Parameters for synthetic neural data

n\_neurons = 50 # Number of neurons

n\_timepoints = 1000 # Number of time points

baseline\_rate = 5 # Baseline firing rate (Hz)

stimulus\_rate = 20 # Stimulus-induced firing rate (Hz)

stimulus\_start = 300 # Stimulus onset time

stimulus\_end = 700 # Stimulus offset time

# Generate baseline activity

neural\_data = np.random.poisson(baseline\_rate, (n\_neurons, n\_timepoints))

# Add stimulus-induced activity

neural\_data[:, stimulus\_start:stimulus\_end] = np.random.poisson(stimulus\_rate, (n\_neurons, stimulus\_end - stimulus\_start))

# Add Gaussian noise to simulate neural variability

neural\_data += np.random.normal(0, 1, (n\_neurons, n\_timepoints))

# Visualize the synthetic neural data

plt.figure(figsize=(10, 6))

plt.imshow(neural\_data, aspect='auto', cmap='viridis', origin='lower')

plt.colorbar(label='Firing Rate (Hz)')

plt.xlabel('Time Points')

plt.ylabel('Neurons')

plt.title('Synthetic Neural Data')

plt.show()

\end{lstlisting}

### Running Neural Simulations in Google Colab

\begin{lstlisting}

# Import necessary libraries

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras import layers

# Define a simple neural network model

def create\_neural\_model(input\_shape):

model = tf.keras.Sequential([

layers.Dense(64, activation='relu', input\_shape=input\_shape),

layers.Dense(64, activation='relu'),

layers.Dense(1, activation='sigmoid')

])

return model

# Generate synthetic data for simulation

def generate\_data(num\_samples=1000, input\_dim=10):

X = np.random.randn(num\_samples, input\_dim)

y = np.random.randint(2, size=(num\_samples, 1))

return X, y

# Train the neural network

def train\_model(model, X\_train, y\_train, epochs=10, batch\_size=32):

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

history = model.fit(X\_train, y\_train, epochs=epochs, batch\_size=batch\_size, verbose=1)

return history

# Visualize training results

def plot\_training\_history(history):

plt.plot(history.history['accuracy'], label='accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()

plt.show()

# Main simulation function

def run\_neural\_simulation():

# Generate data

X, y = generate\_data()

# Create and train the model

model = create\_neural\_model((X.shape[1],))

history = train\_model(model, X, y)

# Visualize training results

plot\_training\_history(history)

# Execute the simulation

run\_neural\_simulation()

\end{lstlisting}

### Real-Time Data Acquisition from Neural Interfaces

\begin{lstlisting}

import numpy as np

import time

import serial

# Initialize serial connection to neural interface device

ser = serial.Serial('COM3', 115200, timeout=1) # Adjust port and baud rate as needed

def acquire\_neural\_data(duration):

"""

Acquire real-time neural data for a specified duration.

Parameters:

duration (float): Duration of data acquisition in seconds.

Returns:

np.ndarray: Array of acquired neural data.

"""

data = []

start\_time = time.time()

while time.time() - start\_time < duration:

if ser.in\_waiting > 0:

# Read and decode data from the serial port

raw\_data = ser.readline().decode('utf-8').strip()

if raw\_data:

# Convert the data to float and append to the list

data.append(float(raw\_data))

return np.array(data)

# Example usage: Acquire data for 10 seconds

neural\_data = acquire\_neural\_data(10)

print("Acquired Neural Data:", neural\_data)

# Close the serial connection

ser.close()

\end{lstlisting}

### Using GPU Acceleration for Neural Simulations

\begin{lstlisting}

import numpy as np

import cupy as cp # CuPy for GPU-accelerated computations

import matplotlib.pyplot as plt

# Define a simple neural network layer with GPU acceleration

class GPUNeuralLayer:

def \_\_init\_\_(self, input\_size, output\_size):

self.weights = cp.random.randn(output\_size, input\_size) \* 0.01 # Initialize weights on GPU

self.bias = cp.zeros((output\_size, 1)) # Initialize bias on GPU

def forward(self, x):

return cp.dot(self.weights, x) + self.bias # Perform matrix multiplication on GPU

# Simulate neural activity using GPU

def simulate\_neural\_activity(layer, input\_data):

return layer.forward(input\_data)

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

input\_size = 100 # Input dimension

output\_size = 50 # Output dimension

layer = GPUNeuralLayer(input\_size, output\_size) # Create a GPU-accelerated layer

# Generate random input data on GPU

input\_data = cp.random.randn(input\_size, 1)

# Simulate neural activity

output = simulate\_neural\_activity(layer, input\_data)

# Transfer results back to CPU for visualization

output\_cpu = cp.asnumpy(output)

plt.plot(output\_cpu)

plt.title("Simulated Neural Activity")

plt.show()

\end{lstlisting}