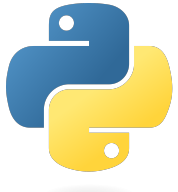
Scientific Python



Indra Ghosh

Computational and Cognitive Neuroscience Summer School

Suzhou, China

June 28 - July 19, 2026

University College Dublin

Email: indra.ghosh@ucd.ie

About me

i. Call me “Indra”: one of your tutors at CCNS 2026
ii. Postdoctoral fellow in computational neuroscience @ University College Dublin

iii. B.Sc. Physics
iv. M.Sc. Physics
v. Ph.D. Applied Mathematics

vi. Visiting fellow in neuroscience @ Trinity College Dublin
vii. Write softwares in and
viii. Research in dynamical systems, network science, and computational science
ix. Find me at https://indrag49.github.io/

Overview

- Introduction
-  basics
- NumPy 
- Numba 
- SciPy  + matplotlib 
- ODEs
- Leaky neural activity model
- LIF model
- Wilson-Cowan rate model
- Signal processing
- Statistics
- scikit-learn  : machine learning
- AI assisted coding

Introduction

1. Conceived in the late 1980's by a Dutch programmer.

2. It was named after the BBC TV show "Monty Python's Flying Circus".

3. Support for Python 2.0 has stopped. We will use Python >=3.10.

4. High-level, general purpose: emphasises code readability.

Guido Van Rossum

Logo, 2006 - present

Source: https://en.wikipedia.org/wiki/History_of_Python

basics

Lists:

a. Storing multiple values in Python: lists.

b. You can also access elements.

c. Indexing starts from 0.

```
[4]: spike_counts = [3, 5, 2, 8, 4]

     print(spike_counts)
```

```
[3, 5, 2, 8, 4]
```

```
[3]: spike_counts[0]    # first element
     spike_counts[1]    # second element
     spike_counts[-1]   # last element
```

```
[3]: 4
```

basics

Variables:

a. Variables store values.

b. Should have meaningful names.

c. Avoid “a=10”, “b=11.2”, until absolutely necessary.

```
[5]: duration = 10
     spike_count = 50

     firing_rate = spike_count / duration
     print(firing_rate)
```

```
5.0
```

basics

Basic data types:

a. Four basic data types:

b. int: whole number, float: decimal,
str: string, bool: True or False

[6]:
```
neuron_id = 101          # integer
firing_rate = 5.3        # float
brain_region = "VISp"    # string
is_active = True         # boolean
```

[8]:
```
print(type(neuron_id))
print(type(firing_rate))
print(type(brain_region))
print(type(is_active))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

basics

Indentation:

a. Instead of using {} to group code Python uses indentation.

b. Use 4 whitespaces (PEP 8)

c. Same spacing = same logical block.

d. Use in if-else statements, loops, functions, etc.

```
[9]: x = 5

if x > 0:
    print("x is positive")
    print("This line is inside the if statement")

print("This line is outside the if statement")
```

```
x is positive
This line is inside the if statement
This line is outside the if statement
```

basics

Loops:

a. Used for repeating an operation.

b. Loop within a loop: “nested” loop.

[10]:
```
firing_rates = []

for count in spike_counts:
    rate = count / duration
    firing_rates.append(rate)

print(firing_rates)
```

```
[0.3, 0.5, 0.2, 0.8, 0.4]
```

[11]:
```
neurons = [0, 1, 2]
trials = [0, 1, 2, 3]

for neuron in neurons:
    for trial in trials:
        print("Neuron:", neuron, "Trial:", trial)
```

```
Neuron: 0 Trial: 0
Neuron: 0 Trial: 1
Neuron: 0 Trial: 2
Neuron: 0 Trial: 3
Neuron: 1 Trial: 0
Neuron: 1 Trial: 1
Neuron: 1 Trial: 2
Neuron: 1 Trial: 3
Neuron: 2 Trial: 0
Neuron: 2 Trial: 1
Neuron: 2 Trial: 2
Neuron: 2 Trial: 3
```

basics

Functions:

a. Reusable piece takes inputs and returns an output.

b. Stops code from becoming long and repetitive.

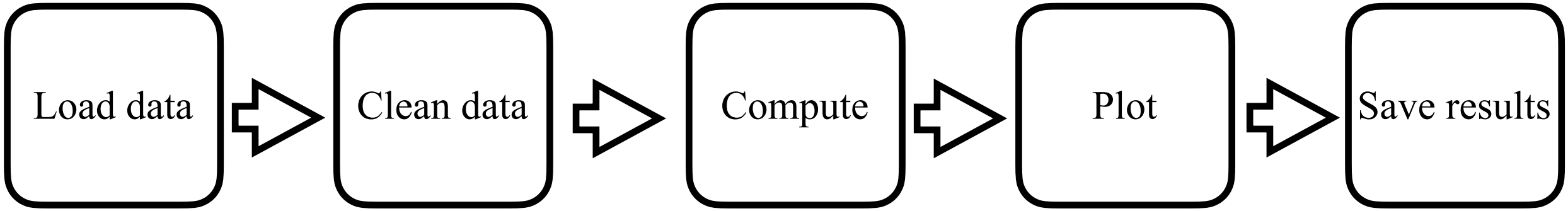
[16]:
```
def compute_firing_rate(spike_count, duration):
    rate = spike_count / duration
    return rate

compute_firing_rate(50, 10)
```

[16]:
```
5.0
```

[17]:
```
spike_counts = [3, 5, 2, 8, 4]
duration = 10

for count in spike_counts:
    rate = compute_firing_rate(count, duration)
    print(rate)
```

```
0.3
0.5
0.2
0.8
0.4
```

modular programming (advanced)



Each box can become one function

modular programming (advanced)

NOT recommended

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

spike_counts = [3, 5, 2, 8, 4]
duration = 10

rates = []
for count in spike_counts:
    rates.append(count / duration)

mean_rate = sum(rates) / len(rates)

plt.plot(rates)
plt.axhline(mean_rate, linestyle="--")
plt.xlabel("Neuron")
plt.ylabel("Firing rate")
plt.title("Firing rates across neurons")
plt.show()
```

Recommended!

```python
import matplotlib.pyplot as plt


def compute_firing_rates(spike_counts, duration):
    firing_rates = []

    for count in spike_counts:
        firing_rates.append(count / duration)

    return firing_rates


def compute_mean(values):
    return sum(values) / len(values)


def plot_firing_rates(firing_rates, mean_rate):
    plt.plot(firing_rates)
    plt.axhline(mean_rate, linestyle="--")
    plt.xlabel("Neuron")
    plt.ylabel("Firing rate")
    plt.title("Firing rates across neurons")
    plt.show()


spike_counts = [3, 5, 2, 8, 4]
duration = 10

firing_rates = compute_firing_rates(spike_counts, duration)
mean_rate = compute_mean(firing_rates)

plot_firing_rates(firing_rates, mean_rate)
```

basics

Dictionaries:

a. Useful for metadata

b. A “dictionary” is a way to store data using labels instead of positions.

[21]:
```
neuron = {
    "unit_id": 101,
    "brain_region": "VISp",
    "firing_rate": 5.3
}

print(neuron["unit_id"])
print(neuron["brain_region"])
print(neuron["firing_rate"])
```

```
101
VISp
5.3
```

basics

Importing packages:

a. You need to know how to import external dependencies

b. Know “pip install <package name>”

```
[22]: import numpy as np
      import pandas as pd
      import matplotlib.pyplot as plt
```

NumPy

1. NumPy (Numerical Python): widely used in science and engineering.

2. Gives us a fast object called an “array”, which is designed for working with numerical data.

3. For more information look at https://numpy.org

4. Pip install it and import it:

```
pip install numpy
```

```
import numpy as np
```

NumPy

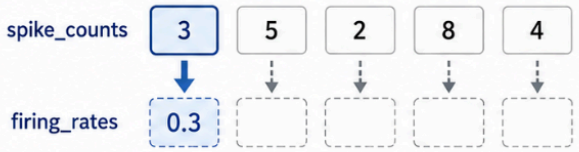
Python list vs NumPy array

list	3	5	2	8	4	* 2 → repeats list
NumPy array	3	5	2	8	4	* 2 → element-wise multiplication
result	6	10	4	16	8	

Loop approach

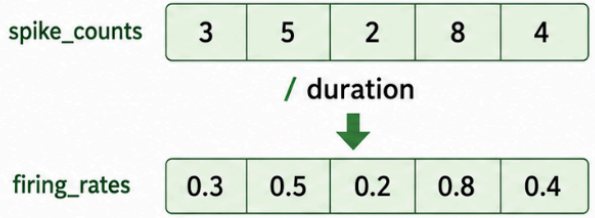
```
firing_rates = []
for count in spike_counts:
    rate = count / duration
    firing_rates.append(rate)
```

More code, repeated step-by-step



Vectorised approach

```
firing_rates = spike_counts / duration
```

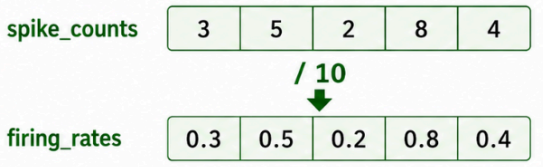


*One operation is applied to every element **at once***

Broadcasting means NumPy can combine arrays of different shapes when the operation makes sense

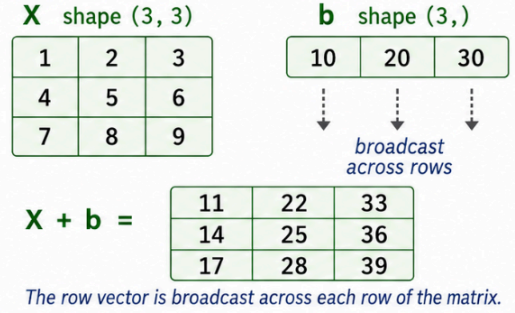
1 Simple example: scalar broadcasting

```
spike_counts = np.array([3, 5, 2, 8, 4])
firing_rates = spike_counts / 10
```

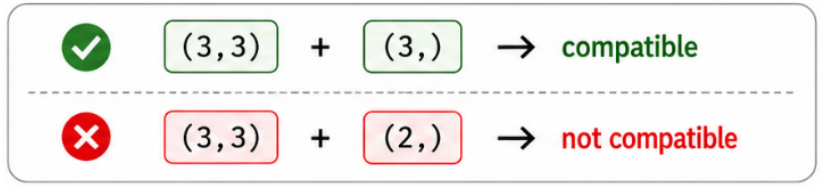


A single value is applied to every element.

2 Matrix + row vector



The row vector is broadcast across each row of the matrix.



NumPy

CSH Asia

Scientific simulations often need random numbers

Distribution	Mathematical notation	NumPy function	Scientific use
Uniform	X~U(a,b)	`np.rng.uniform(a, b, size)`	Random values in an interval
Normal	X~N(μ,σ2)	`np.rng.normal(mu, sigma, size)`	Noise, measurement error
Bernoulli	X~Bernoulli(p)	`np.rng.random(size) < p`	Yes/no events, spikes in bins
Binomial	X~Binomial(n,p)	`np.rng.binomial(n, p, size)`	Number of successes
Poisson	X~Poisson(λ)	`np.rng.poisson(lam, size)`	Spike counts, event counts

Numba

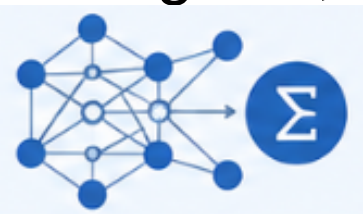
1. Numba is a just-in-time (JIT) compiler for numerical functions 
2. It uses the LLVM compiler project to generate machine code from Python syntax. 

3. Works well with arrays, number, nested loops, if statements, numpy functions
4. For more information look at https://numba.pydata.org 
5. install it (if not already) : `pip install numba` `import numba`
6. First time compiles. The second time, it runs the already compiled version and is faster.

SciPy + matplotlib

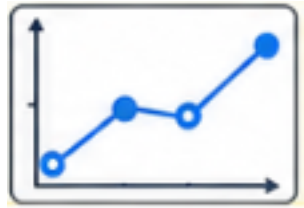
1. “SciPy” gives us a collection of scientific algorithms, very useful in neuroscience.

2. It is useful for solving neural models, processing neural signals, and analyzing experimental data statistically.



3. We use “matplotlib” for visualisation with Python.

4. Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python: creates publication ready plots.



5. Use:

```
import matplotlib.pyplot as plt
import scipy
```

6. More info at https://scipy.org/ and https://matplotlib.org/

Ordinary Differential Equations (ODEs)

1. Many neuroscience models are based on how some properties change over time:
   (a) membrane voltage, (b) synaptic current,
   (c) firing rate, (d) population activity,
   (e) calcium concentration
2. We model this using “ODEs” (Ordinary Differential Equations)
3. Mathematically given by $\frac{dx}{dt} = f(x, t)$
4. In Scipy we solve using: ‘solve_ivp()’ function from the ‘integrate’ suite

```
from scipy.integrate import solve_ivp
```

Ordinary Differential Equations (ODEs)

```
sol = solve_ivp(
    fun=model,
    t_span=[t_start, t_end],
    y0=initial_condition,
    t_eval=time_points
)
```

Parameters

(a) fun: the right-hand side of the ODE, i.e, f()
(b) t_span: start and end time
(c) y0: initial condition
(d) t_eval: time points where we want the solution

Example 1: Leaky neural activity model

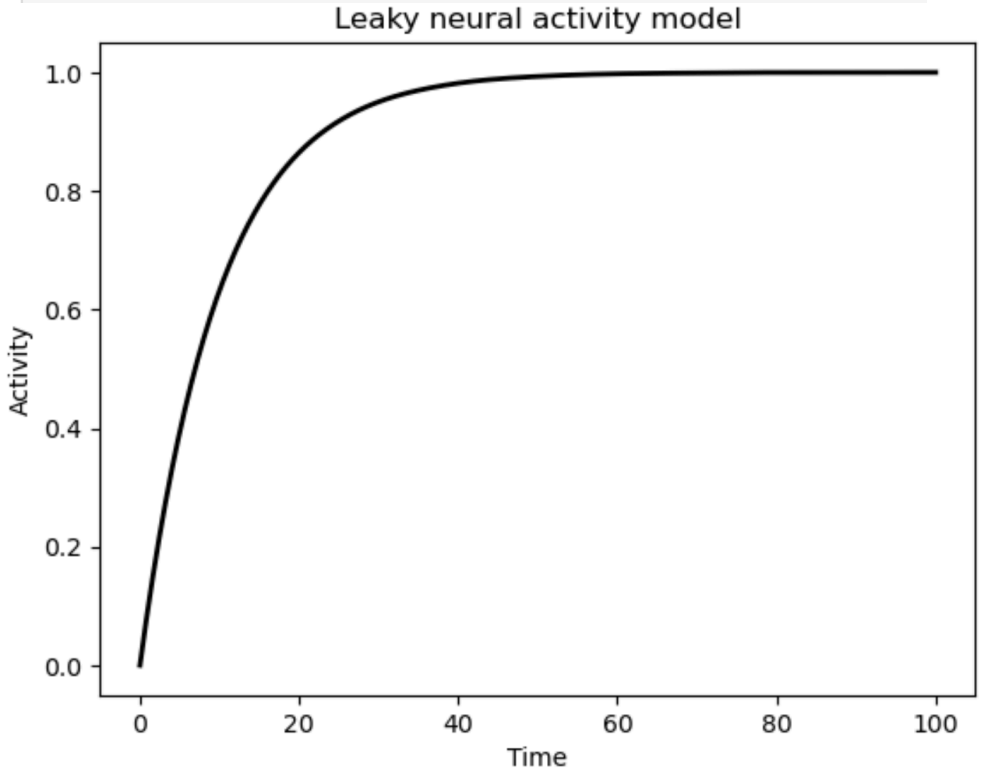
Model: $\tau \frac{dx}{dt} = -x + I,$

$x(t)$: neural activity
I: input
τ: time constant

The activity reaches the input level I. Parameter τ controls how fast that happens!

```python
def leaky_activity(t, x, tau, I):
    dxdt = (-x + I) / tau
    return dxdt


tau = 10.0
I = 1.0
x0 = [0.0]

t_eval = np.linspace(0, 100, 500)

sol = solve_ivp(
    leaky_activity,
    t_span=[0, 100],
    y0=x0,
    t_eval=t_eval,
    args=(tau, I)
)
```

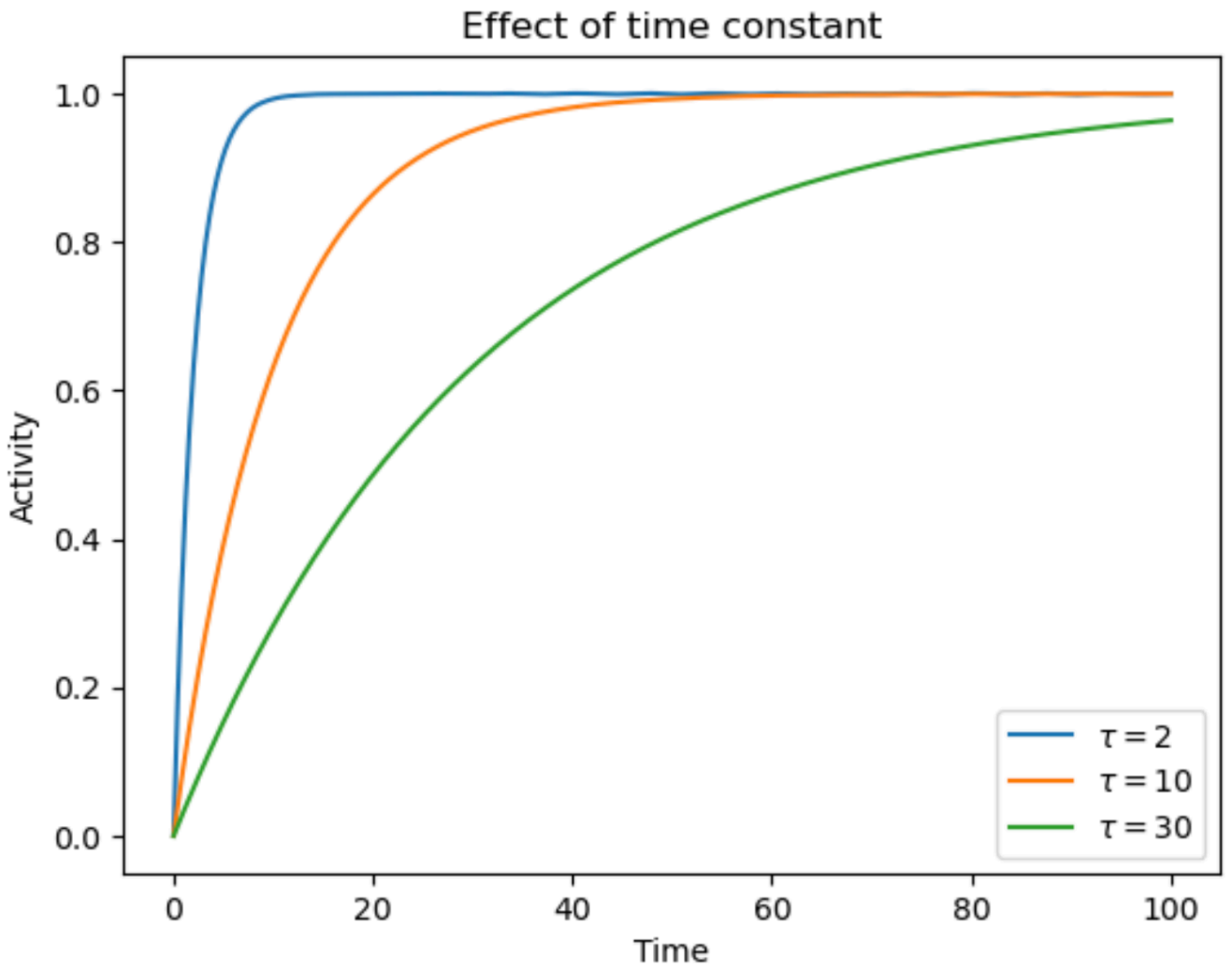
```python
plt.plot(sol.t, sol.y[0], 'k-', lw=2)
plt.xlabel("Time")
plt.ylabel("Activity")
plt.title("Leaky neural activity model")
plt.show()
```



Example 1: Leaky neural activity model

What happens when τ is small or large?

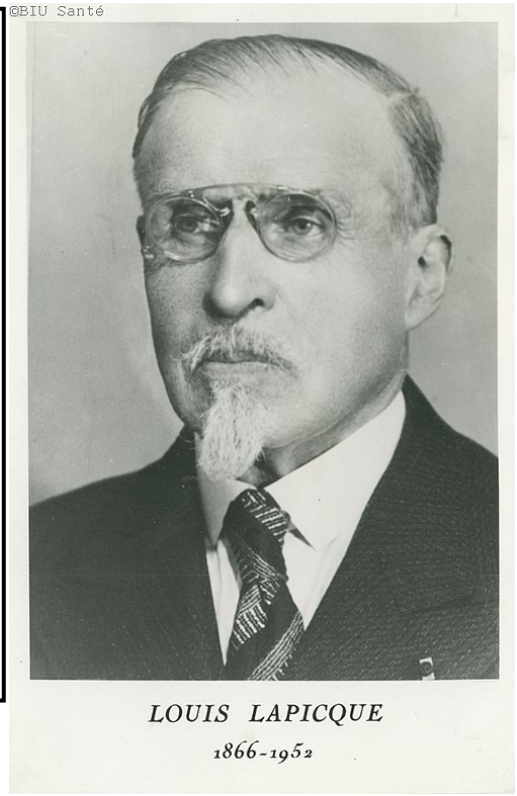
```
taus = [2, 10, 30]
I = 1.0
x0 = [0.0]

t_eval = np.linspace(0, 100, 500)

for tau in taus:
    sol = solve_ivp(
        leaky_activity,
        [0, 100],
        x0,
        t_eval=t_eval,
        args=(tau, I)
    )
    plt.plot(sol.t, sol.y[0], label=rf"$\tau = {tau}$")

plt.xlabel("Time")
plt.ylabel("Activity")
plt.title("Effect of time constant")
plt.legend()
```



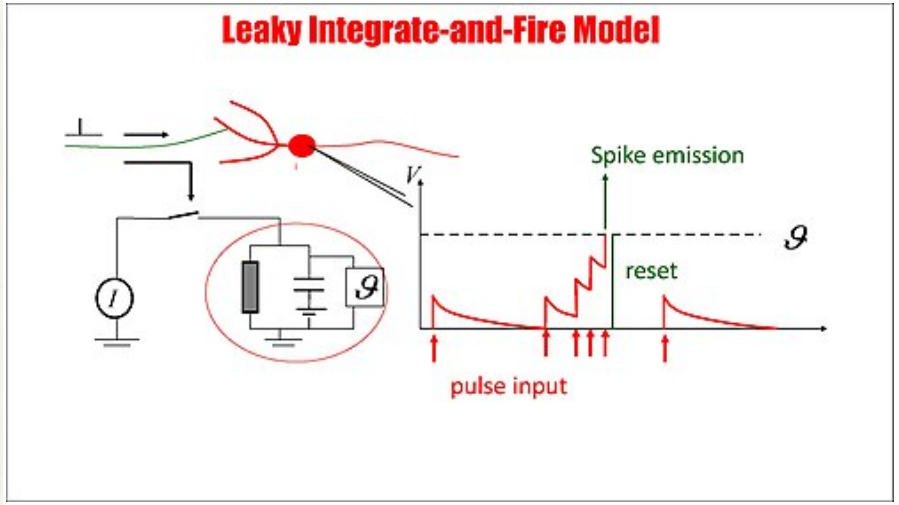
Ex 2: Leaky Integrate and Fire model

Model: $\tau_m \frac{dV}{dt} = -(V - V_{\text{rest}}) + RI,$

$V(t)$: membrane voltage
V_{rest}: resting potential
R: membrane resistance
I : input current
τ_m: membrane time constant



Source: https://en.wikipedia.org/wiki/Louis_Lapicque

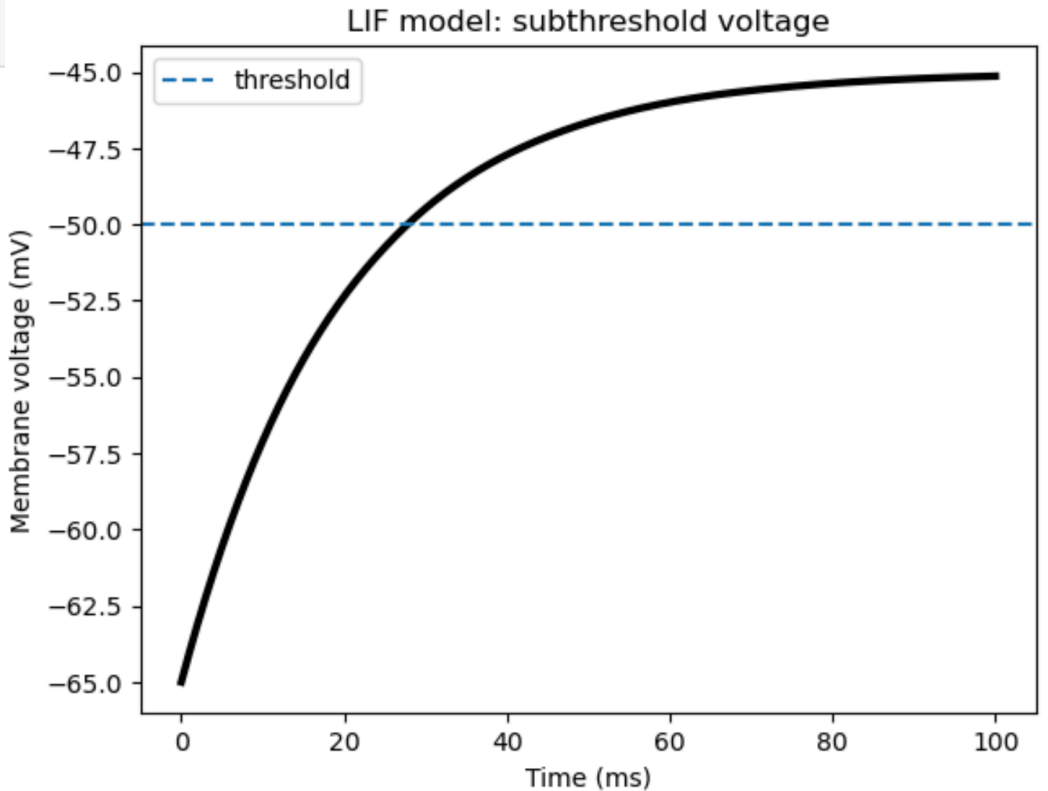


Source: https://en.wikipedia.org/wiki/Biological_neuron_model

Ex 2: Leaky Integrate and Fire model

```python
def lif_subthreshold(t, V, tau_m, V_rest, R, I):
    dVdt = (-(V[0] - V_rest) + R * I) / tau_m
    return [dVdt]


tau_m = 20.0       # ms
V_rest = -65.0     # mV
R = 10.0
I = 2.0

V0 = [-65.0]
t_eval = np.linspace(0, 100, 1000)

sol = solve_ivp(
    lif_subthreshold,
    [0, 100],
    V0,
    t_eval=t_eval,
    args=(tau_m, V_rest, R, I)
)
```

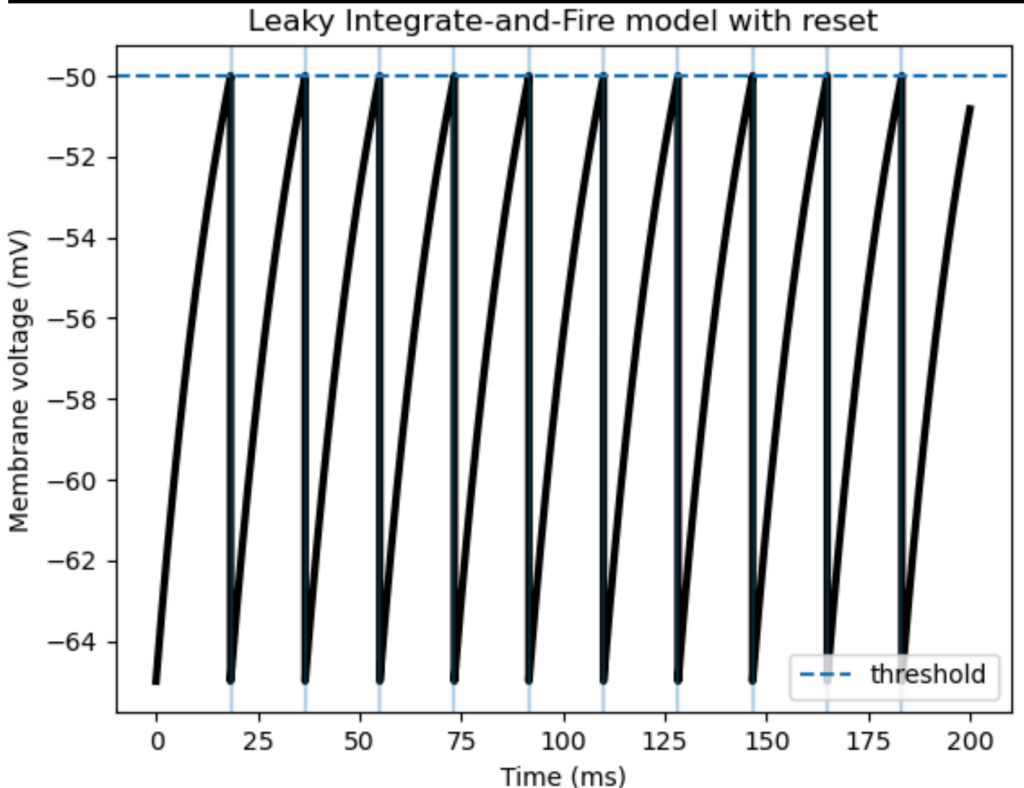
```python
plt.plot(sol.t, sol.y[0], 'k-', lw=3)
plt.axhline(-50, linestyle="--", label="threshold")
plt.xlabel("Time (ms)")
plt.ylabel("Membrane voltage (mV)")
plt.title("LIF model: subthreshold voltage")
plt.legend()
```



Ex 3: LIF model with reset

A bit complex: event based dynamics

If $V \geq V_{\text{threshold}}$ then $V \leftarrow V_{\text{reset}}$




```python
def lif_rhs(t, V, tau_m, V_rest, R, I):
    dVdt = (-(V[0] - V_rest) + R * I) / tau_m
    return [dVdt]


def threshold_event(t, V, tau_m, V_rest, R, I):
    V_th = -50.0
    return V[0] - V_th


threshold_event.terminal = True
threshold_event.direction = 1


tau_m = 20.0
V_rest = -65.0
V_reset = -65.0
V_th = -50.0
R = 10.0
I = 2.5

t_start = 0.0
t_end = 200.0

current_t = t_start
current_V = [V_reset]

all_t = []
all_V = []
spike_times = []
```

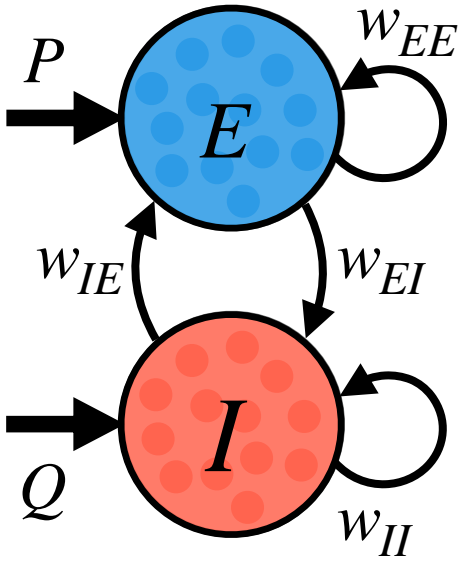
```python
while current_t < t_end:
    sol = solve_ivp(
        lif_rhs,
        [current_t, t_end],
        current_V,
        max_step=0.1,
        events=threshold_event,
        args=(tau_m, V_rest, R, I)
    )

    all_t.extend(sol.t)
    all_V.extend(sol.y[0])

    if sol.t_events[0].size > 0:
        spike_time = sol.t_events[0][0]
        spike_times.append(spike_time)

        current_t = spike_time
        current_V = [V_reset]

        all_t.append(current_t)
        all_V.append(V_reset)
    else:
        break
```

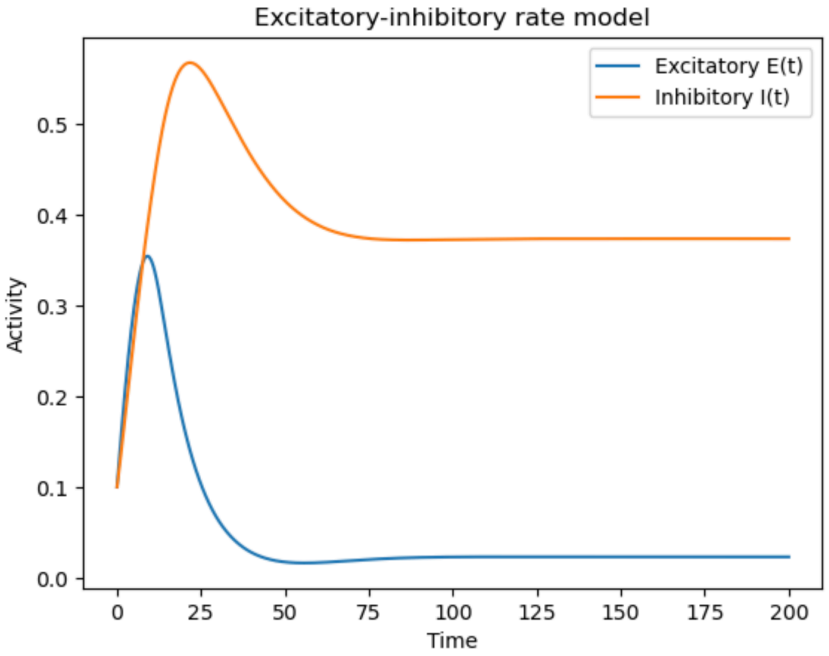
Ex 4: Wilson-Cowan rate model

Model:

$$\tau_E \frac{dE}{dt} = -E + \sigma(w_{EE}E - w_{EI}I + P),$$

$$\tau_I \frac{dI}{dt} = -I + \sigma(w_{IE}E - w_{II}I + Q),$$





here $E(t), I(t)$: activities of the excitatory and inhibitory populations,
τ_E, τ_I are the time constants,
w is the connection weight, with P, Q as the external inputs,
$\sigma()$ is the sigmoid activation function

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

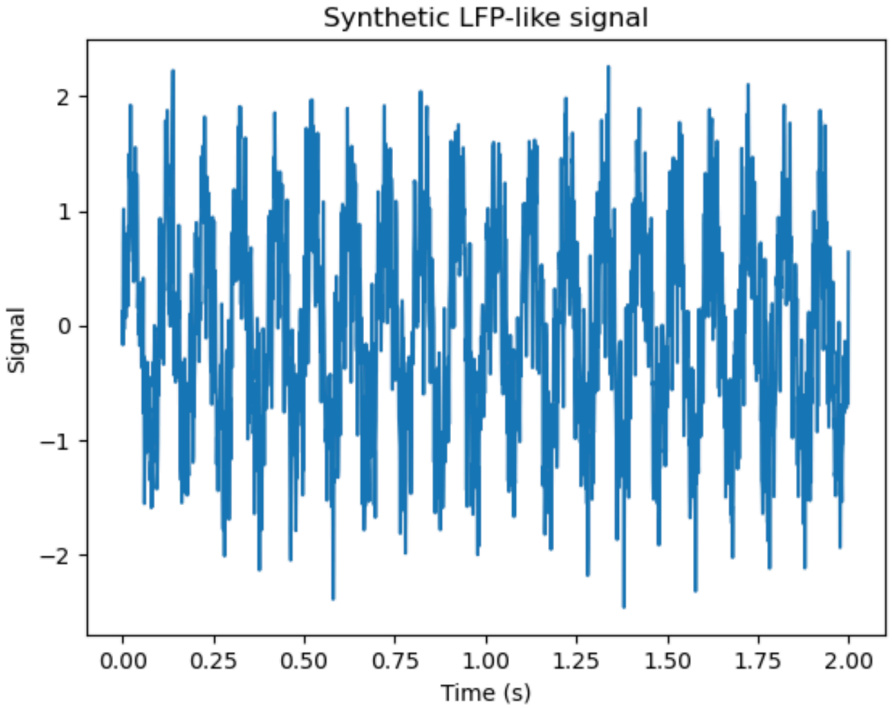
Signal processing with ‘scipy.signal’

Lets create a Local Field Potential like signal with 10 Hz activity, 60 Hz activity, and noise:

$$x(t) = \sin(2\pi 10t) + 0.5\sin(2\pi 60t) + \epsilon(t)$$

We need:

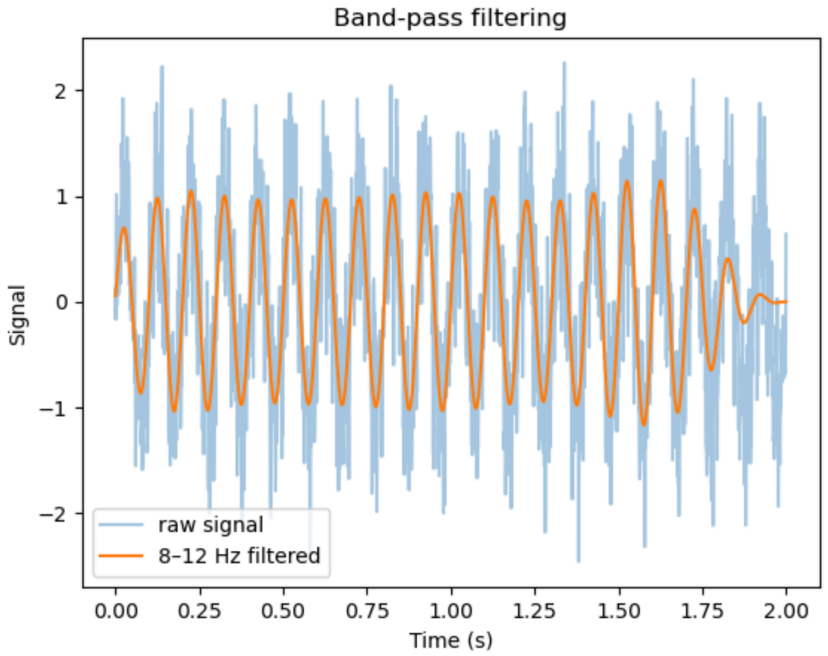
```
from scipy import signal
```



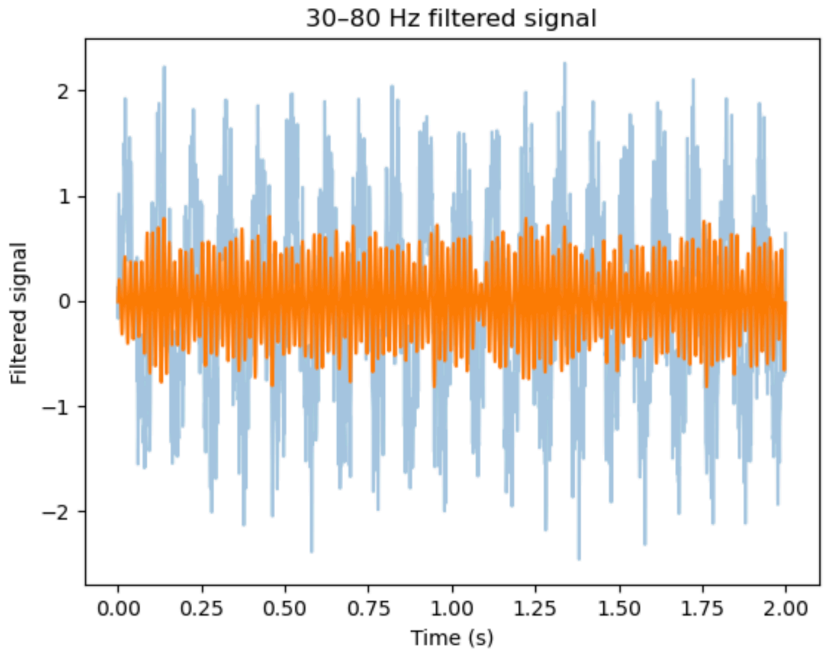

```
rng = np.random.default_rng(42)

fs = 1000
duration = 2.0

t = np.linspace(0, duration, int(fs * duration), endpoint=False)

x = (
    np.sin(2 * np.pi * 10 * t)
    + 0.5 * np.sin(2 * np.pi * 60 * t)
    + rng.normal(0, 0.4, size=len(t))
)

plt.plot(t, x)
plt.xlabel("Time (s)")
plt.ylabel("Signal")
plt.title("Synthetic LFP-like signal")
```

Band pass filtering + peak detection

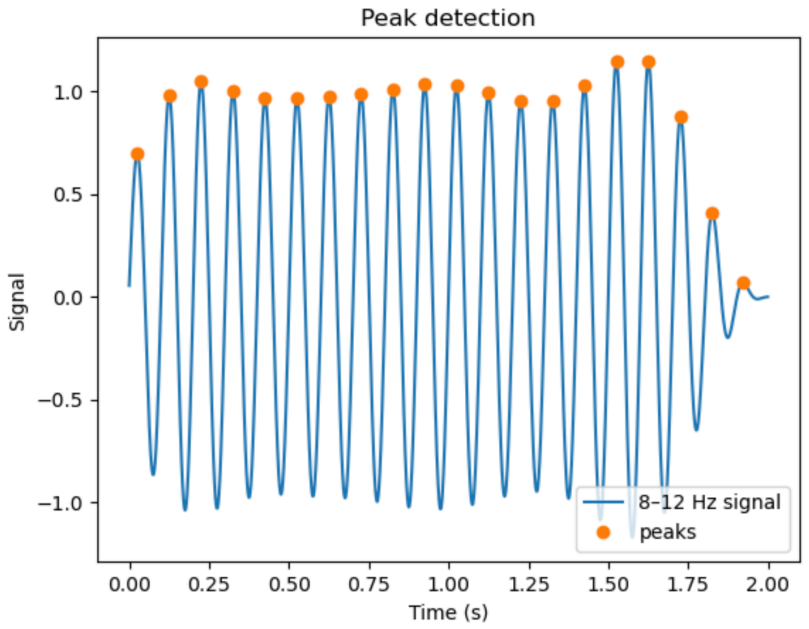
α band: 8-12 Hz



γ band: 30-80 Hz



Peak detection



Statistics with ‘scipy.stats’

1. Applicable in neuroscience because neural data is mostly variable, noisy, and uncertain.

2. Statistics helps us summarise neural data using concepts of ‘mean’, ‘standard deviation’, ‘variance’, ‘median’, ‘confidence interval’, etc.

3. Also helps us compare neural activity: using a paired ’t-test’ or ‘p-value’.

4. Helps in normalise neural data because firing-rate scales can be different.

5. We need:

```
from scipy import stats
```

t-test

1. A 't-test' compares the means of two groups while also considering how variable the data are.

t-statistic	Meaning
Close to 0	The groups are very similar
Positive	The first group mean is larger than the second group mean
Negative	The first group mean is smaller than the second group mean
Large positive or negative	The groups are more different relative to variability

```
result = stats.ttest_ind(baseline_rates, stimulus_rates)

print("t statistic:", result.statistic)
print("p-value:", result.pvalue)
```

```
t statistic: -11.021022700054376
p-value: 7.8553402505156905e-25
```

William Sealy Gosset

Source: https://en.wikipedia.org/wiki/Student%27s_t-test

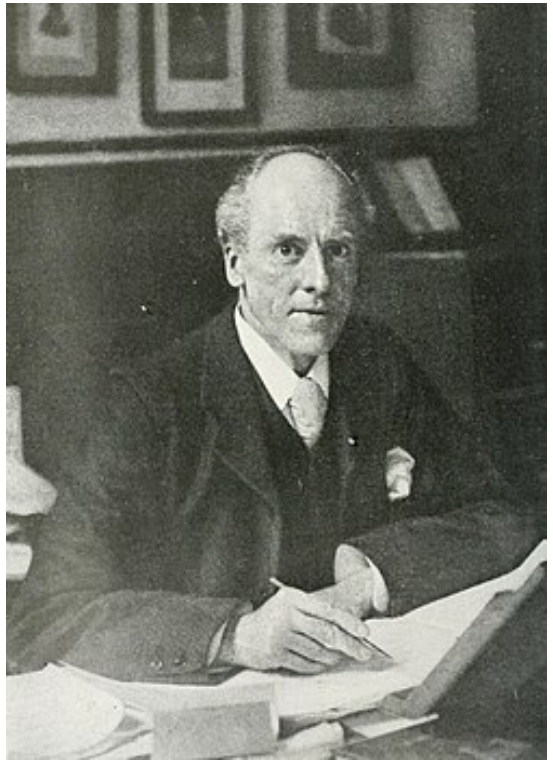
p-value

1. The probability of obtaining test results at least as extreme as the result actually observed, assuming that the “null hypothesis” is correct.

   H_0 : there is no difference between the groups

2. A common threshold is $p < 0.05$: often called “statistically significant”.

p-value	Common interpretation
$p < 0.001$	Very strong evidence against no difference
$p < 0.01$	Strong evidence against no difference
$p < 0.05$	Commonly treated as statistically significant
$p > 0.05$	Not statistically significant



Karl Pearson

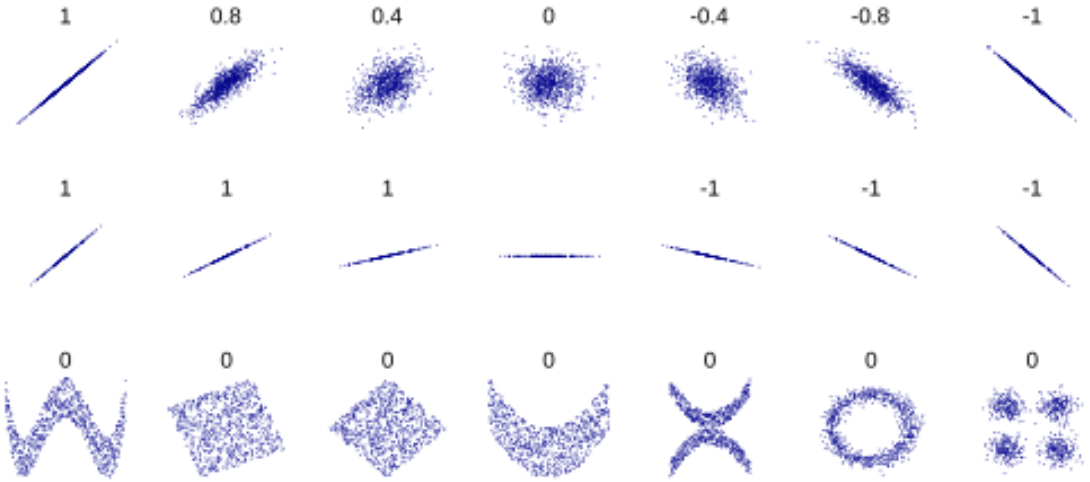
Source: https://en.wikipedia.org/wiki/P-value

Correlation

1. Important: Measures association not causation !!!!

2. Pearson's coefficient

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$



Source: https://en.wikipedia.org/wiki/Correlation

3. Range: [-1, +1]. +1: perfect correlation, -1: perfect inverse. Either extreme represents high correlation

scikit-learn : machine learning

1. Useful when we want to decode stimuli from neural activity, classify brain states, predict behaviour from neural signals, reduce high-dimensional data, or cluster neural responses.

2. Provides multiple built-in machine learning algorithms and models.

3. For more look at : https://scikit-learn.org

4. We need to import: `import sklearn`

5. Most “scikit-learn” workflows have the same structure:

```
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
score = model.score(X_test, y_test)
```

where X = data/features (ex: firing rates)
Y = labels/targets (ex: stimulus conditions)

Classification: decoding stimulus from activity

1. The goal is to learn the function $f : X_i \rightarrow y_i$, where X_i = neural activity vector on trial i, and y_i = stimulus label on trial i.

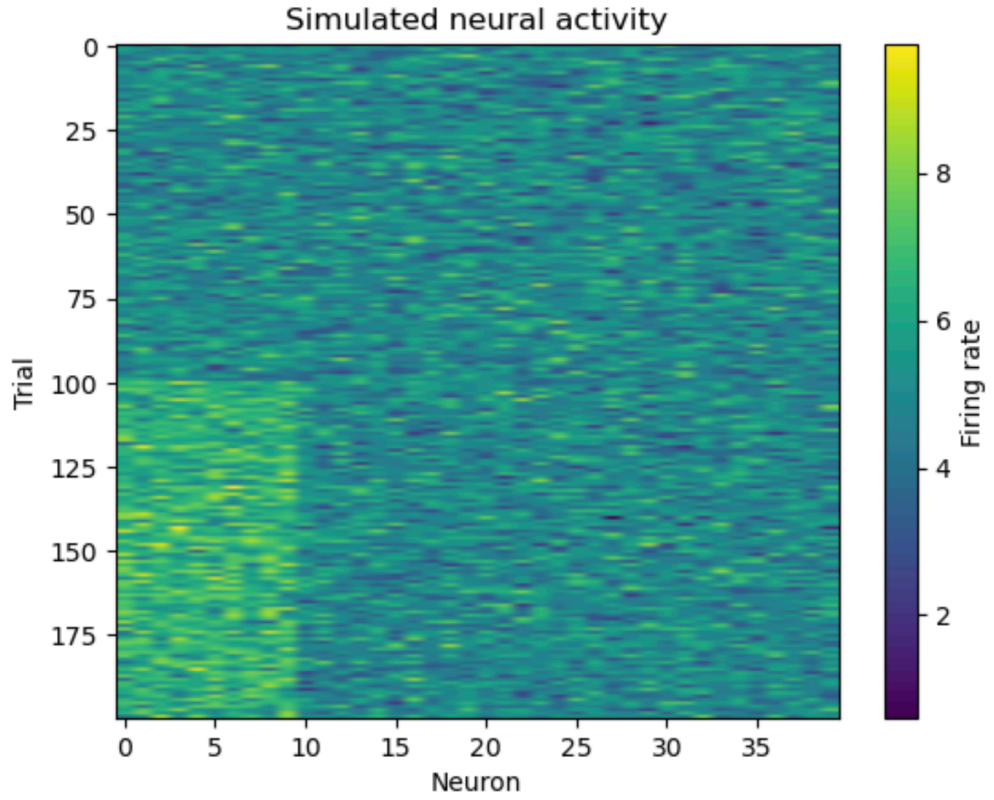
2. We want to predict $y_i \in \{0,1\}$

3. A logistic regression model:

$$P(y_i = 1 \mid X_i) = \frac{1}{1 + e^{-z_i}},$$

with $z_i = \beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2} + \ldots + \beta_p X_{ip}$

4. Chance accuracy = 0.5



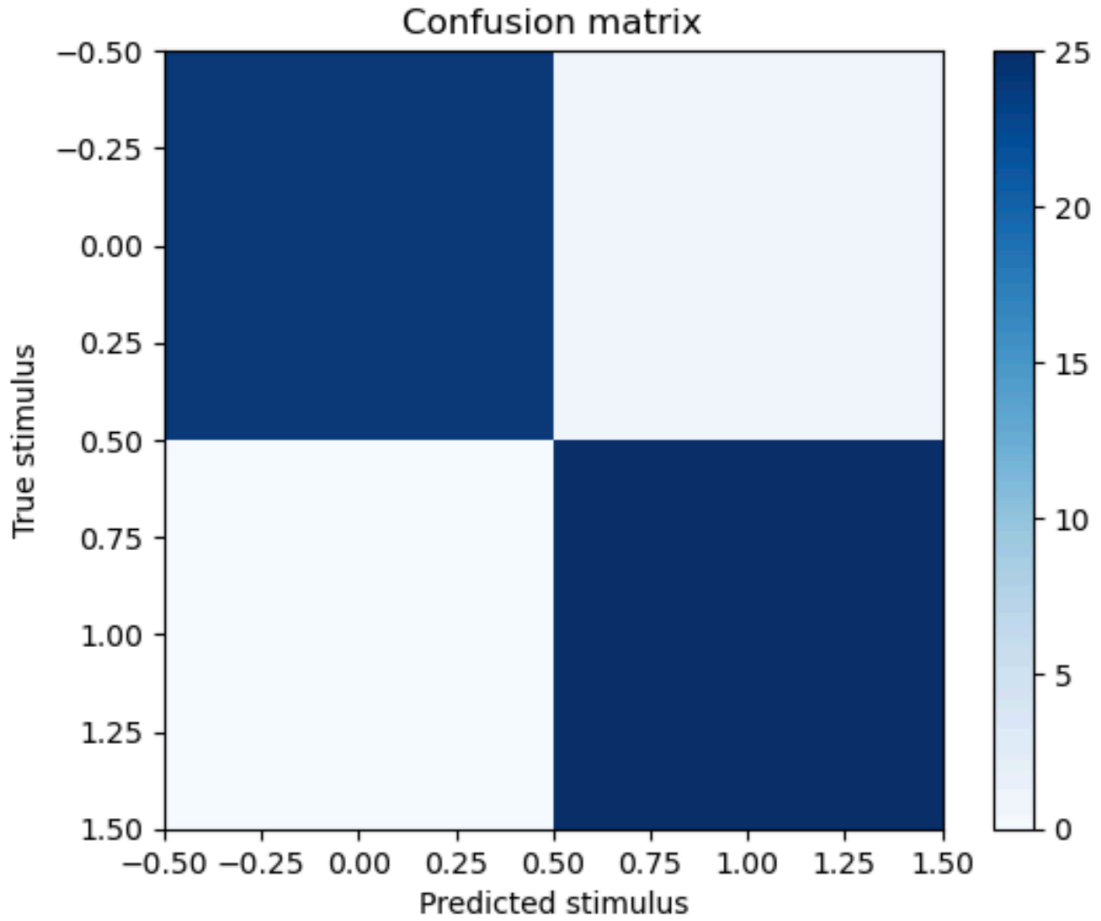
Confusion matrix

1. A confusion matrix counts how often each true class is predicted as each possible class

2. For two classes:

$$C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

C_{00}, C_{11} = correct, C_{10}, C_{01} = incorrect



Cross validation

1. A single test-train might be unstable. We use cross-validation to give a more reliable estimate of model performance across multiple test-train splits.

2. In k-fold cross-validation, the data are divided into k parts $D = D_1 \cup D_2 \cup \ldots \cup D_k$.

3. For each fold i, D_i = test set and $D \backslash D_i$ is the training set.

4. The final performance is the average: $\bar{A} = \frac{1}{k} \sum_{i=1}^{k} A_i$, with A_i the accuracy of fold i

Regression

1. Predicting future behaviour like behaviour from neural activity.

2. The target in neuroscience could be reaction time, movement speed, stimulus intensity, etc.

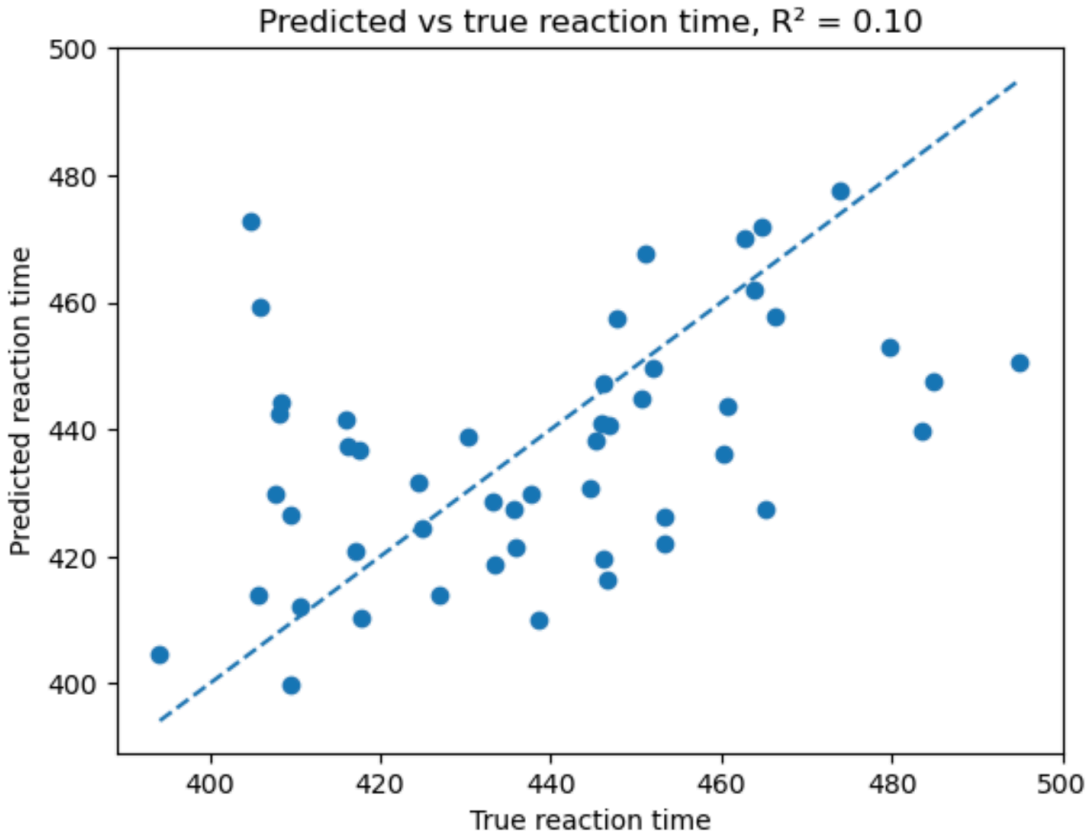
3. The model is
$$\mathbf{y}_i = \beta_0 + \beta_1 \mathbf{X}_{i1} + \beta_2 \mathbf{X}_{i2} + \ldots + \beta_p \mathbf{X}_{ip} + \epsilon_i \,.$$

4. For reaction time $\mathbf{RT}_i = \beta_0 + \sum_{j=1}^{p} \beta_j \mathbf{X}_{ij} + \epsilon_i \,.$

$\mathbf{RT}_i =$ reaction time on trial i,

5. $\mathbf{X}_{ij} =$ activity of neuron j on trial i,

$\epsilon_i =$ noise



R^2: coefficient of determination

1. Tells us how much of the variability in reaction time can be explained by neural activity

2. Given by $R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2}$

   y_i = true value

   here $\hat{y}_i$ = predicted value

   $\bar{y}$ = mean of the true values

3. ~1: strong prediction, ~0: similar to predicting the mean, <0: worse than predicting the mean

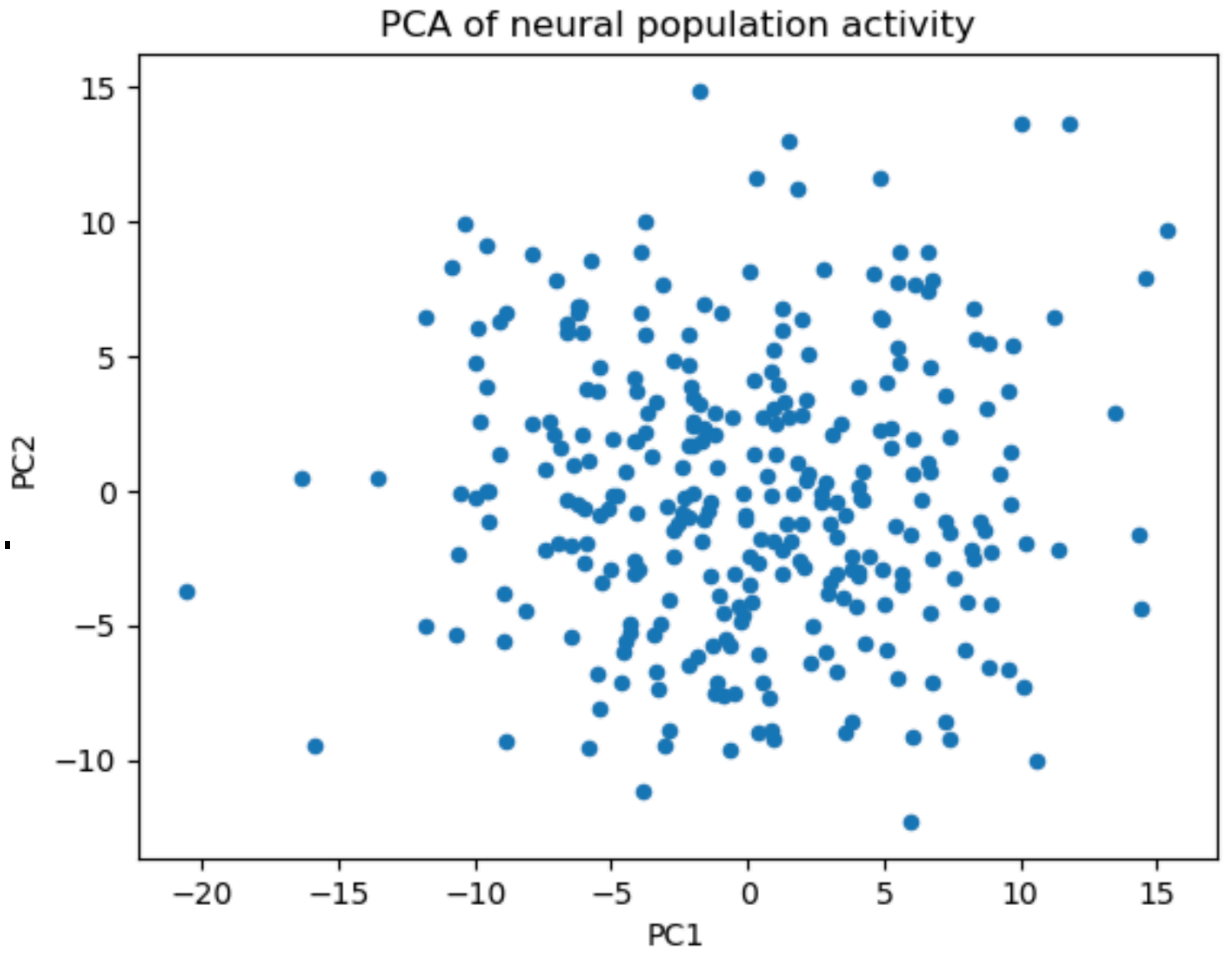
PCA: Principal Component Analysis

1. For dimensionality reduction.

2. Neural data can be high-dimensional; PCA finds new axes: “principal components”

3. Capture directions of maximum variance.

4. The first component is the direction w_1 that maximises Var(Xw_1) subject to $\|w_1\| = 1$.



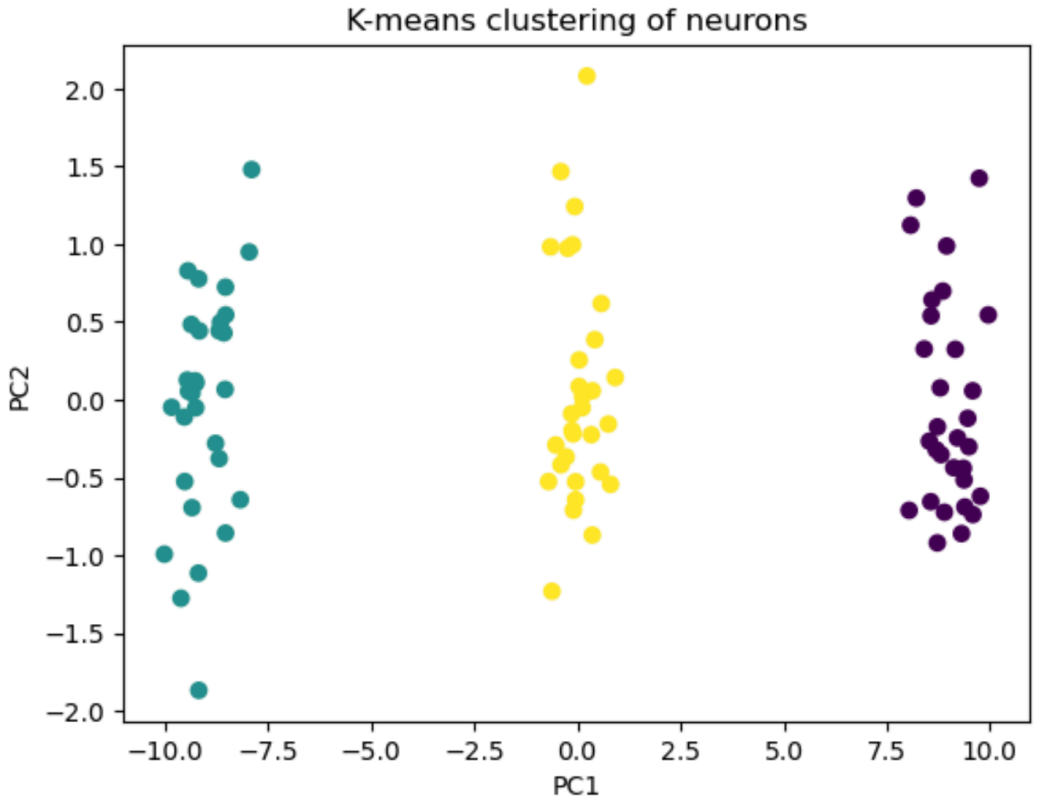
Clustering

1. An unsupervised technique to finding groups without labels

2. "k-means clustering" tries to find k cluster centres $\mu_1, \mu_1, \ldots \mu_k$, and assigns each point to the nearest center,

3. By minimising $\sum_{i=1}^{n} \|X_i - \mu_{c_i}\|^2$, where c_i is the cluster assigned to sample i.

4. It find groups, however does not automatically tell us whether those groups are biologically maningful.



AI assisted coding

Goal is to simulate a network of SNN:

Prompt given to Google gemini, be specific:

Write beginner-friendly Python code to simulate a network of 100 leaky integrate-and-fire neurons.

Use:
- NumPy and matplotlib only
- Euler time stepping
- membrane voltage V_i
- threshold V_th
- reset V_reset
- external input current
- random sparse connectivity matrix

Return:
1. the full code
2. a raster plot
3. a plot of population firing rate
4. comments explaining each step

Here is a complete, beginner-friendly Python script using NumPy and Matplotlib to simulate a network of 100 Leaky Integrate-and-Fire (LIF) neurons with random sparse connectivity.

The code uses Euler integration to update the membrane potentials over time and generates both a spike raster plot and a population firing rate plot.

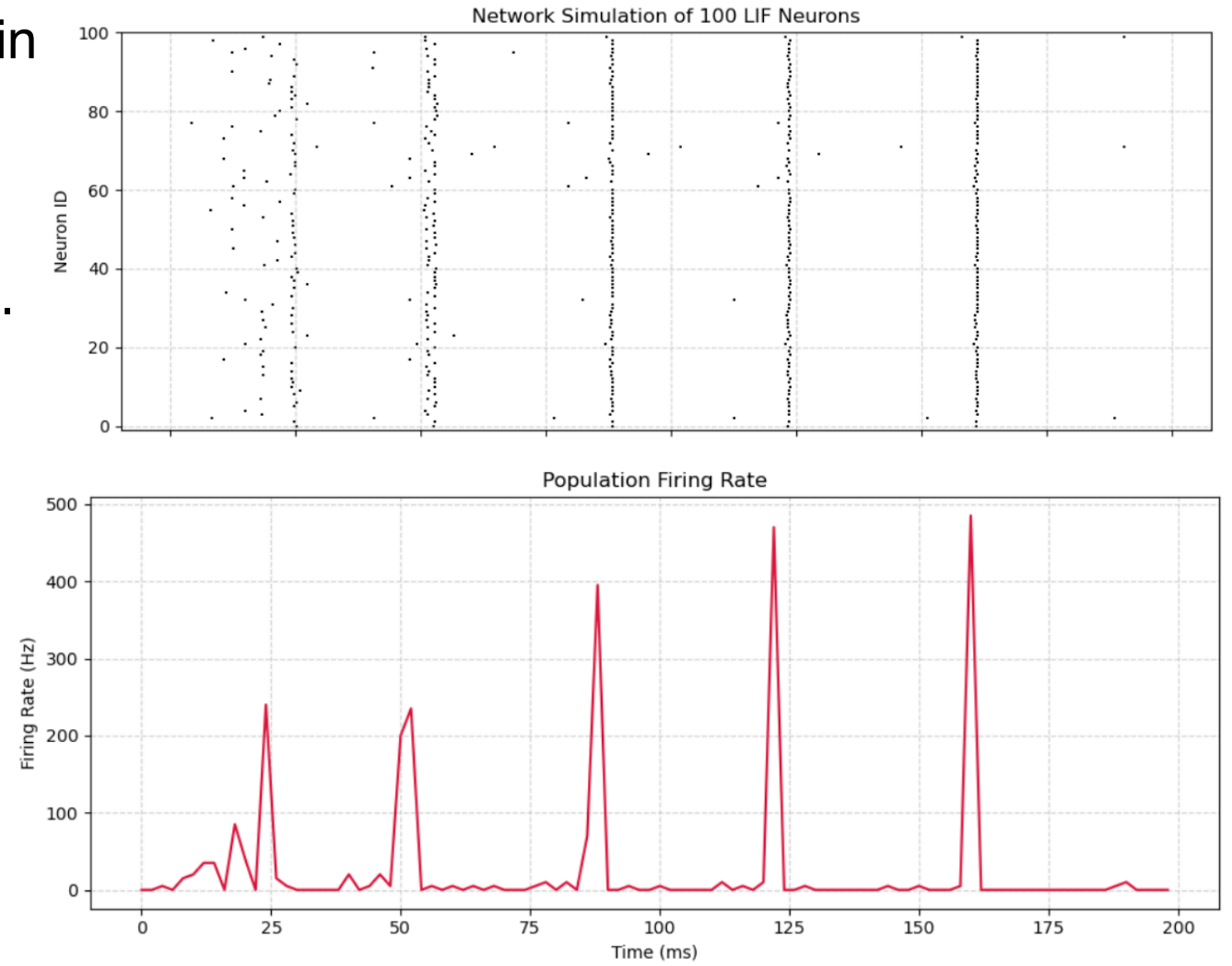
Python

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

# ========================================
# 1. Simulation and Network Parameters
# ========================================
N = 100          # Number of neurons
T = 200.0        # Total simulation time (ms)
dt = 0.1         # Time step for Euler integration (ms)
time = np.arange(0, T, dt)
steps = len(time)

# Neuron parameters
tau_m = 20.0     # Membrane time constant (ms)
V_rest = -70.0   # Resting membrane potential (mV)
V_th = -50.0     # Spike threshold (mV)
V_reset = -65.0  # Reset potential after a spike (mV)
R = 1.0          # Membrane resistance (GOhm)

# External input current (mean + noise to create realistic firing)
I_ext_mean = 21.0  # nA
I_ext_noise = 2.0  # Noise amplitude

# Synaptic parameters
connection_prob = 0.1  # 10% chance of connection between any two neurons
weight_scale = 0.5     # Synaptic weight strength (mV)
```

Ask Gemini Flash

AI assisted coding

1. Ask the LLM to check, improve, and explain
2. Always ask it to explain the code
3. Don’t trust blindly. Ask to look for mistakes. They can be useful for debugging
4. Better to have domain knowledge yourself and double check equations, units, parameter values, etc.
5. Use it as a coding assistant, not as a scientific authority



Questions? I’m all years