◈ Background: The Building Blocks

In DSP, we often want to simulate multiple waveforms:

- Each waveform has a distinct **frequency**
- All waveforms share a common **time base**
- Each waveform needs to be **sampled** uniformly

Using NumPy, we can generate these efficiently using **broadcasting** — a powerful technique where arrays of different shapes are combined automatically to perform element-wise operations.

🛠 Step-by-Step: Building the Signal Matrix

1. Define the Frequency Array

```
frequencies = np.array([5, 10, 20, 50])  # in Hz
```

This array has 4 frequencies, which we'll treat as **4 signals**.

---

2. Define the Time Base

```
t = np.linspace(0, 1, 1000)  # 1 second duration, 1000 samples
```

This gives a common **time vector**:

- Starts at 0 seconds
- Ends at 1 second
- Contains 1000 equally spaced samples (1 ms apart)

---

3. Broadcast Frequency × Time

To compute sine waves:

```
signals = np.sin(2 * np.pi * frequencies[:, None] * t)
```

Here’s what happens:

Expression	Meaning
`frequencies[:, None]`	Shape becomes `(4, 1)` → vertical vector of 4 frequencies
`t`	Shape is `(1000,)` → horizontal vector of 1000 time points
`frequencies[:, None] * t`	Broadcasts to shape `(4, 1000)` → every frequency × all `t`
`sin(...)`	Computes sine at each time sample for each frequency

Result: The Signal Matrix

```
signals.shape  # Output: (4, 1000)
```

This is a **2D array**:

- 4 **rows** (signals)
- 1000 **columns** (samples per signal)

Each row represents a **sine wave** for a specific frequency.

🔶 Interpreting `.shape[0]` and `.shape[1]`

Attribute	Description
`signals.shape[0]`	Number of signals (i.e., 4 frequencies)
`signals.shape[1]`	Samples per signal (1000 time points)

This allows you to:

- Plot each row individually
- Run FFT per signal
- Analyze multi-tone compositions

🖼 Visualizing the Signals

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
for i in range(signals.shape[0]):
    plt.plot(t, signals[i], label=f"{frequencies[i]} Hz")

plt.title("📈 Multi-Tone Signal Generation using Broadcasting")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

This plot shows:

- 4 sine waves of different frequencies
- All sharing the same time base

You can clearly observe:

- 5 cycles in 1s (5 Hz)
- 10 cycles (10 Hz)
- 20 cycles (20 Hz)
- 50 cycles (very dense oscillation)

🔍 Real-World Insight

Term	Physical Meaning
Frequency array	Different tones (e.g., audio frequencies, carriers)
Time vector	Common sampling period (like SDR’s sampling clock)
Broadcasting	Efficiently creates all waveforms without loops
`.shape[0]`	Number of signals you're simulating or processing
`.shape[1]`	Number of time samples (controls resolution/sampling)

✅ Summary

- NumPy broadcasting allows elegant creation of multiple signals
- The result is a 2D array where each **row is a sine wave**
- `.shape[0]` tells you **how many signals**
- `.shape[1]` tells you **how many samples per signal**
- This pattern is **widely used** in DSP, SDR, and machine learning

---

⌺ Upcoming Execises

- 🔁 Add amplitude or phase shifts per row
- 📉 Take FFT row-wise for spectrum analysis

- 🎛 Combine tones to simulate modulated waveforms
- 💾 Save signals as IQ data (interleaved format)
