◈ NumPy for DSP

A Friendly, Hands-On Guide to Using NumPy for Digital Signal Processing

---

🎯 Why NumPy for DSP?

NumPy is like your DSP Swiss Army Knife:

- Fast array & matrix operations (critical for handling signals).
- Built-in mathematical functions (no loops needed!).
- Works hand-in-hand with FFT, filters, and spectrum analysis.

If DSP is a **race car**, NumPy is **the engine**.

📌 Core Concepts

Before jumping into functions, you need to **think in arrays**.

In DSP:

- **Time-domain signal** → an array of samples.
- **Frequency-domain spectrum** → another array (often complex values).
- **Index positions** in arrays map to **time steps** or **frequency bins**.

1️⃣ Creating Time Axes for Signals

```
np.arange()
```

Creates evenly spaced values at a fixed step.

```python
import numpy as np

Fs = 1000   # Sampling frequency
T = 1 / Fs  # Sampling interval
N = 8       # Number of samples

t = np.arange(N) * T
print(t)
# [0.    0.001 0.002 0.003 0.004 0.005 0.006 0.007]
```

Use case in DSP:

Generate **sample time points** for discrete signals.

`np.linspace()`

Creates evenly spaced values over an **interval**, including start & end.

```python
t1 = np.linspace(0, (N-1)*T, N)
print(t1)
# [0.    0.001 0.002 0.003 0.004 0.005 0.006 0.007] {#0----0001-0002-0003-0004-0005-00
```

Difference:

- `np.arange()` → step-based (good for **sample count control**).
- `np.linspace()` → number-of-points-based (good for **plotting**).

---

2 Creating Signals

`np.sin()` & `np.cos()`

For generating sine & cosine waves.

```python
f = 50   # Frequency
y = np.sin(2 * np.pi * f * t)
```

`np.exp()`

For complex exponentials (used in Fourier transforms).

```
y = np.exp(1j * 2 * np.pi * f * t)
```

---

3 Basic Array Operations

- **Add signals**: `y_sum = y1 + y2`
- **Multiply signals**: `y_mul = y1 * y2`
- **Scale**: `y_scaled = 2.5 * y`
- **Element-wise power**: `y_sq = y ** 2`

---

4 Useful Math Functions

Function	DSP Usage Example
`np.abs(x)`	Magnitude of complex FFT result
`np.angle(x)`	Phase of complex FFT result
`np.real(x)`	Get I (in-phase) component
`np.imag(x)`	Get Q (quadrature) component
`np.max(x)`	Peak detection
`np.mean(x)`	Average signal level
`np.std(x)`	Noise measurement

Example:

```
fft_result = np.fft.fft(y)
magnitude = np.abs(fft_result)
phase = np.angle(fft_result)
```

5️⃣ Indexing & Slicing

```
samples_5_to_10 = y[5:11]
every_other_sample = y[::2]
last_sample = y[-1]
```

DSP note:
Slicing helps in **windowing signals** or extracting **frames** from a stream.

6️⃣ Vectorized Operations vs Loops

Good:

```
y = np.sin(2*np.pi*f*t)
```

Bad:

```
y = []
for ti in t:
    y.append(np.sin(2*np.pi*f*ti))
```

NumPy is **much faster** because it works in **C under the hood**.

7️⃣ FFT with NumPy

`np.fft.fft()`

Converts time-domain → frequency-domain.

```
X = np.fft.fft(y)
freqs = np.fft.fftfreq(N, T)

for k in range(N):
    print(f"Bin {k}: freq={freqs[k]:.1f} Hz, mag={np.abs(X[k]):.2f}, phase={np.angle(X
```

`np.fft.ifft()`

Converts frequency-domain → time-domain.

```
y_reconstructed = np.fft.ifft(X)
```

8️⃣ Windowing

```
window = np.hanning(N)
y_windowed = y * window
```

Reduces **spectral leakage** before FFT.

9️⃣ Reshaping & Combining

`np.reshape()`

```
iq_data = np.arange(8)
iq_pairs = iq_data.reshape((len(iq_data)//2, 2))
```

DSP use: Separate **I & Q components**.

`np.concatenate()` **&** `np.stack()`

Merge multiple signals.

🔟 Random Signals (Noise)

```
noise = np.random.normal(0, 1, N)  # Mean=0, StdDev=1
```

Putting It Together – A Mini DSP Example

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

Fs = 1000
T = 1/Fs
N = 256
t = np.arange(N) * T

# Create a signal with 50 Hz and 120 Hz components {#create-a-signal-with-50-hz-and-12
y = np.sin(2*np.pi*50*t) + 0.5*np.sin(2*np.pi*120*t)

# FFT {#fft }
X = np.fft.fft(y)
freqs = np.fft.fftfreq(N, T)

# Plot {#plot }
plt.subplot(2,1,1)
plt.plot(t, y)
plt.title("Time Domain Signal")

plt.subplot(2,1,2)
plt.stem(freqs, np.abs(X))
plt.title("Frequency Domain Spectrum")
plt.show()
```

📖 Cheat Sheet

- `np.arange()` → sample points (step-based)
- `np.linspace()` → sample points (count-based)
- `np.sin` / `np.cos` → waveforms
- `np.exp()` → complex exponentials
- `np.abs()` → magnitude
- `np.angle()` → phase
- `np.real` / `np.imag` → I/Q extraction
- `np.fft.fft()` / `np.fft.ifft()` → domain conversion
- `np.reshape()` → data reformatting

💡 **Remember:**
NumPy isn’t just for numbers — in DSP it’s your **bridge between time-domain samples and frequency-domain insights**.