Fourier Analysis with Python

For turning a time record into a spectrum. Assumes you can work with arrays and plot them — see Arrays and Plotting.

This page is a reference for Fourier analysis: what the Fourier Transform computes, how to use it in Python, and how your acquisition parameters shape the spectrum you see. All examples use synthetic data and need only NumPy and Matplotlib — no hardware required.

Use it as a lookup for topics like windowing and aliasing as they come up in your analysis.

1 What Is the Discrete Fourier Transform?

The discrete Fourier Transform (DFT) of a set of data $\{y_0, y_1, ..., y_{N-1}\}$ is

$$Y_m = \sum_{n=0}^{N-1} y_n \cdot e^{-2\pi i \frac{m}{N} n}$$

The basic idea: the DFT decomposes your data into frequency components. The magnitude of Y_m tells you how much of your signal came from an oscillation at the m-th frequency. If your signal is a pure 100 Hz sine wave, the DFT has a large value at the bin corresponding to 100 Hz and small values everywhere else.

Key facts:

- The DFT output Y_m is *complex-valued* (it encodes both amplitude and phase). To see how much power sits at each frequency, compute $|Y_m|^2$.
- The DFT of N data points produces N complex values.
- For real-valued input (always the case for experimental data) the DFT is symmetric: $|Y_m| = |Y_{N-m}|$. The second half is redundant, so we typically plot only the first half, up to the Nyquist frequency.

1.1 Conceptual Questions

Before computing any FFTs, think through these — they help you interpret results correctly.

1. How do the units of the Fourier Transform array Y_m relate to the units of the data y_n?
2. Does the data y_n have to be taken at equally spaced intervals for the DFT formula above to apply?
3. Is it possible for two different sets of data to have the same Fourier Transform?
4. If a data set has N elements, how many elements does its DFT have?

2 Computing the Power Spectrum in Python

NumPy provides efficient FFT (Fast Fourier Transform) functions. For real-valued signals — which is what you always have in the lab — use `np.fft.rfft()` and `np.fft.rfftfreq()`. They automatically return only the positive-frequency half of the spectrum.

```
import numpy as np

def compute_power_spectrum(signal, sample_rate):
    """
    Compute the one-sided power spectrum of a real-valued signal.
```

```
    Parameters:
        signal: 1D array of signal values
        sample_rate: Sampling rate in Hz

    Returns:
        frequencies: Array of positive frequency values (Hz)
        power: Power spectrum (magnitude squared, normalized)
    """
    n = len(signal)
    frequencies = np.fft.rfftfreq(n, d=1 / sample_rate)
    fft_result = np.fft.rfft(signal)

    # Power spectrum: magnitude squared, normalized by number of points
    power = (np.abs(fft_result) / n) ** 2

    # Double the power for frequencies that have both +/- contributions
    # (all except DC at index 0 and Nyquist at index -1)
    power[1:-1] *= 2

    return frequencies, power
```

2.1 Basic example

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

# Generate a test signal: 50 Hz + 120 Hz + noise
sample_rate = 1000  # Hz
duration = 1.0      # seconds
t = np.arange(0, duration, 1 / sample_rate)

signal = (np.sin(2 * np.pi * 50 * t)
          + 0.5 * np.sin(2 * np.pi * 120 * t)
          + 0.2 * np.random.randn(len(t)))

frequencies, power = compute_power_spectrum(signal, sample_rate)

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 6))

ax1.plot(t * 1000, signal, 'b-', linewidth=0.5)
ax1.set_xlabel('Time (ms)'); ax1.set_ylabel('Amplitude')
ax1.set_title('Time Domain'); ax1.set_xlim(0, 100); ax1.grid(True, alpha=0.3)

ax2.plot(frequencies, power, 'r-')
ax2.set_xlabel('Frequency (Hz)'); ax2.set_ylabel('Power')
ax2.set_title('Power Spectrum'); ax2.set_xlim(0, 200); ax2.grid(True, alpha=0.3)

plt.tight_layout(); plt.show()
```

You should see two clear peaks — a large one at 50 Hz and a smaller one at 120 Hz — on a low noise floor.

2.2 `rfft` vs. `fft`

The functions above use `np.fft.rfft` and `np.fft.rfftfreq`, designed for real-valued input. NumPy also provides `np.fft.fft` and `np.fft.fftfreq` for the general complex-valued case. For real signals, `rfft` is simpler: it returns only the positive-frequency half directly, so you don't slice the output by hand. Both give the same result for the positive frequencies.

3 Key Relationships: Resolution and Nyquist

Two parameters control what you can see in the spectrum:

Parameter	Formula	Meaning
Frequency resolution	$\Delta f = f_s/N = 1/T$	Smallest frequency difference you can distinguish
Maximum frequency (Nyquist)	$f_N = f_s/2$	Highest frequency you can measure
Total duration	$T = N/f_s$	Longer duration = finer resolution

where f_s is the sample rate, N the number of samples, and T the total acquisition duration.

```
sample_rate = 1000     # Hz
num_samples = 2000

freq_resolution = sample_rate / num_samples  # = 1 / duration
max_frequency = sample_rate / 2              # Nyquist frequency

print(f"Duration: {num_samples / sample_rate} s")
print(f"Frequency resolution: {freq_resolution} Hz")
print(f"Maximum frequency: {max_frequency} Hz")
# Duration: 2.0 s
# Frequency resolution: 0.5 Hz
# Maximum frequency: 500.0 Hz
```

A common misconception: increasing the sample rate does *not* improve frequency resolution. Resolution is set by the total acquisition duration $T = N/f_s$. A higher sample rate raises the Nyquist frequency (letting you see higher frequencies), but finer frequency resolution requires recording for longer.

4 Windowing and Spectral Leakage

The DFT assumes your signal is periodic — that the first and last samples connect smoothly. In practice your signal has finite duration and the edges rarely match. This mismatch creates **spectral leakage**: the FFT spreads power from a narrow peak into broad "skirts" across many frequency bins.

4.1 Why it matters

Leakage makes peaks look broader than they are and raises the noise floor around strong peaks. In Fourier-transform spectroscopy, it directly degrades spectral resolution.

4.2 The fix: window functions

Multiplying your signal by a **window function** that tapers smoothly to zero at both ends removes the edge discontinuity. The most common choice is the **Hanning window** (`np.hanning`). The example below shows the problem and the fix: a strong 47 Hz component and a weak 58 Hz component (only 8% of the strong one's amplitude). Without windowing, leakage from the strong peak buries the weak one; with windowing, both are visible.

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

# Two-frequency signal: strong at 47 Hz, weak at 58 Hz (8% amplitude)
sample_rate = 1000
t = np.arange(0, 0.5, 1 / sample_rate)  # 0.5 s (resolution = 2 Hz)
signal = np.sin(2 * np.pi * 47 * t) + 0.08 * np.sin(2 * np.pi * 58 * t)

window = np.hanning(len(signal))
signal_windowed = signal * window

freqs = np.fft.rfftfreq(len(signal), d=1 / sample_rate)
power_raw = np.abs(np.fft.rfft(signal)) ** 2
power_windowed = np.abs(np.fft.rfft(signal_windowed)) ** 2

fig, axes = plt.subplots(2, 2, figsize=(12, 8))

axes[0, 0].plot(t * 1000, signal, 'b-', linewidth=0.5)
axes[0, 0].set_xlabel('Time (ms)'); axes[0, 0].set_ylabel('Amplitude')
axes[0, 0].set_title('Signal (no window)'); axes[0, 0].grid(True, alpha=0.3)

axes[0, 1].plot(t * 1000, window, 'r-', alpha=0.4, linewidth=2, label='Hanning window')
axes[0, 1].plot(t * 1000, signal_windowed, 'b-', linewidth=0.5, label='Windowed signal')
axes[0, 1].set_xlabel('Time (ms)'); axes[0, 1].set_ylabel('Amplitude')
axes[0, 1].set_title('Signal × Hanning window')
axes[0, 1].legend(loc='upper right'); axes[0, 1].grid(True, alpha=0.3)

axes[1, 0].plot(freqs, power_raw)
axes[1, 0].set_xlabel('Frequency (Hz)'); axes[1, 0].set_ylabel('Power')
axes[1, 0].set_title('Spectrum (no window)'); axes[1, 0].set_xlim(30, 75)
axes[1, 0].grid(True, alpha=0.3)

axes[1, 1].plot(freqs, power_windowed)
axes[1, 1].set_xlabel('Frequency (Hz)'); axes[1, 1].set_ylabel('Power')
axes[1, 1].set_title('Spectrum (Hanning window)'); axes[1, 1].set_xlim(30, 75)
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout(); plt.show()
```

Top row (inputs): the raw signal (left) starts and ends at arbitrary values — when the DFT wraps it to repeat, those edges create discontinuities. The windowed signal (right) tapers smoothly to zero at both ends, satisfying the periodicity assumption. The red curve is the Hanning window's shape.

Bottom row (spectra): without windowing (left), leakage skirts from the strong 47 Hz peak spread across the spectrum and bury the weak 58 Hz signal. With windowing (right), the leakage is suppressed and the weak peak emerges. That's the payoff: windowing lets you detect weak signals near strong ones. It does lower overall peak height (it attenuates the signal near the edges), but the gain in peak isolation is well worth it.

Tip

When to window: apply a window function whenever your signal doesn't naturally start and end at zero — almost always the case for experimental data.

5 Aliasing

If your signal contains frequencies above the Nyquist frequency ($f_s/2$), they "fold back" and appear at incorrect positions in the spectrum. This is **aliasing**, and it cannot be undone after the fact.

Example: a 400 Hz signal sampled at 500 Hz appears at 100 Hz in the spectrum (it folds around the 250 Hz Nyquist frequency).

Rule of thumb: sample at least 2× your highest frequency of interest (preferably 5–10× for clean spectra).

You can work out where a given frequency will land without taking any data — the signal folds repeatedly about multiples of the Nyquist frequency:

```python
def apparent_frequency(f_signal, sample_rate):
    """Where a signal of frequency f_signal appears when sampled at sample_rate."""
    folded = abs(f_signal - sample_rate * round(f_signal / sample_rate))
    return folded, f_signal <= sample_rate / 2

for f in (50, 400, 800, 1050):
    apparent, honest = apparent_frequency(f, 1000)
    flag = "" if honest else "   (ALIASED)"
    print(f"{f:5.0f} Hz sampled at 1000 Hz -> appears at {apparent:5.0f} Hz{flag}")
```

```
   50 Hz sampled at 1000 Hz -> appears at    50 Hz
  400 Hz sampled at 1000 Hz -> appears at   400 Hz
  800 Hz sampled at 1000 Hz -> appears at   200 Hz   (ALIASED)
 1050 Hz sampled at 1000 Hz -> appears at    50 Hz   (ALIASED)
```

The last case is the dangerous one: a 1050 Hz signal shows up at 50 Hz, which looks entirely plausible and is indistinguishable from a genuine 50 Hz component once the data is recorded. The only defences are sampling fast enough, or filtering above Nyquist *before* the converter.

Note

Debugging unexpected peaks: if your FFT shows a peak you didn't expect (e.g. at 120 Hz), consider physical sources: power-line interference (60 Hz and harmonics at 120, 180 Hz), fluorescent-light flicker (100–120 Hz), or mechanical vibration. To identify the source, try shielding the detector, turning off lights, or isolating the optics from the table.

Practice Exercises (optional)

These exercises let you verify your understanding using only Python. No hardware needed.

5.1 Exercise 1: Single frequency

Generate a 100 Hz sine wave sampled at 1000 Hz for 1 second. Compute and plot its power spectrum.

```
sample_rate = 1000
duration = 1.0
t = np.arange(0, duration, 1 / sample_rate)
signal = np.sin(2 * np.pi * 100 * t)

freqs, power = compute_power_spectrum(signal, sample_rate)

plt.plot(freqs, power)
plt.xlabel('Frequency (Hz)'); plt.ylabel('Power')
plt.xlim(0, 200); plt.grid(True, alpha=0.3); plt.show()
```

Verify:

- The peak appears at exactly 100 Hz.
- The frequency resolution is $\Delta f = 1000/1000 = 1$ Hz.
- The Nyquist frequency is $f_N = 1000/2 = 500$ Hz.

5.2 Exercise 2: Two frequencies and resolution

Generate a signal containing 95 Hz and 105 Hz components (10 Hz apart). Try two durations:

a) Duration = 1.0 s (resolution = 1 Hz). Can you resolve the two peaks?

b) Duration = 0.05 s (resolution = 20 Hz). Can you still resolve them? Why or why not?

```
sample_rate = 1000

for duration in [1.0, 0.05]:
    t = np.arange(0, duration, 1 / sample_rate)
    signal = np.sin(2 * np.pi * 95 * t) + np.sin(2 * np.pi * 105 * t)

    freqs, power = compute_power_spectrum(signal, sample_rate)

    plt.figure(figsize=(8, 3))
    plt.plot(freqs, power)
    plt.xlabel('Frequency (Hz)'); plt.ylabel('Power')
    plt.title(f'Duration = {duration} s (resolution = {sample_rate / len(t):.0f}
Hz)')
    plt.xlim(50, 150); plt.grid(True, alpha=0.3); plt.show()
```

This demonstrates concretely why longer recordings give better frequency resolution.

5.3 Exercise 3: Aliasing

Generate a 400 Hz sine wave sampled at only 500 Hz. Before running the code, predict where the peak

Now raise the sample rate to 1000 Hz and confirm the peak appears at the correct 400 Hz.

```
    # Undersampled: 400 Hz signal at 500 Hz sample rate
```

Back to Python for Lab Work