Practical Spectral Analysis with Python

Preface

For many students—and even graduate researchers—their first real encounter with spectral analysis often unfolds like this:

One day, they notice an intriguing phenomenon in a time-domain signal and eagerly share their discovery with an advisor or senior colleague. The response is calmly delivered: "You should try some Fourier or wavelet analysis."

Returning to their desk, they dig out an old calculus textbook, flip through several pages, and quickly realize it won't help. Next comes the trusted solution: a swift Google search for "Fourier Analysis tutorial". Eventually, they stumble upon a highly recommended *Digital Signal Processing* textbook with an impressive 4.3/5.0 book rating. After a marathon weekend, they manage to read through the 50-plus pages of Chapter 1, *Signals and Systems*. However, by Chapter 2, Linear Time-Invariant Systems, fatigue sets in—only to realize that the actual Fourier series material is still more than 120 pages away.

At this juncture, most students pragmatically pivot to google *"Fourier Analysis by xxx"* and get an answer with some unfamiliar jargon from *StackOverflow*, grabbing a ready-made code snippet to forge ahead.

Yet, a few determined souls persist—spending days gathering materials, watching lectures online, coding, and compiling a detailed report. Proudly, they present their hard work to their advisor, only to be met with the classic understated response: "Why so little progress this week?"

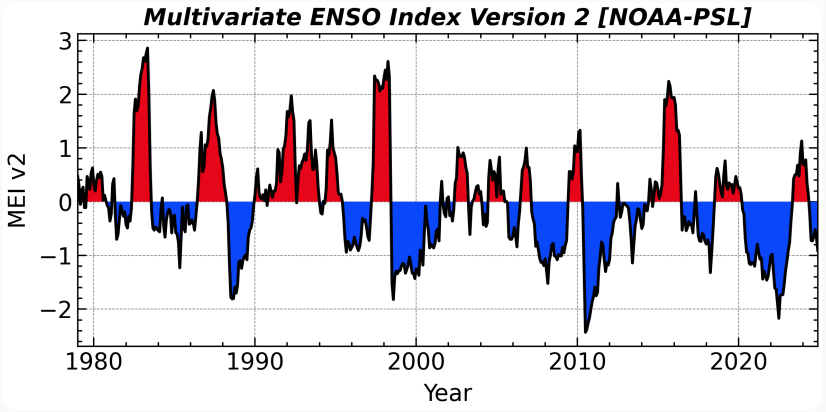
Why Do We Need Spectral Analysis?

Signals and Time Series

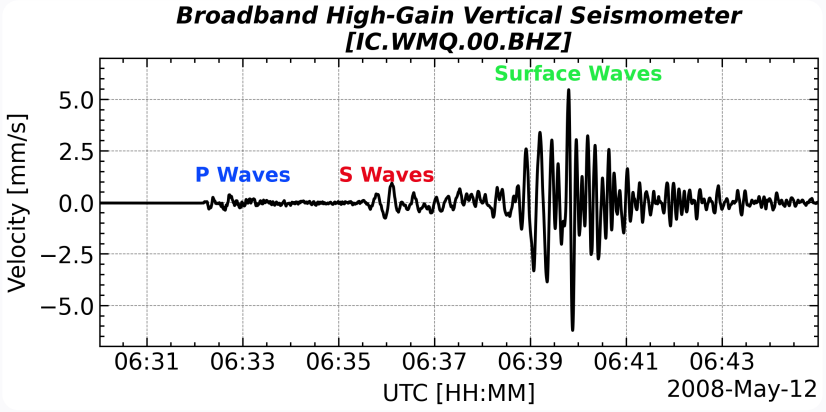
In physics and engineering, we frequently encounter **signals**—mathematical functions representing physical quantities that vary continuously or discretely over time. A signal can be any measurable quantity exhibiting temporal

variation, such as an audio waveform, the voltage output from a sensor, or the magnetic field recorded during a plasma experiment. When signals are observed, sampled, and recorded sequentially, they form a **time series**, capturing how these quantities evolve. Many familiar phenomena can naturally be described as time series, including:

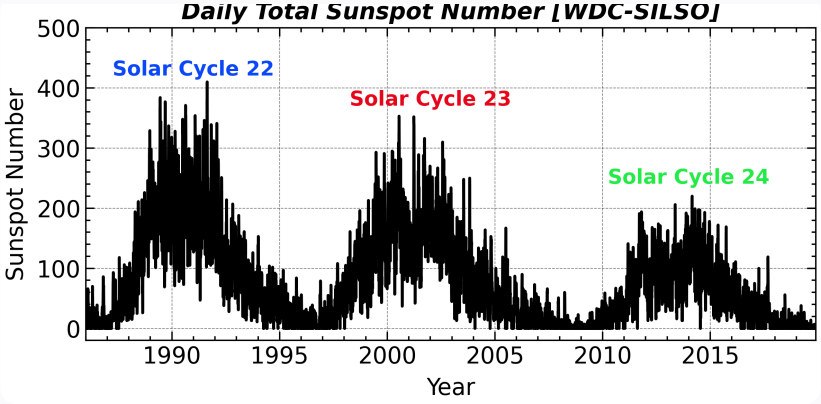
- **Meteorology:** e.g. El-Nino ENSO Index



- **Geophysics:** e.g. Seismic Waves



- **Solar Physics:** e.g. Sunspot Number



Each of these examples is described in the **time domain**—meaning we specify a physical quantity (such as amplitude, voltage, or magnetic field strength) explicitly **as a function of time**.

While understanding the time-domain behavior of a system is fundamental, it can sometimes be challenging to discern **underlying patterns, periodicities, or oscillatory features** directly from time-domain data. This is precisely where **spectral analysis** proves invaluable, as it allows us to examine signals in the frequency domain, clearly identifying and characterizing these hidden structures.

Understand the Signal from Frequency Domain

Spectral analysis examines a signal in the frequency domain instead of the time domain.

Imagine listening to an orchestra. The audio signal is a complex waveform. But your brain can distinguish individual notes — essentially doing spectral analysis!

In **plasma physics**, spectral analysis helps resolve the basic properties of wave (e.g., amplitude, compressibility) by revealing dominant frequencies in electromagnetic field fluctuations.

Spectral analysis helps to:

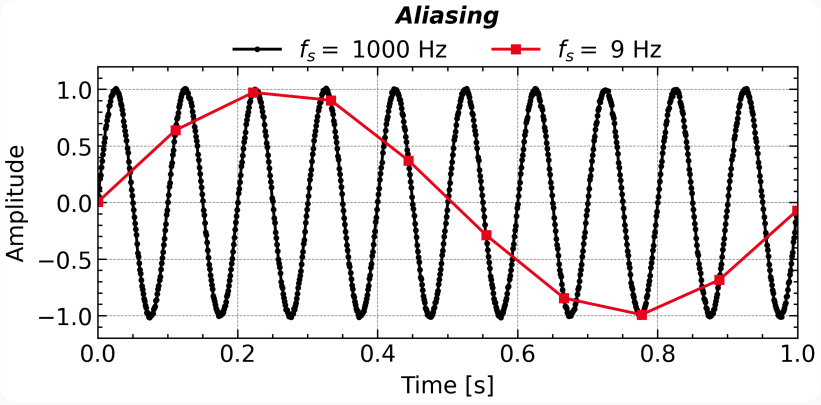
1. **Identify dominant frequencies** in a signal.
2. **Detect multiple overlapping processes**.
3. **Understand system behavior** through resonance.
4. **Filter or reduce noise**.

Sampling

> **Nyquist-Shannon Sampling Theorem:** A band-limited continuous-time signal $x(t)$ containing no frequency components higher than f_{max}, can be perfectly reconstructed from its samples if it is sampled at a rate:
>
> $$f_s \geq 2f_{max}$$

The frequency upper limitation $f_{max} = f_s/2$ is also called ***Nyquist Frequency***.

When you measure a high frequency signal with a low cadence instrument, you will not only miss the high frequency component, **but also measure an erroneous signal**, so called ***Aliasing***.



Such phenomenon is essentially unrelated to the Fourier transform as its frequency range ends up to $f_s/2$ and can be directly observed by naked eye. In real life, aliasing can be visualized by recording the running car wheel (or helicopter propeller) and television (or computer screen) with your smart phone.

This effect always happens when you (down-)sampling the signal, a common way to avoid it is to apply a low pass filter so that the high frequency component doesn't contribute to the unreal signal. In the instrumental implementation, that filter typically consists of a set of resistor, inductor, and capacity and is putted before the analog-digital converter.

- **Generate the Timestamps**

```
import numpy as np

# Length of signal, N
n = 100
fs = 10
T = 1.0

dt = 1 / fs

# Way 1: Given sampling frequency, fs
t =  np.arange(0, n) / fs
# t = 0.0, 0.1, 0.2, ..., 9.9

# Way 2: Given sampling period, dt
t =  np.arange(0, n) / fs

# Way 3: Given Signal Duration, T
t = np.linspace(0, T, n, endpoint = False)
```

  - Tips: Set `endpoint = False` so that the last point is not included in the time array, which ensures that the sampling frequency is equal to f_s.

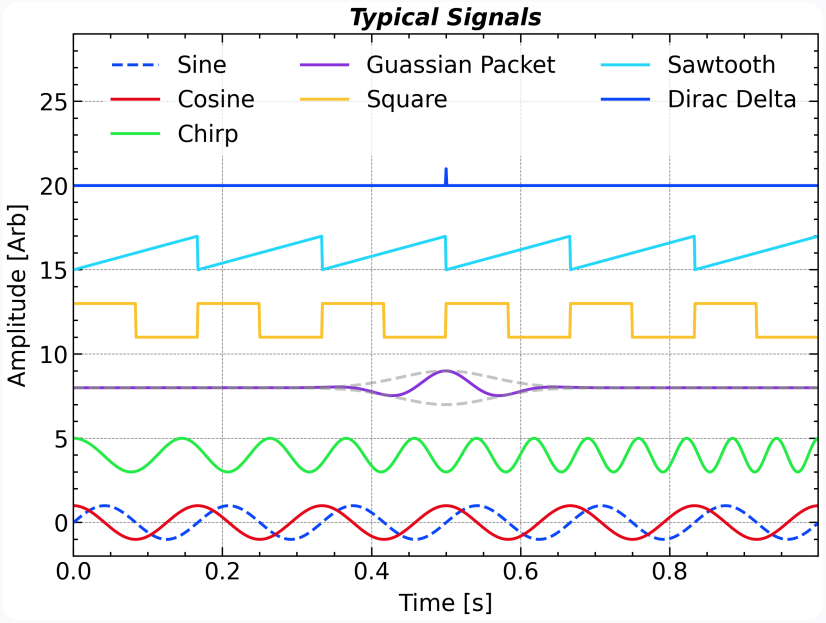
- **Generate the Signal**

```
# Assign wave (angular) frequency (omega = 2 * np.pi * frequency)
f0 = 3
f1 = 6
amp1 = 1
amp2 = 2
omega0 = 2 * np.pi * f0
omega1 = 2 * np.pi * f1

# Way 1: Directly generate the signal
sig = amp1 * np.sin(omega0 * t) + amp2 * np.sin(omega1 * t)

# Way 2: Generate a complex function and then take the real (cosine) or
imaginary (sine) part
sig = (amp1 * np.exp(1j * omega0 * t) + amp2 * np.exp(1j * omega1 *
t)).imag

# Way 3: Use a function or anonymous function to generate the signal
```

```
def sig_func(t):
    return amp1 * np.sin(omega0 * t) + amp2 * np.sin(omega1 * t)
# or
sig_func = lambda t: amp1 * np.sin(omega0 * t) + amp2 * np.sin(omega1 * t)

sig = sig_func(t)
```

Built-in Waveform Functions in `scipy.signal`

In addition to sine waves, there are some other commonly used waveforms built-in functions that are provided by `scipy.signal` module. These functions can be used to generate various types of signals for testing, simulation, and analysis purposes. Below is a brief overview of some typical waveforms:



1. Chirp Waveform (`chirp`)

Generates a swept-frequency (chirp) signal, which is often used in radar, sonar, and frequency response analysis.

```
scipy.signal.chirp(t, f0, t1, f1, method='linear', phi=0)
```

- **Parameters**:

- `t`: Time array.
- `f0`: Initial frequency at time `t=0`.
- `t1`: Final time for frequency sweep.
- `f1`: Final frequency at time `t1`.
- `method`: Type of frequency sweep (`linear`, `quadratic`, `logarithmic`, or `hyperbolic`).
- `phi`: Initial phase in degrees.

---

2. Gaussian Pulse (`gausspulse`)

Generates a Gaussian-modulated sinusoidal pulse, commonly used in ultrasound and narrow-band radar simulations.

```
scipy.signal.gausspulse(t, fc=1000, bw=0.5, bwr=-6, tpr=-60, retquad=False, retenv=False)
```

- **Parameters**:
  - `t`: Time array.
  - `fc`: Center frequency of the Gaussian pulse.
  - `bw`: Fractional bandwidth (typically `0.5`).
  - `bwr`: Bandwidth reference level (in dB, commonly `-6 dB`).
  - `retquad`: Return quadrature (complex) component if `True`.
  - `retenv`: Return envelope if `True`.

---

3. Square Wave (`square`)

Generates a square waveform, useful in digital signal simulations, PWM applications, and modulation experiments.

```
scipy.signal.square(t, duty=0.5)
```

- **Parameters**:
  - `t`: Time array.
  - `duty`: Duty cycle, ratio of pulse duration to total period (0 to 1).

---

4. Sawtooth Wave (`sawtooth`)

Generates a sawtooth waveform, widely used in signal synthesis and electronics simulations.

```
scipy.signal.sawtooth(t, width=1)
```

- **Parameters**:
  - `t`: Time array.
  - `width`: Fraction of the waveform period occupied by the rising ramp (0 to 1). Setting `width=1` yields a fully rising ramp.

---

5. Unit Impulse (`unit_impulse`)

Generates a discrete-time impulse (Dirac delta function), fundamental for impulse response analysis.

```
scipy.signal.unit_impulse(shape, idx=None, dtype=float)
```

- **Parameters**:
  - `shape`: Shape of the output array.
  - `idx`: Index of the impulse position (default is 0).
  - `dtype`: Data type of the output array.

How Do We See Frequencies in Data?

Fourier Transform

Fourier transform provide the perfect way to convert the observed time series into the dual space--Frequency domain. Its definition can be written as follows

$$X(f) = \int_{-\infty}^{+\infty} x(t) e^{-2\pi i f t} \, \mathrm{d}t$$

Correspondingly, the inverse (continuous) Fourier transform can be given as:

$$x(t) = \int_{-\infty}^{+\infty} X(f) e^{2\pi i f t} \mathrm{d}f$$

However, a physical sample can only cover at discrete time nodes. Thus, ***Discrete-Time Fourier Transform (DTFT)*** presents an alternative expression in:

$$X(f) = \sum_{n=-\infty}^{+\infty} x[n\Delta t] \cdot e^{-i2\pi f(n\Delta t)}$$

where $x[n] = x(n\Delta t)$ stands for a discrete signal and T is the sampling period. This signal has infinite length and still unrealistic. For a finite signal, ***Discrete Fourier Transform (DFT)*** is the only one that applicable:

$$
\begin{aligned}
X[k] = X(k\Delta f) &= \sum_{n=0}^{N} x(n\Delta t) e^{-2\pi i k \Delta f t}\, \Delta t \\
&= \sum_{n=0}^{N} x[n] e^{-2\pi i k \Delta f t}\, \Delta t
\end{aligned}
$$

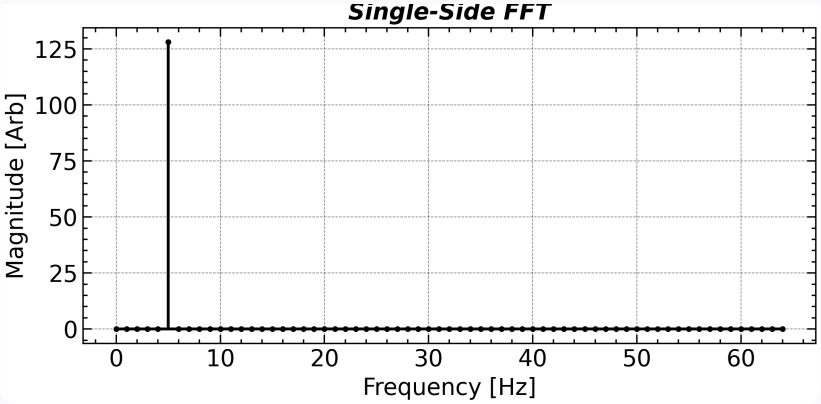
Ideally, according to the periodicity of $e^{-2\pi i f t}$, the DFT actually calculate the DTFT coefficients by extending the original series along and anti-along the time axis.

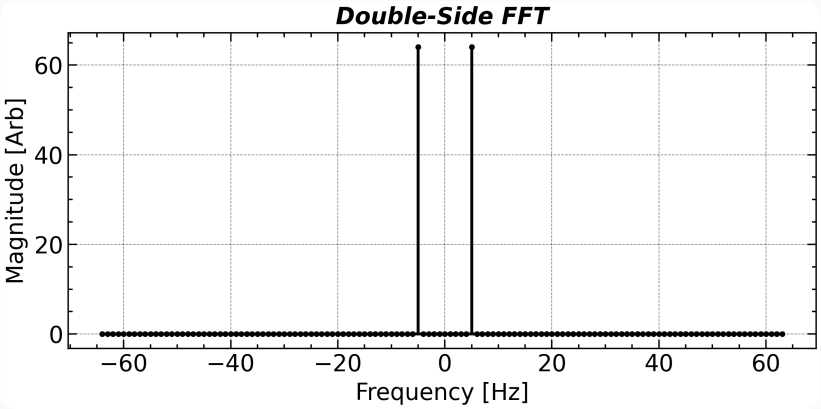
$$
\begin{aligned}
X[k] &= \lim_{M \to +\infty} \frac{1}{2M} \sum_{n=-(M-1)\times N}^{M\times N} x[n] e^{-2\pi i k t/T}\, \Delta t \\
&\propto \sum_{n=-\infty}^{+\infty} x[n] e^{-2\pi i k t/T}\, \Delta t
\end{aligned}
$$

It is worth noting that, Δt is always taken as unity so that the expressions of both DTFT and DFT can be largely simplified as

$$
\begin{aligned}
X(f) &= \int_{-\infty}^{+\infty} x(t) \cdot e^{-i2\pi f t} \mathrm{d}t \\
X(f) &= \sum_{n=-\infty}^{+\infty} x[n] \cdot e^{-i2\pi f n} \\
X[k] &= \sum_{n=0}^{N} x[n] \cdot e^{-2\pi i k n}
\end{aligned}
$$

in most other tutorial. Nevertheless, this tutorial will keep that term as the constant coefficient matters in the real application—The absolute value matters!





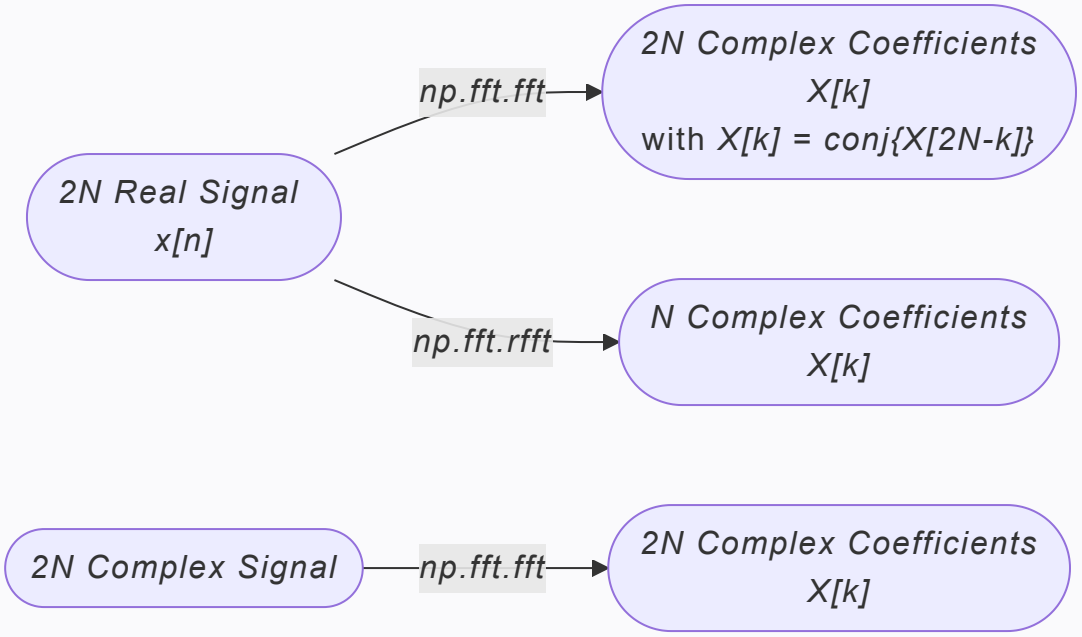
With `numpy`, you can implement DFT through `numpy.fft.fft`:

```
coef = np.fft.fft(sig)
# Corresponding frequency with both zero, positive, and negative frequency
```

Given a window length n and a sample spacing `dt` (i.e., `np.fft.fftfreq(N, dt)`):

```
freq = np.fft.fftfreq(coef.size, dt)
# [0, 1, ...,   N/2-1,     -N/2, ..., -1] / (dt * N)   # if n is even
# [0, 1, ..., (N-1)/2, -(N-1)/2, ..., -1] / (dt * N)   # if n is odd
```

The size of the coefficients is `N` and each coefficient consist of both its real and imaginary parts, which means a `2N` redundancy. That is because `numpy.fft.fft` is designed for not only the real input but also the complex inputs, which can actually represents `2N` variables with a signal size of `N`.



For real input, the aforementioned `2N` redundancy allows you to get that `freqs[1 + i]` = `freqs[-i].conj` and therefore simplify the frequency spectrum but only adopt the positive frequency component.

```
freq[::n // 2]
# [0, 1, ...,   N/2-1] / (DT * N)   if n is even
# [0, 1, ..., (N-1)/2] / (DT * N)   if n is odd
```

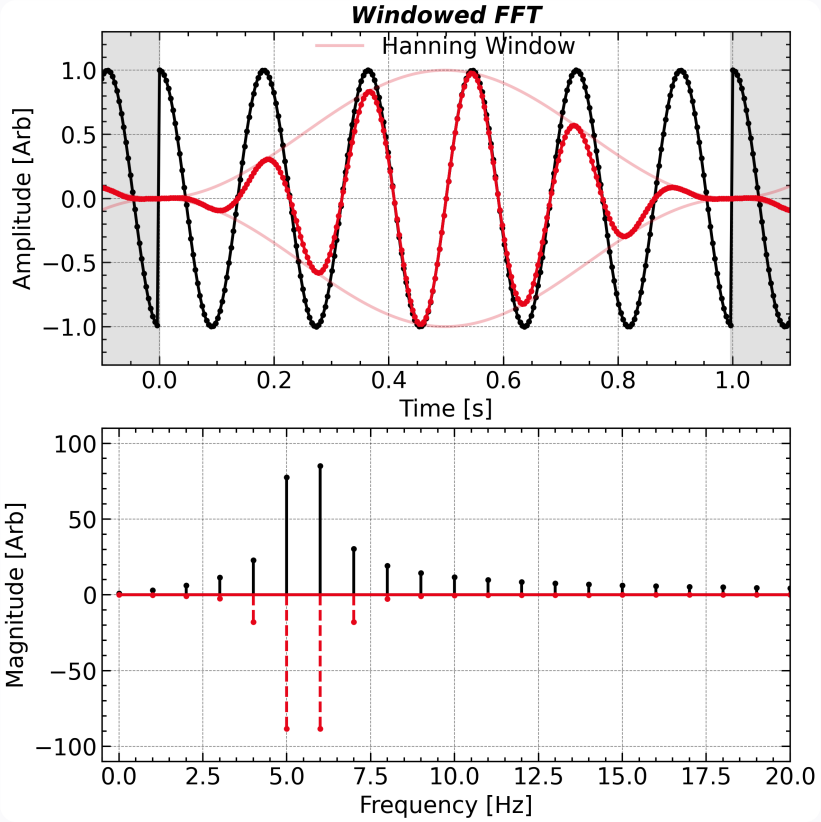
Or, a more suggested way to use `numpy.fft.rfft` (`numpy.fft.rfftfreq`) instead of `numpy.fft.fft` (`numpy.fft.fftfreq`), which is only designed for real input and intrinsically truncate the output coefficients and frequencies.

```
coef = np.fft.rfft(sig)
freq = np.fft.rfftfreq(coef.size, dt)
```

Yet, please remember that only real signal can be used as an input of `numpy.fft.rfft` otherwise the imaginary parts are ignored by default.

Windowing Effect

When performing spectral analysis using the DFT, we implicitly assume that the finite-duration signal is periodically extended. However, if the total sampling duration does not exactly match an integer multiple of the signal's intrinsic period, a mismatch arises between the first and last sample points. This mismatch is interpreted by the DFT as a sharp discontinuity—or a jump—at the signal boundary.



This artificial discontinuity introduces **spectral leakage**, causing energy from specific frequency components to spread into adjacent frequencies, thereby distorting the true spectral content. To mitigate this issue, a **window function**—typically denoted $\mathcal{W}(t)$—is applied to taper the edges of the signal, reducing the contribution of the jump and suppressing leakage.

Typically, the wave function is a bell-shaped function with all non-negative values. Different window functions (e.g., rectangular, Hamming, Hanning, Blackman) offer different trade-offs between **frequency resolution** (main-lobe width) and **leakage suppression** (side-lobe attenuation). Choosing the

appropriate window is essential for ensuring accurate and interpretable spectral results.

Here after, we are going to give an example of window function, the Hanning window, which is written as:

$$w(x) = \frac{1}{2}[1 - \cos(2\pi x)]$$
$$w[n] = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi n}{N}\right)\right]$$

It can be implemented in `numpy` as follow:

```
sig *= np.hanning(sig.size)
```

However, applying a window modifies the signal's amplitude and energy characteristics. This can introduce ambiguity when interpreting the resulting spectrum or comparing analyses across different window shapes. To ensure physical and quantitative consistency, **normalization** of the window function is often necessary. **Amplitude normalization** ensures that the peak value of the window is one, preserving the local signal scale. **Energy normalization** adjusts the window so that the total energy of the signal—defined as the sum of squared values—remains unchanged after windowing. The choice of normalization method depends on the analytical goals, and plays a crucial role in ensuring accurate and meaningful spectral results.

```
# With Normalization
sig *= np.hanning(sig.size) * np.sqrt(8 / 3)
```

it has an average amplitude of $1/2 = \int_0^1 w(x)\mathrm{d}x$ and average energy of

$$\int_0^1 w^2(x)\mathrm{d}x = \frac{1}{4}\left\{\int_0^1 [1 - 2\cos(2\pi x) + \cos^2(2\pi x)]\mathrm{d}x\right\} = \frac{1}{4}(1 - 0 + \frac{1}{2}) = \frac{3}{8}$$

Name and Function / NumPy/SciPy Function	w[n]	Amplitude Normalization	Power Normalization
Rectangular (Boxcar) / `np.ones(N)` or `scipy.signal.windows.boxcar(N)`	$w[n] = 1$	1	1
Hann (Hanning) / `np.hanning(N)` or `scipy.signal.windows.hann(N)`	$w[n] = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right)$	$\frac{1}{2}$	$\sqrt{\frac{3}{8}}$
Hamming / `np.hamming(N)` or `scipy.signal.windows.hamming(N)`	$w[n] = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right)$	0.54	$\sqrt{0.397}$
Blackman / `np.blackman(N)` or `scipy.signal.windows.blackman(N)`	$w[n] = 0.42 - 0.5\cos\left(\frac{2\pi n}{N-1}\right) + 0.08\cos\left(\frac{4\pi n}{N-1}\right)$	0.42	$\sqrt{0.274}$
Kaiser / `scipy.signal.windows.kaiser(N, beta)`	$w[n] = \frac{I_0\left(\beta\sqrt{1-\left(\frac{2n}{N-1}-1\right)^2}\right)}{I_0(\beta)}$	$\frac{1}{2I_0(\beta)}\int_{-1}^{1} I_0\left(\beta\sqrt{1-x^2}\right)dx$	$\sqrt{\frac{1}{2}\int_{-1}^{1}\left[\frac{I_0\left(\beta\sqrt{1-x^2}\right)}{I_0(\beta)}\right]^2 dx}$
Tukey / `scipy.signal.windows.tukey(N, alpha)`	$w[n] = 0.5\left(1 + \cos\left(\frac{\pi(2n)}{\alpha N} - 1\right)\right)$ (for edges)	$1 - \frac{\alpha}{2}$	$\sqrt{1 - \frac{\alpha}{2} + \frac{\alpha}{4}}$
Gaussian / `scipy.signal.windows.gaussian(N, std)`	$w[n] = \exp\left(-\frac{1}{2}\left(\frac{n - \frac{N-1}{2}}{\sigma\frac{N-1}{2}}\right)^2\right)$	$\sigma\sqrt{\frac{\pi}{2}}$	$\sigma\sqrt{\frac{\pi}{4}}$
Bartlett / `np.bartlett(N)` or `scipy.signal.windows.bartlett(N)`	$w[n] = 1 - \frac{2\left\lvert n - \frac{N-1}{2}\right\rvert}{N-1}$	$\frac{1}{2}$	$\sqrt{\frac{1}{3}}$

Definitions

- **Amplitude Normalization (Coherent Gain)**: $\langle w[n] \rangle$ - preserves amplitude of coherent signals (DC component)
- **Power Normalization (Energy)**: $\langle w^2[n] \rangle$ - preserves power of incoherent signals (noise)
- Use amplitude normalization for spectral analysis of tones/periodic signals
- Use power normalization for power spectral density estimation of random signals

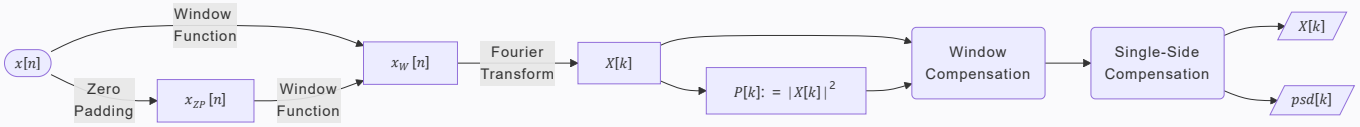
Analytic Values

- **Hamming coefficients**: $0.54 = \frac{25}{46} \approx 0.5435$, $0.46 = \frac{21}{46} \approx 0.4565$
- **Blackman coefficients**: $0.42 = \frac{21}{50}$, $0.5 = \frac{1}{2}$, $0.08 = \frac{2}{25}$
- **Power normalization factors**:
  - Hamming: $\sqrt{0.397} = \sqrt{\frac{25^2+21^2}{2 \cdot 46^2}} = \sqrt{\frac{1066}{4232}} \approx 0.630$
  - Blackman: $\sqrt{0.274} = \sqrt{\frac{21^2+25^2+4^2}{2 \cdot 50^2}} = \sqrt{\frac{1066}{5000}} \approx 0.462$
- $\text{mean} w[n] = \frac{1}{N}\sum_{n=0}^{N-1} w[n]$ and $\text{mean} w[n]^2 = \frac{1}{N}\sum_{n=0}^{N-1} w[n]^2$
- Kaiser and Gaussian expressions involve integrals that depend on shape parameters

- For Tukey: α is the cosine-tapered fraction $(0 \leq \alpha \leq 1)$
- For Kaiser: I_0 is the modified Bessel function of the first kind
- Gaussian approximations assume the window is appropriately scaled

As the magnitude and power spectra have different normalization factors, it is suggested that apply the normalization before the data outputting/plotting but not immediately after you proceed the Fourier transform.

In the end, we show the complete data processing diagram for ***windowed Fourier transform (WFT)***.



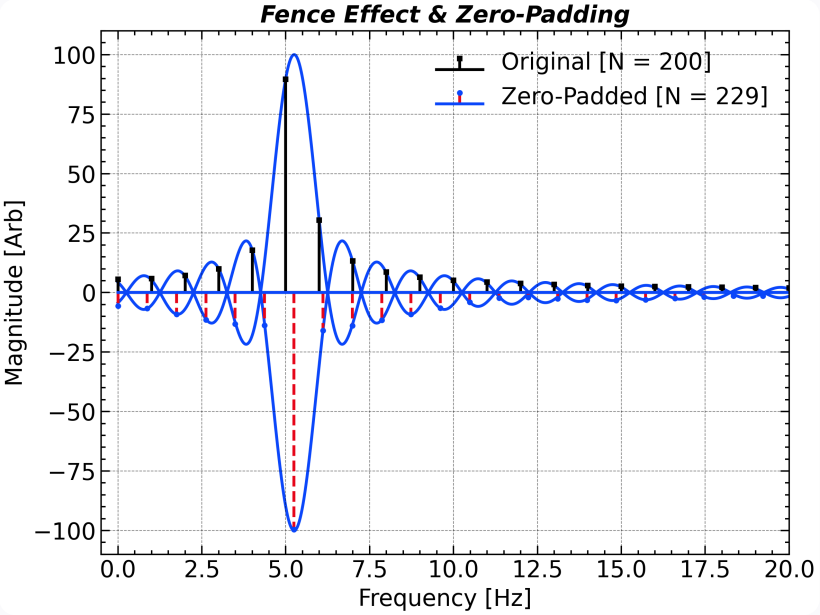
Fence Effect and Zero-Padding

To further improve the interpretability of spectral results, addressing spectral leakage alone is not enough. Another source of distortion arises from the discretization of the frequency axis itself. When a signal's frequency component does not align exactly with the frequency bins defined by the DFT, its energy spreads into neighboring bins—a phenomenon known as the **fence effect**. To reduce this effect and achieve smoother spectral representations, we often apply a technique known as **zero-padding**, which is discussed below.

When using the Discrete Fourier Transform (DFT), we effectively project the signal onto a set of discrete frequency bins. If the actual frequency of a signal component does not align precisely with one of these bins, the energy spreads across multiple nearby bins—an artifact known as the **fence effect**. This leads to inaccurate spectral peak positions and broadening, especially when analyzing short-duration signals or signals with non-integer frequency components.

A common technique to mitigate the visual and analytical impact of the fence effect is **zero-padding**—extending the time-domain signal by appending zeros beyond its original length. While zero-padding does not increase the inherent frequency resolution, it interpolates the spectrum between the original DFT bins, producing a smoother and more detailed frequency-domain representation. This can help in better locating spectral peaks and distinguishing closely spaced features.

Zero-padding is particularly useful in peak detection, cross-spectral analysis, and visualization, where enhanced frequency granularity improves interpretability even though it doesn't extract new information from the signal itself.




```
n_padding = 29

coef = np.fft.rfft(sig, n = signal.size + n_padding)
freq = np.fft.rfftfreq(coef.size, dt)
```

What Else Should You Know About the DFT and FFT?

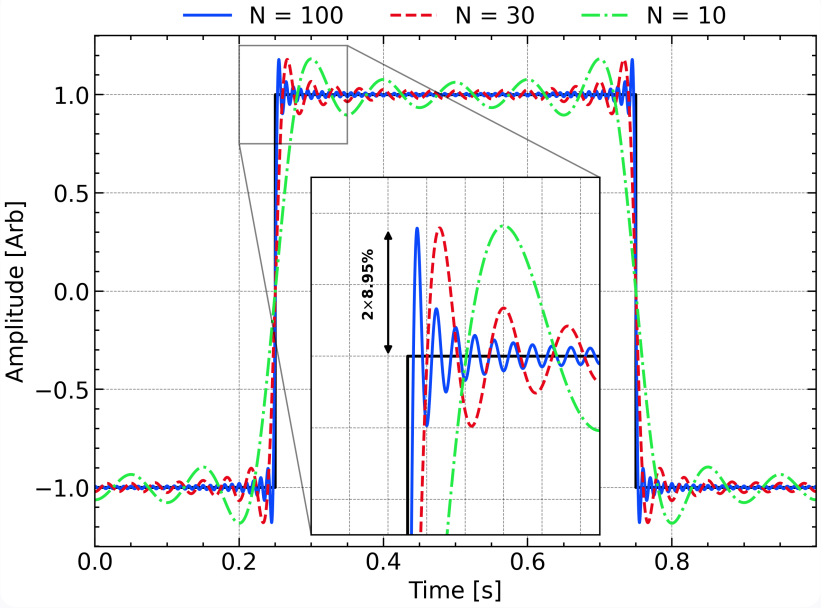
Gibbs Phenomenon

Although a discrete signal can be losslessly transformed using the Fourier transform, it still faces limitations when attempting to represent continuous functions with sharp discontinuities. A continuous function with an infinitely steep jump introduces an infinite derivative at the discontinuity. However, any finite, discrete sampling of that function cannot fully capture its high-frequency behavior near the jump.

From the perspective of the Fourier operator , the discontinuity is perceived as two adjacent sample points with a large difference, and a finite Fourier series will attempt to approximate this using a limited set of basis sinusoids. This leads to **overshooting and ringing** near the discontinuity—an effect known as the **Gibbs phenomenon**.

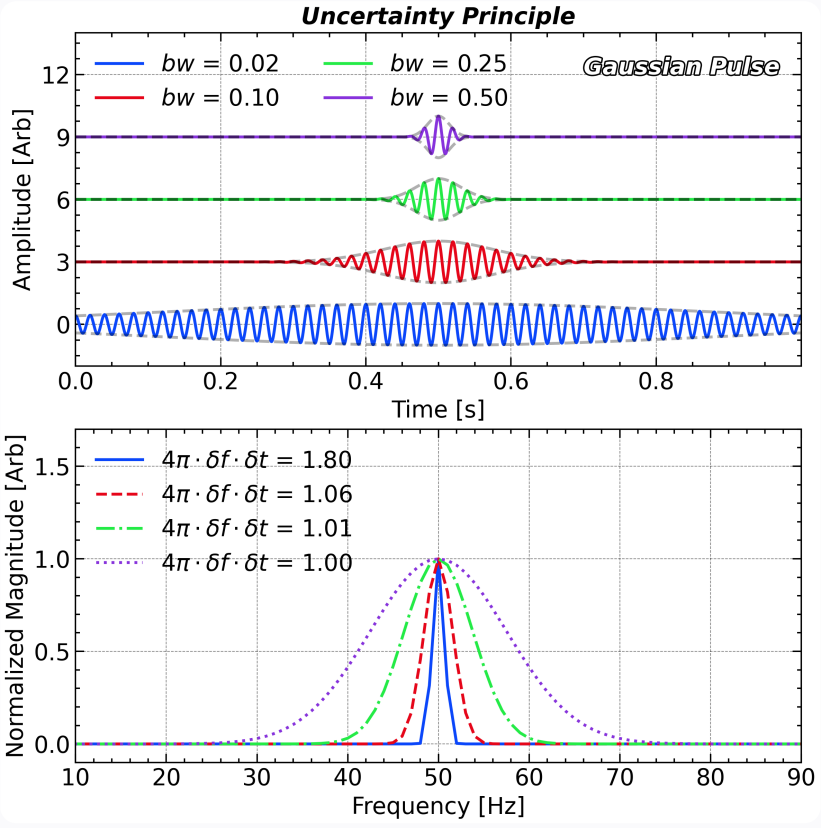
Mathematically speaking, even though the Fourier transform converges to the correct function almost everywhere, at the discontinuity itself it overshoots by a fixed percentage (~9%) regardless of how many harmonics are included. The oscillatory artifacts do not vanish with more terms; they merely become narrower and more localized.

The Gibbs phenomenon is not a numerical artifact but an intrinsic property of the Fourier basis when approximating non-smooth functions. In practical signal processing, techniques such as windowing, filtering, or switching to other basis functions (e.g., wavelets) are used to reduce its visible impact.



Uncertainty Principle

In



Parseval's Theorem and Energy Conservation

Parseval's Theorem for CFT:

$$\int_{-\infty}^{\infty} x^2(t)\,dt = \int_{-\infty}^{\infty} X^2(f)\,df$$

for DFT:

$$\sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N}\sum_{k=0}^{N-1} |X[k]|^2$$

In the physical world, the square power of the amplitude often refers to some kind of ***energy*** or ***power***. For example, the square of the displacement (x) of a spring, x^2 is proportional to the elastic potential energy ($kx^2/2$, where k describes the stiffness). In plasma physics, electromagnetic field contains the energy density (u) written as

$$u = u_E + u_B = \frac{1}{2}(\varepsilon_0 E^2 + \frac{1}{\mu_0} B^2)$$

In this case, the ***energy*** of the signal naturally linked with the ***energy*** of the electromagnetic field. Nevertheless, the energy of a signal is an extensive property as it linearly increases with the length of the sample. In the ordinary investigation, the signal energy is always further converted as signal ***power***, which is an intensive property that describe the amplitude and is independent of signal length. The defition of power, *P*, can be written as:

$$P = \frac{1}{T}\int_{-T/2}^{T/2} |x(t)|^2 \mathrm{d}t$$

or

$$
\begin{aligned}
P &= \frac{1}{N\Delta t}\sum_{n=0}^{N-1}|x(n\Delta t)|^2\Delta t \\
&= \frac{1}{N^2\Delta f}\sum_{k=0}^{N-1}|X(k\Delta f)|^2\Delta f \\
&= \sum_{k=0}^{N-1}\boxed{\frac{1}{Nf_s}|X(k\Delta f)|^2}\Delta f \\
&= \sum_{k=0}^{N-1} PSD[k]\Delta f
\end{aligned}
$$

for DFT. Considering that DFT yields both positive and negative frequency, we typically fold the DFT result. Naturally, the definition of *power spectral density (PSD)* is given as:

$$
\begin{aligned}
\sum_{k=0}^{N-1} PSD[k\Delta f]\Delta f = \\
\text{For Even } N: \ \Delta f\left[PSD[0] + \sum_{k=1}^{N/2-1} 2\cdot PSD[k\Delta f] + PSD[f_{N/2}]\right] \\
\text{For Odd } N: \ \Delta f\left[PSD[0] + \sum_{k=1}^{(N-1)/2} 2\cdot PSD[k\Delta f]\right]
\end{aligned}
$$

$PSD[0]$ represents the DC component and is ignored in the spectral analysis for the most(but not all) time.

```
N = coef.size
fs = 1 / dt
psd = (np.abs(coef) ** 2) / (N * fs)

if N % 2 == 0:
    psd[1:-1] *= 2
else:
    psd[1:] *= 2
```

According to the lineairty of $\mathcal{F}$, $X[k]$ should also be proportional to the signal amplitude. Easily catch that the coefficient at the exact wave frequency has the form of

$$|X[k]| = \frac{1}{2} A_k \cdot f_s \cdot T$$

1/2 in this equation arises from the fact that $\int_0^{2\pi} \sin^2 x \mathrm{d}x = 1/2$.

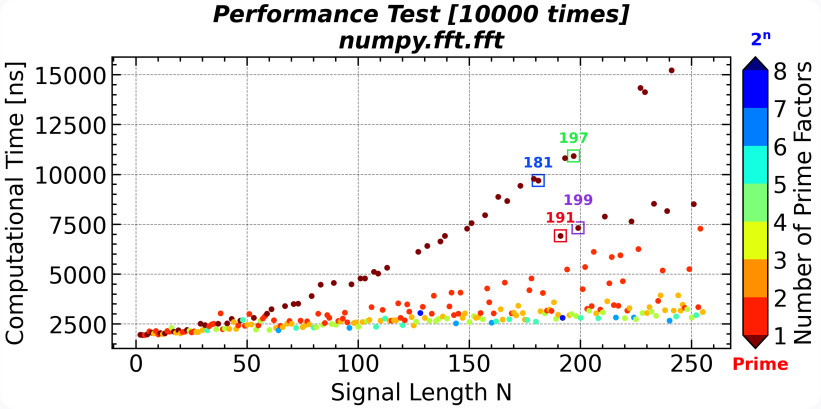
The performance of `numpy.fft.fft` and `scipy.signal.fft`

The invention of the ***(Cooley–Tukey) Fast Fourier Transform (FFT) algorithm*** reduced the time complexity of DFT from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$ by efficiently decomposing the DFT into smaller computations, i.e., 🔗divide-and-conquer.

Most tutorials introduce the ***radix-2*** FFT, which splits the signal into ***two*** sub-signals with exactly the same length and requires the length of the signal to be an integer power of ***2***. This requirement is hard to satisfy in common applications without zero-padding, which actually includes unwanted modification of the original signal. To overcome that, ***radix-3*** and ***radix-5*** FFTs are developed and implemented.

Still, the divide-and-conquer strategy fails when the signal length *N* consists of at least one big prime number factor (e.g, 10007) as the signal is hard to split. In that situation, the ***Bluestein's algorithm***, which is essentially a ***Chirping-Z transform***, is used. This algorithm takes the $\mathcal{F}$ operation as a convolution and then uses the *convolution theorem* in the calculation of DFT

coefficients. The convolution property allows us to extend the signal length to a proper, highly composite number with zero-padding (denoted as *M*), but the coefficients and frequency resolution remain unchanged. The final time complexity of *Bluestein's algorithm* goes to $\mathcal{O}(N + M\log M)$, where the first term originates from the iterate all the frequency component.



From the performance test, we observe that signals with prime-number lengths (dark red dots) often incur higher computational costs. For example:

$$N = 181 = 182 - 1 = 2^1 \times \boxed{7^1 \times 13^1} - 1$$
$$N = 197 = 198 - 1 = 2^1 \times 3^2 \times \boxed{11^1} - 1$$

In contrast, signals with highly composite number lengths (dark blue dots), such as those with lengths being integer powers of 2, usually have the lowest computation time.

However, some prime numbers like:

$$N = 191 = 192 - 1 = 2^6 \times 3^1 - 1$$
$$N = 199 = 200 - 1 = 2^3 \times 5^2 - 1$$

can also exhibit relatively efficient performance due to their proximity to highly factorable numbers.

Modern implementation of the FFT algorithm, such as `pocketfft`, combines the above two methods (*Cooley–Tukey* and *Bluestein*). This *C++* package is used in both `numpy` and `scipy(1.4.0+)` for their FFT implementation. Besides, `fftw`, which stands for the somewhat whimsical title of *"Fastest Fourier Transform in the West"*, is also very popular and used in the `fft/ifft` functions of *MATLAB*. Its *Python* implementation can be found in the `pyfftw` package.

The `scipy.signal.fft` additionally provides an input parameter `workers: int, optional` to assign the maximum number of workers to use for parallel computation. If negative, the value wraps around from `os.cpu_count()`. For parallel computation, you need to input a batch of signals with shape of $N \times K$.

***Reference**:*

1. Cooley, James W., and John W. Tukey, 1965, “An algorithm for the machine calculation of complex Fourier series,” Math. Comput. 19: 297-301.
2. Bluestein, L., 1970, “A linear filtering approach to the computation of discrete Fourier transform”. IEEE Transactions on Audio and Electroacoustics. 18 (4): 451-455.
3. https://dsp.stackexchange.com/questions/24375/fastest-implementation-of-fft-in-c
4. https://www.fftw.org/

Sliding Window

`numpy.lib.stride_tricks.sliding_window_view(x, window_shape, axis=None, *, subok=False, writeable=False)` provides the function for re-organizing the signal into several sub-chunk. This function can only give a stride of one. For a customized stride, you need to use `numpy.lib.stride_tricks.as_strided(x, shape=None, strides=None, subok=False, writeable=True)`. This function can be unsafe and crash your program.

The `bottleneck` package, which is safer and more efficient, is more suggested for common usage of moving windows, like moving-average and moving-maximum. The following code shows how to use the `bottleneck` functions and their expected results.

Properties of Fourier Transform

A super powerful property of Fourier transform is that:

$$\mathcal{F}\left[\frac{\mathrm{d}}{\mathrm{d}t}x(t)\right] = (i2\pi f)\cdot X(f)$$

which can be easily proved by doing derivative to the both sides of the inverse Fourier transform:

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}[x(t)] &= \int_{-\infty}^{+\infty} X(f)(i2\pi f)e^{i2\pi ft}\mathrm{d}f \\ &= \int_{-\infty}^{+\infty} [(i2\pi f)\cdot X(f)]e^{i2\pi ft}\mathrm{d}f \\ &= \mathcal{F}^{-1}\left[(i2\pi f)\cdot X(f)\right]\end{aligned}$$

It can be denoted as

$$\mathrm{d}/\mathrm{d}t \leftrightarrow i2\pi f$$

One can also extend this property to

$$(\mathrm{d}/\mathrm{d}t)^n \leftrightarrow (i2\pi f)^n$$

In plasma physics, the conventional way to express the electromagnetic field.

It should be noted that this derivation property change a little bit for discrete Fourier transform:

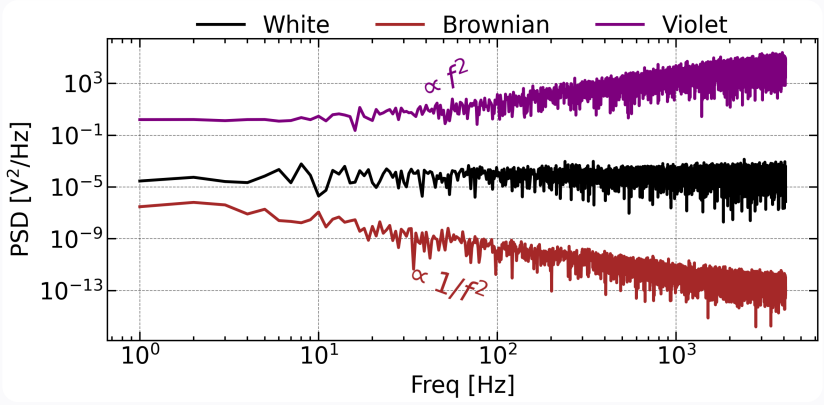
$$\frac{\Delta x(t)}{\Delta t} = \int_{-\infty}^{+\infty} X(f) e^{2\pi i f t} \mathrm{d}f$$

Property	Continuous-time FT	Length-N DFT (circular, arbitrary T_s)
Linearity	$\mathcal{F}\{a\,x + b\,y\} = a\,X + b\,Y$	$a\,x[n] + b\,y[n] \xrightarrow{\text{DFT}} a\,X[k] + b\,Y[k]$
Time shift	$x(t - t_0) \;\longrightarrow\; e^{-j2\pi f t_0} X(f)$	$x[(n - n_0) \bmod N] \xrightarrow{\text{DFT}} e^{-j2\pi k n_0/N}\, X[k]$ ($\Delta t = n_0 T_s$)
Frequency shift / Modulation	$e^{j2\pi f_0 t} x(t) \;\longrightarrow\; X(f - f_0)$	$e^{j2\pi f_0 n T_s} x[n] \xrightarrow{\text{DFT}} X[(k - k_0) \bmod N]$, $k_0 = \dfrac{f_0 N}{f_s}$
Time scaling	(x(a t);\longrightarrow;\tfrac1{	a
Conjugation & Symmetry	Real $x(t) \Rightarrow X(-f) = X^*(f)$	Real $x[n] \Rightarrow X[(N - k) \bmod N] = X^*[k]$
Convolution theorem	$h = x * y \;\Rightarrow\; \mathcal{F}\{h\} = X\,Y$	Circular convolution: $x \circledast y \xrightarrow{\text{DFT}} X[k]\,Y[k]$
Correlation theorem	$h = x \star y \;\Rightarrow\; \mathcal{F}\{h\} = X^*\,Y$	Circular correlation: $x \star y \xrightarrow{\text{DFT}} X^*[k]\,Y[k]$
Differentiation / Multiplication	$\frac{d^n x}{dt^n} \;\longrightarrow\; (j2\pi f)^n X(f)$	Multiply $X[k]$ by $(j2\pi k f_s/N)^n$; conversely, $n T_s\, x[n]$ maps to a discrete derivative of $X[k]$.
Parseval / Plancherel	(\displaystyle\int_{-\infty}^{\infty}	x(t)
Duality & Inversion	$\mathcal{F}^2\{x\} = x(-t),\;\; \mathcal{F}^4 = \mathrm{Id}$	IDFT = DFT with sign reversal + $1/N$; two successive DFTs yield $N\,x[(-n) \bmod N]$.

How to Deal With Noise?

What is Noise?

Noise refers to random or unwanted fluctuations that obscure the true underlying signal in your data. In spectral analysis, understanding the properties and sources of noise is crucial for interpreting results, estimating signal-to-noise ratio (SNR), and designing effective filtering or denoising strategies. In plasma physics, the noise originates from both physical (e.g., plasma turbulence) and non-physical process (e.g., measurement uncertainty).



In audio engineering, electronics, physics, and many other fields, the color of noise or noise spectrum refers to the power spectrum of a noise signal (a signal produced by a stochastic process). Different colors of noise have significantly different properties. For example, as audio signals they will sound different to human ears, and as images they will have a visibly different texture. Therefore, each application typically requires noise of a specific color. This sense of 'color' for noise signals is similar to the concept of timbre in music (which is also called "tone color"; however, the latter is almost always used for sound, and may consider detailed features of the spectrum).

The practice of naming kinds of noise after colors started with white noise, a signal whose spectrum has equal power within any equal interval of frequencies. That name was given by analogy with white light, which was (incorrectly) assumed to have such a flat power spectrum over the visible range. Other color names, such as pink, red, and blue were then given to noise with other spectral profiles, often (but not always) in reference to the color of

light with similar spectra. Some of those names have standard definitions in certain disciplines, while others are informal and poorly defined. Many of these definitions assume a signal with components at all frequencies, with a power spectral density per unit of bandwidth proportional to $1/f^{\beta}$ and hence they are examples of power-law noise. For instance, the spectral density of white noise is flat (β = 0), while flicker or pink noise has β = 1, and Brownian noise has β = 2. Blue noise has β = -1.

Signal-to-Noise Ratio and Decibel

Decibel (dB, Deci-Bel) is frequently used in describing the intensity of the signal. This quantity is defined as the

Decibel	0	1	3	6	10	20
Energy Ratio	1	1.12	1.41	2.00	3.16	10
Amplitude Ratio	1	1.26	2.00	3.98	10	100

Due to the fact that $2^{10} \approx 10^3$, 3 dB corresponds to a energy ratio of $10^{3/10} = \sqrt[10]{1000} \approx \sqrt[10]{1024} = 2$.

The adoption of decibel instead of the conventional physical unit has three advantage:

- It allows the direction addition when compare the amplitude of the signal.
- When you are not confident about the magnitude of the uncalibrated data, you can just use dB to describe the ambiguous intensity.
- The 🔗***Weber–Fechner law*** states that human perception of stimulus intensity follows a logarithmic scale, which is why decibels—being logarithmic units—are used to align physical measurements with human sensory sensitivity, such as in sound and signal strength.

Artificial Noise Generation

Method 1: Approximate $\mathrm{d}x/\mathrm{d}t$ by $\Delta x/\Delta t$

According to the property of Fourier transform, the convolution in the .

```
time = np.linspace(0, 1, 10000, endpoint=False)
dt = time[1] - time[0]

white_noise = np.random.randn(time.size)
brownian_noise = np.cumsum(np.random.randn(time.size)) * dt
violet_noise = np.diff(np.random.randn(time.size + 1)) / dt
```

Method 2: Rescale the frequency spectrum of the white noise [Suggested]

```
time = np.linspace(0, 1, 10000, endpoint=False)
dt = time[1] - time[0]
fs = 1 / dt
freq = np.fft.rfftfreq(len(time), dt)

brownian_noise_fft = np.fft.rfft(np.random.randn(time.size))
brownian_noise_fft[1:] /= freq[1:] ** 1
brownian_noise_fft[0] = 0
brownian_noise = np.fft.irfft(brownian_noise_fft)

violet_noise_fft = np.fft.rfft(np.random.randn(time.size))
violet_noise_fft[1:] /= freq[1:] ** -1
violet_noise_fft[0] = 0
violet_noise = np.fft.irfft(violet_noise_fft)
```

Besides these two methods, one can also get colored noise by filtering white noise. A colored noise that accurately follows its expected power spectrum requires the order of the filter to be high enough. Even though this

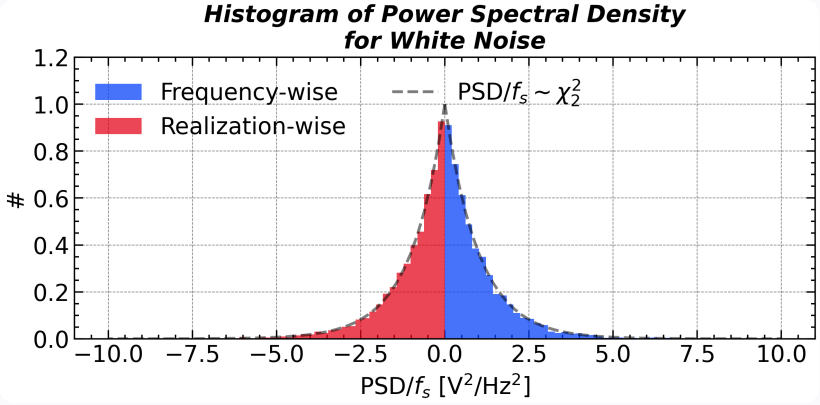
"Noise" of Noise

From the power spectra of noises, one can see that the PSD of the generated noise may randomly deviates from the theoretical expectation, i.e., the exactly power-law PSD.

The Fourier coefficient computed as

$$X[k] := \sum_{0}^{N-1} x[n] \mathrm{e}^{i2\pi nk}$$

can be deemed as a **weighted summation** of the signal $x[n]$. When $x[n]$ are independent identically distributed (*i.i.d*) random variables, their weighted summation approaches the Normal distribution when *N* is large enough, according to the ***Central Limit Theorem***. Thus, the *PSD*, defined as the square sum of the real and imaginary parts, naturally follows the *Kappa* Distribution with the freedom of 2. The above statement requires that he real and imaginary parts are independent to each other, which can be proved by calculating their covariance.



It should be noted that the periodic signals like $\sin\omega t$ are not *i.i.d*. These signals are not even *independent*, which means that even the **Lindeberg (-Feller) CLT** can not guarantee their Fourier coefficients converged to a Normal distribution. Commonly, its *PDF* still follows a bell-shaped curves but

the mean and variance dependent on the *SNR*.

To reduce this kind of uncertainty, we are going to introduce the following three method: 1. Barlett Method; 2. Welch Method; and 3. Blackman–Tukey Method.

Welch Method [`scipy.signal.welch`]

Welch proposed that the averaging the power spectral density instead of the coefficient can largely reduce the flutuation levels of the spectrum. Therefore, we may just get a.

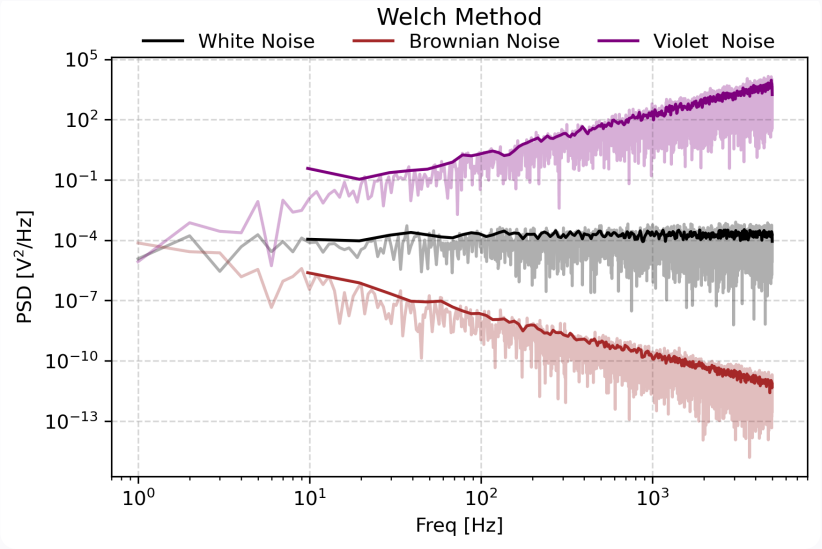
The averaging operation must be taken after the conversion from coefficient to power other wise the averaged coefficients are actually unchanged.

This method can be implemented by `scipy.signal.welch` function:

```
time = np.linspace(0, 1, 10000, endpoint=False)
fs = 1 / (time[1] - time[0])
freq = np.fft.rfftfreq(len(time), time[1] - time[0])

noise_white = np.random.randn(time.size)

coef_white = np.fft.rfft(noise_white, axis=-1).T
psd_white = (np.abs(coef_white) ** 2) / fs / time.size

freq_welch, psd_white_welch = scipy.signal.welch(noise_white, fs, window =
'hann', nperseg=2 ** 10)
```

Except for averaging, one can also choose the median of the PSD across different segements and obtain a less disturbed PSD. This choice can be implemented by `scipy.signal.welch(signal, fs, average = 'median')`. The default parameter for `average` is `mean`, corresponding to the normal Welch method.

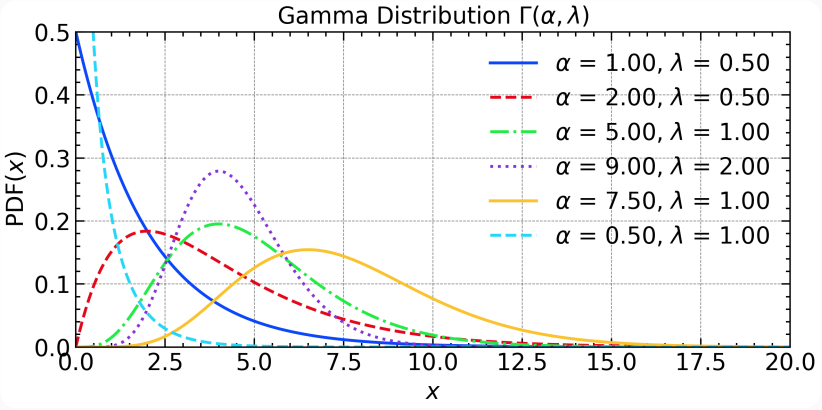
For each segement, you can also chose the window function to reduce the spectral leakage. The result of this method is shown below:



One can also verify that the distribution of the PSD convert to *Gamma* Distribution, which has a ***Probability Density Function (PDF)*** of:

$$PDF(x;\alpha,\lambda) = \frac{\lambda^{\alpha}}{\Gamma(\alpha)} x^{\alpha-1} e^{-\lambda x}$$

The mean and variance of this distribution is α/λ and α/λ^2. When the number of segments (α) decrease/increase to $1/+\infty$, the Gamma distribution degenerate to exponential/normal distribution.



In ***Bartlett Method***, the ratio of `N_STEP` and `N_PER_SEG` is fixed at unity, which means every segement has no overlapping with each other. It can be regarded as a special case of the *Welch Method* while it is actually proposed earlier.

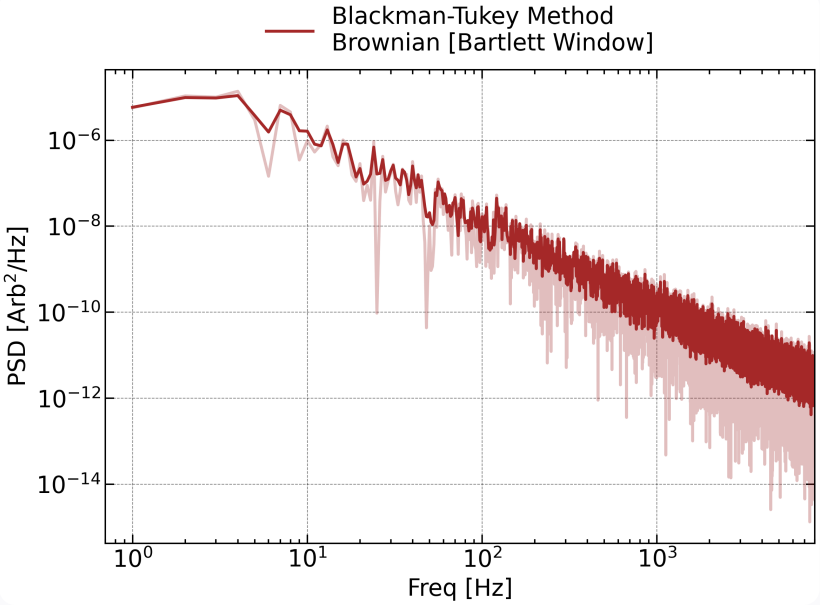
Blackman-Tukey Method

Blackman-Tukey method gives another approach to a high SNR estimation of *PSD* based on the *W.S.S* properties of the signal and *Wiener–Khinchin theorem*. This method consists of three steps:

1. Calculate the (***double-sided***) ACF of the signal
2. Apply a window function to the ACF
3. Do DFT to the windowed ACF.

It should be keep in mind that these methods are all build based on the assumption of wide-sense stationarity of the signal.[Explain WSS here]. A noise signal, no matter its color, is wide-sense stationary. However, a real time series of a physics quantity cannot gurantee its wide-sense stationarity. Since W.S.S is the only presumption of these method, they are also termed ***Nonparametric Estimator***.

Apart from splitting the signal into several segments, one can also downsample the signal and get multiple sub-signal with different startup time. However, the maximum frequency of the yield spectrum will also be reduced by a factor of `N_DOWNSAMPLE`. At the same time, the frequency resolution remains to be $(N\Delta t)^{-1}$.



Signal Over Noise

A signal composed of a deterministic sinusoidal component and additive noise can be written as:

$$x(t) = s(t) + n(t)$$

The Fourier coefficient at frequency f is:

$$\tilde{X}(f) = \tilde{S}(f) + \tilde{N}(f)$$

where $\tilde{S}(f)$ is the deterministic signal component (a fixed complex number), and $\tilde{N}(f)$ is the Fourier transform of the noise. If the noise $n(t)$ is zero-mean wide-sense stationary, then:

$$\tilde{N}(f) \sim \mathcal{CN}(0, \sigma_n^2)$$

That is, $\tilde{X}(f)$ is a complex Gaussian random variable:

$$\tilde{X}(f) \sim \mathcal{CN}(\mu, \sigma_n^2), \quad \mu = \tilde{S}(f)$$

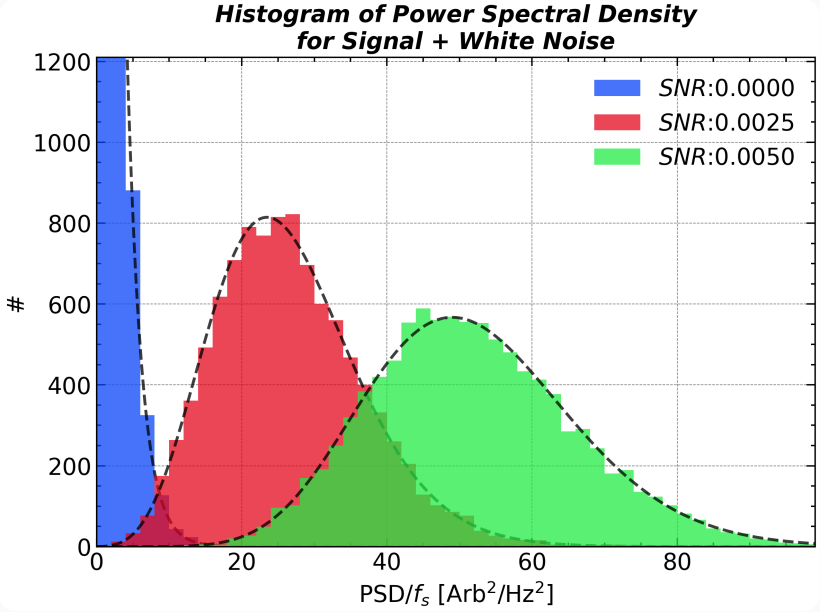
The power spectrum estimate is:

$$\hat{S}_x(f) = |\tilde{X}(f)|^2$$

Since $|\tilde{X}(f)|^2$ is the sum of squares of two independent Gaussian variables (real and imaginary parts), it strictly follows a non-central chi-squared distribution:

$$\hat{S}_x(f) \sim \sigma_n^2 \cdot \chi^2(2, \lambda), \quad \lambda = \frac{|\mu|^2}{\sigma_n^2}$$

In other words, the deterministic signal provides a **complex offset** (mean μ), and the noise determines the **variance** σ_n^2. The resulting power spectrum estimate is exactly a non-central chi-squared distribution with 2 degrees of freedom.



Faulty Sample

Lomb-Scargle Periodogram [*scipy.signal.lombscargle*]

The Lomb-Scargle periodogram is a powerful method for estimating the power spectrum of unevenly sampled time series. Unlike the standard FFT-based periodogram, which requires uniformly spaced data, Lomb-Scargle is widely used in astronomy and geophysics where data gaps are common. This section introduces its mathematical foundation, physical interpretation, and provides practical examples using `scipy.signal.lombscargle`.

The basic idea of the Lomb-Scargle periodogram is to fit the observed time series by a sinusoidal function $A\sin(2\pi ft) + B\cos(2\pi ft)$ with frequency f in the sense of mean square deviation. The yield coefficient A can be a estimate of the signal's magnitude at frequency f.

Minimize the residual sum of squares:

$$\chi^2(A, B) = \sum_{n=1}^{N} \left[x_n - \bar{x} - A\cos(\omega t_n) - B\sin(\omega t_n)\right]^2.$$

Setting $\partial\chi^2/\partial A = \partial\chi^2/\partial B = 0$ yields the normal equations:

$$\begin{pmatrix} \sum \cos^2(\omega t_n) & \sum \cos(\omega t_n)\ \sin(\omega t_n) \\ \sum \cos(\omega t_n)\ \sin(\omega t_n) & \sum \sin^2(\omega t_n) \end{pmatrix} \begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} \sum (x_n - \bar{x})\cos(\omega t_n) \\ \sum (x_n - \bar{x})\sin(\omega t_n) \end{pmatrix}.$$

Define

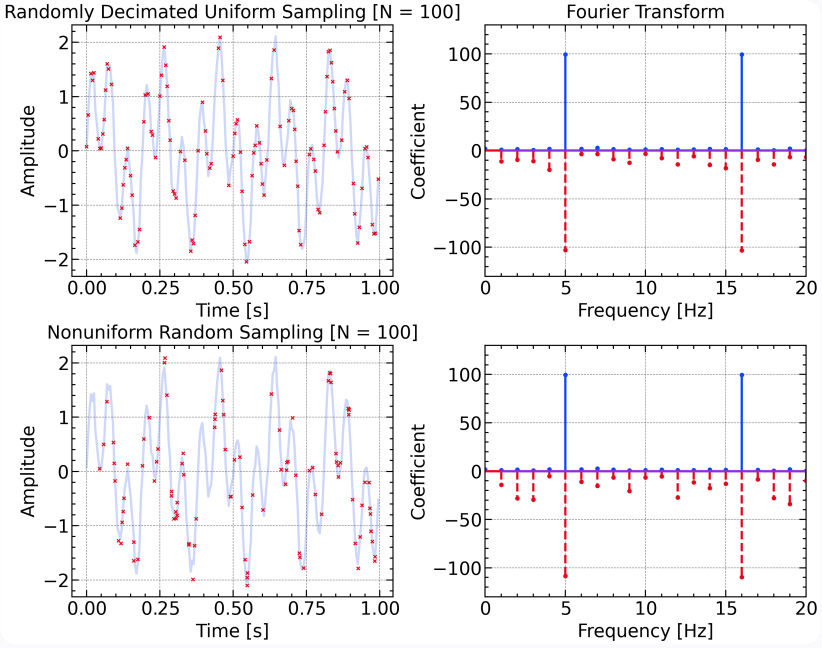
$$
\begin{aligned}
C &= \sum \cos^2(\omega t_n) \\
S &= \sum \sin^2(\omega t_n) \\
D &= \sum \cos(\omega t_n)\sin(\omega t_n) \\
X_c &= \sum (x_n - \bar{x})\cos(\omega t_n) \\
X_s &= \sum (x_n - \bar{x})\sin(\omega t_n)
\end{aligned}
$$

Then

$$
A = \frac{X_c\,S - X_s\,D}{C\,S - D^2}, \quad B = \frac{X_s\,C - X_c\,D}{C\,S - D^2}.
$$

$P(\omega) = \frac{1}{2}\left(A^2 + B^2\right)$.

Substituting the expressions for A and B yields a form that still involves the cross-term D.



Introducing the Phase Offset τ

To eliminate the cross-term, shift the time origin:

$$t'_n = t_n - \tau,$$

and choose τ so that

$$\sum_{n=1}^{N} \sin\left(2\omega t'_n\right) = 0 \quad \Longleftrightarrow \quad \tan(2\omega\tau) = \frac{\sum_{n=1}^{N} \sin(2\omega t_n)}{\sum_{n=1}^{N} \cos(2\omega t_n)}.$$

This makes $\sum \cos(\omega t'_n), \sin(\omega t'_n) = 0$, diagonalizing the normal equations. The power then becomes

$$P(\omega) = \frac{1}{2}\left[\frac{\left[\sum (x_n - \bar{x}) \cos\big(\omega(t_n - \tau)\big)\right]^2}{\sum \cos^2\big(\omega(t_n - \tau)\big)} + \frac{\left[\sum (x_n - \bar{x}) \sin\big(\omega(t_n - \tau)\big)\right]^2}{\sum \sin^2\big(\omega(t_n - \tau)\big)}\right].$$

Compare with the original frequency spectrum, the Lomb-Scargle periodogram contains some irregular frequency leakage. The Lomb-Scargle periodogram finally converge to the Fourier periodogram when the sample time is uniformly distributed.

Correlation Function

> A correlation function is a function that gives the statistical correlation between random variables, contingent on the spatial or temporal distance between those variables. If one considers the correlation function between random variables representing the same quantity measured at two different points, then this is often referred to as an autocorrelation function, which is made up of autocorrelations. Correlation functions of different random variables are sometimes called cross-correlation functions to emphasize that different variables are being considered and because they are made up of cross-correlations. ——Wikipedia

$$R_{XY}(t, t+\tau) := \mathbb{E}\left[X(t)\overline{Y(t+\tau)}\right]$$

where the overline represents the complex conjugate operation when X and Y are complex signal. Specifically, the correlation function between X and itself is called autocorrelation function:

$$R_{XX}(t, t+\tau) := \mathbb{E}\left[X(t)\overline{X(t+\tau)}\right]$$

If X is a wide-sense stationary signal, then $R_{XX}(t_1, t_1+\tau) = R_{XX}(t_2, t_2+\tau)$ for arbitrary t_1, t_2, and τ. Thus, the autocorrelation function can be written as a single-variate function $R_{XX}(\tau) = R_{XX}(t, t+\tau)$.

Wiener–Khinchin theorem

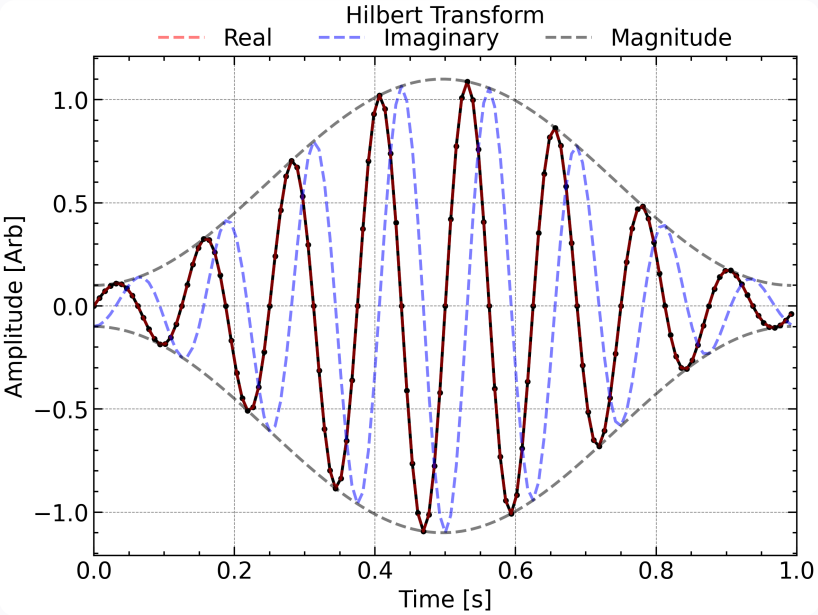
For a wide-sense stationary signal, its power spectral density is equal to the the fourier transform of its autocorrelation function, i.e.,:

$$PSD(f) = \int_{-\infty}^{\infty} R_{XX}(\tau) e^{-2\pi i f \tau} \mathrm{d}\tau$$

This theorem tells the intrinsic relationship between the *PSD* and *ACF*. Its contraposition claims that if the PSD doesn't equal to the Fourier transform of the ACF, the signal is not a *w.s.s* signal. The difference between them signify the nature of the solar wind parameters —— They are different from the NOISE! But, for some specific frequency range, they agree with each other well. It should be noticed that the closeness between them doesn't gurantee the signal to be *w.s.s*.

Hilbert Transform [*scipy.signal.hilbert*]

The Hilbert transform is a fundamental tool for analyzing the instantaneous amplitude and phase of a signal. By constructing the analytic signal, it enables us to extract the envelope and instantaneous frequency, which are essential in the study of modulated waves and transient phenomena. This section demonstrates how to implement the Hilbert transform in Python and interpret its results in both physical and engineering contexts.




```
omega = 2 * np.pi * 8.0
time = np.linspace(0, 1, 2 ** 7, endpoint=False)
# Modulate the Sine Wave with a offseted Hanning Window
signal = np.sin(omega * time) * (0.1 + np.hanning(time.size))
signal_ht = scipy.signal.hilbert(signal)

signal_ht.real, sighal_ht.imag, np.abs(signal_ht)
```

Digital Filter

Digital filters are fundamental tools for shaping, extracting, or suppressing specific features in time series data. In essence, a digital filter is a mathematical algorithm that modifies the amplitude and/or phase of certain frequency components of a discrete signal. Filters can be designed to remove noise, isolate trends, block out-of-band interference, or even simulate the response of a physical system. Anti-aliasing is also a common application, where filters are used to prevent high-frequency components from distorting the signal before downsampling.

Digital filters are divided into two main types:

- **Finite Impulse Response (FIR):** The output depends only on the current and **a finite number of past input samples**. FIR filters are always stable and can have exactly linear phase. A general FIR digital filter is implemented as:

$$y[n] = \sum_{k=0}^{M} h[k]\, x[n-k]$$

  - $x[n], y[n]$: Input, Output signal
  - $h[k]$: Filter coefficients (impulse response), length $M+1$
  - M: Filter order
- **Infinite Impulse Response (IIR):** The output depends on both current and **past input samples** *and* **past outputs**. IIR filters can achieve sharp cutoffs with fewer coefficients, but may be unstable and generally do not preserve linear phase. A general IIR filter has both input and output recursion:

$$y[n] = \sum_{k=0}^{M} b[k]\, x[n-k] - \sum_{l=1}^{N} a[l]\, y[n-l]$$

  - $b[k]$: Feedforward (input) coefficients
  - $a[l]$: Feedback (output) coefficients, usually $a[0] = 1$
  - M, N: Orders for input and output

Example: Moving Average

The **moving average filter** is actually a simple FIR filter. For a window length L, the coefficients are:

$$h[k] = \frac{1}{L}, \quad k = 0, 1, \ldots, L-1$$

So the output is:

$$y[n] = \frac{1}{L} \sum_{k=0}^{L-1} x[n-k]$$

That is, the output is the **average of the most recent L input samples**. **Therefore, the moving average filter is an FIR filter whose coefficients are all equal.**

Example: Low-pass FIR Filtering

Suppose we want to smooth a time series by attenuating frequencies above a certain threshold (e.g., removing noise above 50 Hz). This can be accomplished with an FIR filter designed using `scipy.signal.firwin`:

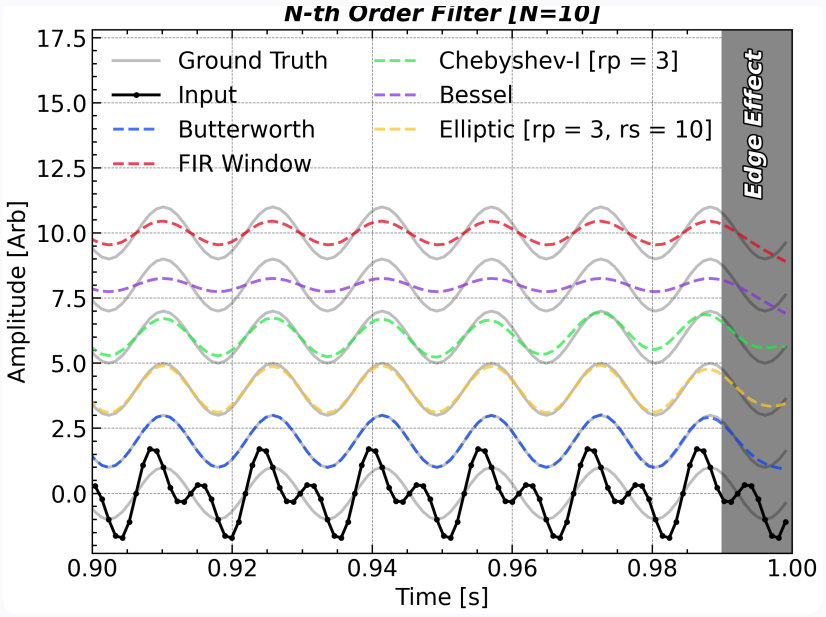
```
import numpy as np
from scipy.signal import firwin, lfilter

fs = 200.0  # Sampling frequency (Hz)
nyq = fs / 2.0
cutoff = 50.0  # Desired cutoff frequency (Hz)
numtaps = 101  # Filter length (number of coefficients)

# Design FIR low-pass filter
fir_coeff = firwin(numtaps, cutoff / nyq)
# Apply to data
filtered_signal = lfilter(fir_coeff, 1.0, raw_signal)
```

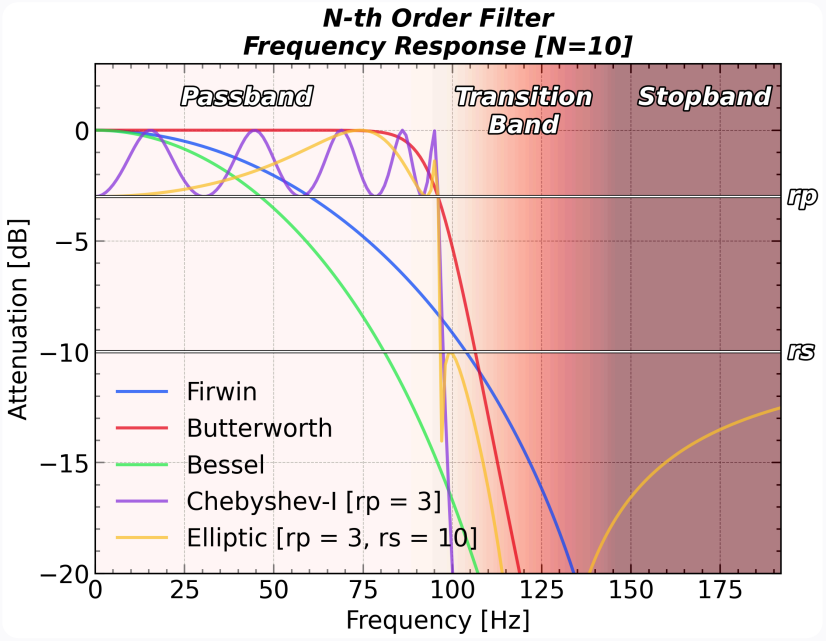
For IIR filters (such as Butterworth, Chebyshev), the `scipy.signal.butter` function is commonly used. **Note:** Filtering can introduce edge effects—always inspect the beginning and end of the filtered signal.

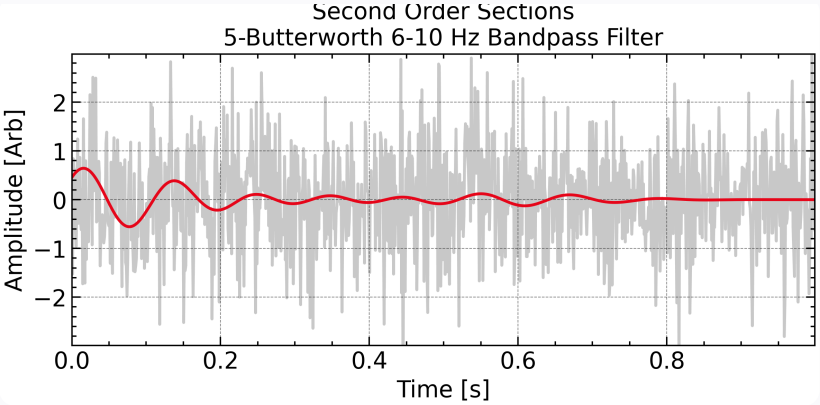
Frequency Response and Interpretation

The effect of a digital filter can be fully characterized by its *frequency response*, i.e., how it amplifies or suppresses each frequency. Use `scipy.signal.freqz` to plot the amplitude and phase response of your filter, and check that it matches your physical requirements (e.g., minimal ripple in the passband, sufficient attenuation in the stopband).

Filter Type	Function (`scipy.signal`)	Main Features	Use Case
FIR (window)	`firwin`, `firwin2`	Stable, linear phase	Smoothing, band selection
Butterworth	`butter`	Smooth, monotonic response	General purpose
Chebyshev I/II	`cheby1`, `cheby2`	Sharper cutoff, ripples	Strong suppression
Elliptic	`ellip`	Fastest cutoff, both ripples	Selective, small band
Bessel	`bessel`	Linear phase, slow rolloff	Transient preservation
Median	`medfilt`, `medfilt1d`	Nonlinear, preserves edges	Spike removal


N-th Order Filter [N=10]
Ground Truth
Input
Butterworth
FIR Window
Chebyshev-I [rp = 3]
Bessel
Elliptic [rp = 3, rs = 10]
Edge Effect
Amplitude [Arb]
Time [s]
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.90
0.92
0.94
0.96
0.98
1.00


N-th Order Filter
Frequency Response [N=10]
Passband
Transition Band
Stopband
rp
rs
Firwin
Butterworth
Bessel
Chebyshev-I [rp = 3]
Elliptic [rp = 3, rs = 10]
Attenuation [dB]
Frequency [Hz]
0
−5
−10
−15
−20
0
25
50
75
100
125
150
175



Practical Tips

- **Zero-phase Filtering:** Use `scipy.signal.filtfilt` for zero-phase filtering to avoid phase distortion, especially for waveform analysis.
- **Edge Effects:** Discard a small number of samples at both ends after filtering, or pad the signal before filtering to reduce transient effects.
- **Causality:** Standard filters are causal (output depends only on current and past inputs). Non-causal (zero-phase) filtering requires processing both forward and backward in time, and is not physically realizable in real-time applications.

Interpolation

Interpolation is the process of estimating unknown values between discrete data points. In scientific data analysis, especially in signal processing and time series studies, interpolation plays a vital role in resampling, aligning datasets, filling gaps, and reconstructing higher-resolution signals from coarse measurements.

Why Do We Need Interpolation?

- **Resampling:** Convert irregularly sampled data to a regular time grid for spectral analysis.
- **Filling Gaps:** Restore missing or corrupted data in a time series.
- **Temporal Alignment:** Synchronize data from different sources with differing sampling rates.

- **Upsampling/Downsampling:** Increase or decrease data resolution, e.g., for visualization or model input.

Common Interpolation Methods

1. Nearest-Neighbor Interpolation

Selects the value of the nearest known data point. Simple and fast, but produces a “blocky” or step-like signal.

```
nearest_interp = scipy.interpolate.interp1d(t, sig, kind = 'nearest', bounds_error = False)
```

2. Linear Interpolation

Connects data points with straight lines. Produces continuous, piecewise linear results; widely used for fast, low-artifact resampling.

```
linear_interp = scipy.interpolate.interp1d(t, sig, bounds_error = False)
```

3. Spline Interpolation

Fits smooth polynomial curves (usually cubic) through the data. Produces smooth and visually appealing results, but can introduce overshoot or ringing near sharp transitions.

```
cubic_interp = scipy.interpolate.CubicSpline(t, sig)
```

4. Akima Interpolation

Akima interpolation is a piecewise method based on fitting local polynomials between data points using adaptive slopes that depend on the trends of neighboring intervals. Unlike cubic splines, it does not enforce global smoothness but instead focuses on avoiding oscillations and overshoots near sharp transitions. This makes Akima interpolation particularly effective for datasets with non-uniform behavior or outliers, where traditional spline methods may produce unwanted ringing. It maintains a good balance between

smoothness and stability and is especially useful in applications requiring visually reliable curve fitting without excessive global influence.

Assumes that the data is periodic and uniformly sampled. The signal is extended using its discrete Fourier transform (DFT), and interpolation is performed in the frequency domain by zero-padding and inverse transforming. Fourier interpolation is ideal for band-limited signals and preserves the frequency content, but it may introduce artifacts if the periodicity assumption is violated.

```
akima_interp = scipy.interpolate.Akima1DInterpolator(t, sig)
```

5. Fourier Interpolation

Assumes data is periodic and uses the Fourier series for reconstruction. Ideal for band-limited signals with uniform sampling, preserves frequency content.

```
sig_interpolated = scipy.signal.resample(sig, t_interp.size)
```

6. Sinc Interpolation

Sinc interpolation is the theoretical ideal method for reconstructing a uniformly sampled, band-limited signal from its discrete samples. According to the Shannon sampling theorem, a continuous signal with no frequency components above the Nyquist frequency can be perfectly reconstructed from its samples using a sinc function as the interpolation kernel:

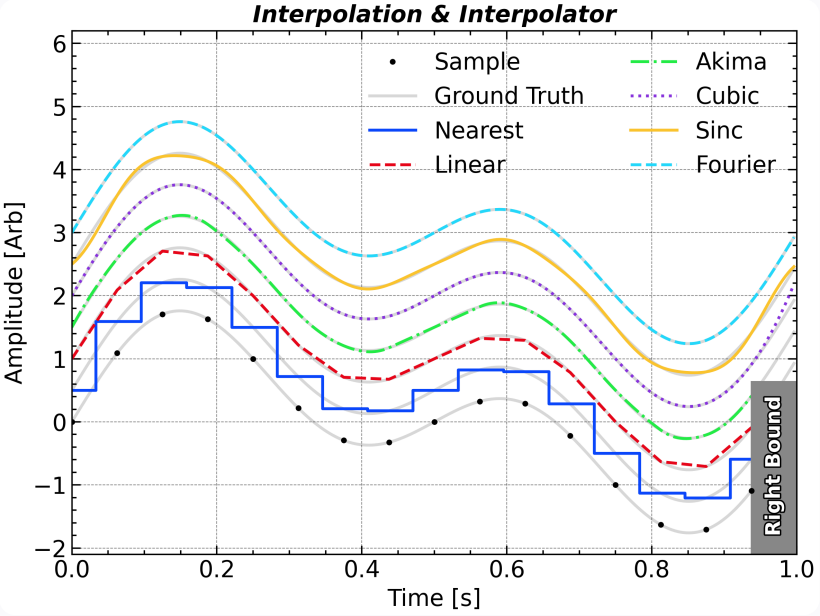
$$x(t) = \sum_{n=-\infty}^{+\infty} x[n] \operatorname{sinc}\left(\frac{t - nT_s}{T_s}\right)$$

where T_s is the sampling interval, and $\operatorname{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$.

- **Perfect for band-limited, uniformly sampled signals** (theoretical limit).
- **Preserves all frequency content up to the Nyquist frequency.**

- The interpolation kernel is infinitely wide (non-local), so true sinc interpolation is not practically achievable (requires truncation or windowing).
- In practice, *windowed sinc* or a finite sum is used.

```
def sinc_interp(t_interp):
    weight = np.sinc((t_interp[:, None] - t[None, :]) / dt)
    return weight @ sig

sinc_interp(t_interp)
```



Interpolation and the Frequency Domain

Interpolation in the time domain directly impacts the signal's frequency content:

- **Nearest-neighbor** acts as a zero-order hold, introducing high-frequency artifacts.
- **Linear** acts as a convolution with a triangle (sinc² in frequency), attenuating high-frequency components.
- **Spline/Polynomial** offers smoother spectra but can still introduce artifacts at sharp features.

- **Sinc interpolation** and **Fourier interpolation** (i.e., zero-padding in the frequency domain) yield the most faithful reconstruction for band-limited signals.

> **Tip:**
> For spectral analysis, prefer linear, sinc, or Fourier interpolation for uniformly sampled, band-limited signals. Spline interpolation is suitable for smooth, low-noise signals, but beware of overshoot and frequency artifacts.

Typical Use Cases

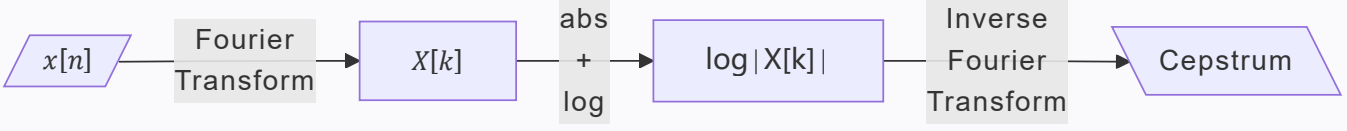
- **Resampling spacecraft data** to a common time base for multi-instrument analysis.
- **Gap filling** in geomagnetic or solar wind time series for uninterrupted spectra.
- **High-resolution reconstruction** for visualization of waveforms.

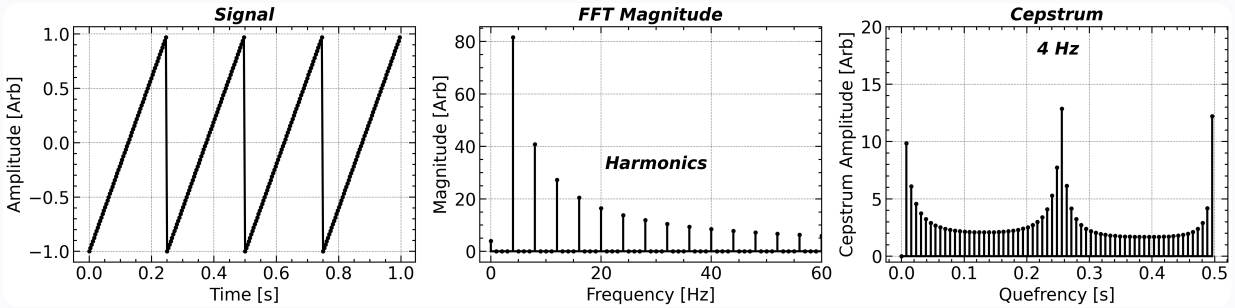
Comparison of Interpolation Methods

Method	`scipy` Function / Implementation	Smoothness	Preserves Frequencies	Typical Use
Nearest Neighbor	`interp1d(kind='nearest')`	Discrete	No	Quick gap fill, step signals
Linear	`interp1d(kind='linear')`	Continuous	Partially	Standard resampling
Cubic Spline	`CubicSpline`, `interp1d(kind='cubic')`	Smooth	Mostly	Smooth signals, visualization
Sinc	*custom, see above*	Smoothest	**Yes (ideal, band-limited)**	Band-limited, uniform samples
Fourier	Zero-padding FFT	Smoothest	Yes	Band-limited, periodic
Akima	`Akima1DInterpolator`	Moderate	No	Robust fitting with minimal overshoot.

Cepstrum

Cepstral analysis provides a unique perspective by applying a Fourier transform to the logarithm of the spectrum. The resulting “Cepstrum” is widely used for echo detection, speech processing, and seismic reflection analysis. This section explains the underlying theory, physical meaning, and demonstrates how to perform cepstral analysis in Python.



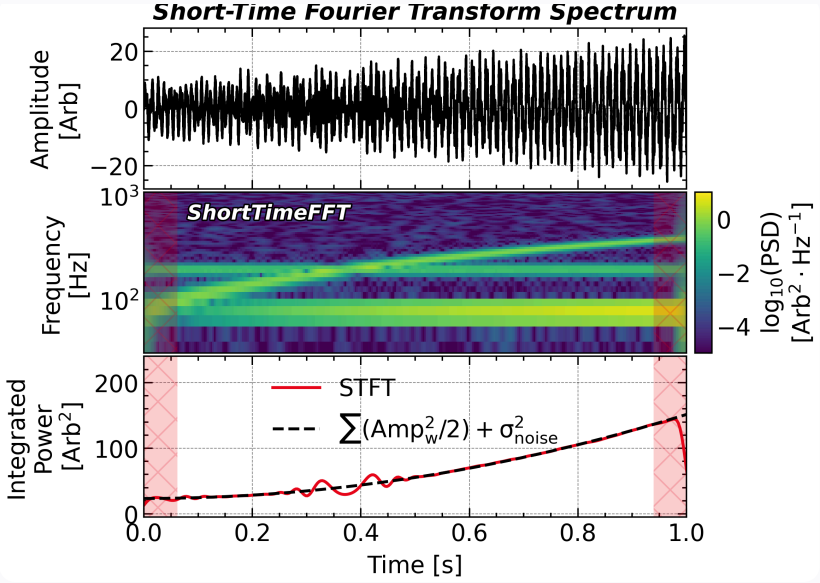


Time-Frequency Spectrum

Short-Time Fourier Transform

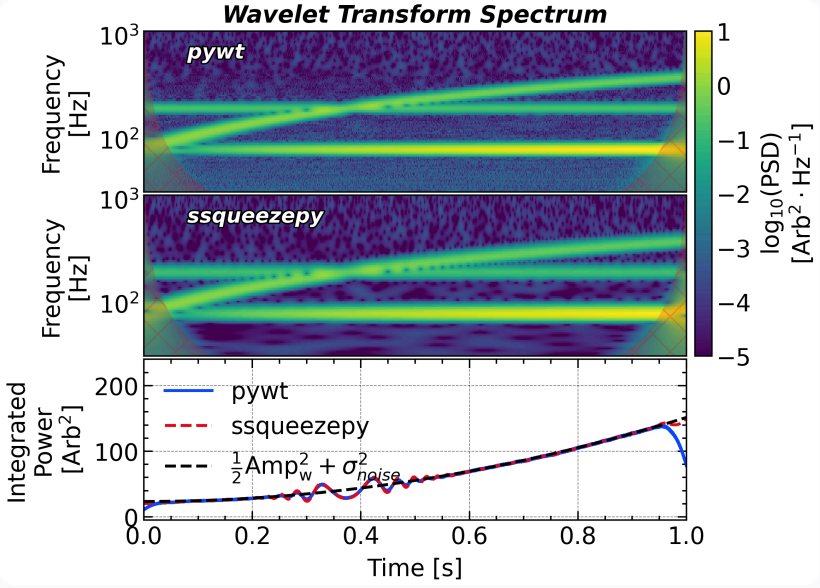
The Short-Time Fourier Transform (STFT) extends traditional Fourier analysis to non-stationary signals by introducing time localization via windowing. This allows us to track how the frequency content of a signal evolves over time. This section explains the trade-off between time and frequency resolution, the role of window functions, and practical implementation with `scipy.signal.shortTimeFFT`. It should be noted that function `scipy.signal.stft` is considered legacy and will no longer receive updates. While `scipy` currently have no plans to remove it, they recommend that new code uses more modern alternatives `shortTimeFFT` instead.

```
window = 4096
step = 100
hann_window = scipy.signal.windows.hann(window, sym = True) # Hanning
Window
STFT = scipy.signal.ShortTimeFFT(hann_window, hop=step, fs = fs,
scale_to='psd', fft_mode = 'onesided2X')  # create the STFT object
stft_psd = STFT.stft(sig)  # perform the STFT
stft_time = np.arange(0, stft_psd.shape[1]) * STFT.hop / fs - STFT.win.size
/ 2 / fs  # time vector for STFT
stft_frequency = np.fft.rfftfreq(window, d=dt)  # frequency vector for STFT
```



Wavelet Analysis

Wavelet analysis offers a versatile framework for multi-resolution time-frequency analysis, especially for signals with localized features or abrupt transitions. By decomposing a signal into wavelets, we gain simultaneous insight into both frequency and time domains. This section introduces the fundamentals of wavelet theory, common wavelet families, and hands-on examples using Python packages such as `pywt`, `scipy`, and `squeezzeypy`.




```
scales = 2 ** np.linspace(4, 12, 160, endpoint = False)
```

```
bandwidth = 12.0
central_frequency = 1.0

# Way 1: pywavelets
wavelet = 'cmor%.1f-%.1f' % (bandwidth, central_frequency)
coef, f = pywt.cwt(sig, scales, wavelet, dt, method = 'fft')
coef *= np.sqrt(np.sqrt(bandwidth) * np.sqrt(2 * np.pi))  # amplitude
normalization for Morlet
psd = (np.abs(coef) ** 2) * (2 * dt)
df = (f[0] / f[1] - 1) * f / np.sqrt(f[0] / f[1])

# Way 2: ssqueezepy
coef, scales = ssqueezepy.cwt(sig, ('morlet', {'mu': bandwidth}), scales =
bandwidth / (2 * np.pi) * scales.astype(np.float32), fs = 1 / dt, l1_norm =
False)
f = bandwidth / (2 * np.pi) / dt / scales
df = (f[0] / f[1] - 1) * f / np.sqrt(f[0] / f[1])
psd = (np.abs(coef) ** 2) * (2 * dt)

# Way 3:
# There is a cwt function provided by scipy.signal. However, this function
is deprecated since version 1.12.0. They recommend using PyWavelets
instead.

# widths = bandwidth * scales / (2 * np.pi)
# coef = scipy.signal.cwt(
#     signal,
#     scipy.signal.morlet2,
#     widths = widths,
#     w = bandwidth,
#     dtype = np.complex128
# )
# frequency = 1 / dt / scales

# Cone of Influence (COI)
coi = (np.sqrt(4) * bandwidth / (2 * np.pi) / f).astype(float)
```

Multi-Dimensional Signal

Principal Component Analysis / Minimum Variance Analysis

Principal Component Analysis (PCA) and Minimum Variance Analysis (MVA) are closely related, eigen-vector–based techniques for extracting the dominant directional structure in multivariate data. PCA is a general statistical tool; MVA is the same mathematics applied to three-component field measurements (e.g., **B** in space physics) with special attention to the minimum-variance direction.

The central ideas of these two methods are:

- **PCA**: Rotate the data into a new orthogonal basis such that each successive axis captures the greatest possible remaining variance.
- **MVA**: Apply PCA to a 3 × N vector time series and interpret the eigenvectors as the directions of maximum, intermediate, and minimum variance—often used to infer boundary normals or wave polarization axes.

Mathematical Formulation

1. **Collect & demean** the data matrix

$$\mathbf{X} = \begin{bmatrix} x_1 & x_2 & \dots & x_N \\ y_1 & y_2 & \dots & y_N \\ z_1 & z_2 & \dots & z_N \end{bmatrix}, \qquad \tilde{\mathbf{X}} = \mathbf{X} - \langle \mathbf{X} \rangle$$

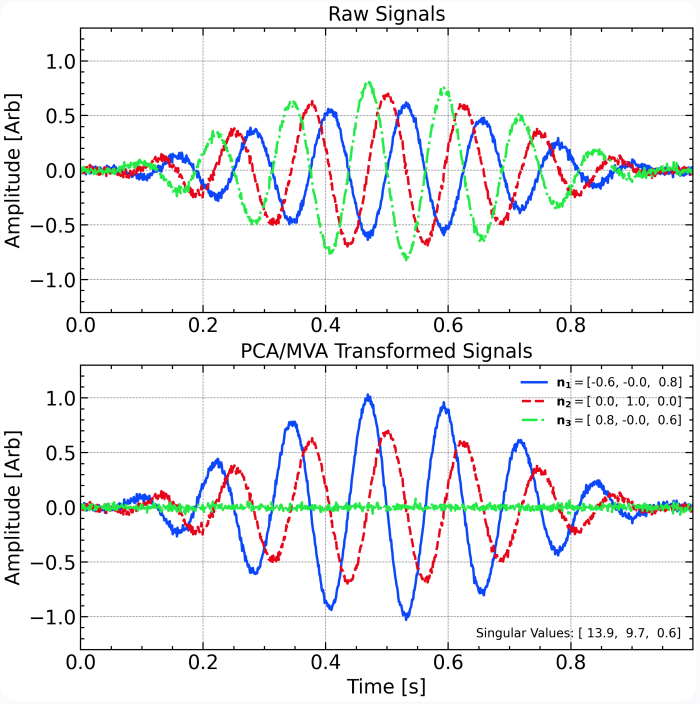
2. **Form the covariance (spectral) matrix**

$$\mathbf{C} = \frac{1}{N-1} \tilde{\mathbf{X}} \tilde{\mathbf{X}}^{\mathsf{T}}$$

3. **Solve the eigenproblem**

$$\mathbf{C}\,\mathbf{e}_i \;=\; \lambda_i\,\mathbf{e}_i, \quad \lambda_1 \geq \lambda_2 \geq \lambda_3 \geq 0$$

4. **Interpretation**
   - **PCA**: Project the data onto the top *k* eigenvectors $\{\mathbf{e}_1, \ldots, \mathbf{e}_k\}$ for dimensionality reduction.
   - **MVA**:
     - $\mathbf{e}_1$: maximum-variance direction (largest fluctuations)
     - $\mathbf{e}_2$: intermediate direction
     - $\mathbf{e}_3$: minimum-variance direction—often taken as the local discontinuity normal or the wave propagation vector.




```
N = 2 ** 10
time = np.linspace(0, 1, N, endpoint=False)
omega = 2 * np.pi * 8.0

signal_x = np.sin(omega * time + np.pi * 0.0) * (0.6 *
np.hanning(time.size)) + np.random.randn(time.size) * 0.02
signal_y = np.sin(omega * time + np.pi * 0.5) * (0.7 *
np.hanning(time.size)) + np.random.randn(time.size) * 0.02
signal_z = np.sin(omega * time + np.pi * 1.0) * (0.8 *
np.hanning(time.size)) + np.random.randn(time.size) * 0.02
```

```
signal = np.vstack((signal_x, signal_y, signal_z)).T
pca = sklearn.decomposition.PCA(n_components=3)
pca.fit(signal)

eigenvalues = pca.singular_values_
eigenvectors = pca.components_
```

Spectral Matrix

A ***spectral matrix*** can be defined as

$$\hat{S}_{ij} = \hat{B}_i \hat{B}_j^*$$

for a time series decomposed with both signal and noise, its Fourier coefficients follow the ***non-central chi-square distribution***, as introduced in the previous section. Taking a moving-average in the time or frequency domain helps improving the SNR as we did in the Welch method.

```
spec = np.einsum('fti,ftj->ftij', coef, coef.conj())

# Average in time or frequency domain
spec_avg = np.copy(spec_avg)
spec_avg = bn.move_mean(spec_avg, window=time_window, min_count=1, axis=1)
spec_avg = bn.move_mean(spec_avg, window=freq_window, min_count=1, axis=0)
```

Coherence

Coherence measures the degree of linear correlation between two signals at each frequency, serving as a frequency-resolved analog of correlation coefficient. High coherence indicates a strong, consistent relationship, which is crucial for studies of wave propagation, coupled systems, and causality analysis. Here, we explain how to calculate and interpret coherence with Python tools.

To be honest, I feel very hard to understand what does *coherent/coherence* means in many of the magnetospheric ULF/VLF waves investigations. It can be easily understood the coherence between two individual light or signal. However, in the *in-situ* observation, the spacecraft can only measure one signal without further distinguishment or separation. In some literature, the coherence between E_x and B_y are used to measure whether the observed VLF waves are coherent. These VLF waves always propagate along the geomagnetic field line, which point to the north near the magnetic equator. It makes some sense as a high coherence suggests the waves have a stable wave vector during this interval. But, it is still hard to expect the occurrence of interference as both E_x and B_y may just be the presence of one single wave. While, some other literatures use the coherence between the magnetic field components to

- Coherency is meaningless without taking an average.

Combination with Maxwell's Equations: SVD Wave Analysis

Spectral analysis gains further physical meaning when interpreted alongside Maxwell’s equations. For electromagnetic signals, the spectral content reflects underlying wave propagation, polarization, and field coupling processes.

$$
\begin{aligned}
\nabla \cdot \mathbf{E}(\mathbf{r}, t) &= -\frac{\rho}{\varepsilon_0} \\
\nabla \cdot \mathbf{B}(\mathbf{r}, t) &= 0 \\
\nabla \times \mathbf{E}(\mathbf{r}, t) &= -\frac{\partial \mathbf{B}(\mathbf{r}, t)}{\partial t} \\
\nabla \times \mathbf{B}(\mathbf{r}, t) &= \mu_0 \mathbf{J}(\mathbf{r}, t) + \mu_0 \varepsilon_0 \frac{\partial \mathbf{E}(\mathbf{r}, t)}{\partial t}
\end{aligned}
$$

As the electromagnetic field $\mathbf{E}(\mathbf{r}, t)$ and $\mathbf{B}(\mathbf{r}, t)$ are square-integrable, Maxwell's equations can be naturally transformed into the $(\mathbf{k}, \omega)$-space with the basic replacement from $\nabla \leftrightarrow i\mathbf{k}$ and $\partial/\partial t \leftrightarrow -i\omega$:

$$
\begin{aligned}
i\mathbf{k}\cdot\hat{\mathbf{E}}(\mathbf{k},\omega) &= -\hat{\rho}_e/\varepsilon_0 \\
i\mathbf{k}\cdot\hat{\mathbf{B}}(\mathbf{k},\omega) &= 0 \\
i\mathbf{k}\times\hat{\mathbf{E}}(\mathbf{k},\omega) &= i\omega\hat{\mathbf{B}}(\mathbf{k},\omega) \\
i\mathbf{k}\times\hat{\mathbf{B}}(\mathbf{k},\omega) &= \mu_0\hat{\mathbf{J}}(\mathbf{k},\omega) - \mu_0\varepsilon_0 i\omega\hat{\mathbf{E}}(\mathbf{k},\omega)
\end{aligned}
$$

However, a single spacecraft measurement only allows you to observe a one-dimensional (time) signal at one position, i.e., the spacecraft position, which literally moves in the space. Thus, the signal can only be converted into the frequency space as $\hat{\mathbf{B}}(\omega)$. The second equation is the only parameter-free equation and states that the wave vector, $\mathbf{k}$ must be perpendicular to the magnetic field disturbance, $\hat{\mathbf{B}}(\omega)$. Obviously, $\mathbf{k}=\mathbf{0}$ is a trivial, but not useful solution for satisfying the divergence-free theorem. By constraining the norm of $\mathbf{k}$ to be unity, $\boldsymbol{\kappa}:=\mathbf{k}/k$, a more meaningful solution comes out. When the real part, $\Re\hat{\mathbf{B}}(\omega)$ and imaginary part, $\Im\hat{\mathbf{B}}(\omega)$ of $\hat{\mathbf{B}}(\omega)$ are highly orthogonal, they can span a linear space whose normal vector is naturally $\boldsymbol{\kappa}$.

$$
\boldsymbol{\kappa} = \frac{\Re\hat{\mathbf{B}}(\omega)\times\Im\hat{\mathbf{B}}(\omega)}{|\Re\hat{\mathbf{B}}(\omega)\times\Im\hat{\mathbf{B}}(\omega)|}
$$

which perfectly satisfy that $\hat{\mathbf{B}}\cdot\boldsymbol{\kappa}=0$.

However, this $\hat{\mathbf{B}}$-based, namely, coefficient-based estimation may be influenced by the noise's contribution and thus is not so practical. Inspired by the Welch method, a spectral-based estimation is preferred as the spectral density is easily denoised. The spectral-based estimation can be given by refining the original proposition:

$$
(\hat{\mathbf{B}}^*\hat{\mathbf{B}})\cdot\boldsymbol{\kappa}=0
$$

$$\hat{S}_{ij} = \langle \hat{B}_i \hat{B}_j^* \rangle$$

which can still be met by the original solution. After averaging the spectral matrix in time and frequency domain, this equation can not be perfectly satisfied any more. Thus, we will look for a weaker solution in the sense of minimization:

$$\begin{aligned} &\min_{||\boldsymbol{\kappa}||_2^2=1} ||\hat{S} \cdot \boldsymbol{\kappa}||_2^2 \\ \Leftrightarrow &\min_{||\boldsymbol{\kappa}||_2^2=1} \{\underline{||\Re\hat{S} \cdot \boldsymbol{\kappa}||_2^2} + \underline{||\Im\hat{S} \cdot \boldsymbol{\kappa}||_2^2}\} \end{aligned}$$

McPherron et al. (1972) and Means (1972) adopts the real and imaginary part in the minimization optimization for the estimation of wave propagation direction, respectively. Both of these two optimization problem can be solved by eigenvalue decomposition. Then, Santolík et al. (2003) combine both terms and construct an augmented matrix A:

$$A = \begin{pmatrix} \Re S_{11} & \Re S_{12} & \Re S_{13} \\ \Re S_{12} & \Re S_{22} & \Re S_{23} \\ \Re S_{13} & \Re S_{23} & \Re S_{33} \\ 0 & -\Im S_{12} & -\Im S_{13} \\ \Im S_{12} & 0 & -\Im S_{23} \\ \Im S_{13} & \Im S_{23} & 0 \end{pmatrix}$$

The optimization problem

$$\min_{||\boldsymbol{\kappa}||_2^2=1} ||A \cdot \mathbf{k}||_2^2$$

is directly solvable by applying a ***singular value decomposition(SVD)*** to matrix A

$$A = U \cdot W \cdot V^T$$

where U is a 6×3 matrix with orthonormal columns, W is a 3×3 diagonal matrix with three nonnegative singular values, and V^T is a 3×3 matrix with orthonormal rows. Diagonal matrix W representing the signal power in a descending order.

- **Compressibility** describe the polarization

$$\text{Compressibility}(f_k) := \frac{PSD(B_{\|})}{\sum_i PSD(B_i)}$$

- **Planarity**

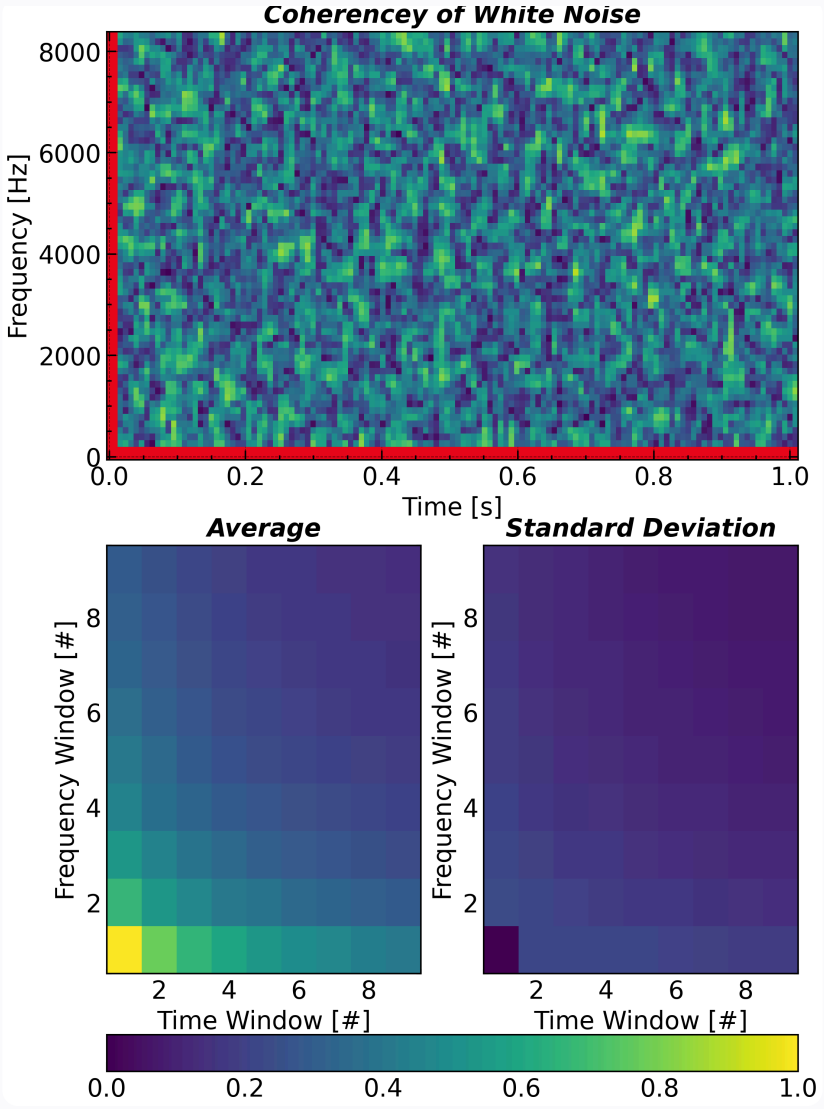
$$F = 1 - \sqrt{W_2/W_0}$$

Without averaging the spectral matrix, the planarity $F(t, f)$ **will be all one**. It means that, when the observer only take one snapshot of the waves, it can not distinguish how does the waves propagate. After the averaging, the planarity actually describe that, **whether the waves that observed at these time periods, frequencies share the common unitary wave vector.**

```
spec = np.einsum('fti,ftj->ftij', coef, coef.conj())
spec = bn.move_mean(spec, window=freq_window, min_count=1, axis=0)
spec = bn.move_mean(spec, window=time_window, min_count=1, axis=1)

spec_63 = np.concatenate([spec.real, spec.imag], axis=-2)
u, s, vh = np.linalg.svd(spec_63, full_matrices=False)

planarity = 1 - np.sqrt(s[:, :, 2] / s[:, :, 0])
ellipticity_along_k = s[:, :, 1] / s[:, :, 0]
```

Similarly, based on the averaged spectral matrix, one may define the coherence (coherency) between different components:

$$Coherency := \frac{|S_{ij}|}{\sqrt{S_{ii}S_{jj}}}$$



One should keep in mind that all interpretation about the observed waves is in the spacecraft inertial reference frame. A proper choice of coordinate system is especially necessary for a spinning spacecraft.

This section explores the synergy between spectral analysis and electromagnetic theory, demonstrating how to derive physical insights and constraints from both perspectives.

Ellipticity along $\mathbf{k}$

Polarization analysis examines the orientation and ellipticity of oscillatory signals, especially electromagnetic or plasma waves. By decomposing the signal into orthogonal components and analyzing their relative amplitude and phase, we can characterize wave mode, propagation direction, and physical

source. This section introduces key polarization parameters, their spectral estimation, and relevant Python implementations.

Ellipticity can be defined as the ratio of the semi-major and semi-minor, which is estimated by:

$$\epsilon = \frac{W_1}{W_0}$$

For a noisy signal, T and S 2019 propose an improved method with a estimation of the noise level based on the eigen decomposition. In this method, they noise level is inferred by decomposing the real part of the spectral density matrix and the maximum/intermediate eigenvalues of the complex spectral density represents the summation of wave power and noise power. Therefore, the improved ellipticity is derived:

$$\epsilon' = \sqrt{\frac{\lambda_{r1} - \lambda_1}{\lambda_{r0} - \lambda_1}}$$

This improved ellipticity performs better than the original one when SNR is low but the still randomly deviates from the ground true. Thus, a moving average in the time or frequency domain is still required as it promote the SNR.

```
coef_wf = np.einsum('ijk,ijlk->ijl', coef, vh)
spec_wf = np.einsum('fti,ftj->ftij', coef_wf, coef_wf.conj())
spec_wf = bn.move_mean(spec_wf, window=freq_window, min_count=1, axis=0)
spec_wf = bn.move_mean(spec_wf, window=time_window, min_count=1, axis=1)

coherence = np.abs(spec_wf[:, :, 0, 1]) / np.sqrt(np.abs(spec_wf[:, :, 0,
0] * spec_wf[:, :, 1, 1]))

eigenvalues_r, _ = np.linalg.eigh(spec_wf[:, :, :2, :2].real) # Ascending
eigenvalues, _ = np.linalg.eigh(spec_wf[:, :, :2, :2]) # Ascending

ellipticity_along_k = np.sqrt((eigenvalues_r[:, :, 0] - eigenvalues[:, :,
0]) \
                              / (eigenvalues_r[:, :, 1] - eigenvalues[:, :,
0]))
```

Ellipticity along $\mathbf{B}$

Both above two ellipticities are unsigned as the singular/eigen values are always non-negative. Another, but not alternative definition of the ellipticity, is the ratio of left-handed polarized signal power to the right-handed polarized power. This definition is signed and the the ellipse is defined in the plane that perpendicular to the background magnetic field:

$$\epsilon_B = \frac{|\hat{B}_L|^2 - |\hat{B}_R|^2}{|\hat{B}_L|^2 + |\hat{B}_R|^2}$$

with B_L and B_R defines below:

$$B_L = \frac{1}{\sqrt{2}}(B_{\perp 1} + iB_{\perp 2})$$
$$B_R = \frac{1}{\sqrt{2}}(B_{\perp 1} - iB_{\perp 2})$$

with $\mathbf{e}_{\perp 1}$, $\mathbf{e}_{\perp 2}$, and $\mathbf{e}_{\parallel}$ constitute a right-hand system, i.e., $\mathbf{e}_{\perp 1} \times \mathbf{e}_{\perp 2} = \mathbf{e}_{\parallel}$.

It is also important as it may unveils the wave excitation mechanism (e.g., wave-particle resonance). This definition is totally irrelevant with the determination of the wave vector direction. Instead, field-aligned coordinates is required for its derivation.

Field-Aligned Coordinate

Field-Aligned Coordinates (FAC) are a specialized coordinate system used in space physics to analyze data collected along the geomagnetic field lines. By transforming measurements into FAC, we can isolate field-aligned structures and dynamics, such as auroral currents and plasma waves. This section explains the transformation process, its physical significance, and practical applications in magnetospheric studies.

To transform a three-dimensional vector field $\mathbf{B}$ into FAC, we first need to calculate the unit vector along the magnetic field line, $\mathbf{b} = \mathbf{B}/|\mathbf{B}|$. The FAC system is then defined as follows:

```
magf = np.array([bx, by, bz])
magf_avg = bn.move_mean(magf, axis=0)

dir_ref = np.array([1., 0., 0.])
dir_para = (magf_avg.T / np.linalg.norm(magf_avg, axis = 1)).T

dir_perp_1 = np.cross(dir_para, dir_ref)
dir_perp_1 = (dir_perp_1.T / np.linalg.norm(dir_perp_1, axis = 1)).T

dir_perp_2 = np.cross(dir_para, dir_perp_1)
dir_perp_2 = (dir_perp_2.T / np.linalg.norm(dir_perp_2, axis = 1)).T

# Amplitude Projection
magf_para = np.einsum(magf, dir_para, 'ij,ij ->i')
magf_perp_1 = np.einsum(magf, dir_perp_1, 'ij,ij ->i')
magf_perp_2 = np.einsum(magf, dir_perp_2, 'ij,ij ->i')

# Coefficient Projection
coef_para = np.einsum('ijk,jk->ij', coef, dir_para)
coef_perp_1 = np.einsum('ijk,jk->ij', coef, dir_perp_1)
coef_perp_2 = np.einsum('ijk,jk->ij', coef, dir_perp_2)
```

```
compressibility = np.abs(coef_para) ** 2 / (np.abs(coef_para) ** 2 +
np.abs(coef_perp_1) ** 2 + np.abs(coef_perp_2) ** 2)

# Polarization
coef_lh = (coef_perp_1 - 1j * coef_perp_2) / np.sqrt(2)
coef_rh = (coef_perp_1 + 1j * coef_perp_2) / np.sqrt(2)

ellipticity_along_b = (np.abs(coef_rh) - np.abs(coef_lh)) /
(np.abs(coef_rh) + np.abs(coef_lh))
```

Degree of Polarization

The **degree of polarization** quantifies the proportion of an electromagnetic fluctuation (such as a plasma wave) that is organized, or polarized, as opposed to random or unpolarized (noise-like) components. It is a fundamental parameter in space plasma physics, characterizing the coherence of observed wave signals.

The degree of polarization is defined as the fraction of the total wave power that is associated with a perfectly polarized (coherent) component. It is mathematically expressed as:

$$D_p = \frac{\text{Power of the Polarized Component}}{\text{Total Power}}$$

- $D_p = 1$: the signal is completely polarized.
- $0 < D_p < 1$: the signal is partially polarized.
- $D_p = 0$: the signal is totally unpolarized (random noise).

A high degree of polarization indicates that the observed fluctuations are dominated by coherent wave processes, while a low degree suggests that random or turbulent components are significant. The degree of polarization is widely used to distinguish wave modes, to separate physical signals from

instrumental or background noise, and to assess the reliability of wave analysis.

In three-dimensional wave analysis, the **degree of polarization** quantifies how much of the measured signal is concentrated along a single, well-defined direction, versus being randomly distributed among all directions.

The 3D eigenvalue-based degree of polarization is defined as:

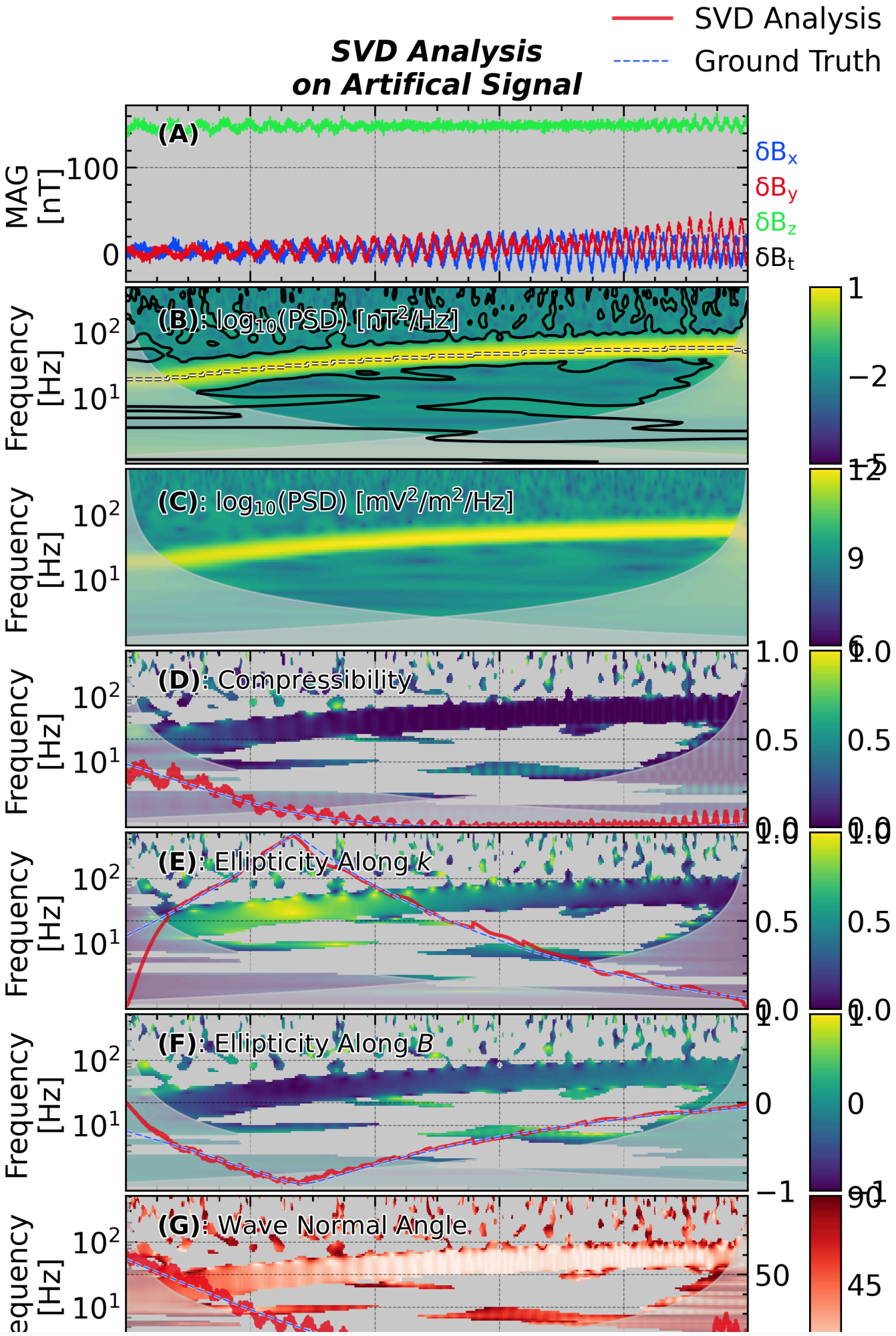
$$D_{p,3D} = \frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2 + \lambda_3}$$

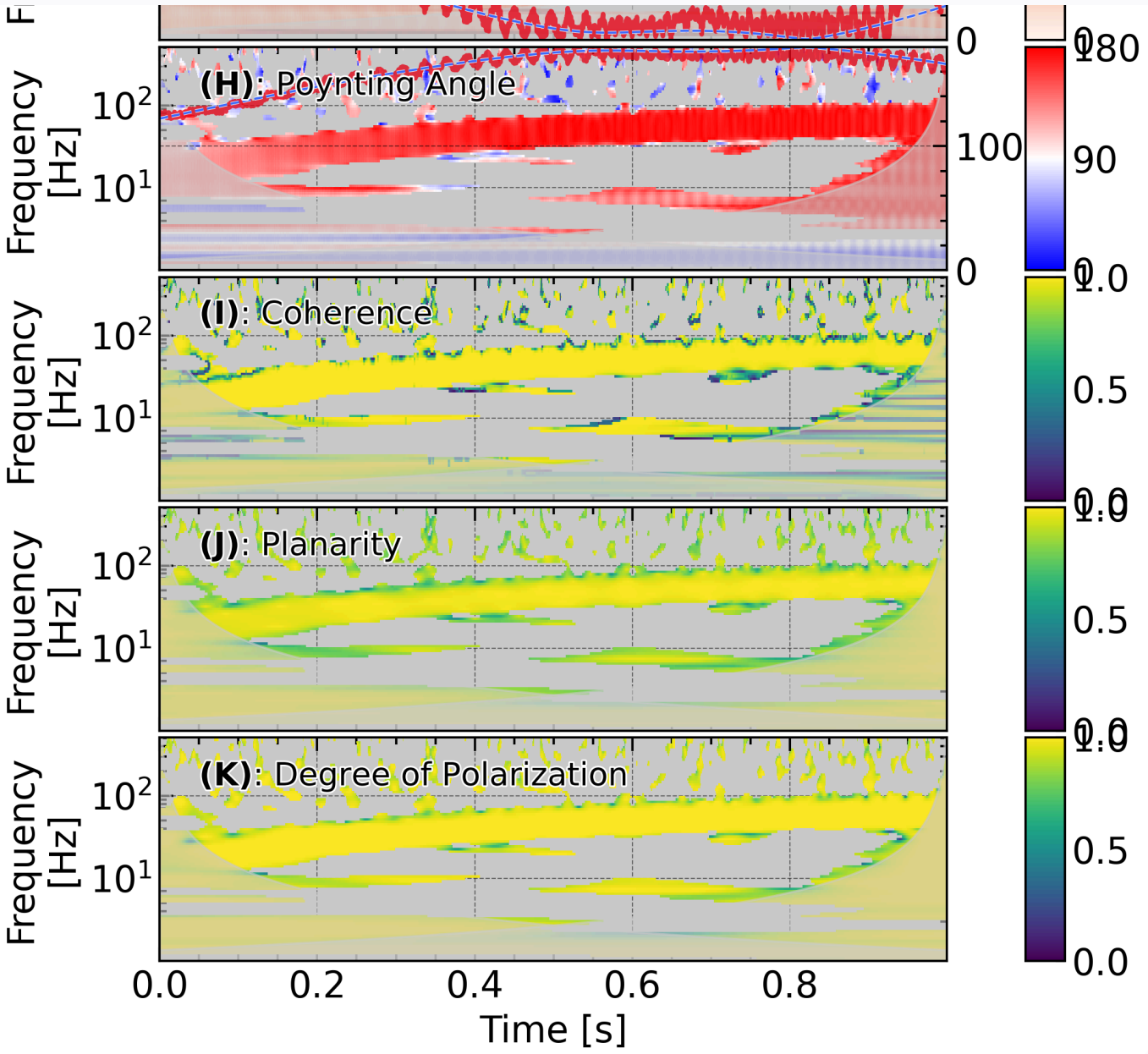
where $\lambda_1 \geq \lambda_2 \geq \lambda_3$ are the eigenvalues of the (power or spectral) matrix constructed from the three orthogonal components of the wave field.

This definition is coordinate-invariant and widely used in space plasma physics to characterize the coherence and organization of wave signals in planetary magnetospheres and the solar wind. It is particularly powerful for distinguishing true wave modes from background turbulence or noise.

```
w, v = np.linalg.eigh(spec)
degree_of_polarization = (w[:, :, 2] - w[:, :, 1]) / np.sum(w, axis = -1)
```

- Notice: `np.linalg.eigh` and `np.linalg.svd` return the eigen/singular values in an ascending / descending order.


SVD Analysis
Ground Truth
SVD Analysis
on Artifical Signal
(A)
MAG [nT]
100
0
δB_x
δB_y
δB_z
δB_t
(B): log10(PSD) [nT²/Hz]
Frequency [Hz]
10²
10¹
1
−2
−5
(C): log10(PSD) [mV²/m²/Hz]
−2
9
6
(D): Compressibility
1.0
0.5
0.0
(E): Ellipticity Along k
(F): Ellipticity Along B
0
−1
(G): Wave Normal Angle
90
50
45



Jargon Sheet and Personal Naming Convention

- When creating a `numpy.ndarray` for a signal, I will always

Notation	Variable Name	Explanation
$(t), (f)$	`t, f`	Continuous input, frequency (f) or time (t)
$[t], [f]$	`t, f`	Discrete input
ω	`omega`	Angular frequency, $\omega = 2\pi f$
$x(t)$	-	Continuous signal x as a function of time (t)
$x[n] = x(nf_S)$	`sig`	An even sample of $x(t)$, the intensity is termed as ***Amplitude***
$X[k] = X(kT^{-1})$	`coef`	Fourier Transform Coefficient of $x(t)$, the intensity is termed as ***Magnitude***
$\mathcal{F}$	-	Fourier Transform Operator
N	`N`	Signal length
Arb	-	Arbitrary Unit
$w[n]$	`window`	

Notation	Variable Name	Explanation

Acknowledgement

This document is finished with the help of ChatGPT and Copilot.