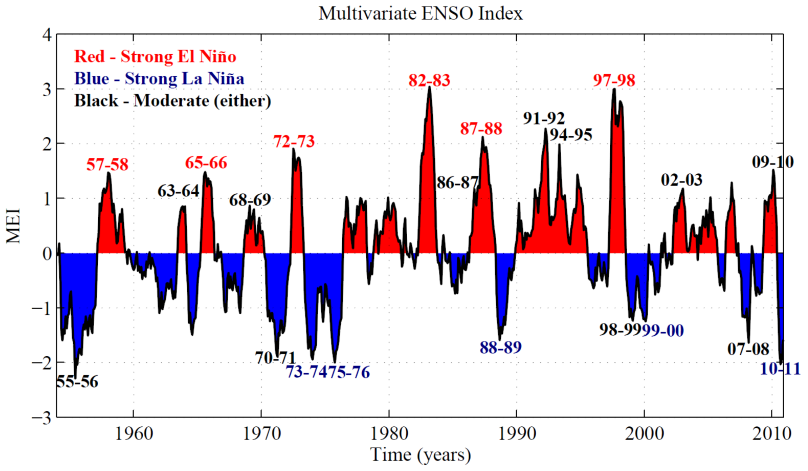
Practical Spectral Analysis in Plasma Physics

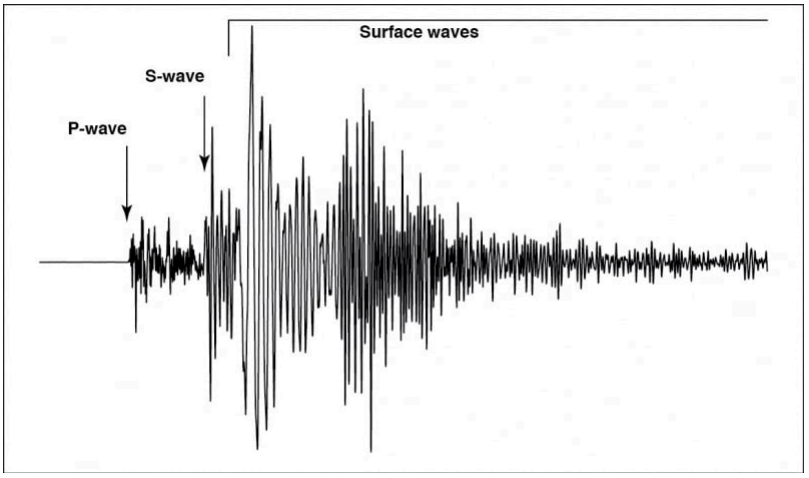
In physics and engineering, we often deal with **signals** – functions that vary over time and represent some physical quantity. A signal can be any measurable quantity that changes with time: for example, an audio waveform, the voltage output of a sensor, or the magnetic field measured in a plasma experiment. When we record such a quantity at successive points in time, we get a **time series**.

Many familiar phenomena are naturally described as time series, including:

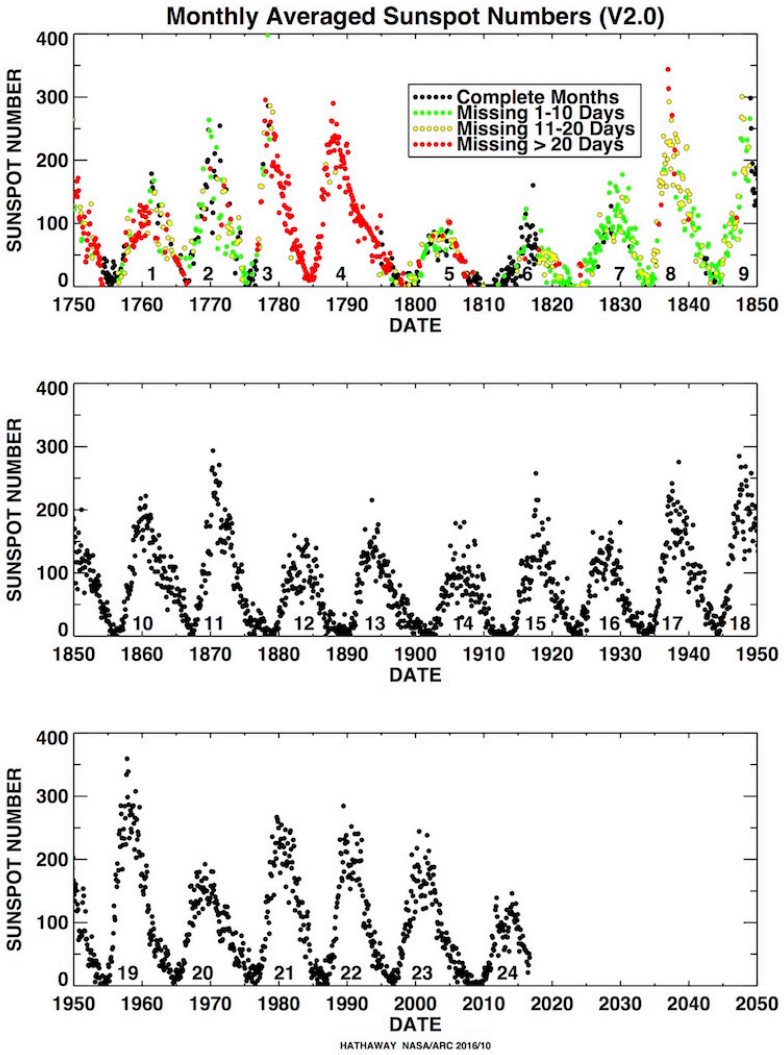
- **Meteorology:** e.g. El-Nino Enso



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



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



Each of these is a time-domain description: we have a quantity (amplitude, voltage, etc.) **as a function of time**.

Understanding the time-domain behavior of a system is important. However, it is often hard to tell **what underlying patterns or oscillations** are present. This is where **spectral analysis** becomes useful.

Why Do We Need Spectral Analysis?

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., amplitutde, compressibility) by revealing dominant frequencies in electric/magnetic 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**.

Nyquist Frequency and Aliasing

> ## 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:
>
> $$f_s \geq 2f_{max}$$

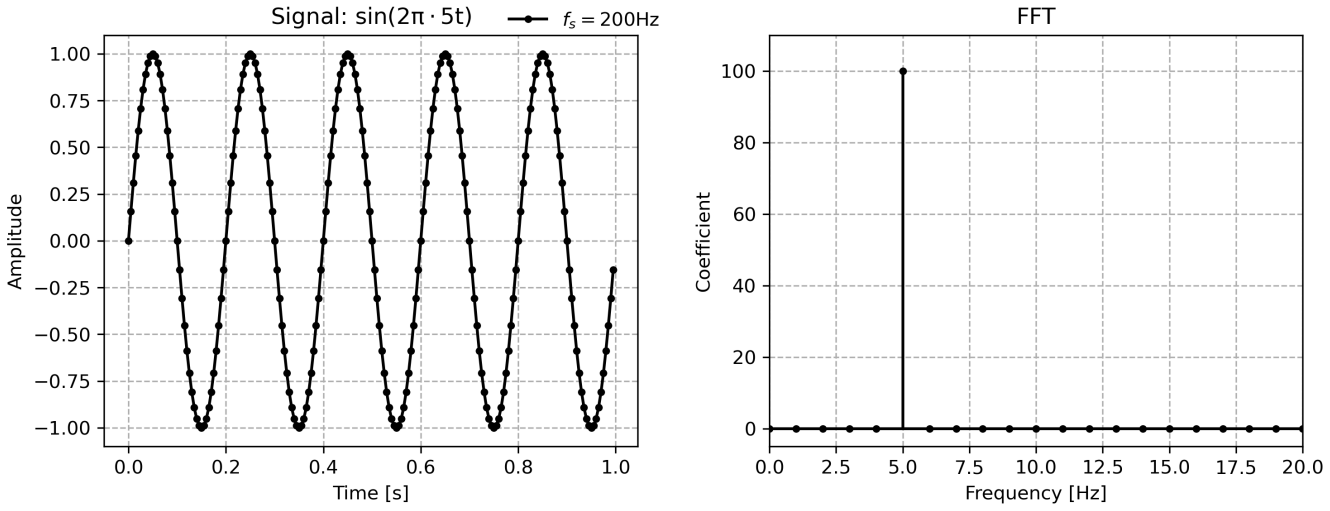
$$X(f) = \int_{-\infty}^{+\infty} x(t) e^{-2\pi i f t}\, dt$$

However, in the real application, the signal is always discrete and finite, and **Discrete Fourier Transform** is actually applied:

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

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

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



An example of DFT

```
# Generate a sinuous signal
omega = 2 * np.pi * 5
time = np.arange(0, 1, 200)
signal = np.sin(omega * time)

dt = time[1] - time[0]
# Complex coefficient
coefs = np.fft.fft(signal)
# Corresponding frequency with both zero, positive, and negative frequency
freqs = np.fft.fftfreq(coefs.size, dt)
```

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

```
f = [0, 1, ...,   n/2-1,     -n/2, ..., -1] / (dt * n)   if n is even
f = [0, 1, ..., (n-1)/2, -(n-1)/2, ..., -1] / (dt * n)   if n is odd
```

So, the sinuous waves can still be well-decomposited by DFT.

```
f[::n // 2] = [0, 1, ...,   n/2-1] / (dt * n)   if n is even
f[::n // 2] = [0, 1, ..., (n-1)/2] / (dt * n)   if n is odd
```

For an even signal length *n*, the Nyquist Frequency and corresponding coefficient is actually ignored. So, it is suggested to use *numpy.fft.rfft (rfftfreq)* instead of *numpy.fft.fft (fftfreq)*, which intrinsically truncate the output coefficients and frequencies.

```
coefs = np.fft.rfft(signal)
freqs = np.fft.rfftfreq(coefs.size, dt)
```

Decibel

Decibel (dB) 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.

Product key: ZFPZRW-YW9P82-CHNEQK-KF3VRS

Frequency Resolution

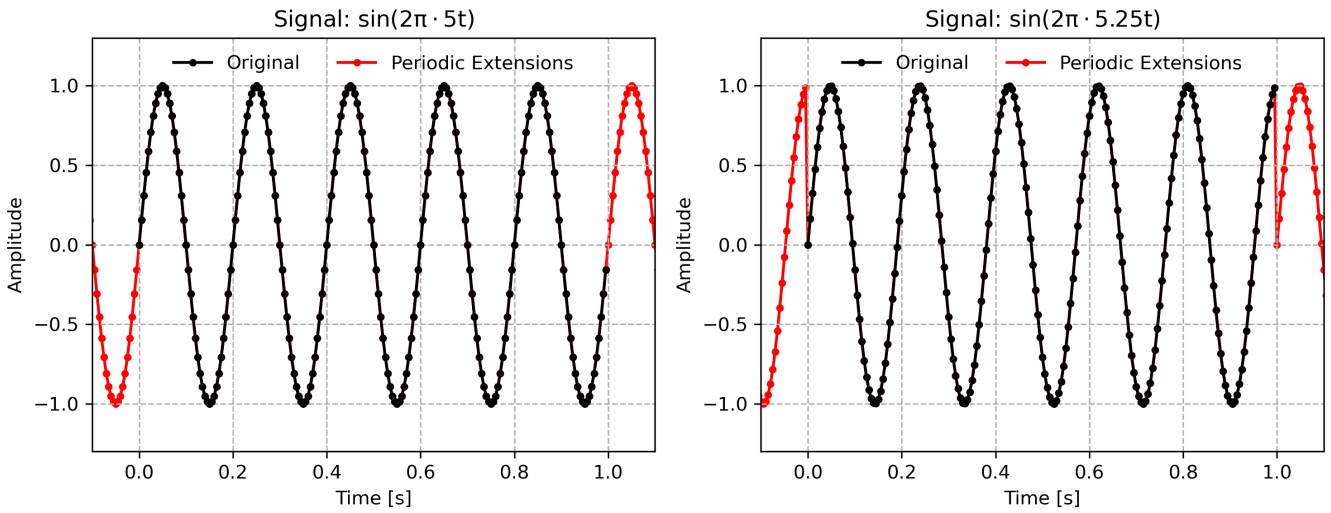
Assuming you already get a prepared signal, a common way to extract the periodicity from the signal is *DFT*. By this transformation, you can perfectly convert the signal to the frequency domain without any loss of the physical information.

The yield spectrum contains the wave coefficient at the frequency of

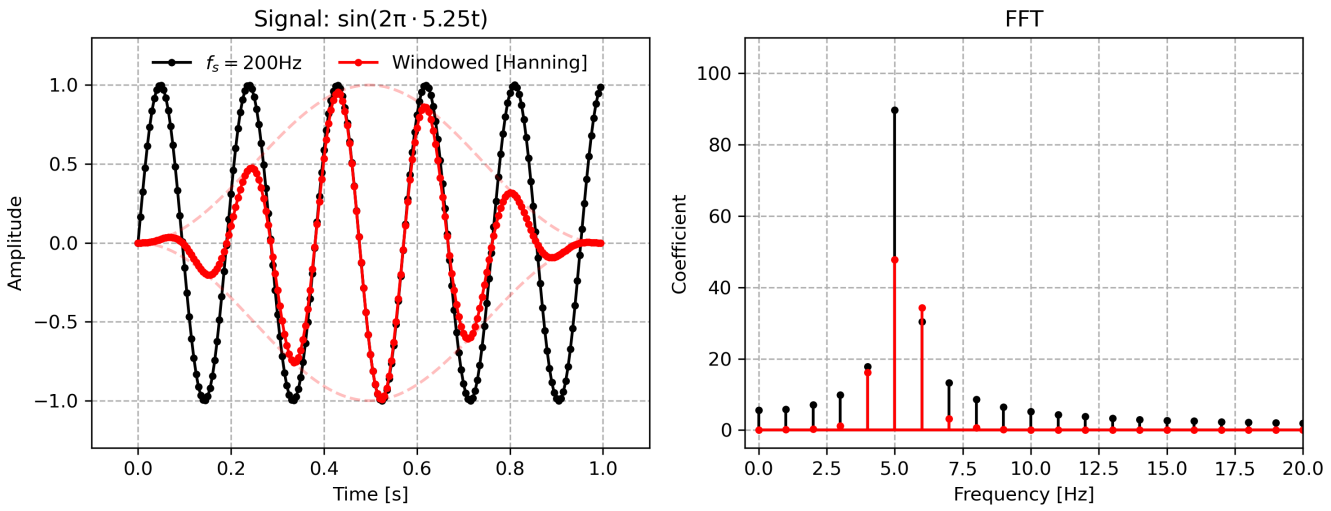
$$\frac{i}{N}\frac{f_s}{2} = \frac{i/2}{N\Delta t} = \frac{i/2}{\Delta T}$$

where $i = 1, 2, \ldots, N$ and $\Delta T = N\Delta t$ is the total duration of the signal.

Periodic Extensions



Windowing Effect




```
# Without Normalization
signal *= np.hanning(signal.size)
# With Normalization
signal *= np.hanning(signal.size) * np.sqrt(8 / 3)
```

The Hanning window is written as:

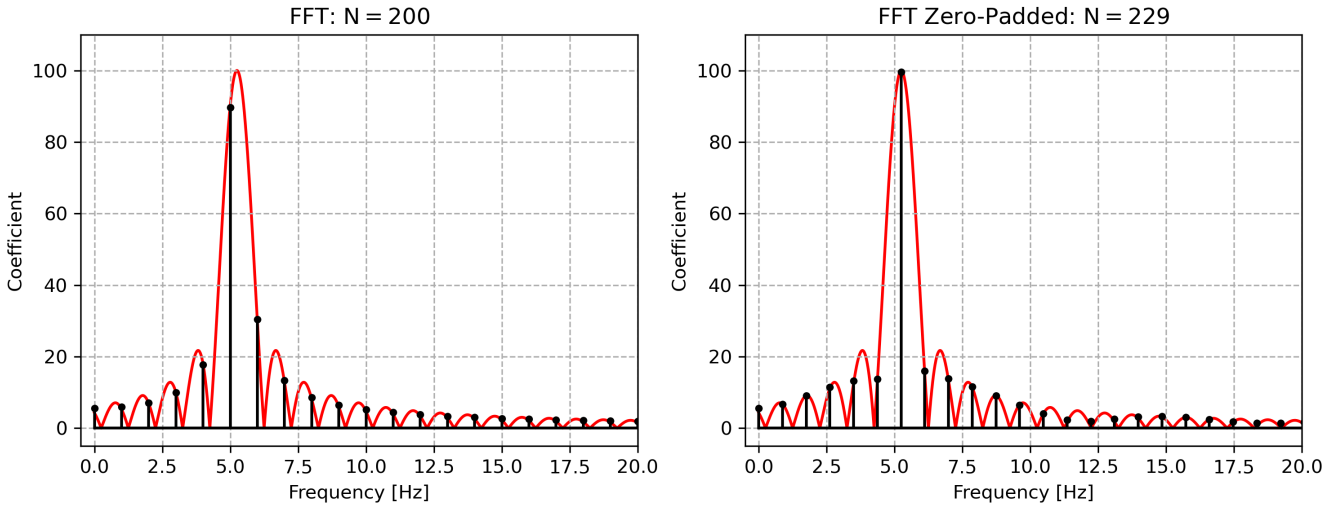
$$w(x) = \frac{1}{2}[1 - \cos(2\pi x)]$$
$$w[n] = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi n}{N}\right)\right]$$

which has an average energy of

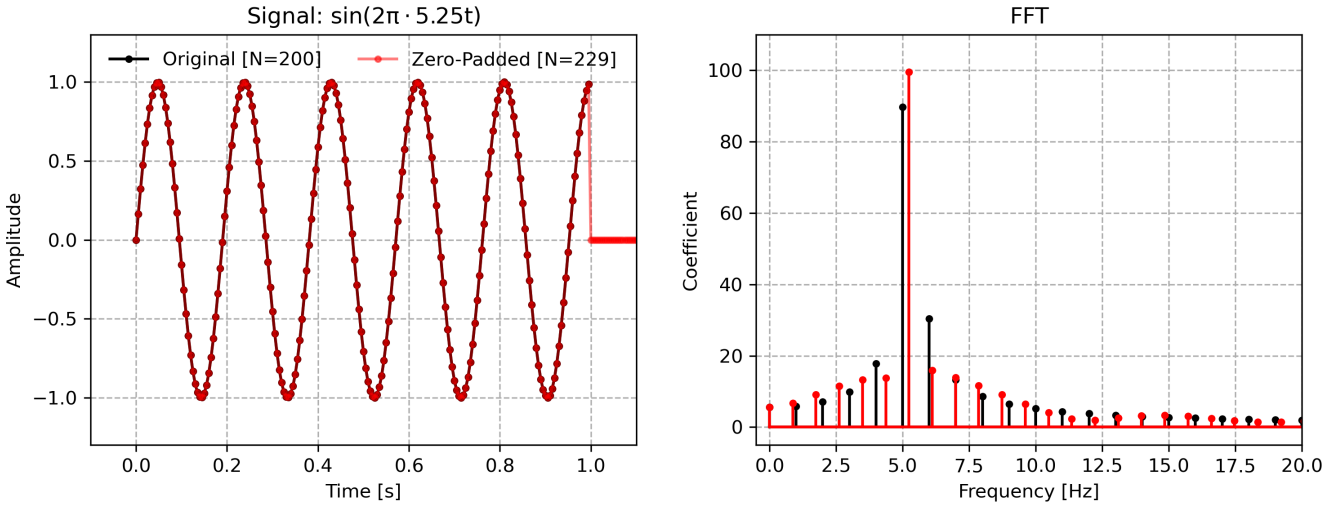
$$\int_0^1 w^2(x)\mathrm{d}x = \frac{1}{4}\left\{\int_0^1 [1 - 2\cos(2\pi x) + \cos^2(2\pi x)]\mathrm{d}x\right\} = \frac{1}{4}(1 - 0 + \frac{1}{2}) = \frac{3}{8}$$

Window Functions	Expression	Energy Normalization Factor
Bartlett [*numpy.bartlett(M)*]	1 - $\left\|\frac{n-N/2}{L/2}\right\|$	
Blackman [*numpy.blackman(M)*]		
Hamming		
Hanning		
Kaiser		

Fence Effect



Zero-Padding




```
dt = time[1] - time[0]
n = signal.size

N_PADDING = 29

coefs = np.fft.fft(signal, n = signal.size + N_PADDING)
freqs = np.fft.fftfreq(coefs.size, dt)
```

Gibbs phenomenon

Uncertainty Principle

In

Parseval's Theorem and Energy Conservation

> **Parseval's Theorem for CFT:**
>
> $$\int_{-\infty}^{\infty} x^2(t)\,dt = \int_{-\infty}^{\infty} X^2(f)\,df$$

> **Parseval's Theorem for DFT:**
>
> $$\sum_{n=0}^{N-1} |x(n\Delta t)|^2 = \frac{1}{N}\sum_{k=0}^{N-1} |X(k\Delta f)|^2$$

In the physical world, the square power of the ampltitude often refers to some kind of ***energy*** or ***power***. For example, the square of the displacement (x) of a spring, x^2 is propotional 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

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

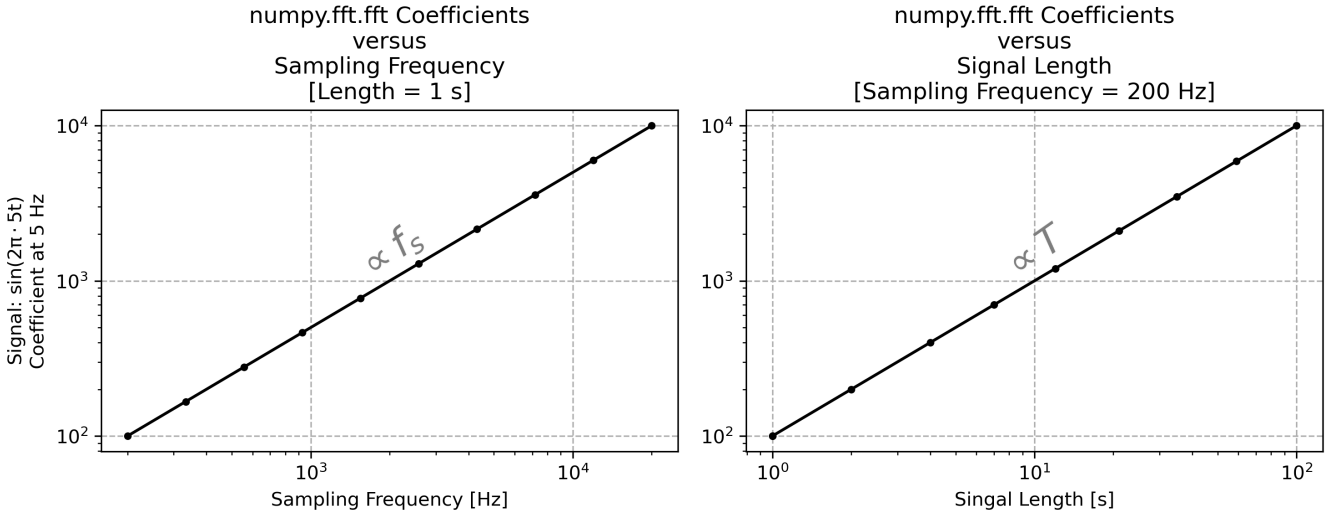
for DFT. Considering that DFT yields both positive and negative frequency, we typically fold the DFT result. Naturally, the definition of *power spectral density (PSD)* is given as:

$$\sum_{k=0}^{N-1} PSD(k\Delta f)\Delta f =$$

$$\text{For Even } N: \ \Delta f \left[PSD(f_0) + \sum_{k=1}^{(N-1)/2} 2 \cdot PSD(k\Delta f) + PSD(f_{N/2}) \right]$$

$$\text{For Odd } N: \ \Delta f \left[PSD(f_0) + \sum_{k=1}^{(N-1)/2} 2 \cdot PSD(k\Delta f) \right]$$

```
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
```



$$|X(k\Delta f)| \propto f_s \cdot T$$

According to the lineairty of $\mathcal{F}$, $X(k\Delta f)$ should also be propotional to the signal ampltitude. Easily catch that the coefficient at the exact wave frequency has the form of

$$|X(k\Delta f)| = \frac{1}{2} A(k\Delta f) \cdot f_s \cdot T$$

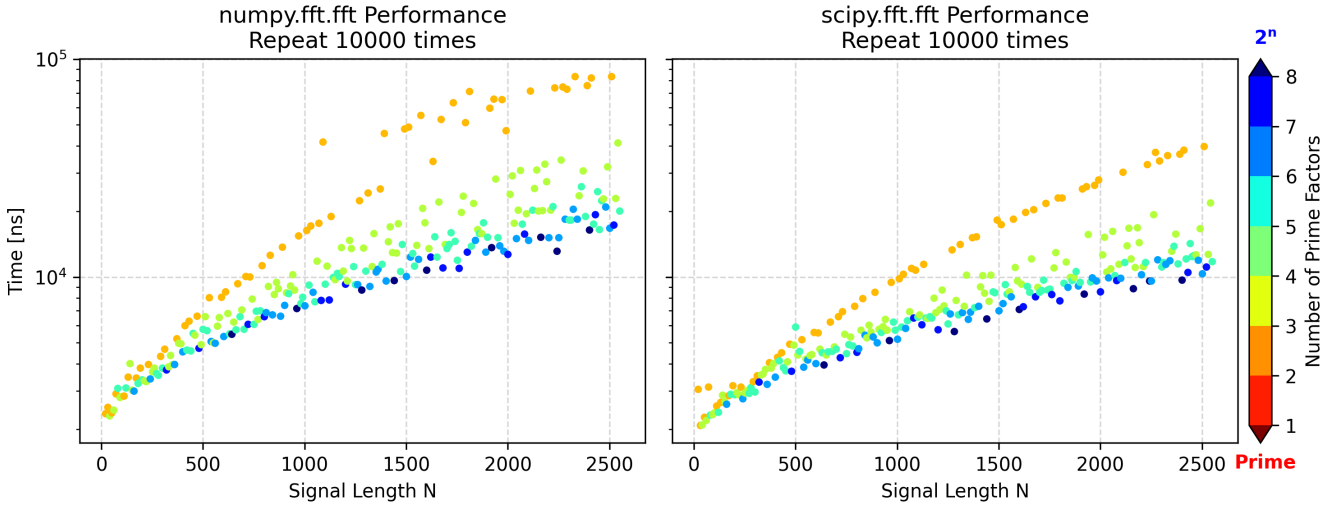
1/2 in this equation arises from the fact that $\int_0^{2\pi} \sin^2 x \mathrm{d}x = 1/2$.

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:

$$N = 197 = 198 - 1 = 2^1 \times 3^2 \times 11^1, \quad N = 241 = 242 - 1 = 2^1 \times 11^2 - 1$$

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 (e.g.):

$$N = 199 = 200 - 1 = 2^3 \times 5^2 - 1, N = 239 = 240 - 1 = 2^4 \times 3^1 \times 5^1 - 1$$

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 $N \times K$.

Takeaway Message:

1. For large-scale FFT computations, choose a proper signal size to speed up.
2. Adopt `scipy.signal.fft(workers = -1)` when you want to do parallel computation.

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/

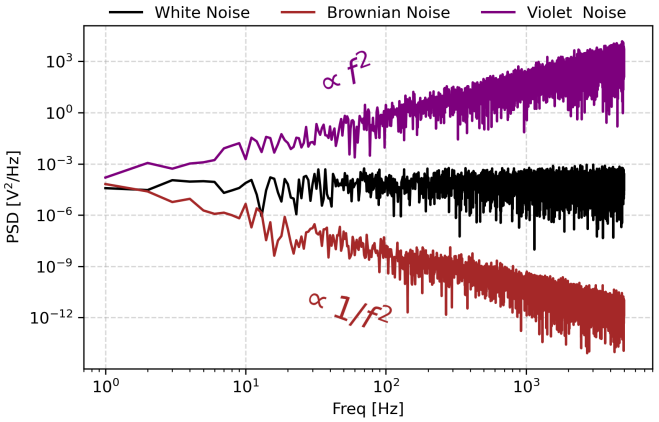
Sliding Window

`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.

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.

How to generate a colored noise?

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

```
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 method, one can also get a colored noise by filtering a white noise. An colored noise that accurately follows its expected power spectrum requires the order of the filter to be high enough. Even though, this

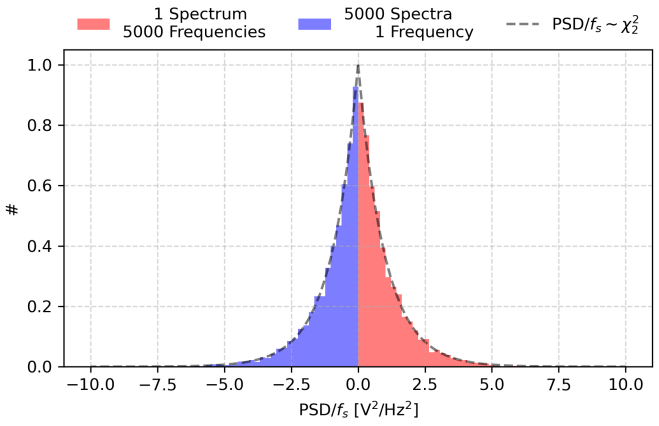
"Noise" of Noise

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

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

can be deemed as a weighted summation of the signal $x[n]$. When $x[n]$ are independent identically distributed 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 part, naturally follows the *Kappa* Distribution with the freedom of 2. The above statement requires the real and imaginary parts are independent to each other, which can be proved by calculating their covariance.



It should be noted that the wave 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.

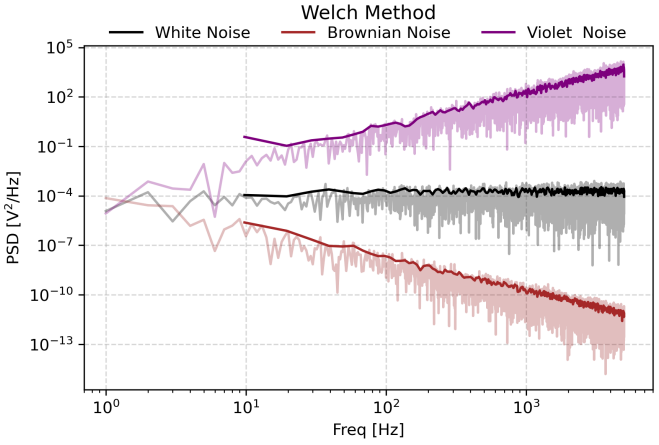
The averaging operation must be taken after the conversion from coefficient to power other wise the averaged coefficients are actually unchanged.

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.

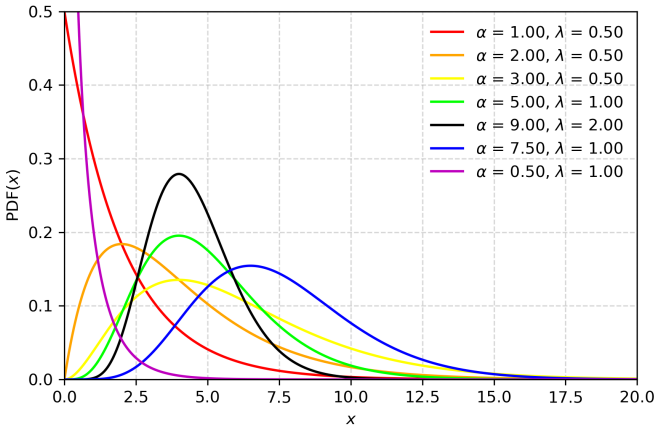
For each segement, you can also chose the window function to reduce the spectral leakage. The result of this method is shown below:



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 $(N\Delta t)^{-1}$.

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`.

(Auto)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

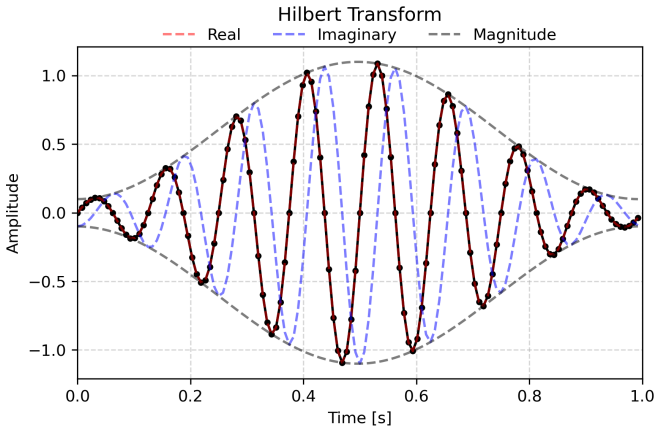
For a wide-sense stationary signal, its power spectral density is equal to the the fourier transform of its autocorrelation function, i.e.,:

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

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

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.

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`.

Moving-Average and Moving-Median

Moving-average and moving-median filters are essential tools for smoothing time series and removing high-frequency noise. They are simple yet effective for trend extraction, baseline correction, and outlier suppression. This section compares these techniques, discusses their strengths and limitations, and provides Python code snippets for practical use. `bottleneck` and `numpy.sliding_window`.

Cross-Spectral Density

Cross-spectral density (CSD) quantifies the frequency-domain relationship between two signals, revealing shared oscillatory components and phase relationships. It forms the basis for advanced techniques such as coherence and transfer function estimation. This section covers the theory behind CSD, its estimation using Welch's method, and real-world applications in system identification and geophysics.

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.

Combination with Maxwell's Equations

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

Polarization

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.

Compressibility

The wave compressibility refers to the energy ratio of the compressional component and the transverse component of the waves, i.e.,

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

It can also be represented by

Jargon Sheet