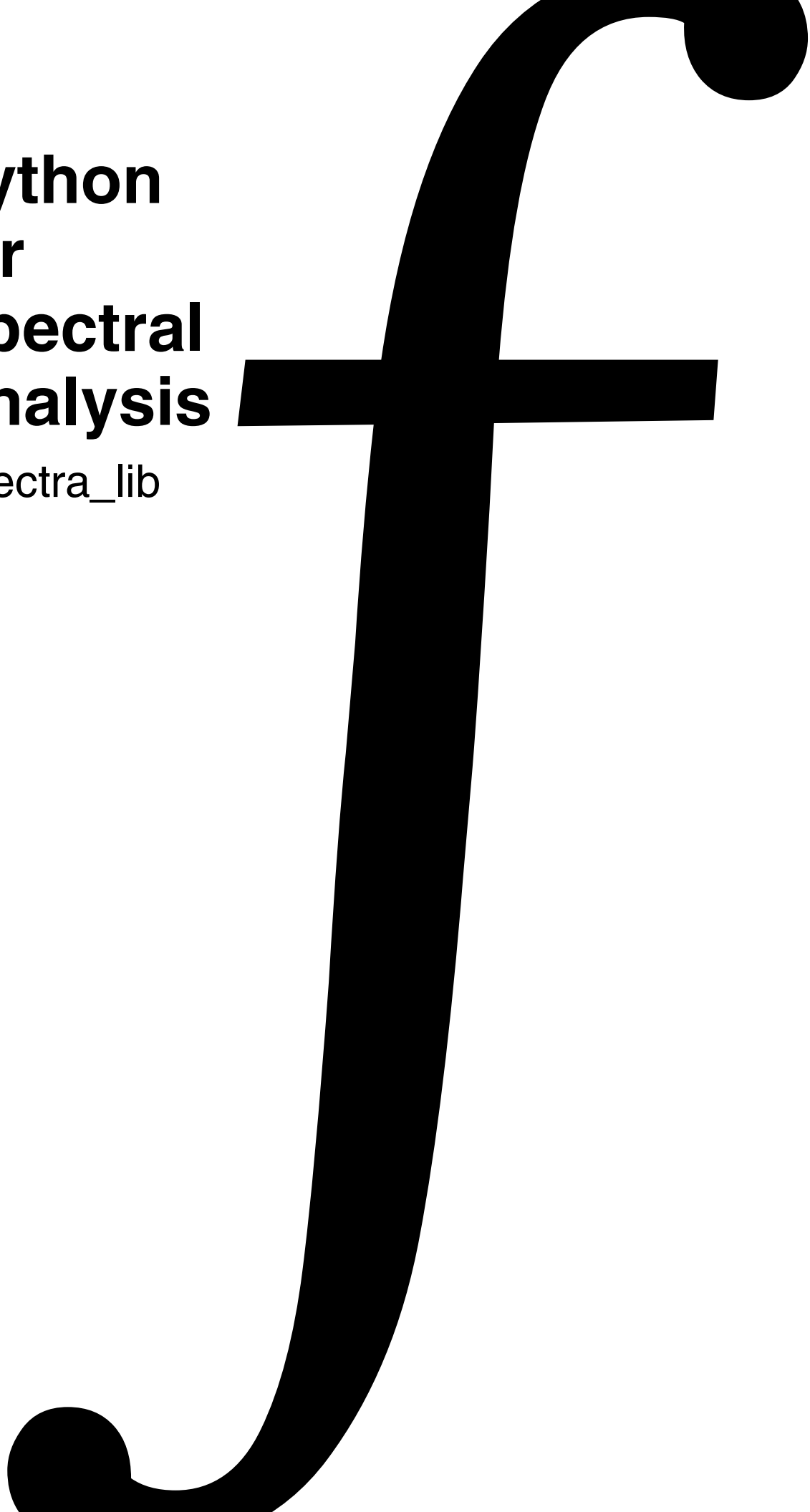
Python for Spectral Analysis

Spectra_lib



Python for Spectral Analysis

Modestino Carbone

October 2024

Contents

Preface

This small essay was made by the author to help people to menage the chaos of the actual mathematical tools commonly used in python for the frequency domain and support the development of custom personalized tools. **Feel free to change this coding ideas and to copy the part of your interest**. This work should be seen only as a starting point like the mathematical references included. If something seems fuzzy or ambiguous probably a real Signal processing book will be better then this rough summary.

Chapter 1

Basic spectrum Functions

1.1 PSD

The PSD (**Power Spectral Density**), usually described as the distribution of power across individual frequency components of a measured signal, is a mathematical tool strongly used in mechanical, electrical, and electronic systems as well as across numerous engineering disciplines. Harmonic analysis is a complete and exhaustive method in order to evaluate the Power distribution of a specific source of noise.

Definition

Given x(t) a stochastic physical quantity and its time windowed version $x_T(t)$ such as:

$$x_T(t) = \begin{cases} \text{x(t)}, & \text{if } |t| < T, \\ 0, & \text{if } |t| > T. \end{cases}$$

The PSD of x(t) is

$$PSD_x(f) = \lim_{T \to \infty} \frac{\mathbb{E}[|X_T(f)|^2]}{2T}$$

The definition of an algorithm to estimate the Power Spectral Density has two main goals: Speed up the computation, display clear plots. There are two main methods of estimation:

1. **Bartlett's Method**, a general purpose Method
2. **Welch's Method**, an efficient variation of the Bartlett's Method

1.1.1 Bartlett's Method

Bartlett's Method consists in computing the DFT (Discrete fourier transform) several times over contiguous and non overlapping time windows.

It is possible the FFT computation in the first window of M samples such as,

$$X_0(k) = \mathcal{F}\{x[0], x[1], x[2], ..., x[M-1]\} \tag{1.1}$$

The second window will be,

$$X_2(k) = \mathcal{F}\{x[M], x[M+1], x[M+2], ..., x[2M]\} \tag{1.2}$$

If N time windows are available a number of N FFTs of distinct time portions can be obtained. The computation of the average of these FFTs is a good estimation of the PSD as follows

$$S_x(k) = \frac{1}{N}\sum_{j=0}^{N} |X_j(k)|^2 \tag{1.3}$$

1.1.2 Welch's Method

Welch's Method is computed over 50 % overlapping windows instead of the consecutive and distinct time windows. The overlap mechanism is clearer in the following equations.

$$X_0(k) = \mathcal{F}\{x[0], x[1], x[2], ..., x[M-1]\} \tag{1.4}$$

$$X_1(k) = \mathcal{F}\{x[M/2], x[1], x[2], ..., x[3/2M-1]\} \tag{1.5}$$

$$X_2(k) = \mathcal{F}\{x[M], x[1], x[2], ..., x[2M-1]\} \tag{1.6}$$

If the number of samples is fixed the Welch's Method has a double number of FFT which means that the equation that estimate PSD (1.7) will obtain a smoother trace.

$$S_x(k) = \frac{1}{N}\sum_{j=0}^{N} |X_j(k)|^2 \tag{1.7}$$

This method is not a good choice if samples are acquired in different time slots with a conspicuous idle time between measures. In the case of a long pulse train of samples or short acquisition this method is perfect because it doubles the number of possible windows of the same input signal. By Fixing the duration of a generic data stream, chopped into $\boldsymbol{m}$ possible windows, the application of Welch method makes the number of windows equal to $\boldsymbol{2m-1}$.

1.1.3 Python code

The code below follows the mathematical notes already exposed. This minimum form can be considered a minimum form of the final algorithm, a small brick. **Psd_basic** has **x** as input value, **fs** as sampling frequency and finally **N** as number of samples. By looking in the box it's possible to observe computational constants in the final formula. Note that:

1. The PSD is divided by **N*fs** the resolution of the FFT because it is a density by definition
2. The PSD is multiplied by 2 because what we want to plot is the right part of the spectrum and the total power must be always the same the 2 factor can compensate the lost contribution.

Psd_basic

```
import numpy as np

def Psd_basic(x,fs,N):

   T = N*(1/fs)
   t = np.arange(0,T,1/fs) #Time vector
   X = np.fft.fft(x)  #fft of the input

   PSD  = 2*(1/fs)*X*np.conj(X)/(N)
   freq = np.fft.fftfreq(t.shape[-1],d=1/fs)

   return freq[0:int(N/2)], PSD[0:int(N/2)]
```

Two fundamental features are still missing to a proper PSD estimator:

1. **Windowing**: Without a windowing method the discontinuity at the boundaries of the acquired samples can not be mitigated.
2. **Averaging**: A good PSD estimator should include an averaging mechanism like shows the Welsh and the Bartlett's methods.

In order to solve the first problem the window generator functions *windows.get_windows* of the *scipy* library can be used as a mask for **x** samples. The window can be represented as a train of pulses with the following discrete summation:

$$W(n) = \sum_{j=0}^{N} w_j \times \delta(n-j) \tag{1.8}$$

Let's define the S factor as the summation of the coefficients of the previous discrete windowing function [1].

$$S = \sum_{j=0}^{N} w_j^2 \tag{1.9}$$

The new generic definition of the PSD which is totally independent from the shape of the window is now complete:

$$PSD_x = \frac{2 \times |X|^2}{S \times f_s} \tag{1.10}$$

Mathematical equations allows a new function definition:

Psd_windowed

```
import numpy as np
from scipy import signal

def Psd_windowed(self,x,fs,N,window = 'hann'):

    win = signal.windows.get_window(window,N)
    T = N*(1/fs)
    t = np.arange(0,T,1/fs)
    X = np.fft.fft(x*win)
    PSD = 2*(1/fs)*X*np.conj(X)/( (win* win).sum())
    freq = np.fft.fftfreq(t.shape[-1],d=1/fs)

    return freq[0:int(N/2)], PSD[0:int(N/2)]
```

Previous operations create a good basis to write a complete usable function comprehensive of an averaging operation and windowing function.

Psd_Bertlett

```
import numpy as np
from scipy import signal

def Psd_Bertlett(self,x,fs,N,avg,window = 'hann'):

    T=N*(1/fs)
    t = np.arange(0,T,1/fs)
    win = signal.windows.get_window(window,N)
    if (len(x)/N >= avg):
        psd_buff = []
        for i in range(avg):
            X = np.fft.fft(x[i*N:i*N+N-1]*win)
            app = 2*(1/fs)*X*np.conj(X)/((win*win).sum())
            psd_buff.append(app)
        buff = np.array(psd_buff)
        PSD = (np.sum(buff , axis=0)/avg)
    else:
        X = np.fft.fft(x)
        PSD = 2*(1/fs)*X*np.conj(X)/(N)
    freq = np.fft.fftfreq(t.shape[-1],d=1/fs)

    return freq[0:int(N/2)], PSD[0:int(N/2)]
```

1.2 The infamous "2/N" factor

For the sake of simplicity the following section will be referred to the Fft (Fast fourier Transform) function, the simplest block that is possible to write in python.

Fft_simple

```
def Fft_simple(x,fs,N):

    T=N*1/fs
    t = np.arange(0,T,1/fs)
    sp = np.fft.fft(x)
    freq = np.fft.fftfreq(t.shape[-1],d=1/fs)
    fft = 2*np.abs(sp)/N

    return freq[0:int(N/2)], fft[0:int(N/2)]
```

Imagine we want to use the fft function of numpy library to compute the absolute value of the fourier trasform of a signal (fig. 1.1, left). For the Physical world negative frequencies are meaningless so, in order to distribute the total power in a complete plot with only the positive axis of frequency, the fft must be multiplied by 2 (fig. 1.1, right). The source signal of both the plots is the following:

```
X(t) = 1.0 + 0.5 * np.sin(2 * np.pi * 5 * t)
```

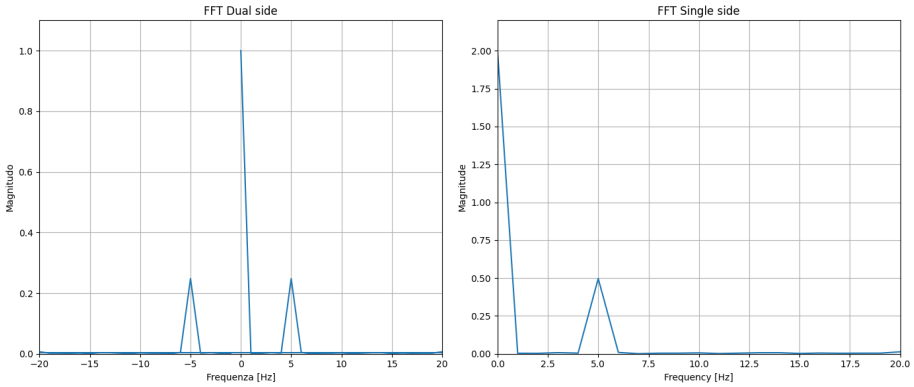


Figure 1.1: visual proof of mistake that the factor "2/N" can cause

Notice that the right plot in the figure present a Magnitude peak of 0.5 with the frequency of 5 HZ (the same Magnitude of the input expression), the Magnitude of the 0 Hz frequency is 2 times the magnitude of the value expected. Starting from this example it is possible to conclude that the continuous component cannot be multiplied by 2 like others harmonics because it is the only point that doesn't mirror in the negative half-plane, like a symmetry axis. The usage of a function with the **infamous** "2/N" factor is legit when the x axis is logarithmic one (which never reaches frequency of 0 Hz).

1.3 ASD

If the PSD is used to observe the distribution of power along frequencies (eg. $[V^2/Hz]$), the ASD (**Amplitude Spectral Density**) is a tool used to describe the amplitude of a specific spectral component expressed in his relative unit (eg. $[\mathrm{V}/\sqrt{Hz}]$).

Definition

Given x(t) a stochastic physical quantity and its time windowed version $x_T(t)$ such as:

$$x_T(t) = \begin{cases} \mathrm{x(t)}, & \text{if } |t| < T, \\ 0, & \text{if } |t| > T. \end{cases}$$

The ASD of x(t) is

$$ASD_x(f) = \sqrt{PSD_X(f)}$$

1.3.1 Python code

By observing the definition in the blue-box, the function is really simple. Notice that in the code below, the ASD uses the Bertlett method to compute the square root of the PSD.

Asd_Bertlett

```
def Asd_Bertlett(x,fs,N = None,avg,window = 'hann'):

    freq, PSD = Psd_Bertlett(x, fs, N, avg, window)
    ASD = np.sqrt(PSD)

    return freq, ASD
```

1.4 CSD

The CSD (**Cross-Spectral Density**) expand possibility unlocked by PSD, the concept of single channel power distribution is substituted with the research of a relationship between two signals. This "relationship" is commonly measured with the cross-correlation function $R_{yx}(t)$ which is an integral method to enhance common characteristic of two signals.

Definition

Given x(t) and y(t), two stochastic physical quantities,
the CSD of x(t) and y(t) is

$$CSD_{yx}(f) = \lim_{T\to\infty} \frac{\mathbb{E}\{\mathcal{F}[R_{y,x}(t)]\}}{2T}$$

It is possible to prove an equivalence between the Fourier transform of the cross-correlation function $R_{yx}(t)$ and the product of the two signals X(f)* (conjugate) and Y(f) in the frequency domain.

$$\mathcal{F}[R_{y,x}(t)] = X(f)^* Y(f) \tag{1.11}$$

By observing this property of cross-correlation (eqn. 1.11) the equation that link PSD and CSD is the following one:

$$PSD_x(f) = CSD_{yx}(f) \qquad if \qquad y(t) = x(t) \tag{1.12}$$

1.4.1 Python code

Like seen in the previous section we want some features in order to write a fully functional python code such as windowing and averaging. By following previous tips and definitions the implementation is really easy to understand.

Csd_basic

```
import numpy as np
from scipy import signal

def Csd_basic(x,y,fs,N,avg,window = 'hann'):

    T=N*(1/fs)
    t = np.arange(0,T,1/fs)
    win = signal.windows.get_window(window,N)

    if (len(x)/N >= avg):
        csd_cross=[]
        for i in range(avg):
            X = np.fft.fft(x[i*N:i*N+N-1]*win)
            Y = np.fft.fft(y[i*N:i*N+N-1]*win)
            app= 2*(1/fs)*Y*np.conj(X)/((win*win).sum())
```

```
            csd_cross.append(app)
        buff = np.array(csd_cross)
        CSD = (np.sum(buff , axis=0)/avg)
    else:
        X = np.fft.fft(x)
        Y = np.fft.fft(y)
        CSD= 2*(1/fs)*Y*np.conj(X)/(N)
    freq = np.fft.fftfreq(t.shape[-1],d=1/fs)

    return freq[0:int(N/2)], CSD[0:int(N/2)]
```

Notice that CSD has an imaginary part and a real part ; PSD, instead, can reach only real values. The real part and an imaginary part of the cross-spectrum have two different meaning that should not be confused. The absolute value of CSD has a simpler and straight forward meaning as it is the measure of how much the two channel are correlated each other.

1.5 Spectral Coherence

By combining the definition of the PSD and the CSD a new tool can be defined: Spectral Coherence. Coherence can be seen as a normalization of CSD to obtain a plot not influenced by the actual power values of the two input signals.

Definition

Given x(t) and y(t), two stochastic physical quantities,
the Coherence of x(t) and y(t) is

$$\text{Coherence}(f) = \frac{|\text{CSD}_{xy}(f)|^2}{\text{PSD}_x(f) \cdot \text{PSD}_y(f)}$$

Notice that Coherence, that due to its mathematical definition, can only be positive and lower or equal to 1.

1.5.1 Python code

The final Python code reuses the old functions and combine them in his last line only to compute the Coherence functions.

Csd_basic

```
def Coherence( x, y, fs, N=None, avg=1, window='hann'):

    freq, CSD = Csd_basic(x, y, fs, avg,  N,window)
    _, PSD_x = Psd_Bertlett(x, fs, avg, N, window)
    _, PSD_y = Psd_Bertlett(y, fs, avg, N, window)
    coherence = (np.abs(CSD) ** 2) / (PSD_x * PSD_y)

    return freq, coherence
```

Chapter 2

Advanced averaging methods

Observing logarithmic plots it is possible to notice that higher frequencies doesn't need the same resolution of the lowest one. It is possible to store a store a smaller amount of frequency points to observe similar quality of an higher resolution one. Resolution is linked to the windowing technique. Final resolution of a FFT plot will be the ratio between the sample frequency and the number of samples per windows. Assuming that the final spectrum plot will be divided into decades, higher decades can use smaller window instead lower decades uses the higher one.
Starting from the idea of a variable windowing system two new algorithms of PDS or CSD are possible:

- **Fixed average**: The number of averages is fixed for every decades (fig. 2.1, a).
- **Maximum average** The number of averages is not fixed for every decades because for higher decades more timing windows are available (fig. 2.2, b).

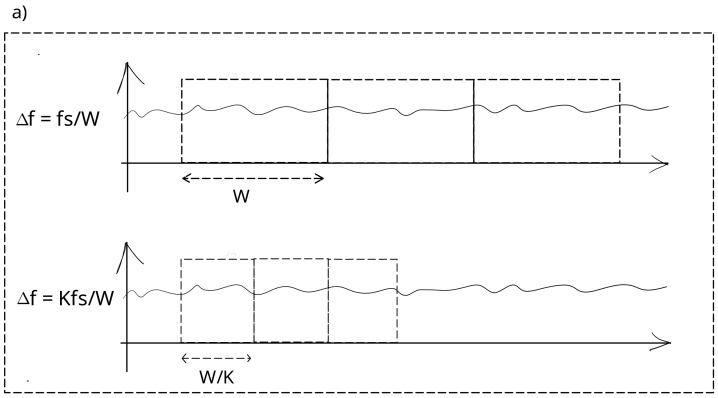


Figure 2.1: Fixed average algorithm

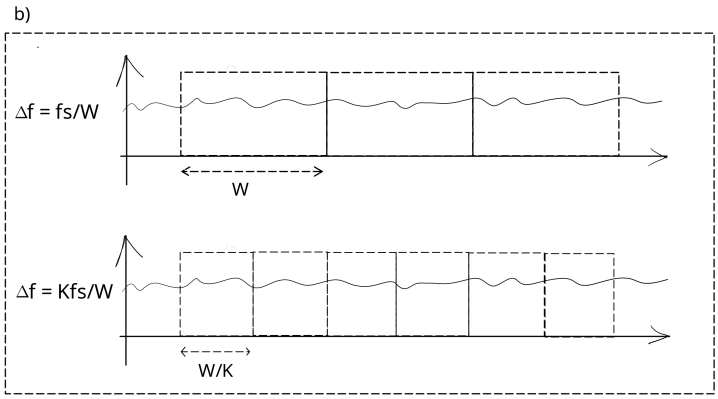


Figure 2.2: Maximum average algorithm

Bibliography

[1] Gerhard Heinzel, Albrecht Rüdiger **and** Roland Schilling. "Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new at-top windows". **in**(2002).