PHY408 Lab 4

Wednesday, April 3, 2024

Jace Alloway - 1006940802 - alloway1

This assignment was compiled in VS Code using the Python and LaTeX extensions. Matplotlib does not produce inline plots in terminal, so the `%matplotlib inline` command was commented out. **Collaborators for Q1: none; for Q2: Juan Miguel L. Padilla**.

Problem 1

(**a**) We begin by defining a cross-correlation function which takes in two input arrays at a sampling interval dt. By the convolution theorem,

$$C_{xy}(\tau) = \int_{-\infty}^{\infty} dt\, \overline{x}(t) y(t+\tau) \tag{1.1}$$

which, by Fourier Transforming, implies that

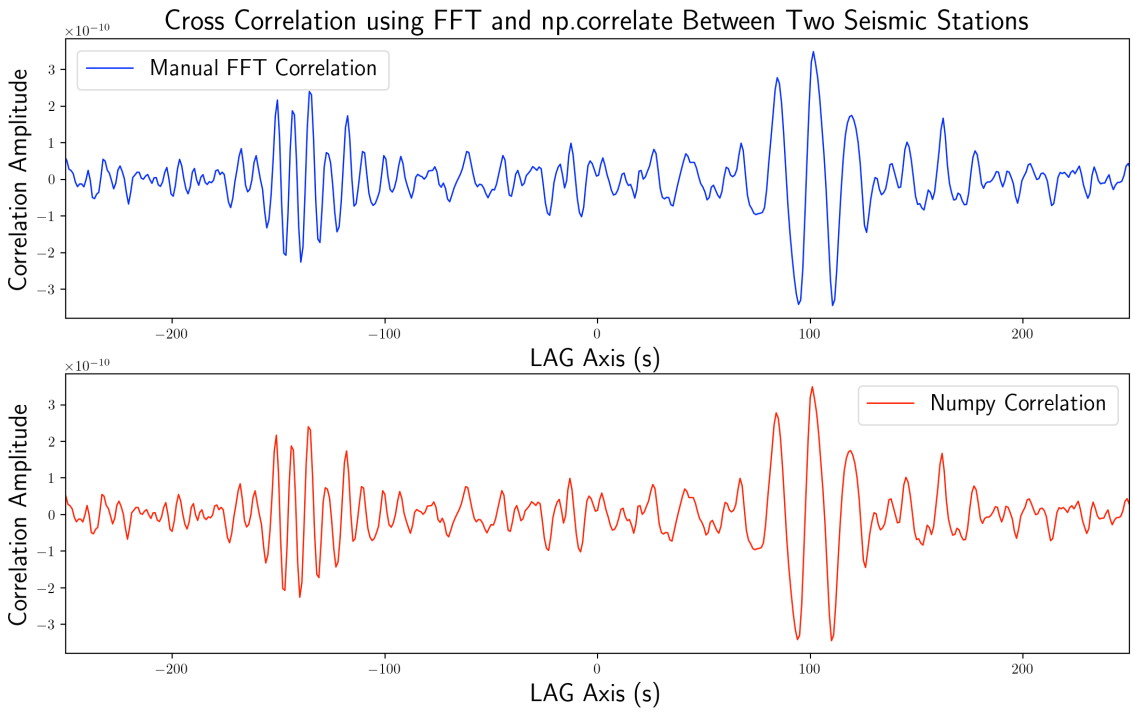
$$C(\omega) = \overline{X}(\omega) Y(\omega) \tag{1.2}$$

where X and Y are the Fourier transforms of x and y. In order to take the correlation in the temporal domain, which is multiplication in the frequency domain by the convolution theorem, we must make sure both arrays are of the same size. With direct convolution, two arrays of length $N-1$ convolve into an array of length $2N-1$. Thus, for Fourier transforms, we must input two arrays of length $2N-1$, not $N-1$. To fix this, we apply `np.pad(x, (0,N), 'constant')` to x_n and y_n arrays to make them the same size.

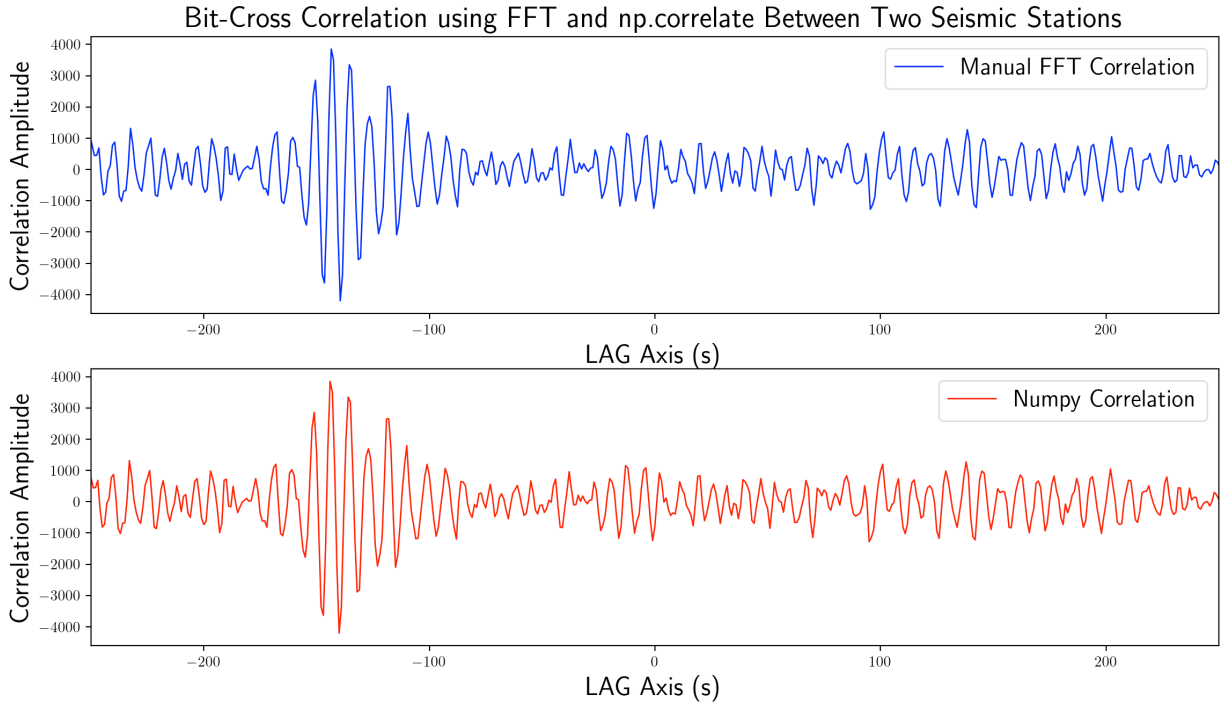
We can proceed by applying `np.fft.fft()` to each of the padded arrays x_n, y_n, then using `np.conjugate()` to take the complex conjugate of X_k (the FFT of x_n). Multiplying both arrays together thus gives C_k, from which we can use `np.fft.fftshift` to shift the output to have the zero-frequency component at $t = 0$. The shifted time axis was defined with `np.arange(-len(out)/2, len(out)/2, dt)` using the direct length of the $2N-1$ output.

After loading the data and stacking it, it was run through our defined correlation function and compared with the `np.correlate()` function with the time domain sliced between $(-250, 250)$ seconds.

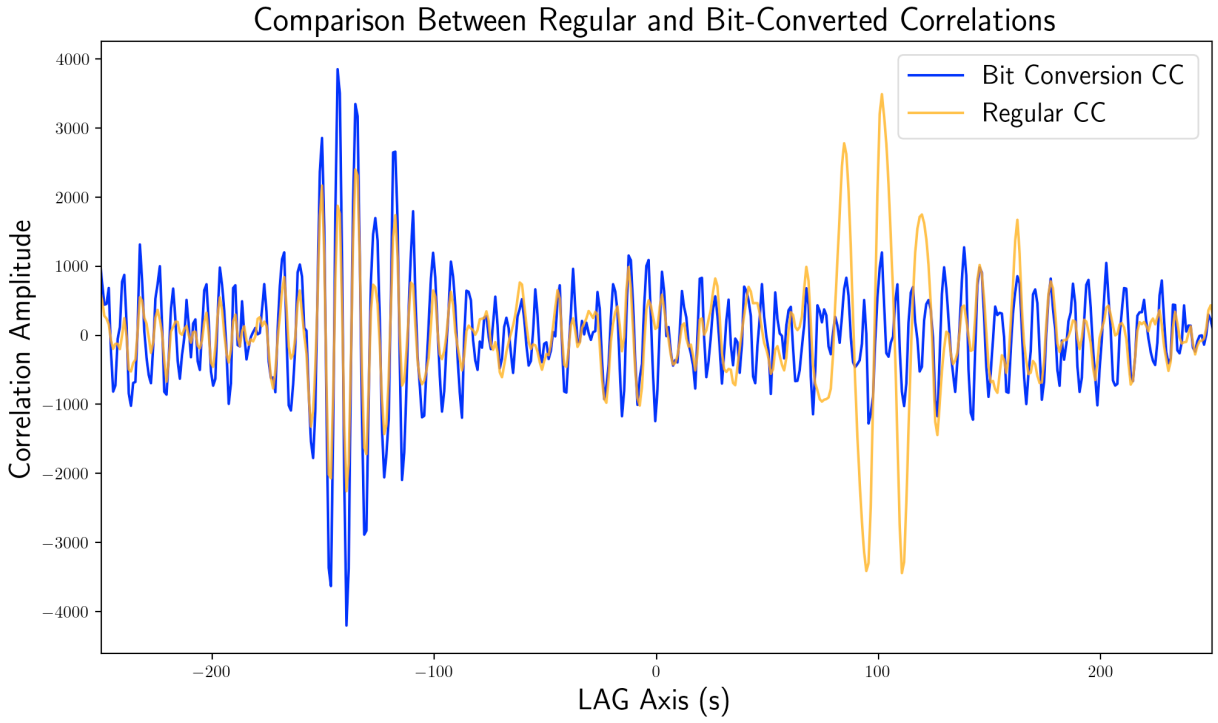
```
def cross_correlate(f, g, dt):
    N = len(f)
    f = np.pad(f, (0, N-1), 'constant') #pad arrays
    g = np.pad(g, (0, N-1), 'constant')
    f_fft = np.fft.fft(f)*dt    #fft arrays into frequency domain
    g_fft = np.fft.fft(g)*dt
    c = np.conjugate(g_fft) * f_fft #multiply arrays
    out = np.fft.ifftshift(np.fft.ifft(c)*dt)   #shift output
    lag_axis = np.arange(-len(out)/2, len(out)/2, dt)   #define axis
    return (lag_axis, out)
```



(**b**) Now using `np.sign()`, the x_n and y_n input arrays were converted to their positive and negative bit arrays depending on the sign of each entry. Again, using our correlation function, the bit-converted arrays were cross-correlated and compared with the output from `np.correlate()`.



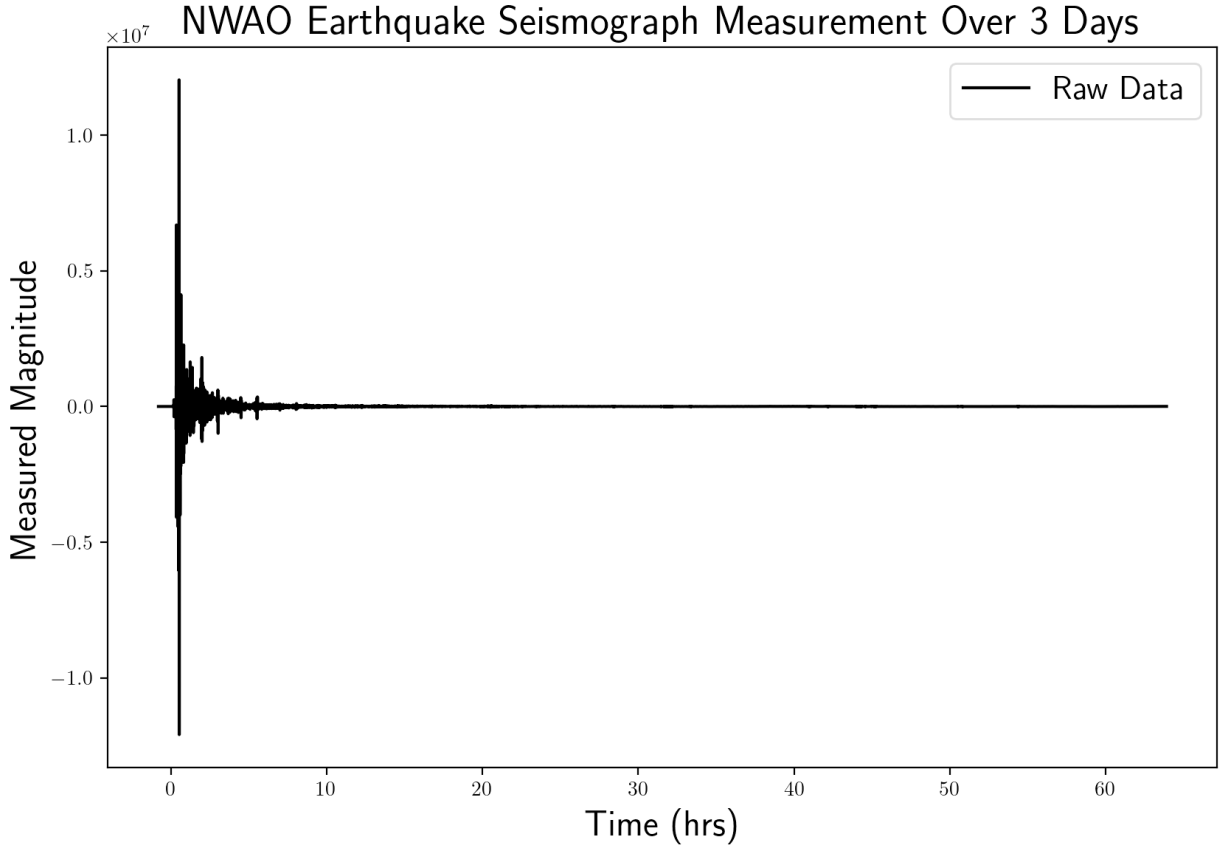
Furthermore, comparing the bit-correlated signals with the original signal from part (a), we find



that the bit-correlated array is not the same as the regular cross-correlation, as the latter half of the peaks and valleys are not as emphasized as in the regular signal. However, upon closer observation, the locations of peaks and valleys are only consistent with the first half of the signal, between $[-250, 0]$ s. In the latter half, we can see that not all antinodes are accounted for and phase information is not preserved entirely. Although there are advantages in simplifying input signals using this method, information is lost.

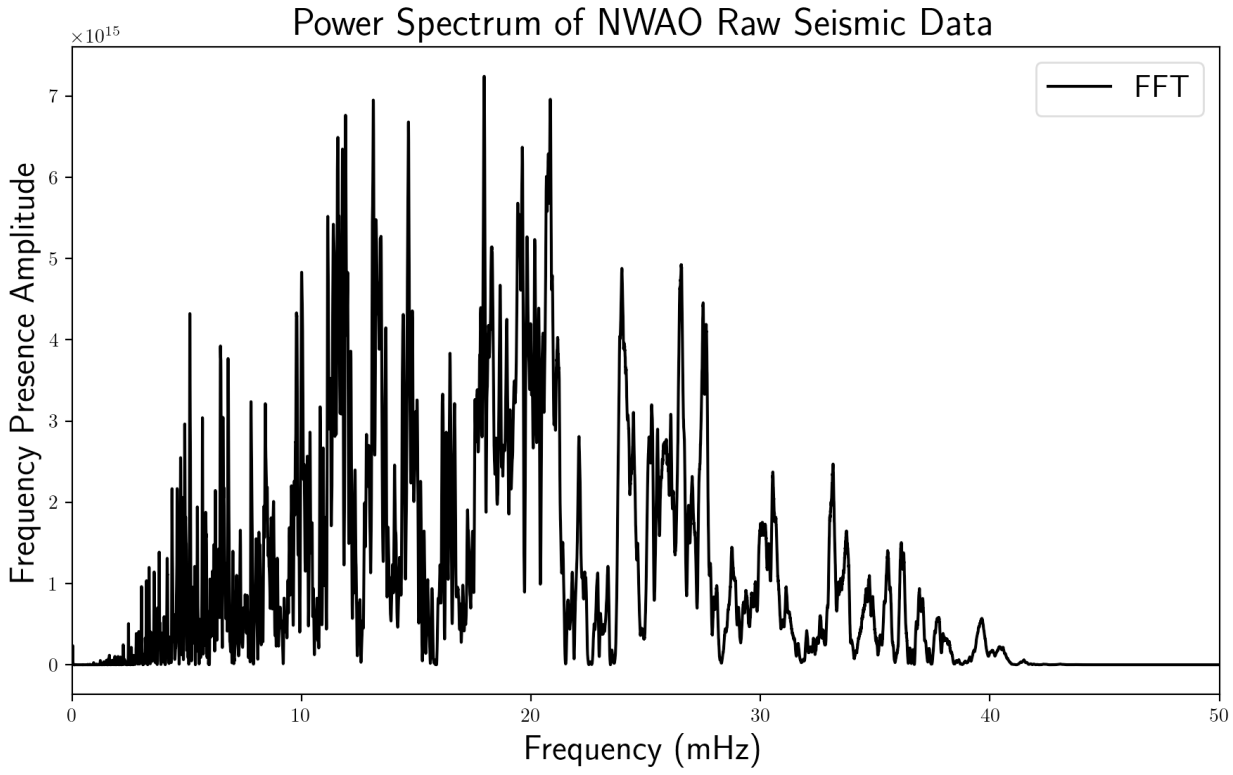
Problem 2

(**a**) In this problem, we consider an analysis of Earth's normal modes from the NWAO station seismograph during the March 11 earthwquake in Japan over a period of 3 days. After loading the data, it was plotted over its entire domain.



The time axis was defined using the output time scaled by $\frac{1}{dt \cdot 6 \cdot 60}$ to give hours.

(**b**) The input signal itself is finite, so does not need to be truncated to compute the power spectrum. In the absence of any additional windowing (just taking the spectrum of the raw signal), $\frac{1}{N}\overline{F}(\omega)F(\omega)$ was plotted where N is the sample number. To clarify the plot, only the positive frequencies were plotted to eliminate redundancy from aliasing:



The frequency axis was computed by noting that `np.fft.fftfreq()/dt` produces a x-axis in Hz/(10s), so to scale to mHz, we multiply by 1000 to obtain mHz.

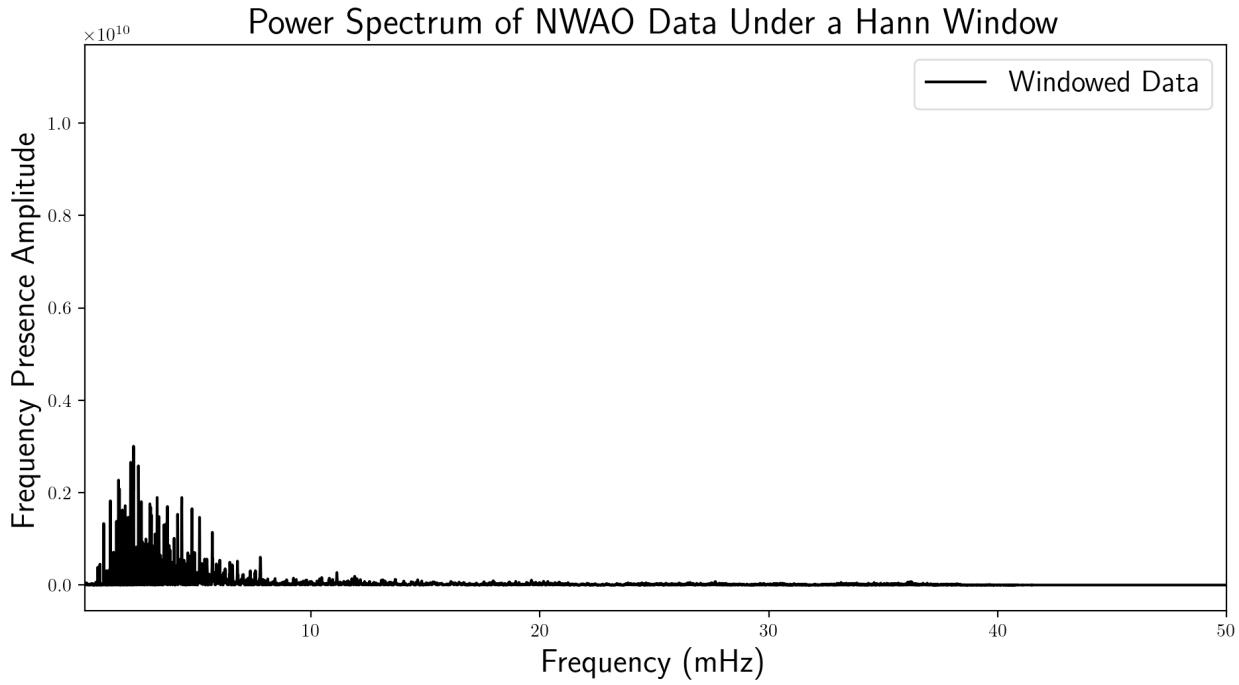
(c) To minimize noise or spectral leakage, we first can remove any linear trend found in the data by using `np.polyfit()` as in Lab 3:

```
slope, y_int = np.polyfit(time, velocity, 1)    #remove trend
line_trend = slope*time + y_int
detrended = velocity - line_trend
```

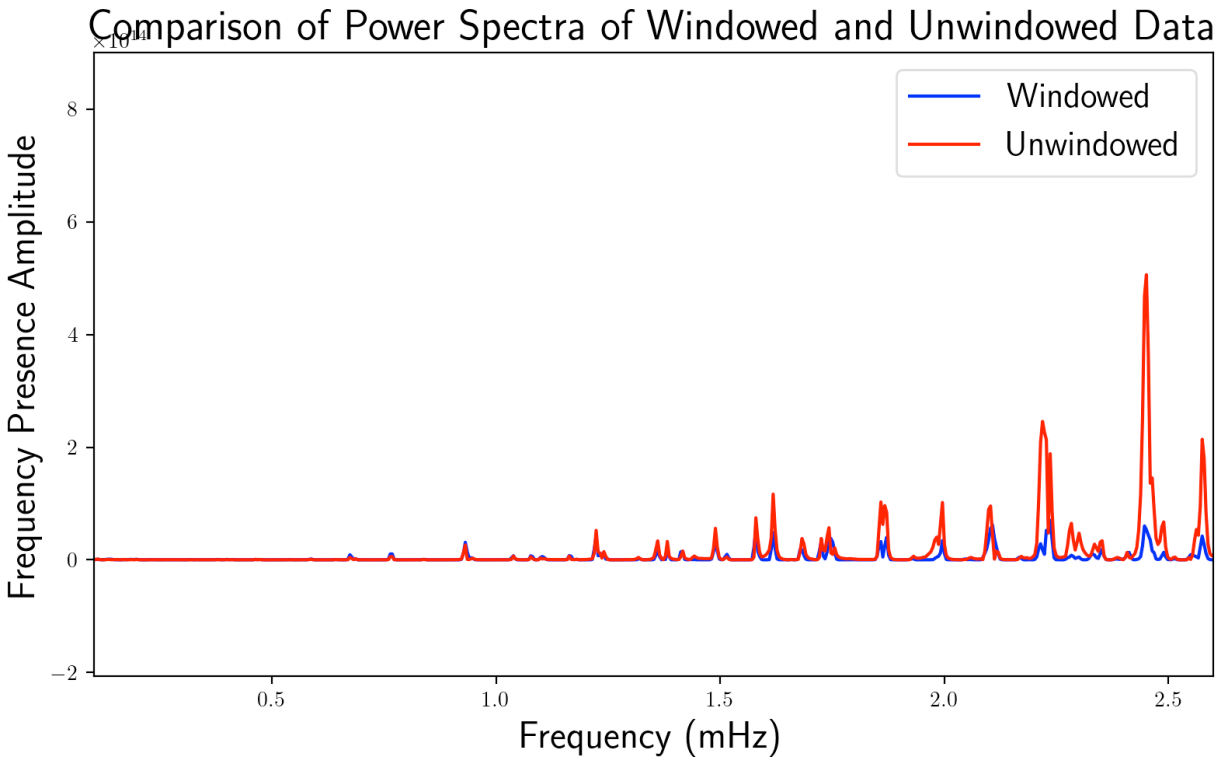
We can window our data by defining a Hann window function for data of length N, whose output elements are dependent of $0 \leq n \leq N$ and return the windowed array.

```
def hann(input):
    N = len(input)
    out=np.zeros(N)
    for n in range(N):
        out[n] = (1 - np.cos(2*np.pi*n/N)) * input[n]
    return out
```

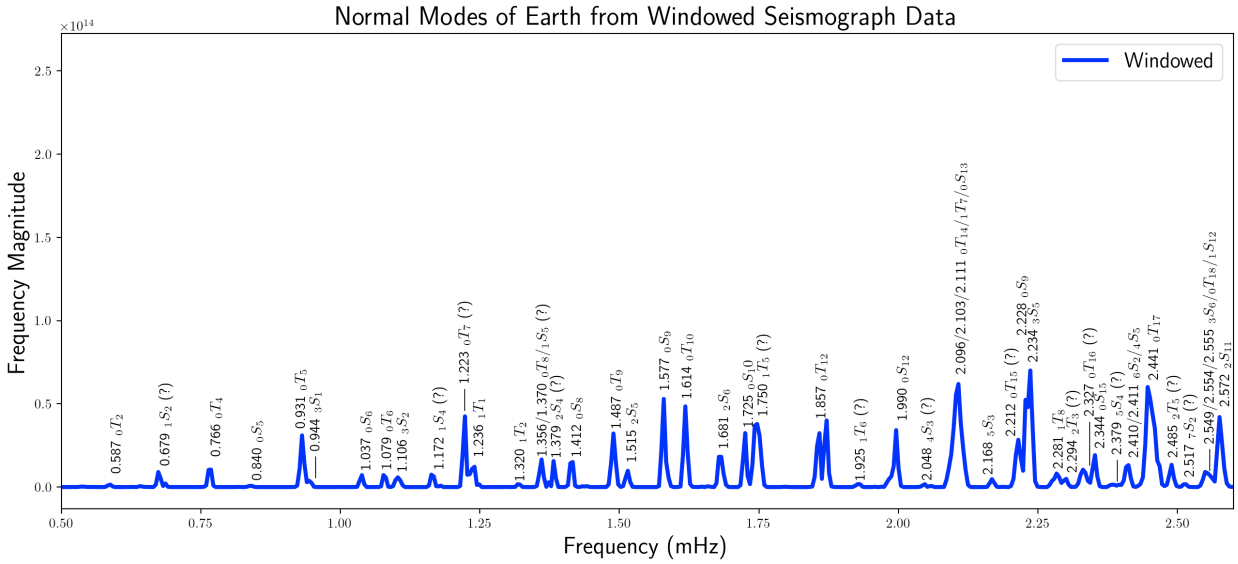
Plotting our result from processing the data, we find that spectral leakage is significantly reduced:



(**d**) Comparing the FFT signal outputs between the windowed data and the raw data including spectral leakage, we notice a significant difference in amplitude and the reduction of access noise found within the signal. Plotting the domain between (0.1, 2.6) mHz, we also observe a more consistent distribution of the frequency presence magnitudes, which is useful for analyzing the frequency composition of the raw data.



(**e**) Lastly, by examining the modes.pdf Table 1, arrays of text and their respective locations were coded into python to be plotted using `plt.annotate()`. Most normal modes found in the data were consistent with what were observed in the table, however any of the other modes which fell slightly out of the uncertainty range recorded were labelled with '?' since I didn't want to leave out the possibility of not mentioning a mode involved in the signal.



I chose to plot $(0.5, 2.6)$ because I didn't find any modes lower than 0.5 mHz. I also note that 100 μHz is equivalent to 0.1 mHz, so while the table gave frequencies in μHz, I just scaled them to mHz.