PHY407 Lab 3

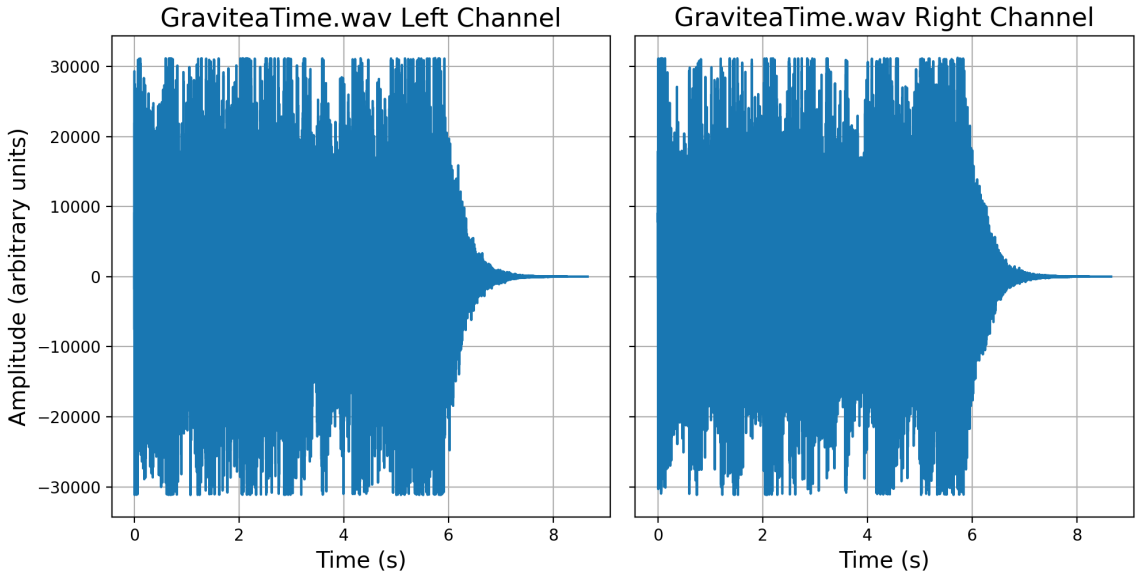
Monday, October 28, 2024

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 all questions: none.**

Problem 1

(**a**) The audio `.wav` file was read into the python file using `scipy.io.wavfile.read`. The left and right channels were extracted by slicing each of the output arrays as mentioned in the lab manual. Since the samplerate is defined as the number of samples per second, the timestep value dt, the length of time between samples, is just the inverse of the samplerate `1/samplerate`. Hence the time array is given by the range `np.arange(0, N*dt, dt)` at spacing dt for `N = len(left)` samples. The left and right audio channels were then plotted using `matplotlib.pyplot`



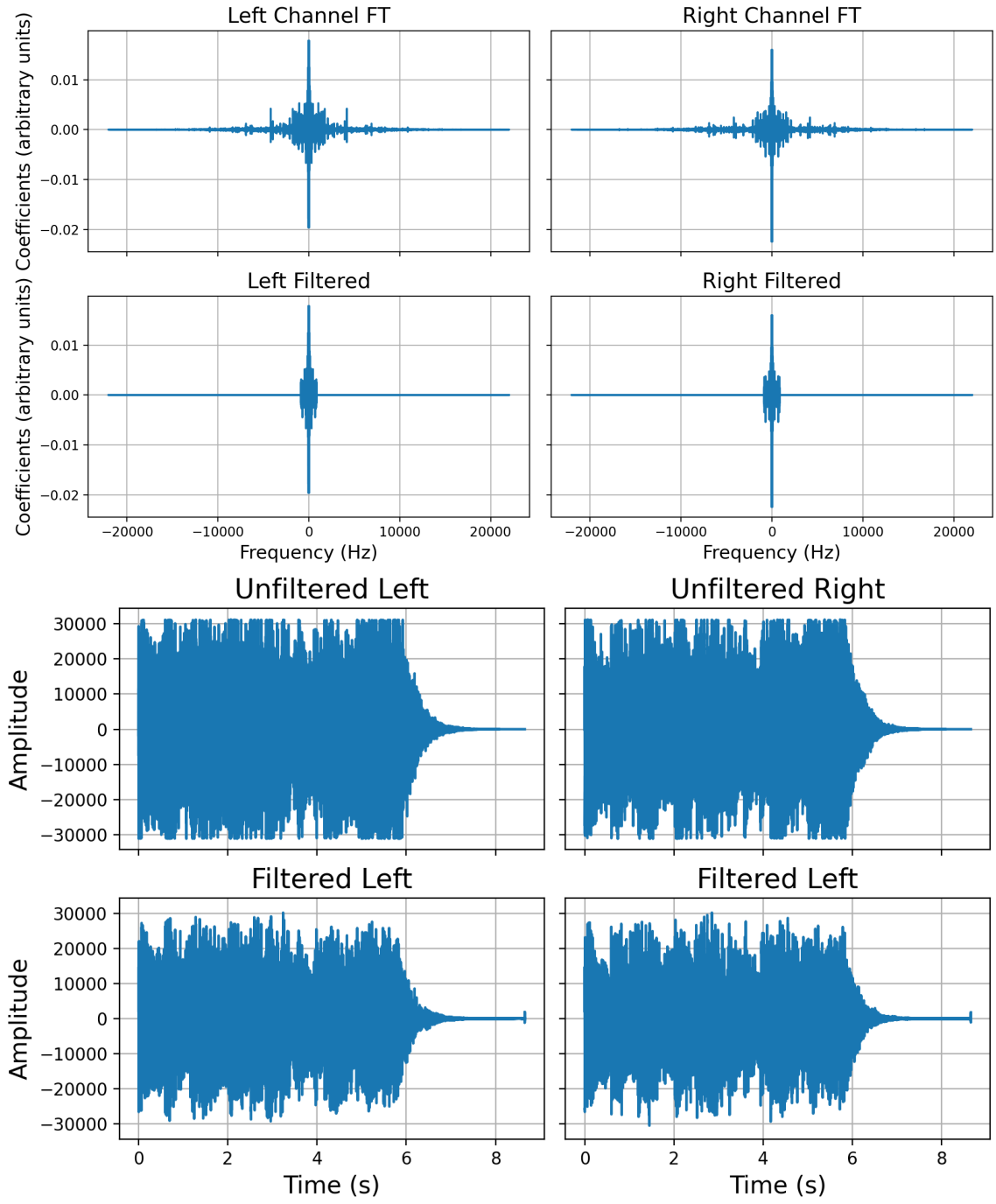
(**b**) Using `np.fft.fft` (the fast Fourier transform with real and complex elements), the left and right channels were transformed into the frequency - Fourier amplitude domain.
The frequency axis was written using `np.fft.fftfreq(N, dt)` for N samples at a spacing dt. This outputs the frequency axis in terms of Hz. The fast Fourier transform itself was taken out according to the numpy documentation, defined as

$$A_\ell = \sum_{k=0}^{N-1} a_k \exp\left[-2\pi i \frac{k\ell}{N}\right] \tag{1}$$

When compared with the exact expression of the Fourier transform, we note that we must multiply by the timestep dt to obtain the transformed values. Furthermore, since the Fourier coefficients are proportional to the weights via

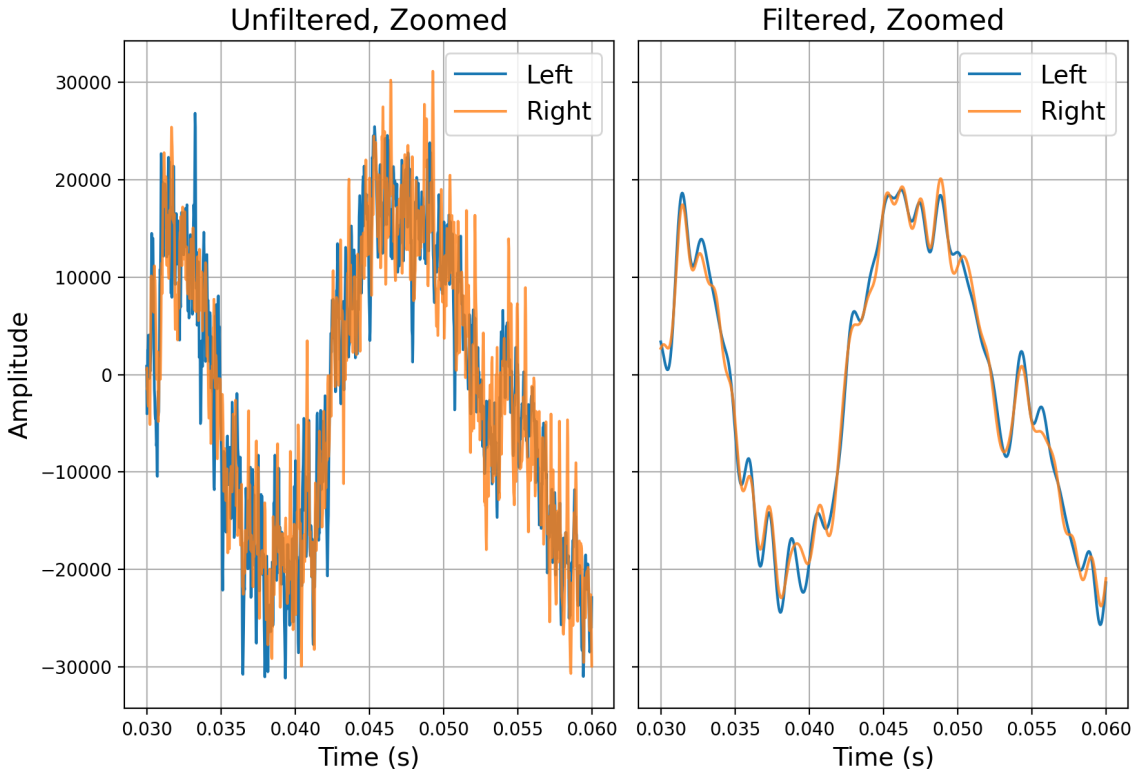
$$A_\ell = N\gamma_\ell, \tag{2}$$

we must then divide by N to obtain γ_ℓ. Hence, for both left and right channels, the FFT was computed as `np.fft.fft(left)*dt/N`. The inverse is then the opposite, `np.fft.ifft(left_FT)*N/dt`. Once the FFT of both channels was computed, a for-loop was written to iterate through the frequency values via indices. If the frequency value entry was greater than $|880|$ (units are both Hz, accounting for both negative and positive frequency values due to the complex domain transformation), the corresponding values in the `left_FT` and `right_FT` arrays were set to zero. The filtered left and right channels were then inversely-FFT'd via `np.fft.ifft(left_FT)*N/dt` as mentioned. All of the results, the filtering, and the FFT spectra, were plotted for both L/R channels and their filtered respectives.



The only note I will make is the error produced at the endpoints by the FFT, and this is due to a lack of padding and truncation from the summation (see question 2 solution).

(**c**) To plot a zoomed instance of the effect of the filtering, the time array just needed to be sliced between two times. The index value is directly proportional to the samplerate, hence for any time t in the array the index could be determined by `t * samplerate`, then making the value an integer (floats cannot be indices for arrays). I just chose $t_1 = 0.03$s and $t2 = 0.06$s (the difference is 3 ms), hence the indices were `i1 = int(t1/dt), i2 = int(t2/dt)` respectively.
The resulting filtering effects were then plotted by slicing between these two indices.



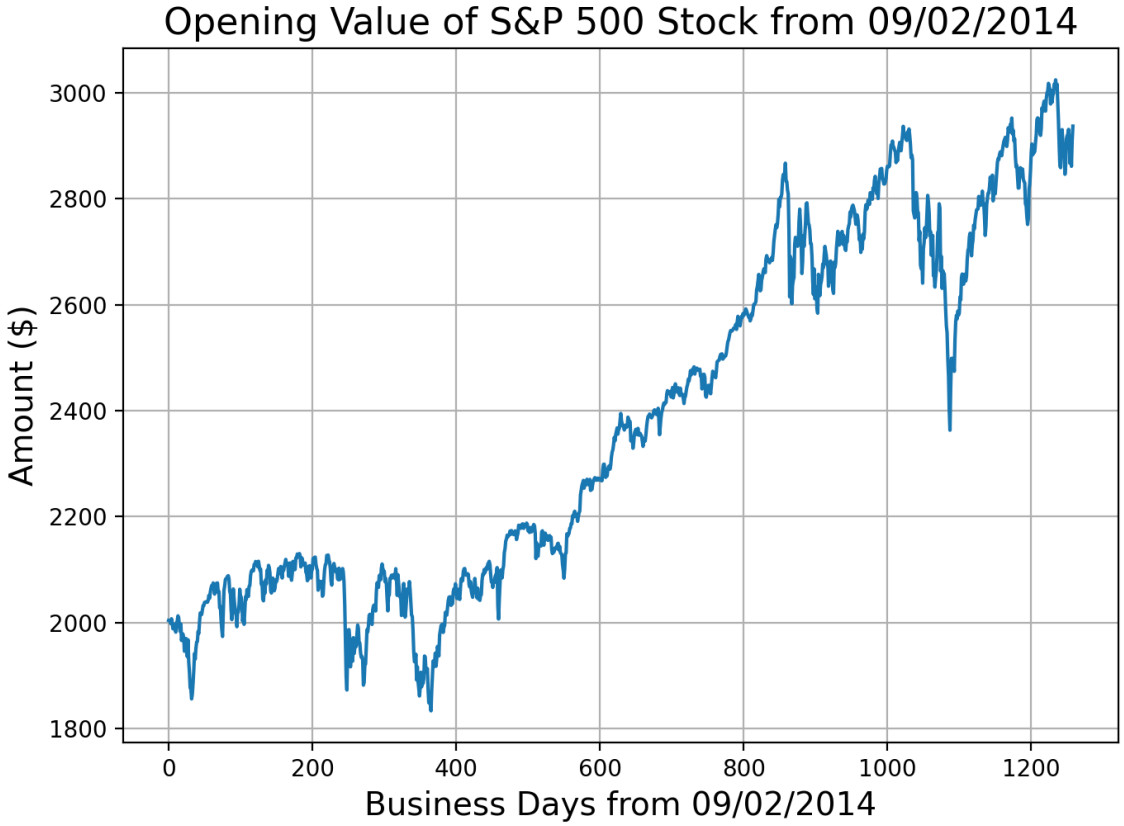
The higher frequencies were removed, hence there is less noise present in the result of filtering.

(**d**) Again, using `scipy.io.wavfile.write`, the resulting filtering was written by appending left and right channels together, taken to be at the same samplerate as before. Only the real components were taken, as the complex values were the same from the FFT output. Taking the absolute just made the file itself clip.

Problem 2

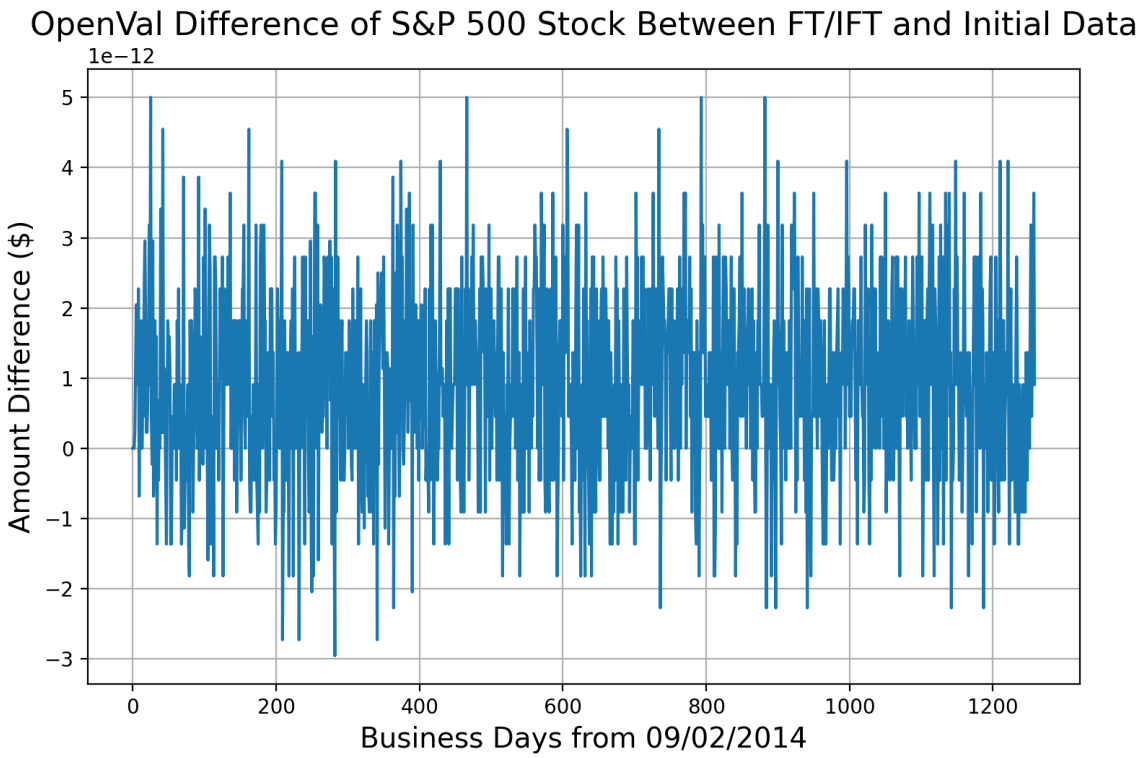
(**a**) Data from a `.csv` file can be read into python using `pandas.read_csv`. Since `.csv` files are indexed using rows/columns, we may extract these columns explicity by adding a '.' to the file followed by the column name. That is, the date values were loaded via `data.Date`, similarly the opening values themselves as `data.Open`.

The business day array is then just `np.arange(0, len(data.Date))`. These were then plotted using `matplotlib.pyplot`.



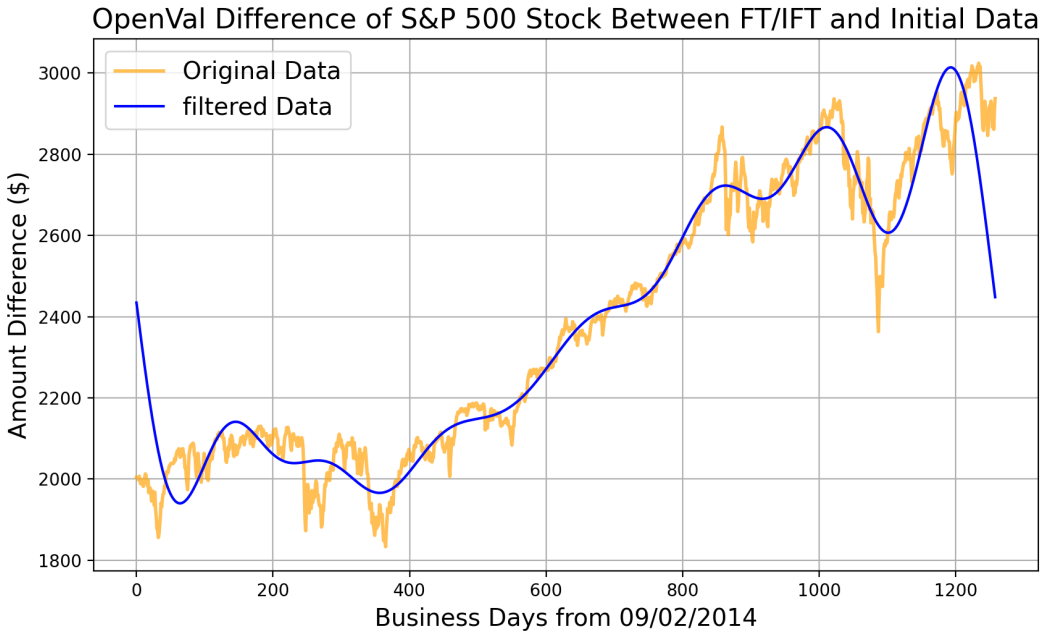
(**b**) The Fourier coefficients are again, given by $\gamma_\ell = A_\ell/N$. We compute the FFT (real or imaginary) as outlined before, `np.fft.rfft(openVal) * dt/N` and it's inverse `np.fft.irfft(openVal) * N/dt`, however there is a caveat. Because the `np.fft.rfft` call only computes the real component of the FFT, we only require half the points necessary, which means the input will be cropped in accordance with the output. To fix this, we just need to pad the surrounding input values with zeros, which is done automatically in `np.fft.rfft` by noting $n = N$ points are being used.

Thus we compute `np.fft.rfft(openVal, N) * dt/N` and `np.fft.irfft(openVal) * N/dt` respectively for the real FFT. To check that they were "identical" (numerically speaking, there will be an accumulation of some kind of error between operations; though I couldn't find what this would be for the numpy algorithm). Using `matplotlib.pyplot`, the difference was plotted. Each value was on the order of $O(10^{-12})$, which I would assumed to be good enough to claim that the operations were identical.



Furthermore, there does not appear to be any trend in the resulting difference, instead just apparent noise of higher frequencies (possibly produced by the endpoints; again, numpy doesn't handle them well). In addition, what is $\$5 \times 10^{-12}$ worth? Realistically, this amount is zero, so we can round and claim that the operations are identical forward as they are backward.

(c) To remove any variations from 6-month long periods, the procedure from question 1 was copied. That is, in the frequency domain, iterating through the axis given by



`np.fft.fftfreq(N, dt)` and setting any corresponding values to zero if the frequency (1/days) is over $1/(21 * 6)$ (on average, the number of business days per month is around 21. During a 6 month period, that is around $21 * 6$ business days).
The resulting filtered array was inverted back into the cost-time domain using the inverse rFFT. It was then plotted overtop the unfiltered data, as shown above.
Overall, the trend of the filtered stock is smoother (less noise) and follows the rough trend of the unfiltered data. There are some minor discrepancies of around $\pm\$200$, is around the 50 - 400 business day mark and the 1200 business day mark, and these were most likely filtered out because their fluctuation was less than the 6 month period. Yes, first and last endpoints are weird, due to truncation error of the edges from the numpy FFT call.

Problem 3

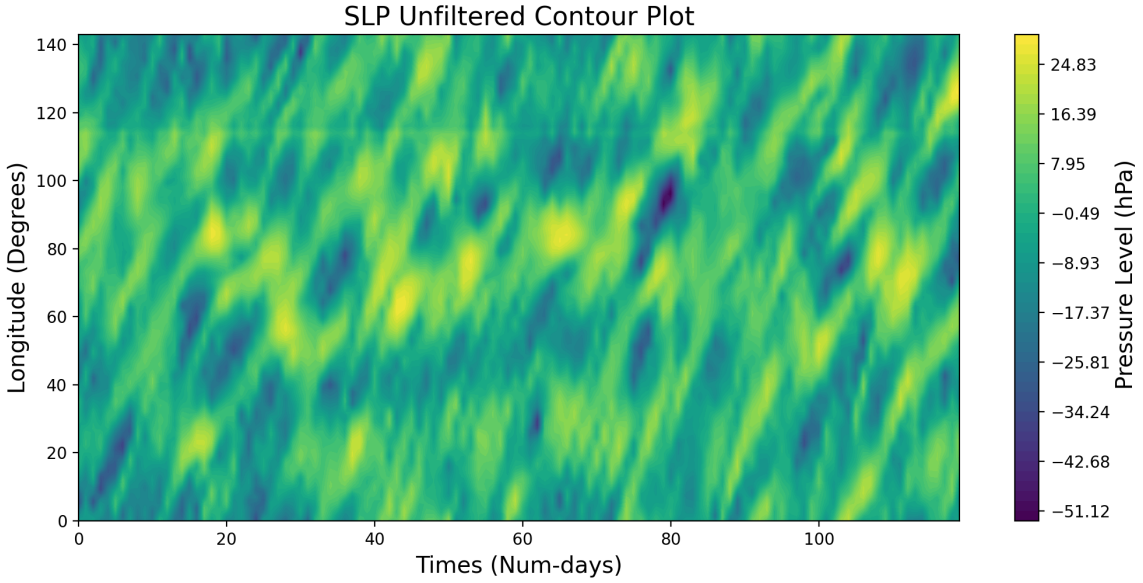
(a) The SLP, longitude, and time data was imported into python by calling the `numpy.loadtxt()` function. The Fourier transform we are wanting to take is along the longitudnal axis, since fluctuations along this axis is what will correspond to the wavenumber m:

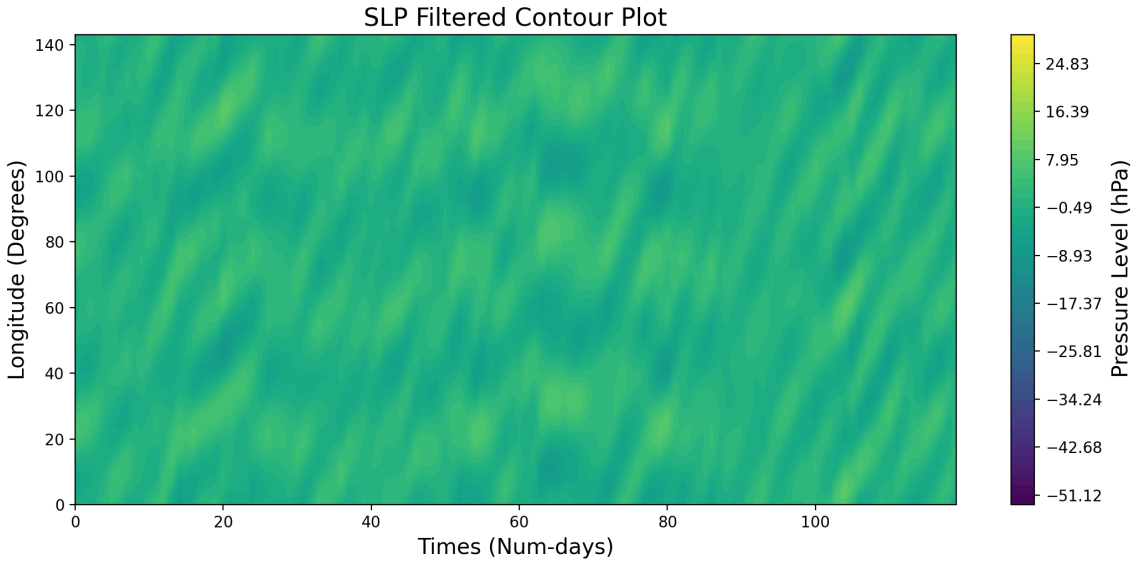
$$A(t)\cos(m\lambda + \varphi(t)). \tag{3}$$

Hence, we are to take the FFT along `axis = 0` if the array is given in (lon - time) format, or `axis = 1` if in (time - lon) format. Luckily, this can be easily done by inserting the axis argument into the `np.fft.fft()` call.
The shape of the SLP array was `SLP.shape = (144, 120)`, $144 \times 2.5^{\circ}$ longitudnal steps and 120 timesteps. Hence we transofmr with respect to `axis = 0`.
Furthermore, implementing the same procedure for calculating the Fourier coefficients, the dl spatial element was calculating via `dl = longitde[1] - longitude[0]`, and the number of elements via the `len()` call. Thus, the FFT is given by `np.fft.fft(SLP, axis = 0) * dl / N`

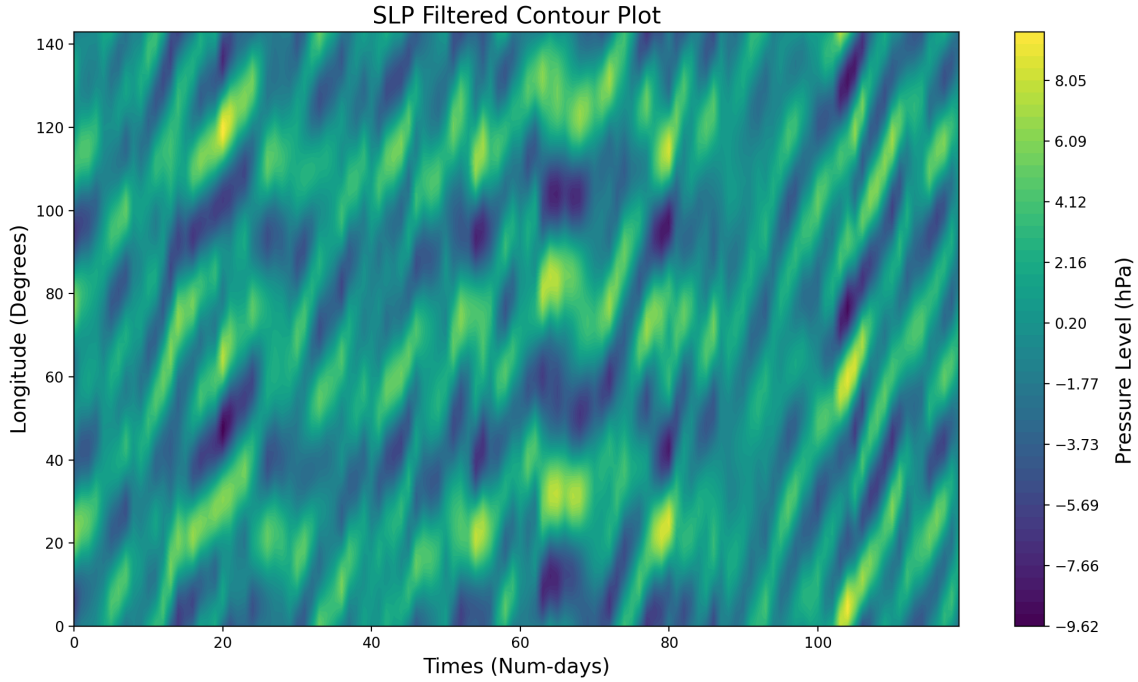
The wavenumbers corresponding to $m = 3$ and $m = 5$ were extracted as the array entry slices $[3]$ and $[5]$, since $m \in (0, \ldots, N-1)$. The rest of the transformed SLP array entries were set to 0, and this was done using a for-loop.

The inverse FFT of the filtered data was calculated, as previously described, `np.fft.ifft(SLP_FT, axis = 0) * N / dl`. Results de to filtering were plotted using `matplotlib.pyplot.contourf`, as well as the initial results using the same scale to show the difference the filtering made.





(**b**) We note the following effect of the filtering: amplitudes are reduced, specific frequencies are isolated for longitudnal waves travelling eastward. One may observe this in the results by noticing that the matters of high and low pressure move upwards (along the positive longitude direction) as time increases. Longer waves appear to move more gradually, and shorter waves appear to move quicker, which is why there are some areas of the plot that are narrow and blurry compared to other areas which are clearly elongated and clear. Compared to the initial plot, this seems to be consistent with the filtered contour plot. It is easier to observe with the initial plot, because the wave pressures are more defined (I tried to use the same scale for both, but it might not be clear). Below I have included a more defined plot.



The more blurry sections correspond to faster wave propagation, and one may again note that the

density of green-purple spots has increased (the wave is shorter), compared to the lower density, less blurry spots (longer waves) that appear to all move eastward (positive longitude).