Computational Physics Homework 4

Neelang Parghi

1. Our first task was to compare different Fourier transform algorithms. I defined three Python functions: slow DFT, an FFT based on the recursive Cooley-Tukey algorithm[1], and an inverse DFT function that returns the original input values.

   The slow DFT function was created according to the definition:

   $$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}. \tag{1}$$

   This implementation was straightforward. An empty array for the transforms was initialized, then a `for` loop went through the input array to calculate the DFT and append these values to the transform array.

   Next was the FFT implementation using the Cooley-Tukey algorithm[2]. This algorithm will recursively break down a DFT of size N into two smaller DFTs of size $N1$ and $N2$, one for the odd-numbered values and the other for the even-numbered values:

   $$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N} \\ &= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i2\pi k(2m)/N} + \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i2\pi k(2m+1)/N} \\ &= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i2\pi km/(N/2)} + e^{-i2\pi k/N} \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i2\pi km/(N/2)} \end{aligned}$$

   Note that we can factor out the common multiplier $e^{-i2\pi k/N}$ from the second sum. We now see that these two sums represent the DFTs of

[1] Based on pseudocode from from the Cooley-Tukey Wikipedia page and from the *Pythonic Perambulations* blog at `https://jakevdp.github.io/`.

[2] Interesting side note: This algorithm was actually invented by Gauss but published posthumously in neo-Latin, so it went largely unnoticed. Cooley and Tukey re-discovered it and re-invented it for use on computers.

the even and odd partitions. After the split, each term still requires $(N/2) * N$ calculations for a total of N^2. This doesn't save us any time, but we can use the symmetry for each to our advantage. Since $0 \leq k < N$ and $0 \leq n < M \equiv N/2$, we only need to perform half the computations for each smaller part. This reduces the $\mathcal{O}(N^2)$ of the slow DFT to $\mathcal{O}(M^2)$, where M is half the size of N. Since the length of the input array will be 2^n, we know that its length will always be even and so we can reapply this strategy recursively and save half the computation time with each cycle. This gives us $\mathcal{O}(N \log N)$. This was implemented using a base case where if a single number was the input, it will simply return the number itself. This explains why my FFT method was much faster than even numpy's FFT for $n = 0$. It was simply returning the input value itself. Things get a bit more interesting when $n = 1$, where we see that the slow DFT is actually the fastest! Keep in mind that FFT actually is DFT, but only executed on a smaller number of input values. When $n = 1$, then there are only two input values, thus there is no time advantage in choosing FFT over DFT in this case.

We see that the DFT and FFT algorithms performed according to their respective theoretical run times. Also note that I used an n value of 12 for the slow DFT calculation and $n = 20$ for the FFT functions. This was due to the widely different performance times for each algorithm. Using these different n numbers allows us to see how their performance times scale without waiting for too long for the code to finish running. From the plot, we see that FFT can perform 10^8 times more work in less time that the slow DFT.

The inverse DFT can be defined as

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{i2\pi kn/N}.$$

As we can see, this is very similar to (1) above and thus the implementation is very similar. We simply need to change the factor $e^{-i2\pi k/N}$ to $e^{i2\pi k/N}$ and take an input of Fourier values. To test this, I fed a random array into numpy's FFT function, then ran the output through an inverse DFT function to see if the original values were returned. For the purpose of this part of the assignment, I used $n = 5$ to make the displayed outputs easier to compare.

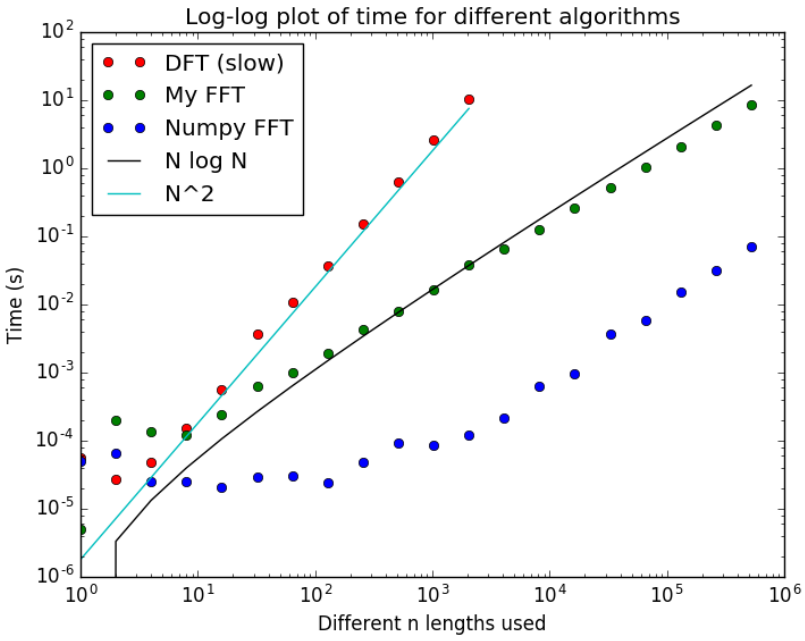

Figure 1: Plot of execution times

2. Next, we applied the above to data from the LIGO project. We were given the strain data as measured by LIGO detectors in Livingston, LA ("L" data) and Hanford, WA ("H" data). The first plot in figure (2) shows the strain data from each detector over 32 seconds. Since the maximum strain from astrophysical sources is 10^{-21}, we instantly know that this data does not directly tell us what we want because the data is on the magnitude of 10^{-18}. The information we want is in there, we just need to de-noise it enough to bring it out.

   The second plot of figure (2) is the periodogram for each data stream. This will be the first step in removing the noise. Since we're told that many of the lines correspond to resonances in the LIGO machinery, we can filter these. We will take our Fourier data and apply a series of transfer functions. We're given two transfer functions to use: step and Gauss[3]. Since these are filters, we need each to be less than 1 but their specific uses are different. The step function will filter data to the right

[3] $n = 8$ for the step function and $\sigma^2 = 1$ for the Gauss function.

of its f_0 input. The Gauss transfer function will filter spectral lines at the input frequency f_0. Using these together will filter the LIGO data enough to produce a distinct signal for a binary black hole merger.

Since we're told that LIGO is most sensitive between 35Hz and 350Hz, we will want to focus on this region. Setting a step function with $f_0 = 35$ seems like a good start but we have to be careful: we want to filter to the left of $f_0 = 35$, not to the right. This can be handled by simply applying $1 - H_{step}$. Another application of H_{step} with $f_0 = 350$ will filter modes outside the band. Next, a large number of lines exceeding 10^{-21} were visible in the periodogram. The H_{Gauss} function will filter lines at the input frequency f_0. A long series of H_{Gauss} functions were applied for each of these frequencies until a clear signal was visible from the inverse FFT plot[4]. We see that the signal from the merger occurs at around 22.5 seconds.

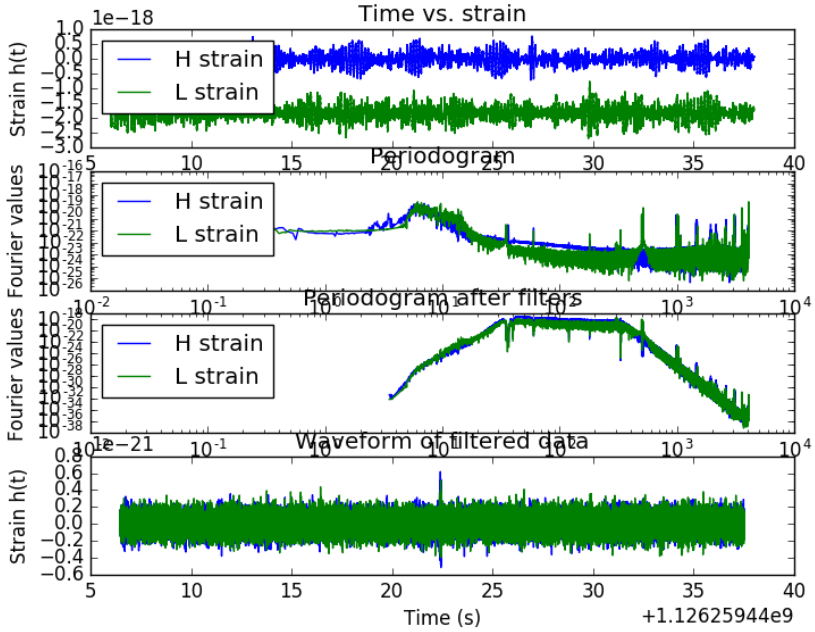

Figure 2: LIGO data

The final step was to create audio files of the data. This was accom-

[4]To make the signal more visible, I trimmed off the first and last 2000 points.

plished by importing `wavfile` from `SciPy` and using the `write_wavfile` and `reqshift` functions found on the LIGO signal processing tutorial page. The first function is used to create an audio file of the filtered data at $\pm$ 2 seconds around the signal, which we know from the LIGO page occurred on September 14, 2015 at 9:50:45 GMT. If you listen carefully, the faint sound of the signal can be heard on files `H1whitenbp.wav` and `L1whitenbp.wav`. This can be enhanced using the `reqshift` function, which increases the frequency by 400Hz. This output can be heard on files `H1shifted.wav` and `L1shifted.wav`. The signal can be heard much more clearly on the files with the shifted frequency.