PHY 324: Computational Project

Meet Chaudhari

April 4, 2023

Abstract

This report describes the idea of fourier and inverse fourier transforms which are used in signal processing to decompose a function with and without noise in it into its frequency components as well as take a fourier transform with noise in it to convert into a signal that produced it using filter functions. These methods are implemented using python programming language and are quite useful when dealing with experimental data in physics and other fields.

1 Introduction

Fourier transform and the related inverse fourier transform are important and very powerful techniques in data and signal analysis. Fourier transform allows one to decompose a complicated function into a sum of sinusoids (sines and cosines) determined by their frequencies and amplitudes. It is simply a plot of amplitude versus frequency. This helps in manipulating the function quite easily using mathematical techniques. For discrete data, that is, data that's not continous such as experimental data, we use a method called Fast Fourier Transform (fft) to find the discrete fourier transform of the function and inverse fourier tranform as well. These apply to non-periodic functions as well. For a continous function $y(t)$, the fourier transform is defined by the formula:

$$Y(\omega) = \int_{-\infty}^{\infty} y(t) \cdot e^{-i\omega t} dt \tag{1}$$

and the inverse fourier transform is determined by:

$$y(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} Y(\omega) \cdot e^{i\omega t} d\omega \tag{2}$$

Observe that from eq. (1) and (2), the fourier transform of a function and the inverse fourier transform are complex in nature due to the identity $e^{i\theta} = \cos(\theta) + i\sin(\theta)$.

Also note that the Fourier transform (1) gives a function Y which depends on the angular frequency (and hence normal frequency f as $\omega = 2\pi \cdot f$) whereas the inverse fourier transform (eq 2) takes the fourier transoform $Y(\omega)$ to give back $y(t)$ i.e. the function that was fourier transformed to $Y(\omega)$. In this sense, both $y(t)$ and $Y(\omega)$ essentially have the same information because we can easily convert one to another by using (1) and (2). Neither contains more information than the other. A fourier transform only contains the information about the frequency components of sinusoids, their amplitudes and relative phases which is already present in the original function. It is simply represented in a different format.

For N-discrete values of $y(t)$: $y_0, y_1, y_2, ..., y_{N-1}$, we can convert the integral in (1) to a finite sum with the differential term dt turning into time sampling interval Δ as:

$$Y_j = Y(\omega_j) = \frac{2}{N} \left(\sum_{k=0}^{N-1} y_k e^{-i2\pi \frac{jk}{N}} \right) \tag{3}$$

Equation (3) above is the discrete-fourier transform. The $2/N$ factor is a normalization factor which ensures that the peaks in the $|Y(\omega)|$ graph give the correct amplitudes of the frequency components in the signal $y(t)$. The corresponding frequency values is given by $\omega_j = j\frac{2\pi}{N\Delta}$ where j is the index of the peak. We have set $\Delta = 1$ without loss of generality because we could choose units such that our time array would have a spacing of $\Delta = 1$ in those units. Now, the discrete inverse fourier transform of eq. (2) is given by:

$$y_k = \frac{2}{N} \cdot \left(\sum_{j=0}^{N-1} Y_j \cdot e^{i2\pi \frac{jk}{N}} \right) \tag{4}$$

Both of these can be easily calculated by using the fast fourier transform. The fast fourier transform basically separates the sums in eq. (3) and (4) into even and odd terms. If we set $W = e^{-i\frac{2\pi}{N}}$, then in terms of even and odd k, we can write equation (3) as:

$$Y_j = Y(\omega_j) = \frac{2}{N}\left(\sum_{k=0}^{\frac{N}{2}-1} y_{2k} e^{-i2\pi\frac{j(2k)}{N}}\right) + \frac{2}{N}\left(\sum_{k=0}^{\frac{N}{2}-1} y_{2k+1} e^{-i2\pi\frac{j(2k+1)}{N}}\right) \tag{5}$$

$$= Y_j^{k\ \text{even}} + W^j \cdot Y_j^{k\ \text{odd}}$$

This way, the fft algorithm keeps dividing the set of data points into even and odd terms. When the number of points in a given sum is a power of 2, it is able to directly compute the sum very quickly using a set of previously computed exponentials. Now let's see some examples that show how useful these techniques are.

2 Procedure and Results

First, let's analyze a function which is a sum of 2 sine waves with different frequencies. Say we have a time array of size $N = 200$ which runs from $t_0 = 0$ s to $t_{199} = 199$ s.

Now say

$$y_1(t_k) = A_1 \sin(2\pi\frac{t_k}{T_1}) = 2\sin(2\pi\frac{t_k}{4}) \tag{6}$$

which has a frequency of $f_1 = \frac{1}{T_1} = 0.25$ Hz and amplitude $A_1 = 2$. Also, set

$$y_2(t_k) = A_2 \sin(2\pi\frac{t_k}{T_2}) = 3\sin(2\pi\frac{t_k}{5}) \tag{7}$$

which has a frequency of $f_2 = \frac{1}{T_2} = 0.2$ Hz and amplitude $A_2 = 3$. Now, we can define a new function $y(t_k)$ which is a sum of these sine waves:

$$y(t_k) = y_1(t_k) + y_2(t_k) = 2\sin(2\pi\frac{t_k}{4}) + 3\sin(2\pi\frac{t_k}{5}) \tag{8}$$

The position vs time graph of $y(t_k)$ is shown in Figure 1.

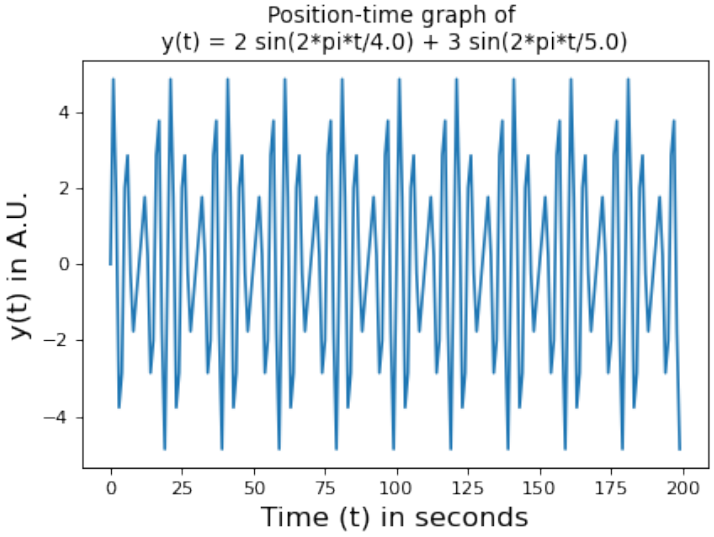

Figure 1: The position vs time graph of $y(t_k) = y_1(t_k) + y_2(t_k) = 2\sin(2\pi\frac{t_k}{4}) + 3\sin(2\pi\frac{t_k}{5})$ which is a sum of sine waves of frequencies 0.25 and 0.2 with amplitudes 2 and 3 respectively. This complicated looking graph can be fourier transformed to know the underlying sine waves that created it.

Now, we can generate the fourier transform of $y(t_k)$ using Numpy's (or np) fft.fft command. We can then set $z = \frac{2}{N}$ np.fft.fft(y). Now since z would be generally complex valued, we can plot its

absolute value i.e. np.abs(z) as a function of frequency as python outputs fft results in terms of frequency f and not angular frequency ω that appears in equation (3). Now, although we have calculated the fourier transform z we haven't yet calculated the frequencies that correspond to it. This is done by Numpy's fft.fftfreq command which takes in the size of our time array and the spacing between adjanent points in it to give the corresponding frequency values. In our case $N = 200$ and $\Delta = 1$ s. It assumes that the function we just fourier transformed has a period of N. After this, we can plot the absolute value of z against the frequency values. This graph is shown in figure 2.

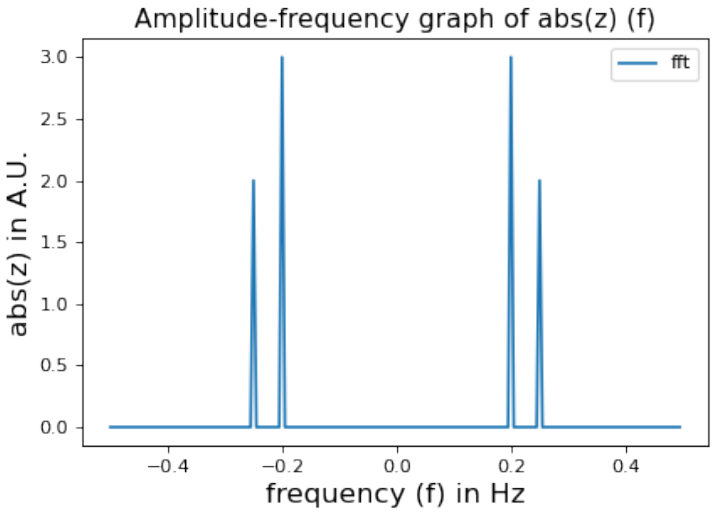

Figure 2: Plot of the absolute value of the fourier transform z = np.fft.fft(y) as a function of frequency. Although $y(t)$ is a sum of only 2 sine waves, the plot shows 4 peaks because there is a mirror symmetry at play. The corresponding heights of the peaks give the amplitudes of the sine wave with frequency equal to the x-coordinate of the peak.

Note how there are 4 peaks in the grapha although there are just 2 sine waves in the signal $y(t_k)$. This is due to a mirror symmetry in the fourier transform. Now, let's check how the fourier transform gives the information about the 2 sine waves which were added to create $y(t_k)$. Using Scipy's find_peaks method, which takes the array abs(z) and the range of peak heights that needed, which can be approximated from figure 2, to give us the indices of abs(z) where the peaks occur and their heights. The indices could then be used to find the corresponding frequencies of the peaks with the use of *np.fft.fftfreq*. The calculated peak coordinates were: $(0.2 \text{ Hz}\ , 2.99) = (0.2 \text{ Hz}\ , 3)$ and $(0.25 \text{ Hz}\ , 2.0)$ which match exactly with $(f_1, A_1) = (0.2 \text{ Hz}\ , 3)$ and $(f_2, A_2) = (0.25 \text{ Hz}\ , 2.0)$.

We can also demonstrate the use of fourier transform to decipher the signals hidden in noisy real world data. For example, let's say we have a sine wave of form $y(t_k) = 5\sin(2\pi\frac{t_k}{17 \text{ s}})$. If random gaussian noise is added to it, then we get a new function $x(t_k) = y(t_k) + noise(t_k)$. Its position vs time graph is shown in figure 3 below.

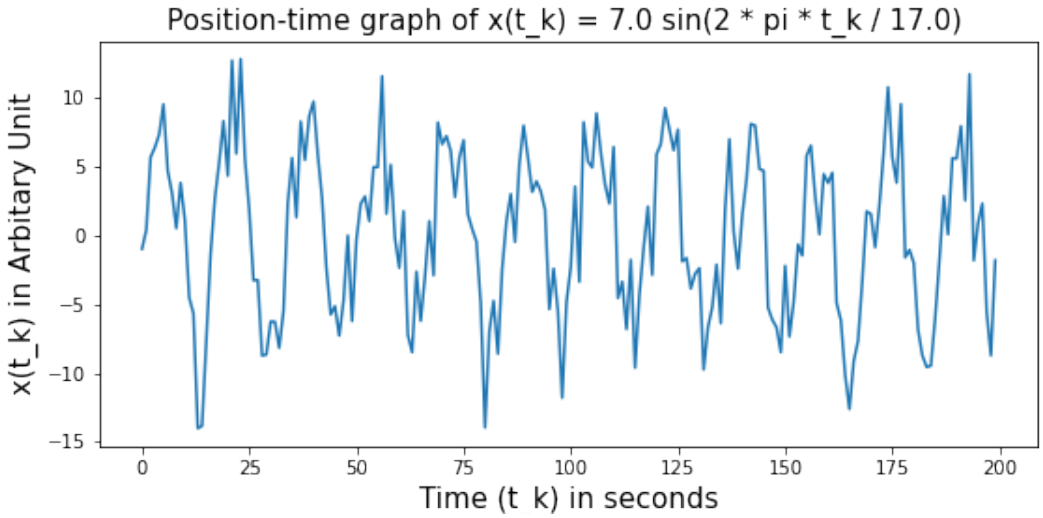

Figure 3: Position vs time plot of the noisy sine wave $x(t_k) = 5\sin(2\pi\frac{t_k}{17\text{ s}}) +$ noise
The noise added to the sine wave has a gaussian distribution with an amplitude of $\frac{5}{2} = 2.5$.

If we wish to extract the cleaned signal $y(t_k)$ out of $x(t_k)$, then we can use fast fourier fourier transform. Setting $q(f) =$ np.fft.fft(x) and plotting its absolute value with respect to frequency values, we get 2 duplicate peaks with noise in between (see figure 4) indicating the presence of one prominent sinusoid in the signal.

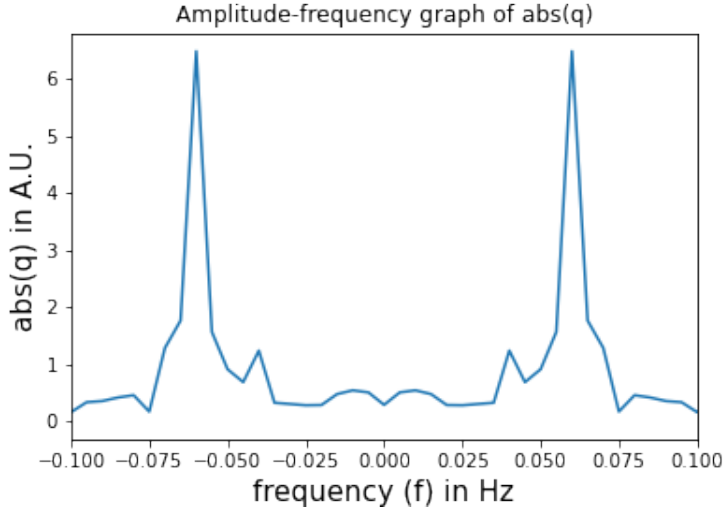

Figure 4: Ampltitude vs frequency plot of the fourier transform $q(f)$ of a noisy sine wave $x(t_k) = 5\sin(2\pi\frac{t_k}{17\text{ s}}) +$ noise. Observe how despite the noise the fft graph shows 2 equal peaks due to mirror symmetry indicating that there is only 1 frequency sinusoid present in the signal.

To smooth out the fast fourier transform $q(f)$, we can create a filter function which would have peaks at the same frequencies as $q(f)$ and would vanish smoothly at other points. Multiplying this filter function with $q(f)$ then would lead to the elimination of the noise between the peaks seen in figure 4. A fairly common method to create "bump" functions such as the filter function would be to use a sum of positive and negative exponentials which are determined by their width and frequency. These are called gaussian functions. The filter function used was determined by:

$$\text{filter\_function} = e^{\frac{-(\text{freq}-\text{peak})^2}{\text{width}}} + e^{\frac{-(\text{freq}+\text{peak}-N)^2}{\text{width}}} \tag{9}$$

and is shown in the second plot from top of figure 5. The variable *peak* is the frequency at which the first peak should occur. The variable *freq* is an array from 0 to $N-1$ and is useful because when peak equals to an element in freq, the first exponential in (9) becomes 1 (a peak) and then the second exponential would replicate that peak at an equal distance to the left of the maximum value in *freq*.

Thus, this would capture the position of both the peaks in the fourier transform. By eyeballing the positions of peaks in the graph of abs($q(f)$) (see top plot in figure 5) and by trial-error, I adjusted the *peak* variable to line up such that the filtered fft $p(f)$ was quite smooth while making sure that the filter function wasn't too narrow to not allow other frequencies around the peak which might very well be present in the original signal. The proof of some additional frequencies can be seen by the 2 small non-negligible bumbs around the peaks in top plot in figure 5. I kept the $width = 10$ Hz initially.

The filter function with these parameters was then multiplied with $q(f)$ to generate a filtered fourier transform $p(f)$ of the noisy sine wave data $x(t_k)$:

$$p = q \cdot \textbf{filter\_function}$$

The absolute value abs($p(f)$) of $p(f)$ is plotted against frequency at the bottom subplot of figure 5. Observe how the noise between the peaks of abs($q(f)$) has vanished because the filter_function is 0 there.

Actually, since the frequency of the pure sinusoid underlying $x(t_k)$ was known, the x-coordinates (frequencies) of the peaks in abs($q(f)$) (see top subplot in figure 5) are also known as then $peak = \frac{1}{17}$ Hz ≈ 11.7 Hz where 17 s is the time period of the pure sinusoid. Thus, setting that same frequency for the filter function would align it exactly with the noisy fourier transform $q(f)$. But, without that knowledge, the filtered fft $p(f)$ looks quite good.

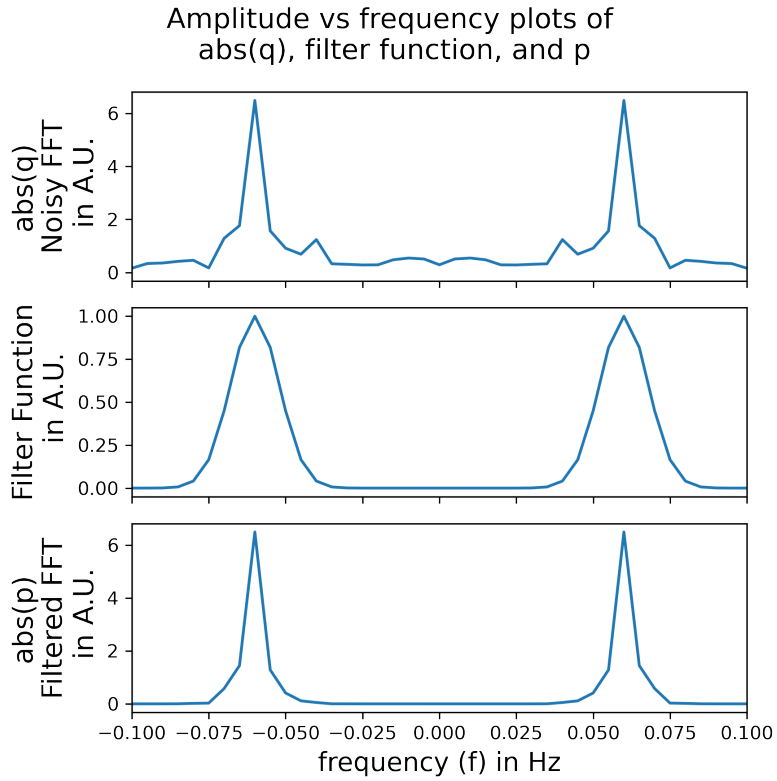

Figure 5: From top to bottom: Plots of the absolute value of fourier transform of noisy sine wave i.e. $q(f)$, Gaussian filter function, and absolute value of filtered fourier transform $p(f)$ versus frequency. The peaks of the Gaussian filter function have a height of exactly 1 and are position exactly at the peak positions of abs($q(f)$) so that when multiplied together peak height doesn't change. The absolute value of filtered fft graph at the bottom is smoother than abs(q(f)) as the noise between peaks gets eliminated.

But before finalizing the parameters of the filter function, I used inverse fast fourier transform to plot the signal that would have created filtered fft p gave and compared it against the ideal sinusoid function $y(t_k)$. Calling the result of inverse fourier transform $cleaned(t_k)$, to compare *cleaned* with $y(t_k)$, I plotted the residual plot i.e. a position-time graph of $cleaned(t_k) - y(t_k)$. The ideal $cleaned(t_k)$ would have to be exactly equal to $y(t_k)$ and hence ideally the residual plot should

be equal to a zero function. But since there is noise, the residual plot would have small fluctuations. By varying *width* and *peak* parameters of the filter function, I found that *peak* = 12 and *width*= 5 Hz made the residual plot as close to the zero function as possible. Thus, these parameters were finalized as this made the inverse fourier transform of p give nearly the ideal signal $y(t_k)$ back as possible. The 3 signals $x(t_k)$, $cleaned(t_k)$, $y(t_k)$ and the best residual plot $cleaned(t_k) - y(t_k)$ are plotted in figure 6. As can be seen from the second and third plots from top in figure 6, the $cleaned(t_k)$ signal very closely resembles the ideal function $y(t_k)$ that was originally present in the noisy signal $x(t_k)$. We can observe that the filtered signal $cleaned(t_k)$ has small modulations in its amplitude compared to the constant amplitude of $y(t_k)$. This shows that it's not perfect. However, it is appreciable how close it is to the actual signal.

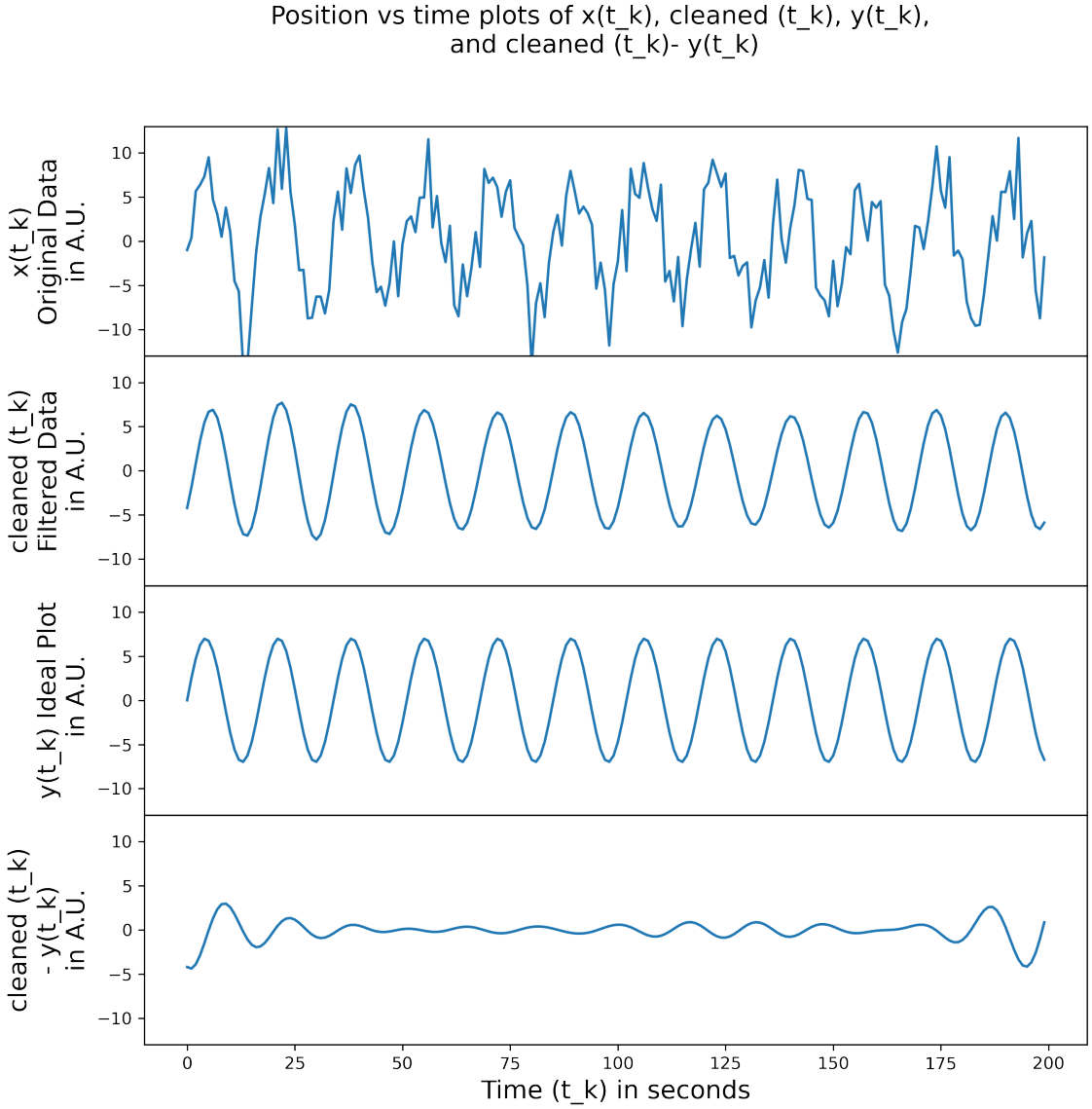

Figure 6: From top to bottom: Position vs time plots of $x(t_k)$, $cleaned(t_k)$, $y(t_k)$, and $cleaned(t_k) - y(t_k)$. The signal $cleaned(t_k)$ was extracted by using inverse fourier transform on the filtered fft p of the noisy data $x(t_k)$. Due to filtering, the signal $cleaned(t_k)$ plotted in the second plot from top closely resembles the ideal sinusoid $y(t_k)$ plotted in the third plot from top. This is also indicated by the residual plot at the bottom which has small fluctuations around the line $x = 0$.

Now, we can apply the same techniques of fourier transform and inverse fourier transform to study an unknown set of data which is what real experimental data is like. By putting the provided files 'noisy sine wave' and 'pickle-example.py' in the same folder and running the latter, we get a

noisy plot of data which is shown in figure 7. Although there are 2000 points in this original noisy data set, only the first 300 values are plotted for simplicity.

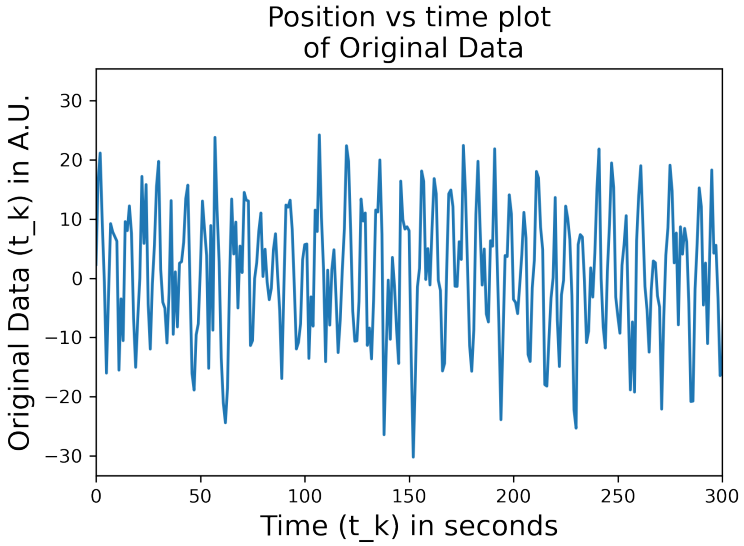

Figure 7: Position vs time plot of a noisy original signal generated by the file 'pickle-example.py' using data from the file 'noisy sine wave'. The data set coontains 2000 values but only the first 300 are shown above.

To decipher the sinusoids of various frequencies and amplitudes hidden in this noisy signal and compute a theoretical curve based on that information, it's first necessary to first fast fourier transform this original data to create the fourier transform array $J(f)$. This was done and half of the plot of $J(f)$ is shown in the top graph of figure 8 because the graph is repeated afterwards. Clearly, there are 3 prominent frequencies of different amplitudes involved in the signal with noise in between the visible peaks. To filter out the noise, it was necessary to create a filter function made up of 3 gaussians each of which aligned along separate peaks. But to make it easier computationally, I first used scipy's *find_peaks* method on the noisy fft (top subplot in figure 8: see next page) to find the amplitudes and indices of the noisy fft array where the peaks occur. Using the indices and the length $N = 2000$ frequency array generated by *fft.fftfreq*, the corresponding frequencies of the peaks were also found. Now, using these, I created a filter function as:

$$\text{filter\_function} = \text{filter\_func\_1} + \text{filter\_func\_2} + \text{filter\_func\_3} \tag{10}$$

where for the three peaks $i = 1, 2, 3$,

$$\text{filter\_func\_i} = \exp \frac{-(\text{freq} - \text{peak}_i)^2}{\text{width}_i} + \exp \frac{-(\text{freq} + \text{peak}_i - N)^2}{\text{width}_i}$$

Here peak_i are the frequencies corresponding to the three peaks found by running *find_peaks* initially on $J(f)$. The array *freq* is an array from 0 to 1999 which performs the same function as it did when we were dealing with only one gaussian function (eq. 9). All the widths were set to $\text{width}_i = 5$ initially. The generated filter function aligned very closely with the peaks of the $J(f)$ as can be seen in the second plot from top in figure 8. Finally, $J(f)$ was filtered by multiplying it with the generated filter function to create the filtered fft $K(f)$:

$$K(f) = J(f) \times \text{filter\_function} \tag{11}$$

Half of the graph of the filtered fft $K(f)$ is shown in the bottom plot of figure 8 because the fourier transform is repeated after the last.

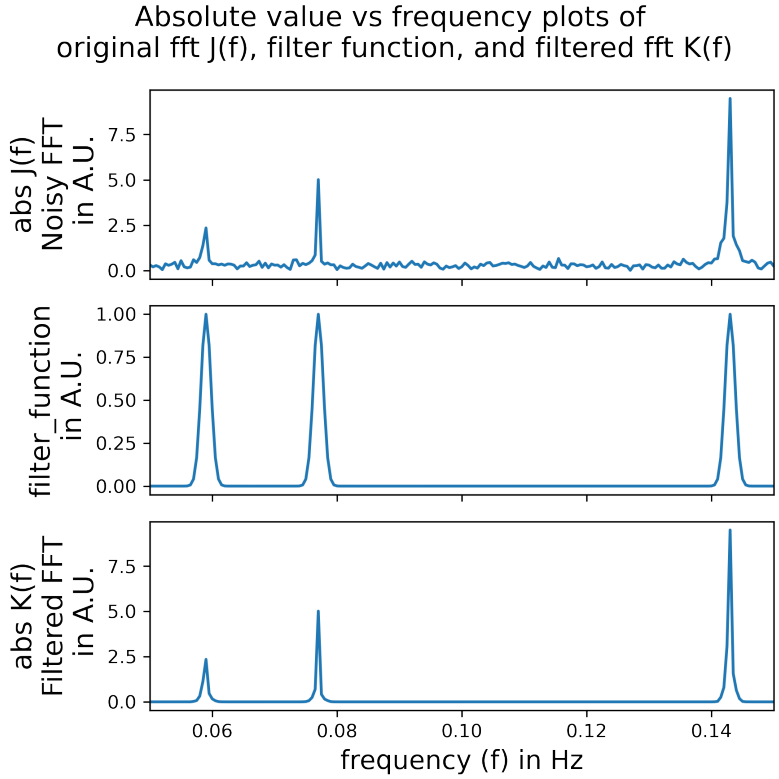

Figure 8: From top to bottom: Plots of the absolute value of fourier transform of noisy original data i.e. $J(f)$, Gaussian filter function *filter_function* (eq. 10), and absolute value of filtered fourier transform $K(f)$ versus frequency. The peaks of the Gaussian filter function have a height of exactly 1 and are position exactly at the peak positions of abs $J(f)$ so that when multiplied together peak height doesn't change. The graph of $K(f)$ at the bottom is smoother than abs $J(f)$ as the noise between peaks gets eliminated.

Now that we have the filtered fourier transform $K(f)$, we can perform inverse fft on it to get the ifft signal that would have generated it. The result is shown in the middle graph in figure 9. By using *find_peaks* on the filtered array abs $K(f)$, the best possible amplitudes and frequencies of sinusoids that would explain the original data were found as:

$$y_1(t_k) = 2.34 \sin(2\pi \cdot 059 \text{ Hz } \cdot t_k)$$

$$y_2(t_k) = 5.01 \sin(2\pi \cdot 0.077 \text{ Hz } \cdot t_k)$$

$$y_3(t_k) = 9.48 \sin(2\pi \cdot 0.143 \text{ Hz } \cdot t_k)$$

Hence, the best theoretical curve based on our analysis is given by:

$$\text{theoretical\_curve}(t_k) = 2.34 \sin(2\pi \cdot 0.059 \cdot t_k) + 5.01 \sin(2\pi \cdot 0.077 \cdot t_k) + 9.48 \sin(2\pi \cdot 0.143 \cdot t_k) \quad (12)$$

which is plotted in the bottom subplot of figure 9. Clearly, the theoretical curve and ifft signal only differ by a scaling factor which might be a result of normalization conditons.

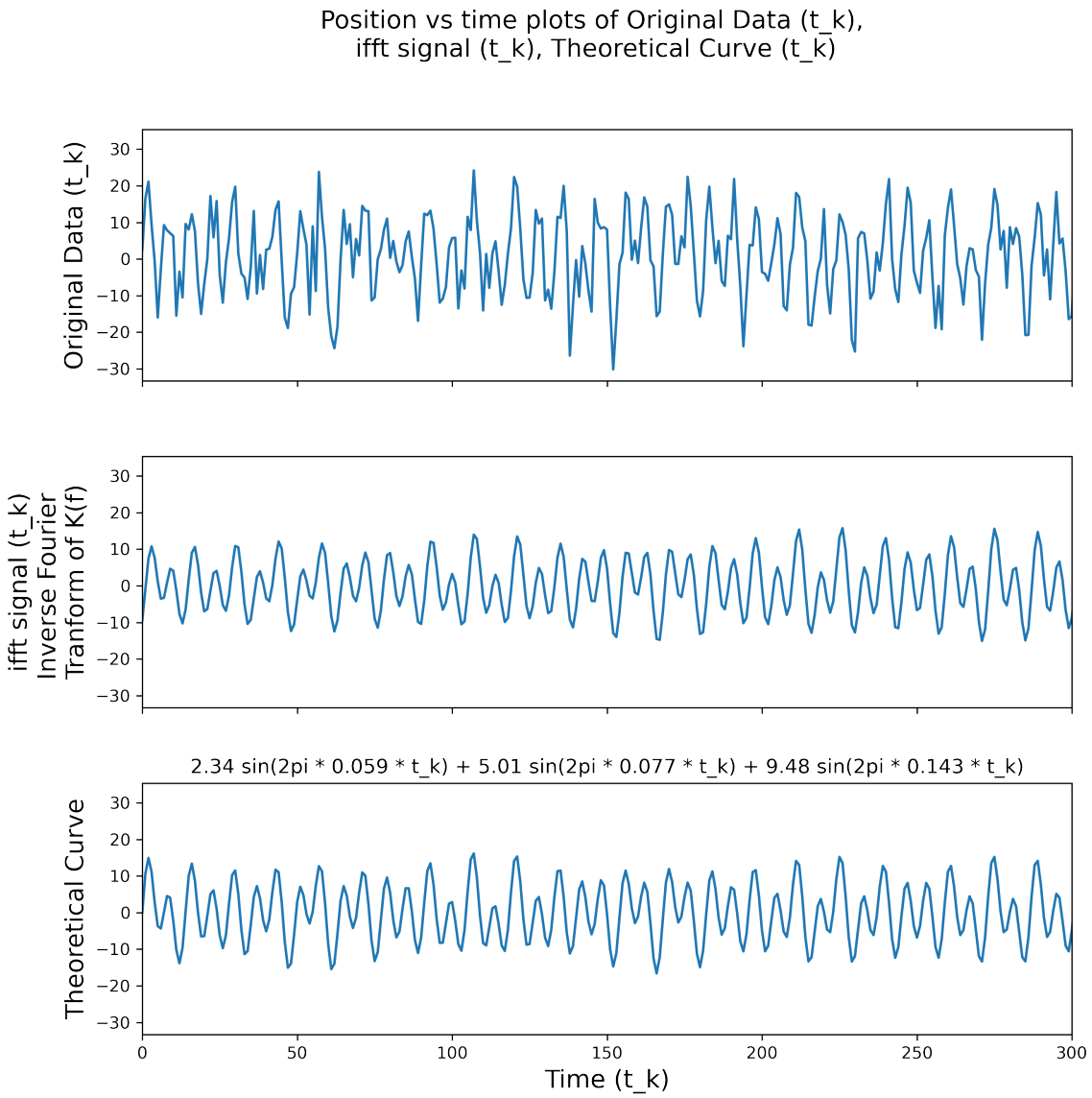

Figure 9: From top to bottom: Position vs time plots of Original Data(t_k), ifft signal(t_k) calculated by apply inverse fft on $K(f)$, and theoretical_curve(t_k)) calculated from the sum of the sinusoids of amplitudes and frequencies inferred by applying *find_peaks* on $K(f)$. Clearly, the theoretical curve and ifft signal only differ by a scaling factor which might be a result of normalization conditons.

These techniques can also be applied to functions which do not have a fixed periodicity. For example, consider the function $y(t_k)$ with linearly increasing frequency:

$$y(t_k) = \sin(2\pi \cdot f(t_k) \cdot t_k) \text{ with } f(t_k) = \frac{0.5}{2700 \text{ s}^2} t_k \tag{13}$$

The position versus time graph of $y(t_k)$ is plotted in figure 10. It clearly shows the time period of the sine wave decreasing implying that the frequency is increasing.

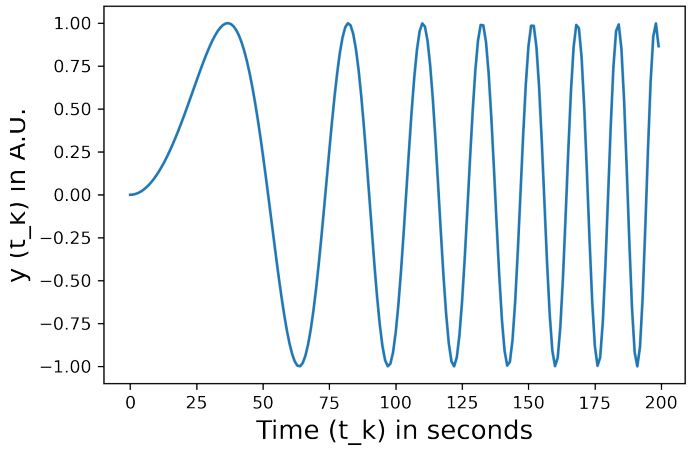

Figure 10: Position vs time plot of a sine wave with linearly increasing frequency.

We can then plot the absolute value of the fourier transform $Y(f)$ of the sine wave $y(t_k)$. This is done in figure 11.

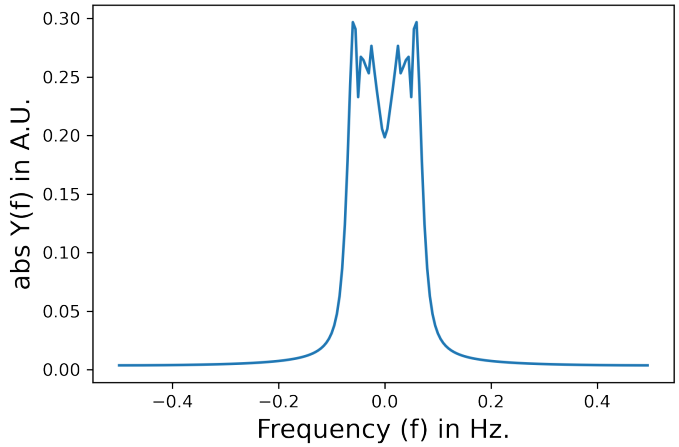

Figure 11: Amplitude vs frequency plot of the fourier transform abs $Y(f)$ of the sine wave $y(t_k) = \sin(2\pi \cdot f(t_k) \cdot t_k)$ with $f(t_k) = \dfrac{0.5}{2700\ \mathrm{s}^2} t_k$.

From figure 11, we can observe that there is a large peak at a particular frequency and a range of frequencies are involved. This indicates that to produce a sine wave with increasing frequency, we would have to add multiple sine waves of different frequencies which are quite close to each other. This ensures that the sum of those sine waves has a frequency that gradually increasing. There are also sine waves with higher frequencies but have very small amplitudes which means that it takes a long time for those high frequencies to affect the signal.

3 Conclusion

Over the course of this paper, we have explored the various ways in which Fourier transform and inverse fourier transform are useful in decomposing discrete data into various sinusoids of different frequencies and also composing all the data from a functions fourier transform to recreate the function itself. These techniques are very useful when data is plagued with noise as seen in some of the demonstrations shown here. Hence, these prove monumental when analyzing real world experimental data.