THE DISCRETE FOURIER TRANSFORM

This lab will introduce you to the Discrete Fourier Transform (DFT) and also the Inverse Discrete Fourier Transform (IDFT). This is essentially how one ventures into the frequency domain on a digital computer and it is important for you to understand the concept at both a high level and a low level. You will be computing the DFT from scratch and comparing your results to the functions provided by various libraries. You will also be brought into confrontation with the practicalities one must take into account while implementing the DFT on arbitrary signals.

LAB INSTRUCTIONS

1. You might have to spend time outside the allocated lab hours to finish the lab. In doing so, you can approach any of the course instructors, or TA(s).

2. You may work in teams or groups of 1-3 members and are not allowed to collaborate with anyone outside your group except the instructors of the course. You may change group or team members for different labs but you cannot change group members for the given lab you are working on.

3. Please document your code well by using appropriate comments, variable names, spacing, indentation, etc.

4. The starter code is not binding on you. Feel free to modify it as you wish. Everything is fine so long as you're getting the right results.

5. Please upload the `.ipynb` file to canvas. One notebook per team is fine and any one team member can upload the file. The required file(s) must be uploaded by the deadline.

1 Complex numbers and Complex Sinusoids (3 Points)

To start things off, we will learn how to define and use complex numbers and consequently complex sinusoids as they are the building blocks of the Fourier basis.

1.1 Complex Numbers

Defining complex numbers can be done in a number of ways. But before discussing these, one must be aware of how python handles $j = \sqrt{-1}$. It is represented as `1j` or more generally, `bj` where `b` is any scalar. Now, to define a complex number, one can simply write `z=4+3j` in python code or use the `complex()` function where you can pass in the real and imaginary parts separately, i.e, `complex(4,3)` or type the number in single quotes,i.e, `complex('4+3j')`. To extract the real and imaginary components of any complex number one can use numpy's `real()` and `imag()` functions respectively.

Now, proceed to finishing the first set of tasks:

(a) Define and operate on the required complex numbers as directed by the starter code.

(b) Define the complex number `4+3j` and plot it in the complex plane or Argand diagram.

(c) Compute and print the magnitude of the complex number defined above using the Pythagorean theorem (the plot from (b) can help) and by using numpy's `abs()` function. Compute and print the phase of the complex number using trigonometry (the plot from (b) can help) and by using numpy's `angle()` function.

You may want to verify that your answers are correct by working these tasks out on paper.

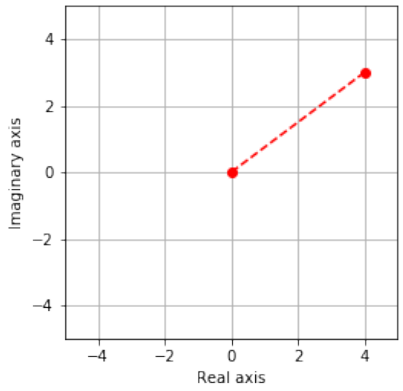

Figure 1: Expected result for 1.1 (c)

1.2 Complex Sinusoids

Now we move on to complex sinusoids and take one step closer to implementing a DFT from scratch.

(a) Define the complex exponential e^{kj} where k is the angle or phase value and $j = \sqrt{-1}$. Let k be a vector of two to three phase values like say, $\frac{\pi}{4}, -\frac{\pi}{2}$ etc. You may want to use a lambda expression/function to define you complex exponential (Check APPENDIX A if you're not familiar with lambda expressions in python).

(b) Plot a unit circle (unit magnitude or radius) on the Argand diagram or complex plane. Now overlay the points contained in k on the unit circle. Referring to the the plot of the complex number will help. Also, try different colors of dots by changing the first letter of the parameter in the plot function appropriately. For example, `'ro'` is a red circle and `'go'` is a green circle.

To plot the unit circle, plot the real and imaginary parts of e^{xj} where x is a linearly spaced vector from $-\pi$ to π. To implement this `np.linspace()` could prove useful.

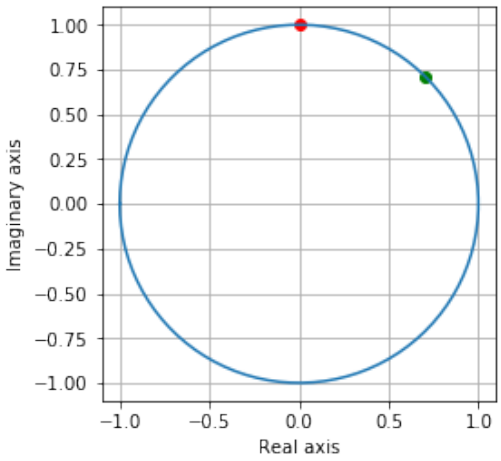

Figure 2: An example of a result for 1.2 (b)

(c) Define a complex sinusoid in the form $Ae^{-j(2\pi ft+\phi)}$ which lasts for a duration of 2s and is sampled with $fs = 500$ samples per second. A is the amplitude of the sinusoid and you set that to be a value of 2. f is the frequency of the sinusoid and you use 5 Hz. ϕ represents the phase of the sinusoid and you can use a phase value of $\frac{\pi}{3}$.

Now, plot the real and imaginary parts of this complex sinusoid and clearly label the axes and show the legend. The expected result given below can be used for reference.

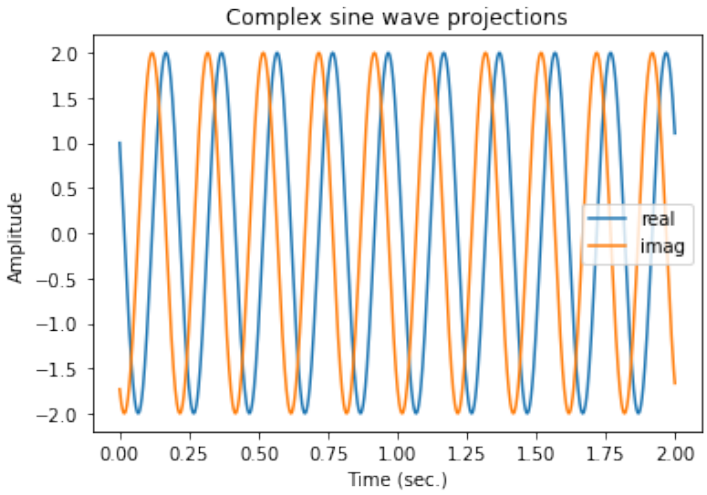

Figure 3: An example of a result for 1.2 (c)

(d) Whenever one finds the DFT of any signal, a pertinent question is that of the size of the DFT, or how many points (representing frequency bins) are in the DFT. We will look at the beginning of a 16 point DFT.

Recall that the DFT is essentially a projection of the signal onto a sinusoidal basis. So, we are going to plot the entire basis of a 16 point DFT building on top of the starter code as follows:

- Put the slowest and fastest frequency values possible (in Hz) for a sinusoid given the number of points in the variables `slowest` and `fastest` respectively.
- In the loop that is given, define the complex sinusoid in the same form as before but this time with unit amplitude. Also, $\phi = 0$ and t in this case is the time vector that has been defined for you in the starter code. The frequency vector will increment by one from the slowest to the fastest frequency.

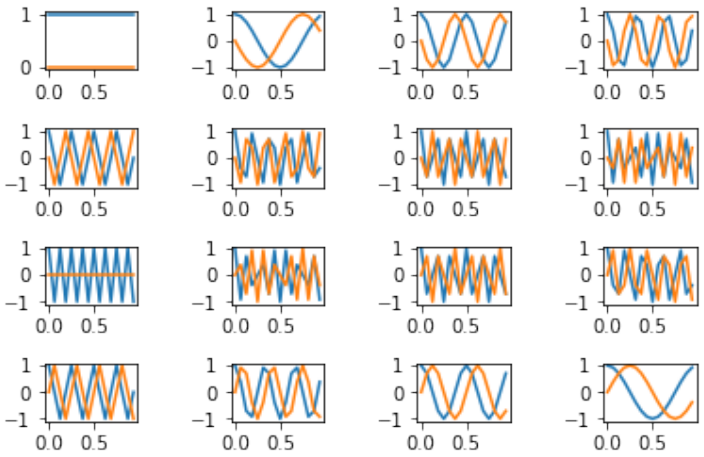

Figure 4: Expected result for 1.2 (d)

Discussion (3 Points)

- Justify your choice for the fastest and the slowest frequencies you chose for your N point DFT.
- When you print the Fourier basis for your N point DFT (as in Figure 4), how many unique frequencies are displayed in the plot? (Ignore the phase shifts)
- If the number of unique frequencies does not equal the number of points, then why is that the case despite your code that creates complex sinusoids of frequencies from 0 to N-1.

2 Naive Computation of the DFT and IDFT from First Principles (Vector Form) (5 Points)

2.1 DFT (Vector Form)

In this section, you will implement a DFT from scratch by creating complex exponentials that are orthonormal to each other (this is the Fourier basis) and projecting the signal onto this basis by taking dot products with each of the basis vectors. The Fourier coefficients are nothing but the dot product between the signal and the respective Fourier basis vectors. Recall that the

analysis formula to obtain the Fourier coefficients is given by

$$a_k = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{-jk\frac{2\pi}{N}n}$$

The steps that will take you through this for two basic sinusoids are as follows:

(a) Define two signals with the following specifications:

- A 4s long composite sinusoidal signal composed of two frequencies 4Hz and 6.5Hz with amplitudes 2.5 and 1.5 respectively. Please use a sampling rate of 1000 samples per second. Be sure to define a variable called `pnts1` that contains the number of points in the signal as it will be needed later in the code. Note that you can use `np.sin` or `np.cos` to create the signals.
- A 2s long sinusoidal signal with amplitude 2.5 and frequency 4 Hz and a DC offset (0 Hz frequency) of 1.5 . Be sure to define a variable called `pnts2` that contains the number of points in the signal as it will be needed later in the code. Similarly, use a sampling rate of 1000 samples per second. You can use `np.sin` or `np.cos` to create the signals.

(b) Given the time vector and initialization of the associated Fourier coefficient vectors, fill in the for loop that populates the Fourier coefficient arrays for both signals as follows:

- fill in the complex exponential by defining the complex exponential required as in the previous parts with the time vector being `fourTime1` and `fourTime2` respectively.
- compute the dot product between the signal and the complex exponential and sum the result (don't forget to normalize - this is discrete time after all). Here, you are implementing the analysis equation to obtain the Fourier coefficients

(c) Obtain the magnitude of the spectrum for both signals and store them as directed.

(d) Plot the result by running the code given. Note that in this case we are only plotting the positive frequencies in the DFT. The DFT is symmetric and you will be plotting the full spectra in the coming sections.

(e) Verify your results to be correct in terms of the frequency and magnitude of the sinusoids. You see the half spectrum indicate the magnitudes and frequencies you defined as mentioned in the aforementioned step.

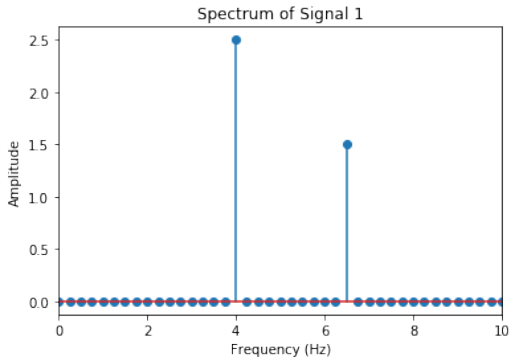

Figure 5: Expected result for Signal 1 in 2.1

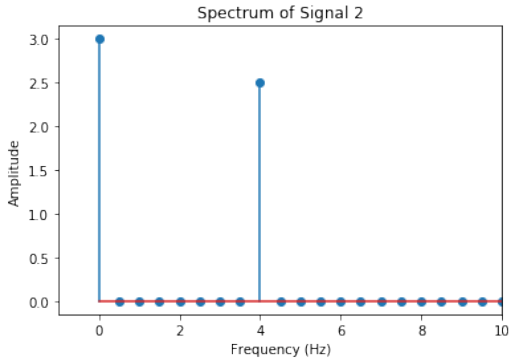

Figure 6: Expected result for Signal 2 in 2.1

2.2 IDFT (Vector Form)

In this section you will be implementing the IDFT using the synthesis equation. Recall that the synthesis equation is given by

$$x[n] = \sum_{n=0}^{N-1} a_k e^{jk\frac{2\pi}{N}n}$$

Using this one, can reconstruct the time domain signal that produced the given Fourier coefficients in the spectrum. Reconstruct the original signals defined with the specifications in the section before this with the IDFT by following the procedure outlined below:

(a) Initialize the vectors to store the reconstructed signals as a vector of zeros of same length as the corresponding original signals.

(b) In the for loops for both signals, implement the multiplication part of the synthesis equation and store it in the variables as directed. Next, implement the summation part by cumulatively adding the result to the variable for the reconstructed signal.

For both signals, plot, in a 3 by 1 subplot, the original signal, the reconstructed signal and the original signal overlaid on top of the reconstructed signal. The results expected for both the signals are given in the figures below.

Note: You may need to flip the order in which you plot the original and reconstructed signal together.

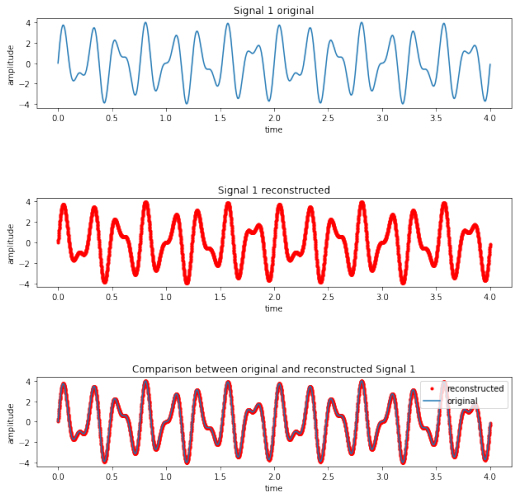

Figure 7: Expected result for Signal 1 in 2.2

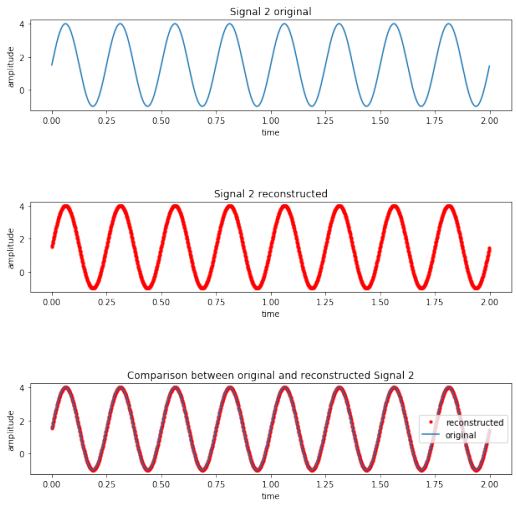

Figure 8: Expected result for Signal 2 in 3.2

Discussion (2 Points)

- Notice that the absolute value of the Fourier coefficients had to be multiplied by 2 to obtain the correct magnitudes of the frequencies in the signal. Why was this multiplication by 2 required for the magnitude spectrum to be correct?

- Let's say you wanted shift the level of a regular sinusoid of a given frequency that oscillates in amplitude from -1 to 1 so that it oscillates from 0 to 2 instead. How would you manipulate it's magnitude spectrum (given by your code) to do this?

3 Naive Computation of the DFT and IDFT from First Principles (Matrix Form) (5 Points)

3.1 DFT (Matrix Form)

3.1.1 Defining the DFT Matrix

The above vector operations can together be combined into one matrix operation by defining the appropriate "DFT Matrix" as:

$$W = \begin{bmatrix} 1 & 1 & 1 & 1 & \ldots\ldots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \ldots\ldots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \ldots\ldots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \ldots\ldots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \ldots\ldots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

where $\omega = e^{-\frac{2\pi j}{N}}$. Fill in the function `dft_function` to return such a DFT matrix of size N.

One way to do this is to make use of broadcasting in python and the other way is to use a nested loop to populate an $N \times N$ matrix with each element given by

$$W(n,k) = e^{-j\frac{2\pi}{N}nk}$$

If you're going to use a loop to create W, then don't forget to initialize it's data type to be complex. Otherwise, it will discard the imaginary part of it's entries. An example to illustrate this initialization is `a=np.zeros(3,2,dtype=complex)` which creates a 3×2 matrix of zeros with provision for entering complex numbers.

3.1.2 Defining the `dft` function

In the `dft` function, you will obtain the DFT by using obtaining the DFT matrix and then multiplying it with the signal. So, $X = Wx$ is all that

needs to be done to obtain the spectrum of the signal, which the function will return (Don't forget to normalize!).

3.1.3 Implementing shifting in the DFT

By default custom FFT functions return spectra that are not centred or symmetric. The `dft` function written in the previous subsection is no different. But it is important to do so and this is the focus of this subsection.

The coefficient number k indicates the contribution (in amplitude and phase) of a sinusoidal component of frequency

$$\omega_k = \frac{2\pi}{N}k$$

Because of the rotational symmetry of complex exponentials, a positive frequency ω between π and 2π is equivalent to a negative frequency of $\omega - 2\pi$; this means that half of the DFT coefficients correspond to negative frequencies and when we concentrate on the physical properties of the DFT it would probably make more sense to plot the coefficients centered around zero with positive frequencies on the right and negative frequencies on the left.

There is also another subtle point that we must take into account when shifting a DFT vector: we need to differentiate between odd and even length signals. With $k = 0$ as the center point, odd-length vectors will produce symmetric data sets with $(N-1)/2$ points left and right of the origin, whereas even-length vectors will be asymmetric, with one more point on the positive axis; indeed, the highest positive frequency for even-length signals will be equal to $\omega_{N/2} = \pi$. Since the frequencies of π and $-\pi$ are identical, we can copy the top frequency data point to the negative axis and obtain a symmetric vector also for even-length signals.

You can fill in the `dft_shift` function as follows:

- In the `if` statement, test whether the signal is of even length or odd length.
- Define the frequency bins of the shifted signal which contains both negative and positive frequencies. You can use `np.range()` to define the vector to have frequencies from $-\frac{N}{2}$ to $\frac{N}{2}$ but be mindful of the even or odd nature of N and make the required adjustments.

- Now, create the shifted spectrum. For this, the spectrum is essentially split into two parts, the first half in the range of 0 to $N/2$ and the other from $N/2$ to N. The full signal must slide to the right and warp over to the other side. So, the second half that goes from the middle to the end now runs from the beginning to the middle and the first half that went from the beginning to the middle now goes from the middle to the end. You can use `np.concatenate()` for this.

You have been provided a test signal to verify the correctness of your in-house shift and DFT functions defined thus far. The plot below can also be used to verify correctness.

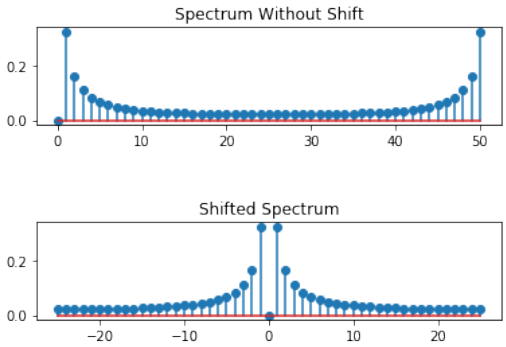

Figure 9: Expected result for 3.1.3 showing the shift of the spectrum

3.1.4 Mapping the DFT index to real world frequencies

In order to look at the spectrum of a signal with a DFT, we need to map the digital frequency "bins" of the DFT to real-world frequencies.

The k-th basis function over $\mathbb{C}^N$ completes k periods over N samples. If the time between samples is $\frac{1}{f_s}$, then the real-world frequency of the k-th basis function is periods over time, namely $k(\frac{f_s}{N})$.

From the aforementioned formula, the frequency resolution naturally follow as the the sampling frequency per sample. Define this in your `dft_map` function. If a shift is required, just use `dft_shift` to obtain the defined n and Y vectors. If no shift is required or if the shift condition is false, then return the spectrum that was passed to the function and the frequency bin

array without shift as directed.

Finally, obtain the frequency vector using the aforementioned formula and return it and the shifted spectrum.

3.1.5 Testing it all out

Now, test your newly defined functions to obtain the double sided spectra of Signal 1 and Signal 2 just like you did in Section 2. The frequency mapping, spectrum shift, spectrum magnitude should all be correct in that they represent the parameters you coded when defining the signals. Figure 10 and Figure 11 can be used to verify correctness.

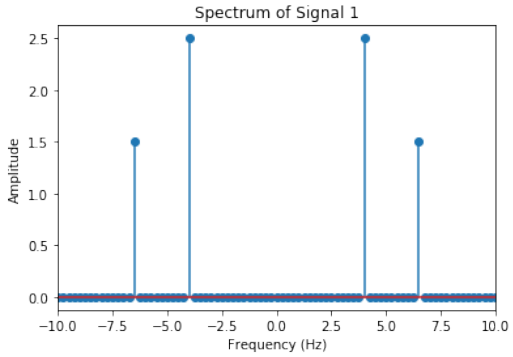

Figure 10: Expected result for Signal 1 in 3.1.5

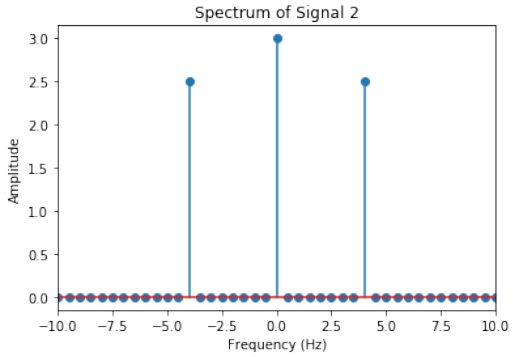

Figure 11: Expected result for Signal 2 in 3.1.5

3.2 IDFT (Matrix Form)

To obtain the IDFT of a spectrum, say X, one need only invert the operation performed in obtaining the DFT, i.e, pre-multiplying the DFT by the DFT matrix W^{-1}. This serves to undo the DFT, which was done by pre-multiplying by W and reconstruct the original time domain signal.

Note that the DFT matrix is an orthogonal matrix and this means that it's inverse is equal to it's conjugate transpose. Finding the conjugate transpose of a matrix easier than inverting it and hence, implement $x = W^*X$ to reconstruct both signal 1 and signal 2. In python, one of many ways to find the conjugate transpose of a matrix, say W is, `W.T.conjugate()`. Plot them like you did in the previous section. In this case, just the two plots with the original signal overlaid on the reconstructed signal will suffice.

Note: You may need to flip the order in which you plot the original and reconstructed signal together.

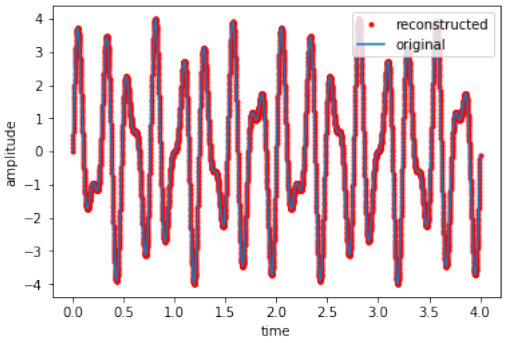

Figure 12: Expected result for Signal 1 in 3.2

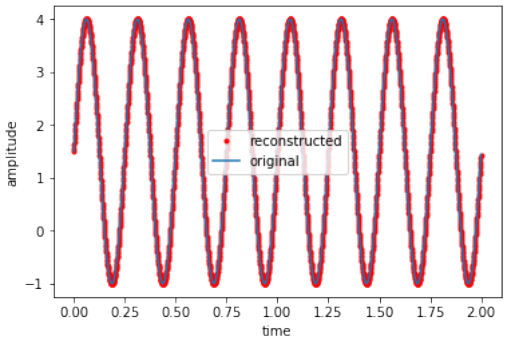

Figure 13: Expected result for Signal 2 in 3.2

Discussion (5 Points)

- notice that the `fft` functions of `numpy` and `scipy` do not have a shift in the spectrum by default and neither does your `dft` function. When or why would you want to work with a spectrum that is not shifted?
- Notice that `numpy` and `scipy` have provision to find the N point DFT of a signal when N is greater than the length of the signal but your `dft` function breaks down if N is not the length of the signal. What can you do to circumvent this problem?
- You've learned about CTFT (Continuous Time Fourier Transforms), DTFT(Discrete Time Fourier Transform), DFT and FFT. What is the relationship between these transforms? How are they different from each other and in what ways are they similar? Do this comparison for each one against the rest.

4 Numerical Precision Issues With the DFT and IDFT (2 Points)

In this section we will investigate the numerical precision issues associated with the naive computation of the DFT. We will do so by comparing the method you implemented with the Fast Fourier Transform (FFT) function in `scipy` or `numpy`.

The signal we will be working with is a discrete pulse as shown in Figure 14. It is a discrete signal of length 128 samples and has the first 64 samples

to be unit height. The remaining are zero. Define a discrete step function as directed in the starter code.

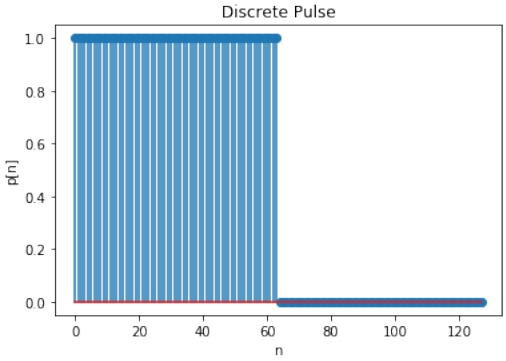

Figure 14: Discrete pulse

Now, using the `dft` function you defined in the previous part, find the DFT of the discrete pulse and plot it's magnitude and phase. Your results should look something like what's shown in Figure 15.

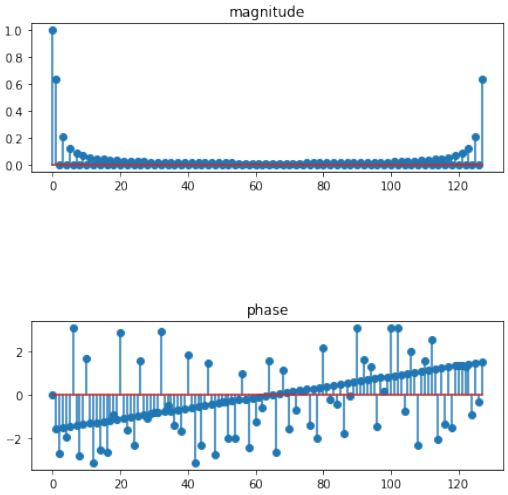

Figure 15: Expected result for magnitude and phase with in house DFT function

Repeat the same but using the fft function from `nunmpy` or `scipy`. They are invoked as `numpy.fft.fft(signal,N)` and `scipy.fftpack.fft(signal,N)`

where `signal` is the discrete pulse and `N` is the length of the FFT, in this case, it's the same as that of the signal. Your results should look something like that shown in Figure 16.

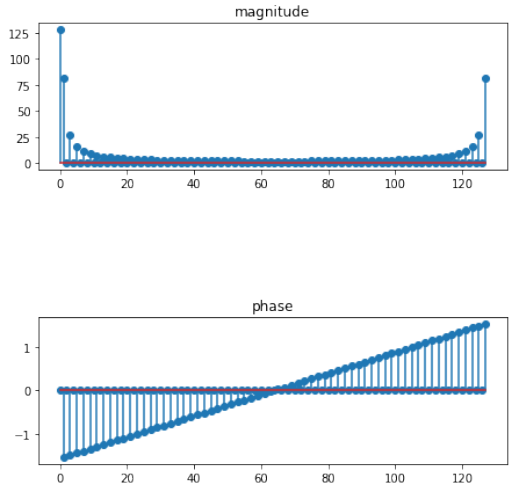

Figure 16: Expected result for magnitude and phase with FFT functions from libraries

Now, just as in the previous section, reconstruct the discrete pulse from the DFT you obtained in this section using your `dft` function by using the IDFT. Plot the stem plot of the error between the real and imaginary parts of the original and the reconstructed signal. Your results should look something like that shown in Figure 17.

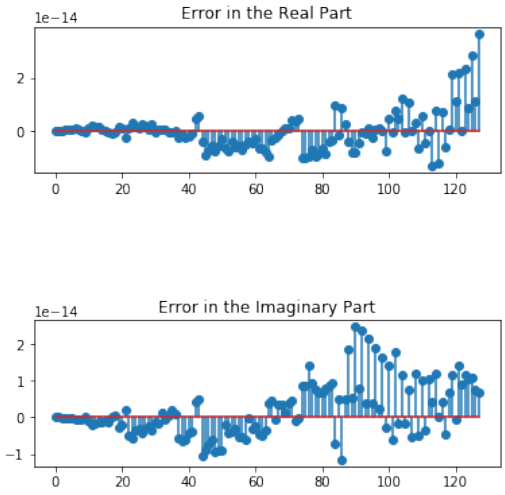

Figure 17: Error plots when using IDFT as defined in this lab

Repeat the same but use the IDFT functions defined in `numpy` or `scipy` instead of what you implemented in the previous section. They are invoked as `numpy.fft.ifft(X)` and `scipy.fftpack.ifft(X)` where X if the array of DFT coefficients . Your results should look something like that shown in Figure 18.

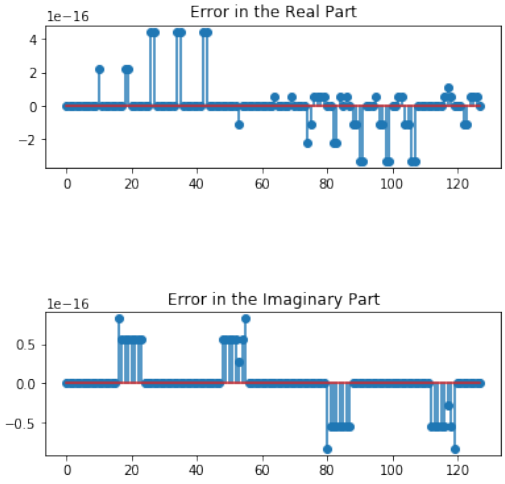

Figure 18: Error plots when using numpy's or scipy's IDFT function

Discussion (2 Points)

- What inherent problem do you see with your naive implementation of the DFT compared to the DFT obtained from using FFT in `numpy` or `scipy`?
- Given that the naive method has the problem hinted at in the previous item, when would you still be OK with using your `dft` function to find the Fourier transform of a signal instead of relying on `numpy` or `scipy`?

5 Minimizing Energy Spread and Zero Padding (5 Points)

5.1 Minimizing Energy Spread or Spectral Leakage

In this section we will be looking at one of the practicalities associated with the DFT in that there could be some spurious spikes in the DFT spectrum indicative of frequencies we know are not in the signal. To deal with this you will write a function called `minimizeEnergySpreadDFT` which has been defined in the starter code. But before you start filling it in, define a 2s long signal composed of two sinusoids with frequencies 300 Hz and 800 Hz. Use a sampling rate of 48000 samples per second.

The DFT magnitude spectrum of a sinusoid has only one pair of non-zero values (in the positive and negative halves of the DFT spectrum) when its frequency coincides with one of the DFT bin frequencies. This happens when the DFT size (M in this question) contains exactly an integer number of periods of the sinusoid. Since the signal in this question consists of two sinusoids, this condition should hold true for each of the sinusoids, so that the DFT magnitude spectrum has only two non-zero values, one per sinusoid.

So, your next step is to start filling in the function provided as follows:

- Find the time period of each sinusoid in the sampled composite signal and store those in `t1` and `t2` respectively.
  To do so for the standard continuous time sinusoid $\sin(2\pi ft)$, follow the standard procedure of replacing t with nT_s where T_s is the sampling time. Do this for each sinusoidal component individually and simplify them to the form $\sin(2\pi fn)$ and fine the time periods from there (you may want to work this out with pen and paper).
- Calculate M, which can be computed as the Least Common Multiple (LCM) of the sinusoid periods (in samples). The LCM of two numbers

x, y can be computed as: $x \times y/GCD(x, y)$, where GCD denotes the greatest common divisor. You can use the `gcd` function from the math library in python.

- Find the M points DFT of the first M samples of the signal using the `dft` function you defined in an earlier section.
- Obtain the shifted spectrum and the frequency vector using the function `dft_map` that you defined in an earlier section and have the function return them.

Now, obtain two plots to show the effect of minimizing spectral leakage. The first plot will be the DFT without any minimization of spectral leakage and can be done by obtaining the spectrum of the defined signal using the `dft` function and plotting the double sided and shifted spectrum. The second plot will be the DFT with minimization of spectral leakage and to do so use the `minimizeEnergySpreadDFT` function and plot the result.

Also, you may need to interchange the order of your plotting , the energy minimized plot may have to come first followed by the regular spectrum plot. Do this if you run into an error.If you run into a memory error, then take only the first 1024 points of the signal and use it both, for the regular DFT and the energy minimized DFT.

You can use the two figures below to verify your result.

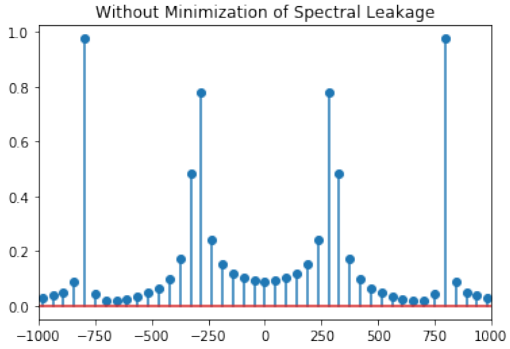

Figure 19: Expected result for Signal 2 in 3.2

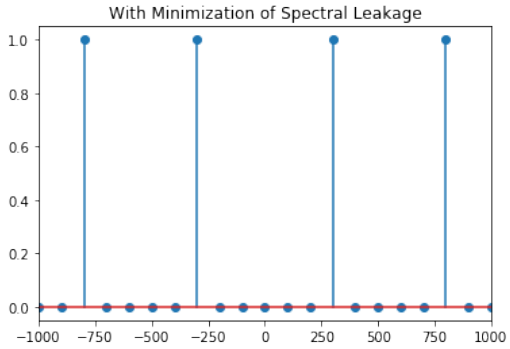

Figure 20: Expected result for Signal 2 in 3.2

5.2 Zero Padding

5.2.1 Optimal Zero Padding

In this section we will see how to zero pad the signal in the time domain to help get a less noisy spectrum.

Zero-padding needs to be done such that one of the bin frequencies of the DFT coincides with the frequency of the sinusoid. One of the DFT bin frequencies coincides with the frequency f of a sinusoid when the DFT size (N in this question) contains exactly an integer number of periods of the sinusoid.Choose the minimum zero-padding length for which this condition is satisfied.

For example, if $f = 100$ Hz and $fs = 1000$ Hz, one period of the sinusoid has 10 samples. Then given a signal of length $M = 25$ samples, there are 2.5 periods in it. The minimum zero-padding length here would be 5 samples (0.5 period), so that the DFT size $N = 30$ corresponds to 3 periods of a sinusoid of frequency $f = 100$ Hz.

Now fill in the `optimalZeropad` function as follows:

- Define `M` to be the length of the signal.
- Calculate the number of zeros to be padded to the signal and store it in some variable, say, `pad`.
- Define `N` to be the length of the signal after zero padding

- Find the `N` points DFT of the first `N` samples of the signal using the `dft` function.
- Use `dft_map` to find the shifted spectrum and the associated frequency vector and return them.

Just as in section 5.1, find the DFT before and after applying the extra step, which is zero padding in this case. Obtain the spectrum plots for the signal before and after zero padding.

You can verify your answers with the plots below. In general, the DFT should be cleaner and better,i.e, have a lower threshold to differentiate spurious spikes from the required spikes, if the zero padding is done right.

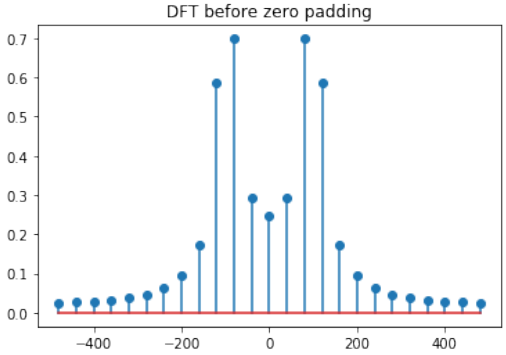

Figure 21: Expected result for 5.2.1 before zero padding the signal

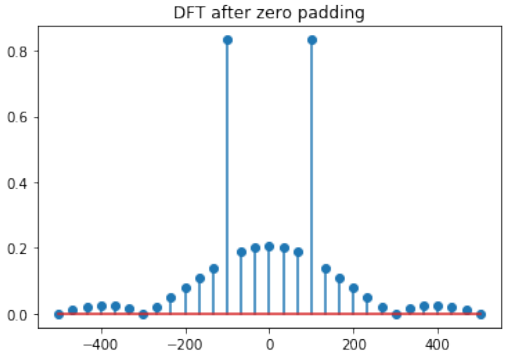

Figure 22: Expected result for 5.2.1 after zero padding the signal

5.2.2 Frequency Resolution

Since the resolution of a DFT depends on the length of the data vector, one may erroneously assume that, by artificially extending a given data set, the resulting resolution would improve. Note that here we're not talking about collecting more data but rather, we have a data set and we append zeros (or any other constant value) to the end of it.

Assume we're in $\mathbb{C}^N$ with $N = 256$. The resolution of the DFT in this space is

$$\Delta = 2\pi/N = 2\pi/256 \approx 0.0245$$

Build a signal with two discrete sinusoids with frequencies more than Δ apart and let's look at the spectrum. The discrete sinusoids can be of the form $\cos(\Omega n)$ and you can use an angular frequency of $\frac{\pi}{5}$ radians per second. Next, find it's DFT using the `dft` function and plot the first half of it's magnitude spectrum. You should be able to see the two frequencies distinctly in the plot as shown below.

Note: You are given freedom in choosing how far apart or close the sinusoids should be to each other and so, your results may not be exactly the same as those shown, but similar.

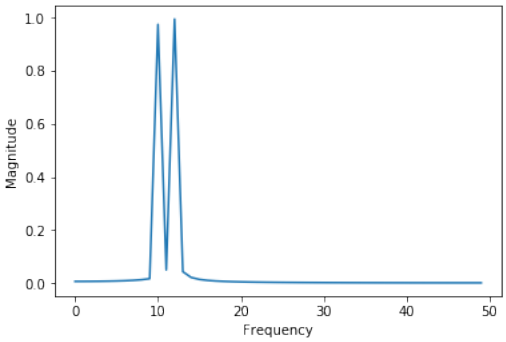

Figure 23: Expected result for 5.2.2 where the composite frequencies can be detected given the frequency resolution

Next, repeat the same thing but let the signal have sinusoids that are less than Δ apart. The plot should show poor or no differentiation between the composite sinusoids and will be as below.

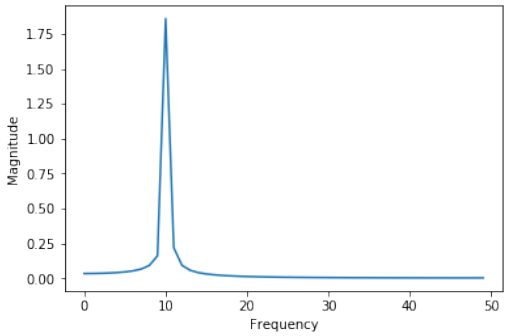

Figure 24: Expected result for 5.2.2 where the composite frequencies cannot be detected given the frequency resolution

As seen in the previous plot, the two frequencies cannot be resolved by the DFT. If you try to increase the data vector by zero padding, the plot will still display just one peak. Append 2000 zeros to the signal and repeat the above process and see if that helps improve resolution and consequently display the two peaks clearly.

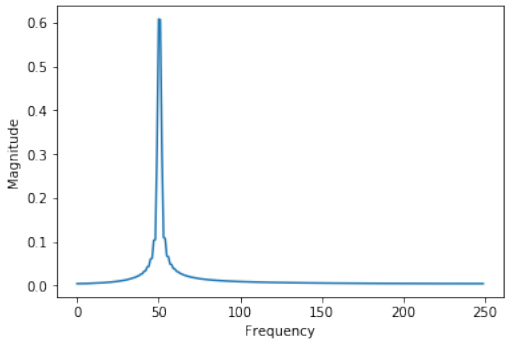

Figure 25: Expected result for 5.2.2 where zero padding is done to attempt to improve frequency resolution

Discussion (3 Points)

- Mention any ONE of the adverse effects of spectral leakage.
- Why does zero padding not increase frequency resolution and separate out the peaks as seen in Section 5.2.2?

- How would you modify the code to show that frequency resolution does indeed decrease with signal length? (Hint: try taking the DFT of only a part of the signal and see how you can fail to resolve the two sinusoids that are more than delta apart.)