Deev Patel & Wei Tong

CS 474: Image Processing & Interpretation

Wednesday, October 31, 2018

Programming Assignment # 3

The Discrete Fourier Transform

(Implemented in 1-D & 2-D via the FFT Algorithm)

Division of Work

Experiment 1, of both the code and the report, was done by Deev. Experiment 2, of both the code and the report, was done by Wei. Additionally, both Deev and Wei helped in implementing the 2-D FFT function. We believe this was close to an equal division of work because it gave each person responsibility over one of the two major parts of the project.

Technical Discussion

(1) The 1-Dimensional Fourier Transform & Its Discrete Implementation

The Fourier transform is complex transformation that takes a signal and decomposes it into a linear combination of its basic frequencies. The transformation allows us to convert signals from the time/spatial domain to the frequency domain. This can be helpful because many operations, such as the removal of certain types of noise and large convolutions, are easier and faster to implement in the frequency domain. Mathematically, the Fourier transform and its inverse can be defined for any continuous signal f(x) in the time domain and its frequency domain counterpart F(u) as shown below in (Eq 1) and (Eq 2) respectively.

$$\mathcal{F}\{f(x)\} = F(u) = \int_{-\infty}^{\infty} f(x) e^{-j2\pi ux}\, \mathrm{d}x \quad \text{(Eq 1)}$$

$$\mathcal{F}^{-1}\{F(u)\} = f(x) = \int_{-\infty}^{\infty} F(u) e^{j2\pi ux}\, \mathrm{d}u \quad \text{(Eq 2)}$$

In order for the Fourier transform to be of any practical use with real world signals, we must consider the Discrete Fourier Transform (DFT). Here, if we take the case of having a time domain signal f(x) with N equally spaced out samples — say f(0), f(1), … , f(N-1) — we can consider each sample to be an impulse. Doing this allows us to expand the integral from (Eq 1) into the summation shown below, where Δx is the time/space between each sample.

$$\mathcal{F}\{f(x)\} = F(u) = f(0)e^{-j2\pi u(0)\Delta x} + \ldots + f(k)e^{-j2\pi u(k)\Delta x} + \ldots + f(N-1)e^{-j2\pi u(N-1)\Delta x}$$

This summation can then be simplified and normalized by a factor of $1/N$ to get the forward DFT equation shown in (Eq 3). Similarly, we can derive the inverse DFT equation shown in (Eq 4) by expanding (Eq 2).

$$\text{Forward DFT: } F(u) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)e^{-\frac{j2\pi ux}{N}} \quad \text{for } u = 0,1,\ldots,N-1 \quad \text{(Eq 3)}$$

$$\text{Inverse DFT: } f(x) = \sum_{u=0}^{N-1} F(u)e^{\frac{j2\pi ux}{N}} \quad \text{for } x = 0,1,\ldots,N-1 \quad \text{(Eq 4)}$$

While (Eq 3) and (Eq 4) allow us to convert a discrete signal between the frequency and time domains, they are computationally expensive in their raw forms. This is because each summation has *N* components and we have to compute *N* summations. Thus, both the forward and inverse DFT equations are $O(N^2)$. However, this can be reduced to $O(N \log_2(N))$ via the Fast Fourier Transform (FFT) algorithm. At a high level, the FFT is a divide and conquer algorithm that assumes the number of samples to be a power of two and successively splits the signal up into its even and odd parts.

More specifically, the FFT is based on the fact that for a discrete signal f(*x*) with 2*M* samples, we can compute the DFT for $u = 0,1, \ldots, M-1$ and for $u = M, M+1, \ldots, 2M-1$ using the same basic values with a few minor sign changes. If we let $F_{even}(u)$ and $F_{odd}(u)$ be the DFT of the even and odds terms of f(*x*) respectively, then we can let simplify (Eq 3) to obtain (Eq 5). For simplicity and clarity, we let $W_{2M} = e^{-\frac{j\pi}{M}}$.

$$\text{For } u = 0,1,\ldots,M-1: \quad F(u) = \frac{1}{2}\left[F_{even}(u) + W_{2M}{}^{u} F_{odd}(u)\right]$$

$$\text{For } u = M, M+1,\ldots,2M-1: F(u) = \frac{1}{2}\left[F_{even}(u) - W_{2M}{}^{u} F_{odd}(u)\right] \quad (Eq\ 5)$$

From (Eq 5), it is clear that the full DFT can be computed by finding the DFT of the odd and even terms and changing the sign related to the odd terms when $u \geq M$. This essentially

allows us to reduce the computational complexity of the DFT equation when the number of samples is a power of two because we can successively split the signal all the way down to a single impulse. Then, we can compute the DFT of the impulse (which is itself) and gradually reconstruct the whole DFT.

The inverse DFT can be computed in a similar fashion. Moreover, since the forward and and inverse DFT equations only differ by the sign of the exponent in the complex exponential, we can use the same implementation for both. All we have to do is add a flag to change the sign of the exponent from positive to negative when computing the inverse transformation.

(1a) Experiment 1, Part A

Given a real discrete signal f = [2 , 3 , 4 , 4], we can compute its fundamental frequencies via the DFT as described in (Eq 3). Here, it helps to use Euler's formulas for complex exponentials, given in (Eq 6), to simplify and compute values.

$$e^{\pm j\theta} = \cos(\theta) \pm j\,\sin(\theta) \quad (Eq\ 6a)$$

$$\cos(\theta) = \frac{1}{2}\left[e^{j\theta} + e^{-j\theta}\right] \quad (Eq\ 6b)$$

$$\sin(\theta) = \frac{1}{2j}\left[e^{j\theta} - e^{-j\theta}\right] \quad (Eq\ 6c)$$

The resulting signal in the frequency domain is F = [3.25 , -0.5+0.25j , -0.25, -0.5-0.25j]. For reference, the calculations at each of the four discrete values are shown below.

$$\begin{aligned} F(0) &= \tfrac{1}{4}[f(0)e^{\frac{-j2\pi(0)(0)}{4}}) + f(1)e^{\frac{-j2\pi(0)(1)}{4}}) + f(2)e^{\frac{-j2\pi(0)(2)}{4}}) + f(3)e^{\frac{-j2\pi(0)(3)}{4}})] \\ &= \tfrac{1}{4}[f(0)e^0 + f(1)e^0 + f(2)e^0 + f(3)e^0] = \tfrac{1}{4}[f(0) + f(1) + f(2) + f(3)] \\ &= \tfrac{1}{4}[2 + 3 + 4 + 4] = \tfrac{13}{4} \end{aligned}$$

$$F(1) = \frac{1}{4}[f(0)e^{\frac{-j2\pi(1)(0)}{4})} + f(1)e^{\frac{-j2\pi(1)(1)}{4})} + f(2)e^{\frac{-j2\pi(1)(2)}{4})} + f(3)e^{\frac{-j2\pi(1)(3)}{4})}]$$
$$= \frac{1}{4}[f(0)e^{0} + f(1)e^{\frac{-j\pi}{2}} + f(2)e^{-j\pi} + f(3)e^{\frac{-j3\pi}{2}}]$$
$$= \frac{1}{4}[f(0) + f(1)(cos(\frac{\pi}{2}) - jsin(\frac{\pi}{2})) + f(2)(cos(\pi) - jsin(\pi)) + f(3)(cos(\frac{3\pi}{2}) - jsin(\frac{3\pi}{2}))]$$
$$= \frac{1}{4}[2 + 3(-j)) + 4(-1)) + 4(j))] = \frac{1}{4}[-2 + j] = \frac{-1}{2} + j\frac{1}{4}$$

$$F(2) = \frac{1}{4}[f(0)e^{\frac{-j2\pi(2)(0)}{4})} + f(1)e^{\frac{-j2\pi(2)(1)}{4})} + f(2)e^{\frac{-j2\pi(2)(2)}{4})} + f(3)e^{\frac{-j2\pi(2)(3)}{4})}]$$
$$= \frac{1}{4}[f(0)e^{0} + f(1)e^{-j\pi} + f(2)e^{-j2\pi} + f(3)e^{-j3\pi}]$$
$$= \frac{1}{4}[f(0) + f(1)(cos(\pi) - jsin(\pi)) + f(2)(cos(2\pi) - jsin(2\pi)) + f(3)(cos(3\pi) - jsin(3\pi))]$$
$$= \frac{1}{4}[2 + 3(-1)) + 4(1)) + 4(-1))] = \frac{-1}{4}$$

$$F(3) = \frac{1}{4}[f(0)e^{\frac{-j2\pi(3)(0)}{4})} + f(1)e^{\frac{-j2\pi(3)(1)}{4})} + f(2)e^{\frac{-j2\pi(3)(2)}{4})} + f(3)e^{\frac{-j2\pi(3)(3)}{4})}]$$
$$= \frac{1}{4}[f(0)e^{0} + f(1)e^{\frac{-j3\pi}{2}} + f(2)e^{-j3\pi} + f(3)e^{\frac{-j9\pi}{2}}]$$
$$= \frac{1}{4}[f(0) + f(1)(cos(\frac{3\pi}{2}) - jsin(\frac{3\pi}{2})) + f(2)(cos(3\pi) - jsin(3\pi)) + f(3)(cos(\frac{9\pi}{2}) - jsin(\frac{9\pi}{2}))]$$
$$= \frac{1}{4}[2 + 3(j)) + 4(-1)) + 4(-j))] = \frac{1}{4}[-2 - j] = \frac{-1}{2} + j\frac{-1}{4}$$

From the resulting signal, we can also calculate the magnitude and phase at each discrete value using (Eq 7) and (Eq 8).

$$\text{Magnitude}(a + jb) = \sqrt{a^2 + b^2} \quad (Eq\ 7)$$

$$\text{Phase}(a + jb) = \tan^{-1}\left(\frac{b}{a}\right) \quad (Eq\ 8)$$

For reference, the magnitude turns out to be |F(*u*)| = [3.25 , 0.559 , 0.25 , 0.559].

(1b) Experiment 1, Part B

Since the Fourier transform breaks down a signal into its fundamental frequencies, the Fourier transform of a function that only consists of one frequency will simply be that value. Thus we should expect the Fourier transform of a function in the form of f(*x*) = cos(2π*ax*) for some constant *a* to yield an impulse, or delta, function. We can prove this by applying the definition of

the forward Fourier transform from (Eq 1). Then, we can convert the cosine to a complex exponential using (Eq 6b) and split the integral.

$$\mathcal{F}\{\cos(2\pi ax)\} = \int_{-\infty}^{\infty} \cos(2\pi \mathrm{ax}) \mathrm{e}^{-\mathrm{j}2\pi \mathrm{ux}}\, \mathrm{dx} = \frac{1}{2}\int_{-\infty}^{\infty} \left[e^{j2\pi \mathrm{ax}} + e^{-j2\pi \mathrm{ax}}\right] \mathrm{e}^{-\mathrm{j}2\pi \mathrm{ux}} \mathrm{dx}$$
$$= \frac{1}{2}\left[\int_{-\infty}^{\infty} \mathrm{e}^{-\mathrm{j}2\pi \mathrm{x}(u-a)} \mathrm{dx} + \int_{\infty}^{-\infty} \mathrm{e}^{-\mathrm{j}2\pi \mathrm{x}(u+x)} \mathrm{dx}\right]$$

From here, we can see that the Fourier transform of cosine is nothing more than two impulses, or delta functions, at *a* and *-a* with a height of 0.5. This is shown in (Eq 9) below.

$$\mathcal{F}\{\cos(2\pi ax)\} = \frac{1}{2}[\delta(u-a) + \delta(u+a)] \quad \text{(Eq 9)}$$

The fact that cosine yields two impulses instead of the expected one is just a small side effect of the complex nature of the Fourier transform. In reality, the negative frequency of *-a* still corresponds to the same fundamental frequency of *a*.

Since we are necessarily interested in the discrete case, we must look at how this applies to the DFT. While the DFT of a discrete sampling of a cosine function will still result in two impulses, we have to take the periodic nature of the DFT into account to fully understand the results. Despite only having a fixed number of sampling points, both the inputs and outputs of the DFT can be considered periodic. This is a direct result of the way we define the DFT in (Eq 3). If we take (Eq 3) and consider the value of F(*u* + *N*), we get the following.

$$F(u+N) = \frac{1}{N}\sum_{x=0}^{N-1} f(x) e^{-\frac{j2\pi(u+N)x}{N}} = \frac{1}{N}\sum_{x=0}^{N-1} f(x) e^{\frac{-j2\pi \mathrm{ux}}{N}} e^{-j2\pi \mathrm{x}} = \frac{1}{N}\sum_{x=0}^{N-1} f(x) e^{\frac{-j2\pi \mathrm{ux}}{N}} [\cos(2\pi \mathrm{x}) - j\, \sin(2\pi \mathrm{x})]$$

From here, we can use that fact that *x* only takes integer values to simplify cos(2π*x*) to 1 and sin(2π*x*) to 0. Doing this simplification yields (Eq 10).

$$F(u+N) = \frac{1}{N}\sum_{x=0}^{N-1} f(x) e^{\frac{-j2\pi ux}{N}} (1 - j0) \; = \; F(u) \quad \text{(Eq 10)}$$

From (Eq 10) it is evident that the DFT is periodic with a period of *N*. This is of importance because when we compute the DFT, we want to see the full period. However, since we can only compute the DFT at integer values in the range [0 , *N*-1] we will not see a full period. Instead we will see two half periods. To solve this problem, we must shift the period of the DFT by *N*/2. In order to do this, we can compute F(u + *N*/2) as shown below.

$$F\left(u+\frac{N}{2}\right) = \frac{1}{N}\sum_{x=0}^{N-1} f(x) e^{-\frac{j2\pi\left(u+\frac{N}{2}\right)x}{N}} = \frac{1}{N}\sum_{x=0}^{N-1} f(x) e^{\frac{-j2\pi ux}{N}} e^{-j\pi x} = \frac{1}{N}\sum_{x=0}^{N-1} f(x) e^{\frac{-j2\pi ux}{N}} [\cos(\pi x) - j\,\sin(\pi x)]$$

Here, we can again use the fact that *x* only takes integer values to further simplify the right hand side and obtain (Eq 11).

$$F\left(u+\frac{N}{2}\right) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)(-1)^x e^{\frac{-j2\pi ux}{N}} \quad \text{(Eq 11)}$$

(Eq 11) shows us that we can obtain a full period by multiplying the original function by $(-1)^x$ before computing the DFT.

(1c) Experiment 1, Part C

The Fourier transform of a rectangular function yields a sinc function.

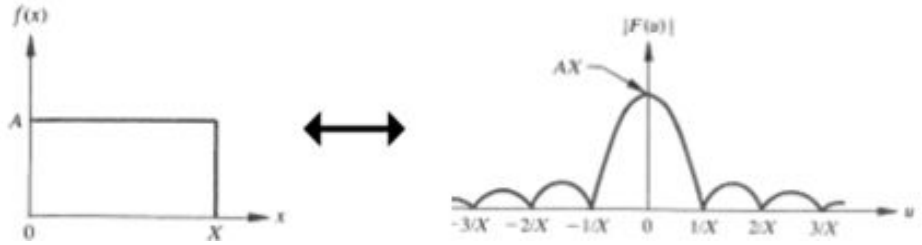

Figure 1: The rectangular function (left) and its sinc frequency domain counterpart (right).

We can prove this relationship by letting f(x) be the rectangular function, as shown in Fig. 1, and applying the definition of the Fourier transform from (Eq 1). We can also make use of the fact that f(x) = A in the range [0 X] and 0 everywhere else.

$$\mathcal{F}\{f(x)\} = \int_{-\infty}^{\infty} f(x)\mathrm{e}^{-\mathrm{j}2\pi\mathrm{u}x}\,\mathrm{d}x = \int_{0}^{X} A\mathrm{e}^{-\mathrm{j}2\pi\mathrm{u}x}\mathrm{d}x = \mathrm{A}\left[\frac{\mathrm{e}^{-\mathrm{j}2\pi\mathrm{u}x}}{-\mathrm{j}2\pi\mathrm{u}}\right]_{0}^{\mathrm{X}} = \frac{\mathrm{A}}{-\mathrm{j}2\pi\mathrm{u}}\left[\mathrm{e}^{-\mathrm{j}2\pi\mathrm{u}\mathrm{X}} - 1\right]$$

Now we can factor the right hand side and apply (Eq 6c).

$$\frac{-\mathrm{A}}{\mathrm{j}2\pi\mathrm{u}}\left[-1 + \mathrm{e}^{-\mathrm{j}2\pi\mathrm{u}\mathrm{X}}\right] = \frac{Ae^{-j\pi\mathrm{u}\mathrm{X}}}{\pi\mathrm{u}}\left[\frac{1}{2j}\left(e^{j\pi\mathrm{u}\mathrm{X}} - e^{-j\pi\mathrm{u}\mathrm{X}}\right)\right] = \frac{Ae^{-j\pi\mathrm{u}\mathrm{X}}}{\pi\mathrm{u}}\sin(\pi\mathrm{u}\mathrm{X}) = AXe^{-j\pi\mathrm{u}\mathrm{X}}\,\frac{\sin(\pi uX)}{\pi\mathrm{u}\mathrm{X}}$$

Finally we can expand the complex exponential using (Eq 6a) and find the magnitude of the Fourier Transform using (Eq 7).

$$\left|AXe^{-j\pi\mathrm{u}\mathrm{X}}\,\frac{\sin(\pi uX)}{\pi\mathrm{u}\mathrm{X}}\right| = AX\frac{\left|\sin(\pi\mathrm{u}\mathrm{X})\right|}{\pi\mathrm{u}\mathrm{X}}\left|\cos(\pi\mathrm{u}\mathrm{X}) - j\sin(\pi\mathrm{u}\mathrm{X})\right| = AX\frac{\left|\sin(\pi\mathrm{u}\mathrm{X})\right|}{\pi\mathrm{u}\mathrm{X}}$$

Thus, we have shown that the magnitude of the Fourier Transform of the rectangular function is the absolute value of the sinc function.

$$\left|\mathcal{F}\{\mathrm{rect}(\mathrm{x})\}\right| = AX\frac{\left|\sin(\pi\mathrm{u}\mathrm{X})\right|}{\pi\mathrm{u}\mathrm{X}} = AX\left|\mathrm{sinc}(\mathrm{ux})\right| \quad \text{(Eq 12)}$$

(2) The 2-Dimensional Fourier Transform & Its Discrete Implementation

The continuous 1-dimensional Fourier transform, and thus the 1-D DFT described earlier, can be logically extended to the 2-dimensional case. We can see this in (Eq 13) and (Eq 14), which show the equations for the 2-D DFT and its inverse for a discrete function f(x,y) in the time domain and its counterpart F(u,v) in the frequency domain. Note that these equations assume an NxN sampling for the sake of simplicity and further simplification.

$$\text{Forward DFT: } F(u,v) = \frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)e^{-\frac{j2\pi(ux+vy)}{N}} \quad \text{for } u,v = 0,\ldots,N-1 \quad \text{(Eq 13)}$$

$$\text{Inverse DFT: } f(x,y) = \frac{1}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1} F(u,v)e^{\frac{j2\pi(ux+vy)}{N}} \quad \text{for } x,y = 0,\ldots,N-1 \quad \text{(Eq 14)}$$

While the above equations compute the DFT, they are very computationally expensive. To solve this, we can separate the kernel of the forward transformation as shown in (Eq 15).

$$e^{-\frac{j2\pi(ux+vy)}{N}} = e^{-j2\pi\left(\frac{ux}{N}\right)} e^{-j2\pi\left(\frac{vy}{N}\right)} \quad \text{(Eq 15)}$$

Then, we can substitute the kernel separation into (Eq 13) and rearrange the terms.

$$F(u,v) = \frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)e^{-\frac{j2\pi ux}{N}} e^{-\frac{j2\pi vy}{N}} = \frac{1}{N}\sum_{x=0}^{N-1} e^{-\frac{j2\pi ux}{N}} \sum_{y=0}^{N-1} f(x,y)\, e^{-\frac{j2\pi vy}{N}}$$

Now, if we let F(x,v) be N times the 1-D DFT of f(x,y) with respect to the rows, it becomes clear that the 2-D DFT is nothing more an 1-D DFT with respect to the function F(x,v).

$$\text{Let } F(x,v) = N\left[\frac{1}{N}\sum_{y=0}^{N-1} f(x,y)e^{-\frac{j\pi vy}{N}}\right] = \sum_{y=0}^{N-1} f(x,y)\, e^{-\frac{j2\pi vy}{N}}$$

$$\text{Then } F(u,v) = \frac{1}{N}\sum_{x=0}^{N-1} F(x,v)\, e^{-\frac{j2\pi ux}{N}}$$

In essence, the 2-D DFT is a set of 1-D DFT transformations with respect to the rows followed by another set of 1-D DFT transformations with respect to the columns. Notice that this allows us to use the FFT algorithm described earlier in part (1) for even more computational efficiency. The only difference is that we have to compute two successive sets of FFT computations to get the result.

We can do a similar kernel separation in the inverse 2-D DFT to see that it is also nothing more than two successive sets of 1-D DFT inverse transformations.

(2a,b,c) Experiment 2, Parts A, B, & C

As described earlier, the 2-D DFT is calculated by computing the 1-D DFT with respect to F(*x,v*). However, when doing this, we must taking into account the periodic nature of the DFT. Just like in the case of the 1-D DFT, we must alter the original function in order the center the spectrum and fully visualize one period. (Eq 16) below gives the 2-D DFT property that we can use to accomplish this. The equation mirrors (Eq 12) since the 2-D DFT is nothing more than an extension of the 1-D DFT.

$$f(x,y)(-1)^{x+y} \Leftrightarrow F\left(u-\frac{M}{2}, v-\frac{N}{2}\right)$$
$$f\left(x-\frac{M}{2}, y-\frac{N}{2}\right) \Leftrightarrow F(u,v)(-1)^{x+y} \quad \text{(Eq 16)}$$

Using (Eq 16) allows us to shift the center of the frequency domain signal from the edges to the center of the spectrum. Notice that this makes a checkerboard look in terms of alternating the values. In comparison, the 1-dimensional DFT shifting only requires taking the negative of every other sample.

Another minor issue that we must take into account with the 2-D DFT is the large range of frequencies that it tends to produce. Because of this, we often transform the spectrum when visualizing it as an image. For this project, we only consider the approach given in (Eq 17) below. Note that this is used in Fig. 10, Fig. 11, & Fig. 12 when showing the spectrums.

$$\text{Displayed Value} = \log_{10}[1+F(u,v)] \quad \text{(Eq 17)}$$

Implementation & Results

(1) Experiment 1 Implementation Notes

For experiment 1, the given FFT implementation was used in all three parts to compute the results. To make things easier, a vector of floats was used to store the signals before passing the required addresses to the given *fft()* function for both the forward and inverse computations. To be efficient, the address of the value directly before the first element in the vector was passed to the function at each call. This was to avoid the unnecessary case of having an extra unused zero index.

For each of the three parts of the experiment, a helper function was defined to generate the required signal. For part A, the signal was hard coded since it only consisted of 4 samples. For part B, a generator was defined with the given cosine wave and sampled 128 times. For part C, the "Rect_128.dat" file was read in to obtain all the values. In all the parts, the imaginary section of the signal was set to zero. Thus, the signal vector had a size of 8 for part A and a size of 256 for part B and C.

To compute the forward DFT, the given function was used with -1 as the sign flag. Afterwards, the entire signal was divided by the number of samples as required by (Eq 3). For parts B and C (Eq 11) was used to obtain a full period in the result. Specifically, every second real and imaginary sample was inverted starting at the second sample. For part A, the inverse DFT was computed by using the given *fft()* function with +1 as the sign flag.

To make everything easier, various helper functions were defined to display and save the signal values. This was very helpful in verifying the computations and creating the final figures in

this report. The implementation and usage of these functions can be seen in the *Program Listings* section.

(1a) Experiment 1, Part A

The result of Part A is shown below in Fig. 2.

```
PART A||||||||||||||||||||
Original Signal (Real + j*Imaginary):  [ 2+j*0  3+j*0  4+j*0  4+j*0 ]
FFT of Signal (Real + j*Imaginary):  [ 3.25+j*0  -0.5+j*0.25  -0.25+j*0  -0.5+j*-0.25 ]
FFT of Signal (Magnitude):  [ 3.25  0.559017  0.25  0.559017 ]
Original after inverse (Real + j*Imaginary):  [ 2+j*0  3+j*-3.33067e-16  4+j*0  4+j*3.33067e-16 ]
```

Figure 2: The terminal output of the implementation for experiment 1, part A. It shows the result after computing the FFT and inverse FFT on the given signal. It was obtained by calling the *partA()* function defined in the *Program Listings* section.

(1b) Experiment 1, Part B

The results of Part B are shown in Fig. 3 through Fig. 5.

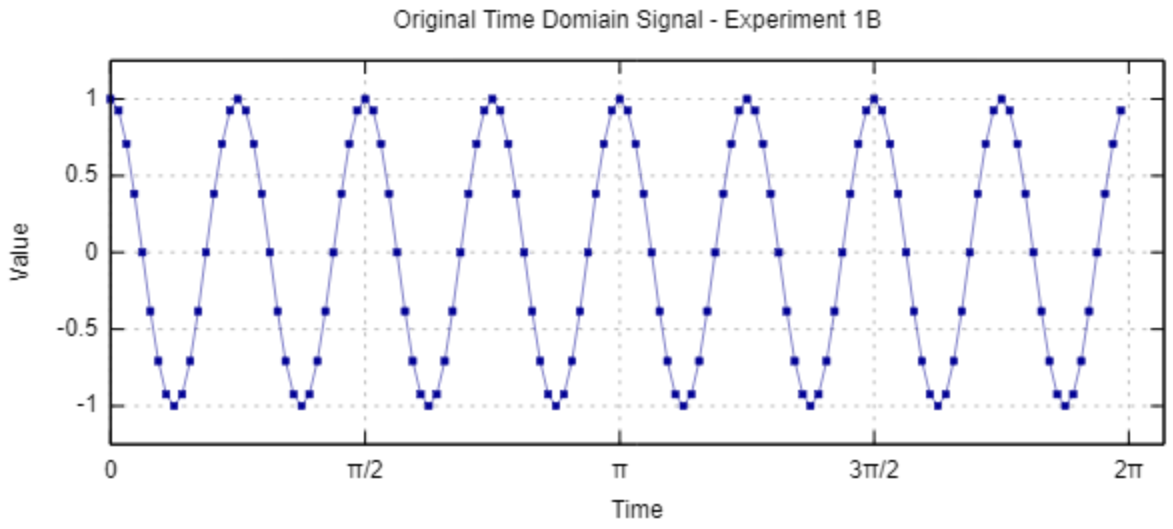

Figure 3: The sampling of the given cosine function for experiment 1, part B. There are 128 samples and 8 periods as required.

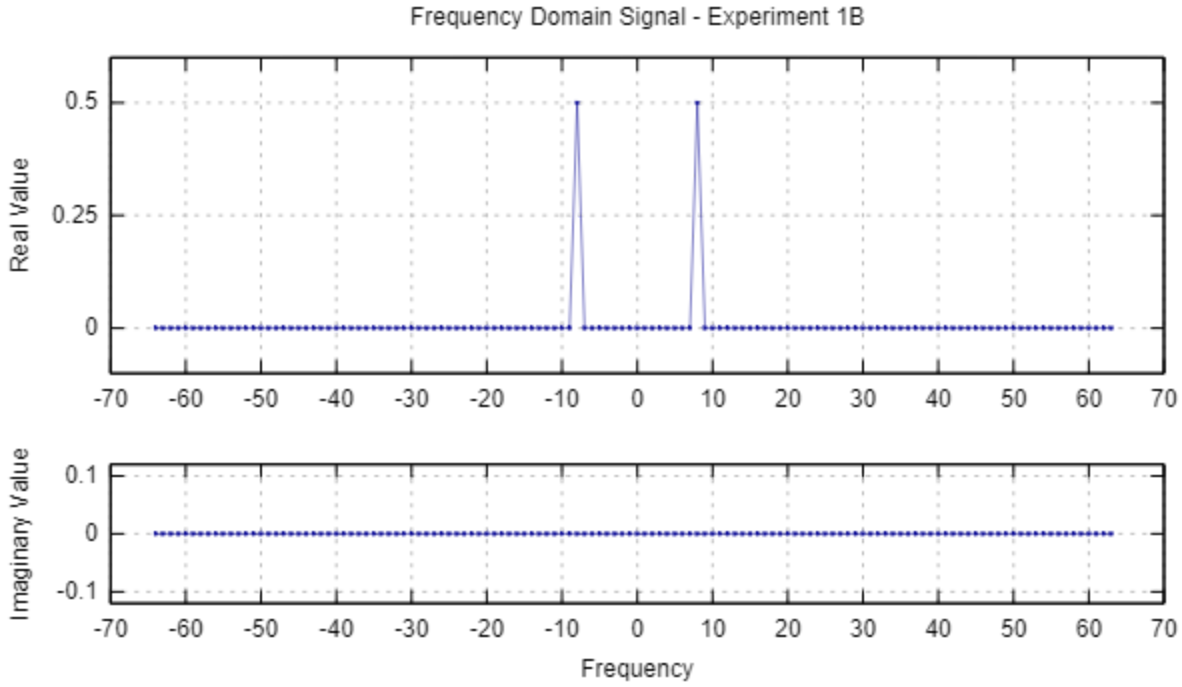

Figure 4: The DFT (real and imaginary parts) of the signal shown in Fig 3.

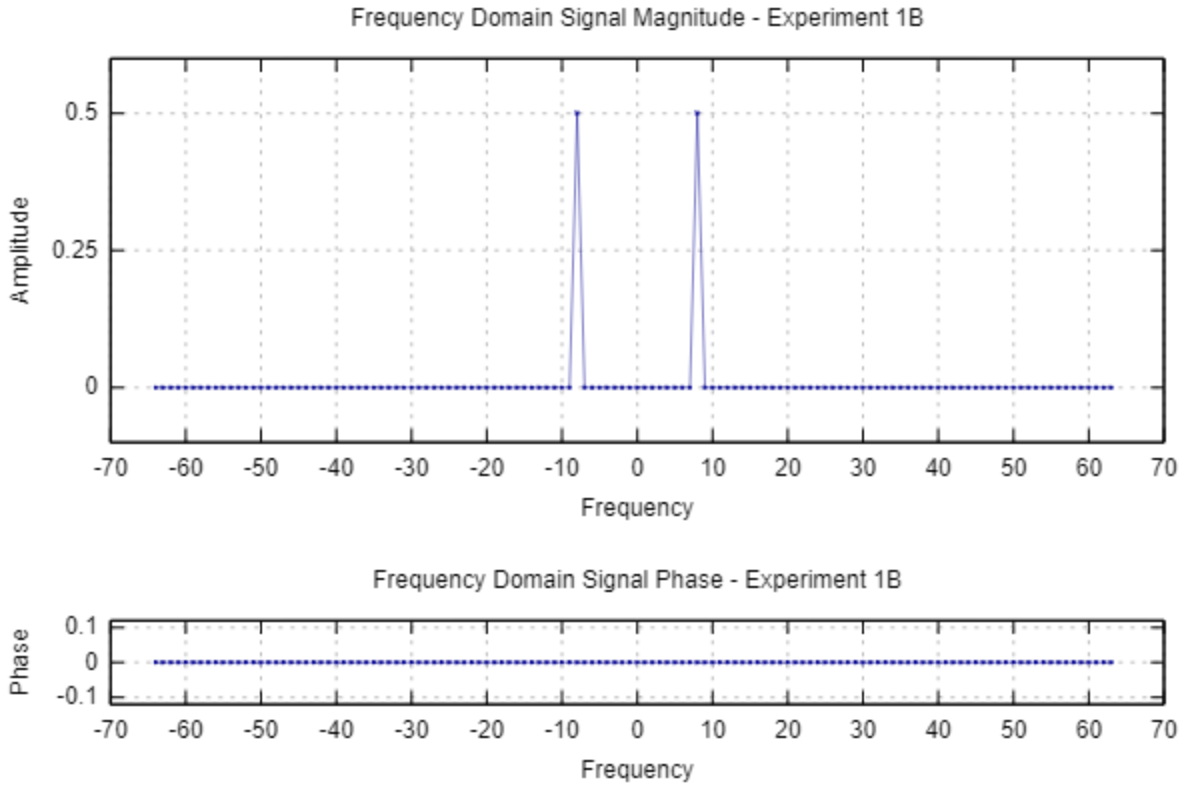

Figure 5: The magnitude and phase of the DFT signal shown in Fig 4.

(1c) Experiment 1, Part C

The results of Part C are shown in Fig. 6 through Fig. 8.

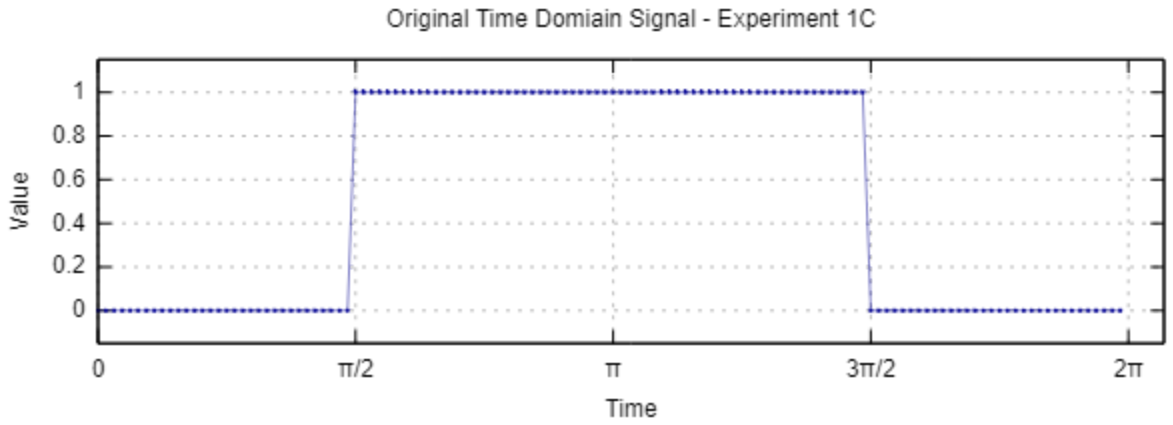

Figure 6: The 128 point sampling of the rectangular function from the "Rect_128.dat" file.

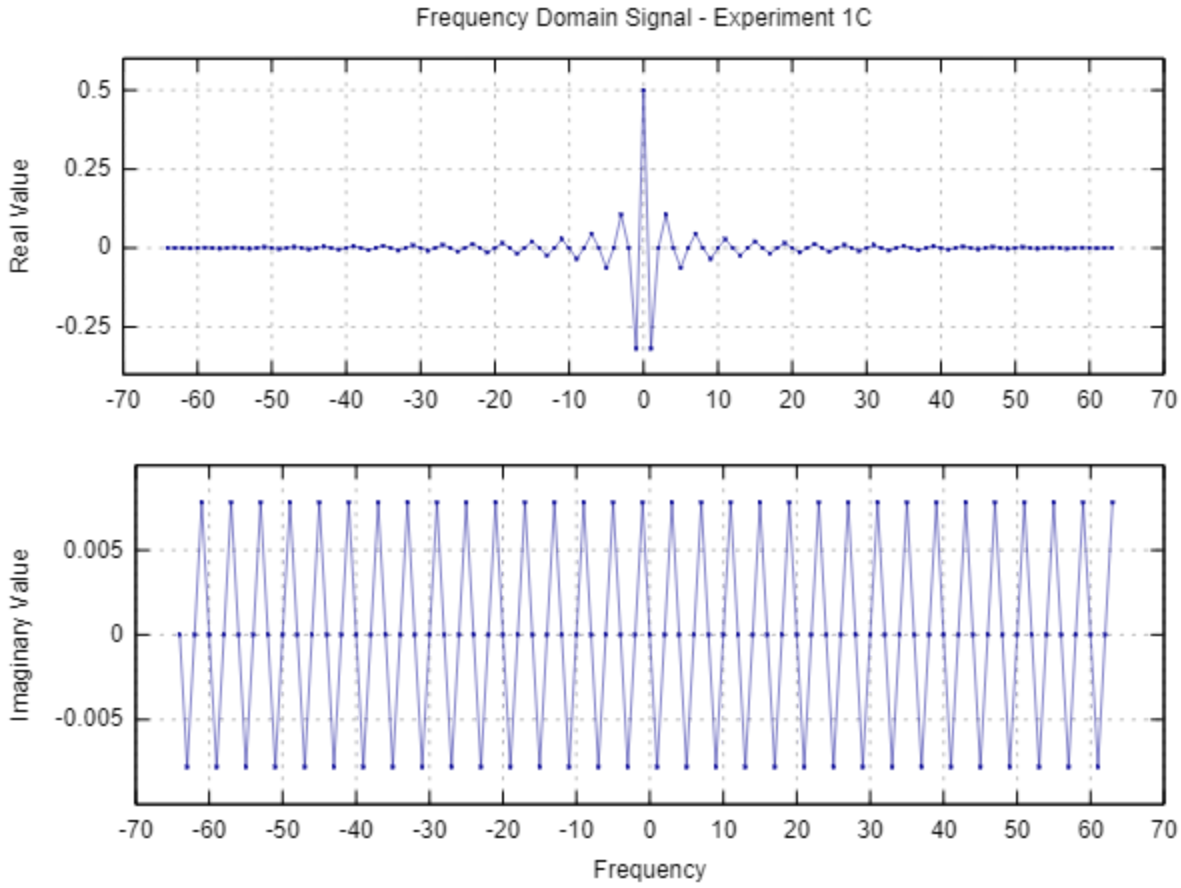

Figure 7: The DFT (real and imaginary parts) of the signal shown in Fig 6.

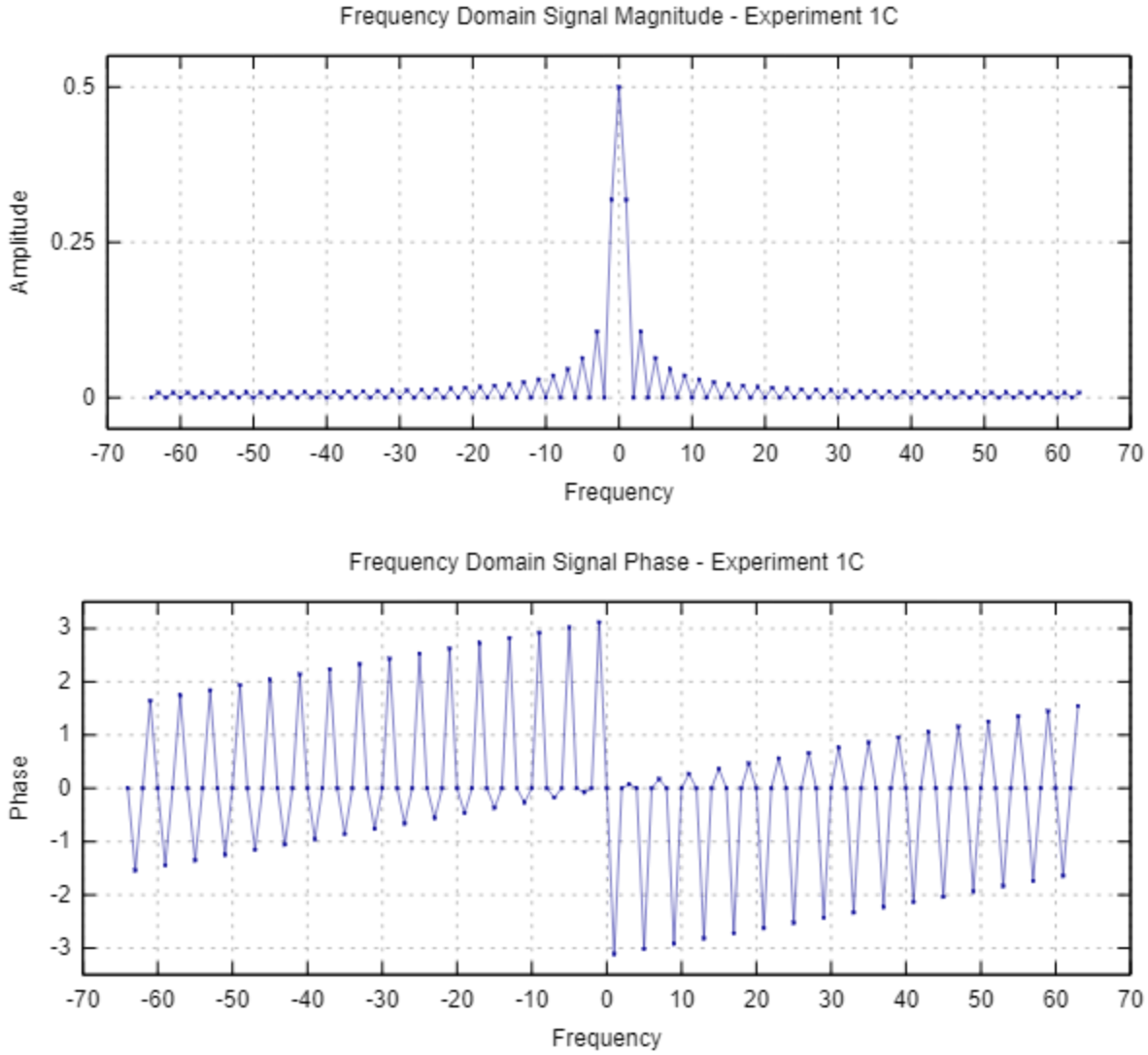

Figure 8: The magnitude and phase of the DFT signal shown in Fig 7.

Implementation of 2-D fft function

At a high level, the required *fft2D()* function was implemented using the separability property shown in (Eq 15). This property, as discussed in section (2) of the *Technical Discussion* section, allows us to compute the 2-D FFT by applying the 1-D FFT on the rows followed by the columns. So, the majority of this function had to deal with iterating over first the rows and then

the columns and computing the 1-D FFT with each subset of the data. Furthermore, the function simply deferred the 1-D FFT computations to the given *fft()* function. However, in order to do this correctly, the image data had to copied and moved into a format usable by the given 1-D implementation. This could not be avoided because the *fft2D()* function requires the real and imaginary part of the signal/image to be passed in separately while the given *fft()* function requires it to be all in one continuous array with alternating real and imaginary parts. In order to solve this issue, a temporary vector of floats, called *tempArr*, was defined. Then, *tempArr* was iteratively populated with the data of each row and passed to the *fft()* function. Afterwards, each transformed row was copied back into the original data. Finally, the same process was done again over all the columns. Here, the returned data from the *fft()* function was scaled by a factor of $1/N$ as required by (Eq 13) and (Eq 14) before being copied back. In order to make all these data movements easier, a few lambda helper functions were defined to get and set the real and imaginary parts of the original data by row and column.

Notice that since this 2D implementation of the FFT defers all the computations to the 1-D FFT, it doesn't change based on whether the forward or inverse transformation is being computed. Instead the forward/inverse flag is simply passed down to the 1-D FFT function. The full code of this function can be seen in the "2-Dimensional FFT Function" section of the *Program Listings.*

(2) Experiment 2 Implementation Notes

For experiment 2, the 2D FFT implementation was tested on the 2x2 signal $f(x,y) = [\{0\ 1\}\ \{1\ 0\}]$ before being put to use. After running through the 2-D function with the -1 modifier to

signal a forward transformation, the signal was run through the 2-D function again with the 1 modifier to signal an inverse transformation. Since the processed signal was the same as the original, we concluded that the function was working as intended. The test that we conducted can be seen in the figure below.

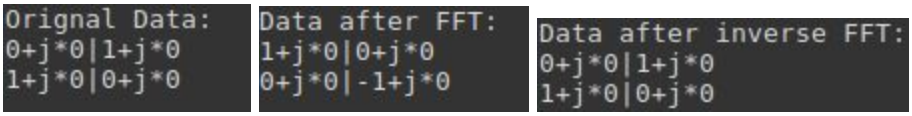

Figure 9: The results of the forward and inverse transformation on a 2x2 test signal

All three parts of experiment 2 were the same, with the exception of the image used. For part A, a 512x512 image was created and filled with black (0 value). Afterwards, a 32x32 square in the middle of the image was filled with white (255 value). The square was filled in a loop that took the center of the image by dividing 512 in half and setting the lower and left boundaries by subtracting 16 (32 / 2) and setting the upper and right boundaries by adding 16. For parts B and C, the boundaries were found by subtracting and adding 32 (64 / 2) and 64 (128 / 2) respectively. The images were then run through the 2-D FFT with and without shifting the center of frequency.

(2a) Experiment 2, Part A

The results of the experiment 2, part A is shown in Fig. 10 below.

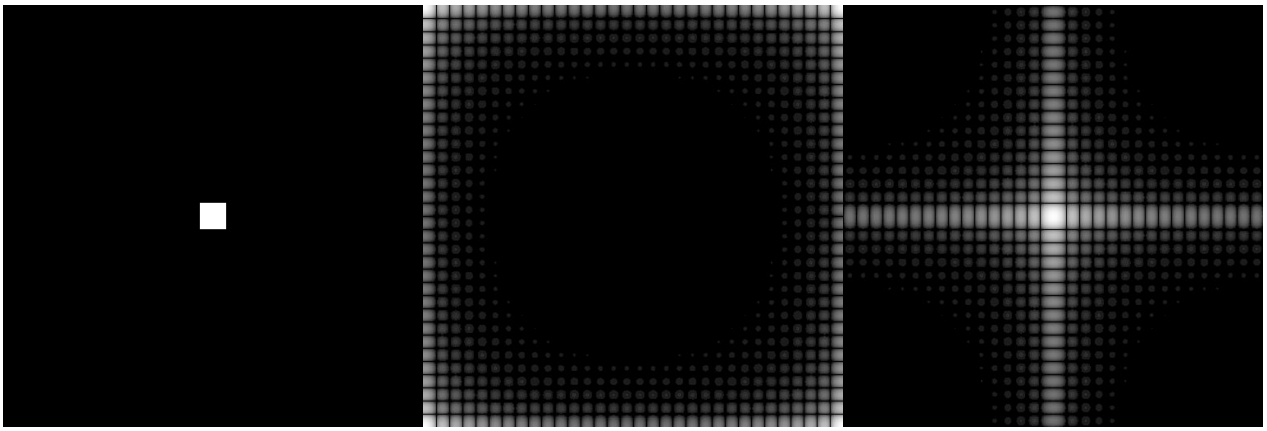

Figure 10: The image with a 32x32 white square in the middle. From left to right: original image, unshifted magnitude, shifted magnitude

(2b) Experiment 2, Part B

The results of the experiment 2, part B is shown in Fig. 11 below.

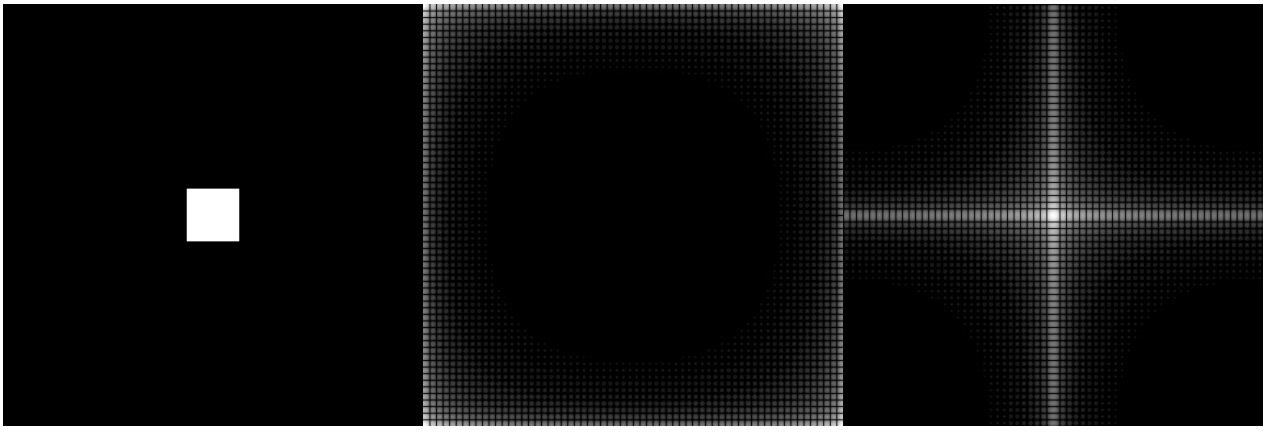

Figure 11: The image with a 64x64 white square in the middle. From left to right: original image, unshifted magnitude, shifted magnitude.

(2c) Experiment 2, Part C

The results of the experiment 2, part C is shown in Fig. 11 below.

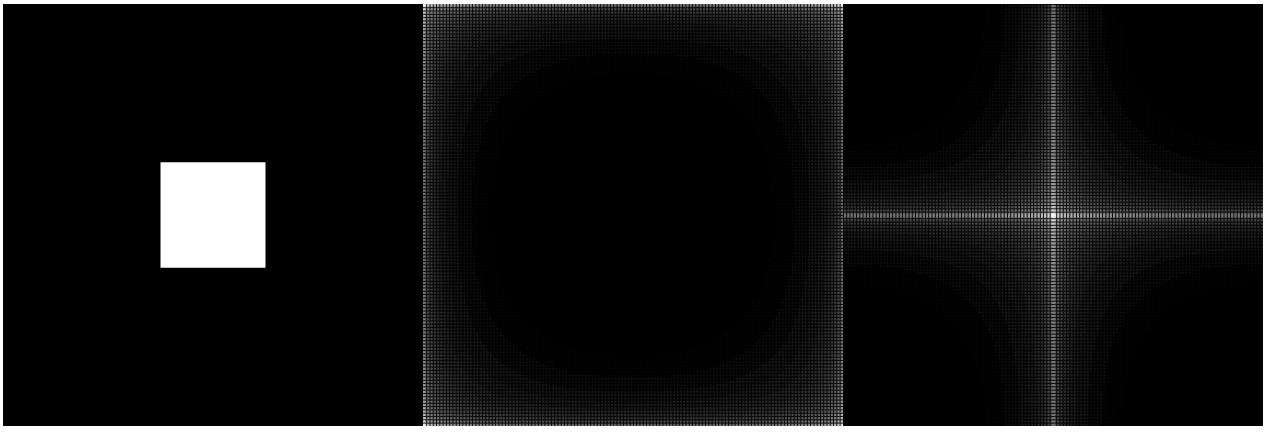

Figure 12: The image with a 128x128 white square in the middle. From left to right: original image, unshifted magnitude, shifted magnitude

Discussion

(1a) Experiment 1, Part A

In part A of experiment 1, the given *fft()* function was verified to work correctly. As shown in Fig. 2, the algorithm was successful in calculating the DFT and inverse DFT for a signal defined by [2 3 4 4].

For the forward DFT, the results matched perfectly with the expected results derived in section (1a) of the *Technical Discussion*. The real, imaginary, and magnitude parts were correct at all 4 data points.

For the inverse DFT the results were once again as expected. The original signal was obtained from the DFT with only a few minor rounding errors. The errors, which appear in the imaginary parts of the second and third data points, were mostly likely due to rounding errors in the floating point math being done by the FFT algorithm. Even so, the error was on the magnitude of 10^{-16} and would probably make no difference for any real world applications.

(1b) Experiment 1, Part B

In part B of experiment 1, the DFT was used to verify (Eq 9) . From the sampling of the cosine function in Fig. 3, it is clear that there are 8 periods. Thus, because of (Eq 9), it is expected that the Fourier transform will produce a spike and -8 and 8 in the real part and contain no imaginary part. This matches perfectly with the results shown in Fig. 4. Here, it is important to note the DFT signal plots were generated by applying (Eq 11) before any transformations so that a full period could be viewed. Additionally, the origin was shifted to be the center of the period

for visualization purposes. This visualization process was used for all the frequency domain graphs in experiment 1.

The magnitude and phase of the DFT, shown in Fig. 5, do not give any new information in this specific case. Since the imaginary part is all zero, the magnitude is the just the real part. Moreover, the phase, while technically undefined when the real part is zero by its definition in (Eq 8), is simply zero because there is no imaginary part in the DFT.

(1c) Experiment 1, Part C

In part C of experiment 1, the DFT was used to find the Fourier transform of the rectangular function. In Fig. 6, we can see that the data points in the given "Rect_128.dat" file are indeed a good sampling of the rectangular function.

From section (1c) of the *Technical Discussion* we know the following.

$$\mathcal{F}\{\text{rectangular function}\} = AX\text{sinc}(\pi uX)e^{-j\pi uX}$$

Using this, we can expand the complex exponential into its real and imaginary parts using (Eq 6a).

$$AX\text{sinc}(\pi uX)e^{-j\pi ux} = AX\text{sinc}(\pi uX)\cos(\pi ux) - jAX\text{sinc}(\pi uX)\sin(\pi ux)$$

Now we can use that fact that *u* only takes integer values to compare this with the results obtained in Fig. 7.

For the real part, we can see that everything more or less matches up. According to the above equation, we should have a sinc function being multiplied by an alternating series of 1 and -1. However, since the -1 weights correspond to points where the function is already 0, we essentially get a sinc function, as shown in the top of Fig. 7. The imaginary part, on the other

hand, is not as perfect. According to our equation, the imaginary part should be zero since $\sin(\pi u x)$ for an integer u is zero. However, the bottom of Fig. 7 clearly shows a pattern. But, since the values of the imaginary part are so close to zero, the pattern was probably obtained from the imprecise nature of the floating point math required for the computation of the DFT.

In Fig. 8, we can see the magnitude and phase of the DFT. The magnitude appears very much like the absolute value of the sinc function, as predicted by (Eq 12). Moreover, we can see that the peak occurs at AX = (1)(0.5) = 0.5. So, everything seems to line up with the expected results except for a few minor errors from the imprecise nature of floating point math.

(2a) Experiment 2, Part A

Before starting part A of experiment 2, the 2-D FFT function was verified to work correctly. This is shown in Fig. 9. Once this was down, the 2-D DFT of a 2-D rectangular function was computed. This resulted in a white cross shape after the magnitude was centered in the frequency domain. As shown in the right of Fig.10, the rows and columns in the center are very white, and the brightness decreases as the distance from the center increases. This shows that the original image primarily consists of low frequencies. In the middle of Fig. 10, we can see that the shifted results are the same as the unshifted results if the unshifted image is translated halfway horizontally and vertically. This is as expected and verifies that everything from (Eq 16) worked correctly. Also, for verification, the inverse function was used to get the original image back.

(2b) Experiment 2, Part B

In part B of experiment 2, the same process described in (1a) was done, except with a larger white square at the center of the image. The results are very similar, but it seems as if the individual blocks of the image are smaller, as seen in Fig. 11. In the unshifted image, the black circle showed about 3 layers of varying darkness. This is in contrast to part A, where only 2 layers are present. Both of these differences can be attributed to a larger portion of the image being white. Essentially, the larger rectangle seems to decrease the number off high frequencies in the image.

(2c) Experiment 2, Part C

In part C of experiment 2, the same process described in (1b) was done, except with an even larger white square at the center of the image. The results are a continuation of the trend established earlier. There are increasingly smaller blocks of white compared to part B and A, as seen in Fig. 12. The unshifted image shows 4 distinct black rings in comparison to part B's 3 rings. The white radiating from the center column and row in the shifted image is much more defined. This shows the prevalence of low frequencies in white images. Although not shown, running the altered image through the inverse function created the original image, verifying that the 2D-DFT worked properly.

Program Listings

(1) Experiment 1 Helper Functions

The following helper functions were defined and used throughout experiment 1.

```
//scales all values in a signal by a given factor
void scaleSignal(std::vector<float> & signal,
                 const float scaleFactor){
   for (float & val : signal)
      val *= scaleFactor;
}

//function to print signal. Prints 'real + j*imaginary'
void printSignal(const std::vector<float> & signal,
                 const std::string & startLable = "signal: "){
   std::cout << std::endl << startLable << " [";
   for (unsigned int i = 0; i < signal.size(); i+=2)
      std::cout << ' ' << signal[i] << "+j*" << signal[i+1] << ' ';
   std::cout << ']' << std::endl;
}

//function to print magnitude of signal.
void printSignalMagnitude(const std::vector<float> & signal,
                          const std::string & startLable = "mag: "){
   std::cout << std::endl << startLable << " [";
   for (unsigned int i = 0; i < signal.size(); i += 2)
      std::cout << ' ' << std::sqrt(signal[i] * signal[i] +
                          signal[i + 1] * signal[i + 1]) << ' ';
   std::cout << ']' << std::endl;
}

//function to save signal to external file
void saveSignal(const std::vector<float> & signal,
                const std::string & fileName,
                bool printReal = true, bool printImaginary = true) {
   std::ofstream outputFile(fileName);
```

```
    for (unsigned int i = 0; i < signal.size(); i += 2) {
        if (printReal) //save real part if necessary
            outputFile << signal[i] << std::endl;
        if (printImaginary) //save imaginary part if necessary
            outputFile << signal[i + 1] << std::endl;
    }
    outputFile.close();
}

//function to save magnitude or phase of signal to external file
void saveSignalCharacterisitic(const std::vector<float> & signal,
                               const std::string & fileName,
                               bool saveMagnitude = true){
    std::ofstream outputFile(fileName);

    for (unsigned int i = 0; i < signal.size(); i += 2)
        outputFile << ((saveMagnitude) ?
                       std::sqrt(signal[i] * signal[i] +
                                 signal[i + 1] * signal[i + 1]) :
                       std::atan2(signal[i + 1], signal[i]))
                   << std::endl;

    outputFile.close();
}
```

(1a) Experiment 1, Part A

The following two functions implemented part A with the help of the already mentioned helper functions.

```
//function to implement part A
void partA(void){
    //generate required dataset and print it for verification
    std::vector<float> signal;
    generateSignalPartA(signal);
    printSignal(signal, "Original Signal (Real + j*Imaginary): ");
```

```
    //compute forward fft with help of given function
    fft(&signal[0] - 1, signal.size() / 2, -1);
    scaleSignal(signal, float(1)/(signal.size()/2)); //scale by 1/N

    //print signal's fft result for verification
    printSignal(signal, "FFT of Signal (Real + j*Imaginary): ");
    printSignalMagnitude(signal, "FFT of Signal (Magnitude): ");

    //compute inverse fft with given function
    fft(&signal[0] - 1, signal.size() / 2, 1);

    printSignal(signal, "Original after inverse: "); //print result
}

//generates required signal for part A (with 0 for imaginary part)
void generateSignalPartA(std::vector<float> & signal){
    signal.resize(8);
    //real part of data
    signal[0] = 2;
    signal[2] = 3;
    signal[4] = 4;
    signal[6] = 4;
    //imaginary part of data - all zero
    signal[1] = signal[3] = signal[5] = signal[7] = 0;
}
```

(1b) Experiment 1, Part B

The following two functions implemented part B with the help of the already mentioned helper functions.

```
//function to implement part B
void partB(void) {
    //generate required dataset
    std::vector<float> signal;
    generateSignalPartB(signal);
```

```
   //save signal for verification purposes - only save real part
   saveSignal(signal, "Ex1PtB_original_real.dat", true, false);

   for (unsigned int i = 2; i < signal.size(); i += 4)
      signal[i] *= -1; //invert every other sample see full period

   //compute forward fft with help of given function
   fft(&signal[0] - 1, signal.size() / 2, -1);
   scaleSignal(signal, float(1)/(signal.size()/2)); //scale by 1/N

   //save real, imaginary, magnitude, & phase part for verification
   saveSignal(signal, "Ex1PtB_fft_real.dat", true, false);
   saveSignal(signal, "Ex1PtB_fft_imaginary.dat", false, true);
   saveSignalCharacterisitic(signal, "Ex1PtB_fft_magnitude.dat");
   saveSignalCharacterisitic(signal, "Ex1PtB_fft_phase.dat", false);
}

//generates required signal for part B (with 0 for imaginary part)
void generateSignalPartB(std::vector<float> & signal) {
   const int N = 128, u = 8;
   signal.resize(N * 2);

   //generator function for signal
   auto signalGenerator = [&N, &u](int sampleNum) ->float {
      return std::cos(2 * M_PI * u * sampleNum / N);
   };

   unsigned int i = 0;
   for (int sample = 0; sample < N; ++sample) {
      signal[i++] = signalGenerator(sample); //add real part
      signal[i++] = 0; //add imaginary part
   }
}
```

(1c) Experiment 1, Part C

The following two functions implemented part C with the help of the already mentioned helper functions.

```
//function to implement part C
void partC(void){
   //generate required dataset and print it for verification
   std::vector<float> signal;
   generateSignalPartC(signal);

   //save signal for verification purposes - only save real part
   saveSignal(signal, "Ex1PtC_original_real.dat", true, false);

   for (unsigned int i = 2; i < signal.size(); i += 4)
      signal[i] *= -1; //invert every other sample see full period

   //compute forward fft with help of given function
   fft(&signal[0] - 1, signal.size() / 2, -1);
   scaleSignal(signal, float(1)/(signal.size()/2)); //scale by 1/N

   //save real, imaginary, magnitude, & phase for verification
   saveSignal(signal, "Ex1PtC_fft_real.dat", true, false);
   saveSignal(signal, "Ex1PtC_fft_imaginary.dat", false, true);
   saveSignalCharacterisitic(signal, "Ex1PtC_fft_magnitude.dat");
   saveSignalCharacterisitic(signal, "Ex1PtC_fft_phase.dat", false);
}

//generates required signal for part B(with 0 for imaginary part)
void generateSignalPartC(std::vector<float> & signal) {
   signal.clear();
   std::ifstream inputFile("images/PA03/Rect_128.dat");

   if (!inputFile.is_open()) {
      std::cerr << "Could not open input file for Part C!";
      Return;
   }
```

```
   //get each data value in file until the end.
   std::string tempLine;
   std::getline(inputFile, tempLine); //get first line
   while (!inputFile.eof()) {
      signal.push_back(std::stof(tempLine)); //real part from file
      signal.push_back(0); //set imaginary part to 0
      std::getline(inputFile, tempLine); //get next line
   }
}
```

2-Dimensional FFT Function

Below is the implementation of the required 2D FFT function. It uses the 1D FFT repeatedly to compute the results

```
/* Function to implement 2D fft via given 1D fft function.
 * Used to calculate the DFT of an NxM image with N rows and M cols.
 * N and M must be a a power of 2. Otherwise function will not work.
 * The data arrays must be 1 dimensional with a length of N*M
 *     ie: first row will be in the range [0,M-1].
 *         second row will be in the range [M,2M-1] and so on.
 * isign: use -1 for forward DFT and 1 for inverse DFT
 */
inline void fft2D(unsigned long N, unsigned long M,
                  float real_Fuv[], float imag_Fuv[], int isign) {
   //helpers to get and set values by row and column number
   auto getReal = [&M, real_Fuv](unsigned int r, unsigned int c)
                  ->float {return real_Fuv[r * M + c];};
   auto getImag = [&M, imag_Fuv](unsigned int r, unsigned int c)
                    ->float {return imag_Fuv[r * M + c];};
   auto setValue = [&M, real_Fuv, imag_Fuv](unsigned int r,
                      unsigned int c, float realVal, float imagVal)
                      ->void {
                          real_Fuv[r * M + c] = realVal;
                          imag_Fuv[r * M + c] = imagVal;};

   std::vector<float> tempArr(2 * M); //to hold data for 1-D fft
```

```
   //transform each row using 1D fft
   for (unsigned int r = 0; r < N; ++r) {
      //copy over real and imaginary part to temporary array
      unsigned int i = 0;
      for (unsigned int c = 0; c < M; ++c) {
         tempArr[i++] = getReal(r, c);
         tempArr[i++] = getImag(r, c);
      }
      //compute fft with help of given function
      fft(&tempArr[0] - 1, M, isign);

      //store result back in original arrays
      i = 0;
      for (unsigned int c = 0; c < M; ++c, i += 2)
         setValue(r, c, tempArr[i], tempArr[i + 1]);
   }

   //transform each column using 1-D fft
   tempArr.resize(2 * N);
   for (unsigned int c = 0; c < M; ++c) {
      //copy over real and imaginary part to temporary array
      unsigned int i = 0;
      for (unsigned int r = 0; r < N; ++r) {
         tempArr[i++] = getReal(r, c);
         tempArr[i++] = getImag(r, c);
      }
      //compute fft with help of given function. Normalize after
      fft(&tempArr[0] - 1, N, isign);
      for (float & tempVal : tempArr)
         tempVal /= N; //normalize by 1/N

      //store result back in original arrays
      i = 0;
      for (unsigned int r = 0; r < N; ++r, i += 2)
         setValue(r, c, tempArr[i], tempArr[i + 1]);
   }
}
```

(2) Experiment 2 Helper Functions

This function output the magnitudes after the 2D FFT in case the user wants to implement the image or graphs on the data themselves.

```
void outputMagnitude(const std::vector<float> & signal,
                     const std::string & fileName){
   std::ofstream outputFile(fileName);
   for(unsigned int i = 0; i < signal.size(); ++i)
       outputFile << signal[i] << std::endl;
   outputFile.close();
}
```

This function generated the original image with the white square in the center.

```
void generate_image(ImageType & image, int row, int col, int middle){
   // Set everything 0 initially
   for(int i = 0; i < row; ++i)
      for(int j = 0; j < col; ++j)
         image.setPixelVal(i, j, 0);

   // Set middle square to 255
   for(int i = (row/2)-(middle/2); i < (row/2)+(middle/2); ++i)
      for(int j = (col/2)-(middle/2); j < (col/2)+(middle/2); ++j)
         image.setPixelVal(i, j, 255);
}
```

The next helper function was basically the main code for the program. After taking in the original image and checking whether the user wants to shift the signal or not, the values are processed into a format useable for the 2D FFT helper function mentioned earlier. The resulting values are then rescaled to the normal 256 gray scale values and output to an image.

```
void FFT2D_Main(const std::string & dataFile,
                const std::string & imageFile,
                ImageType & image, bool shifted){

   int N, M, Q; // M = Columns, N = Rows, Q = Levels
   image.getImageInfo(N, M, Q);

   std::vector<float> imgDataReal, imgDataImag, imgDataCombined;

   // Obtain real values and set imaginary to 0
   int px_val;
   for(int i = 0; i < N; ++i)
      for(int j = 0; j < M; ++j){
         image.getPixelVal(i, j, px_val);
         imgDataReal.push_back(px_val);
         imgDataImag.push_back(0);
       }

   if(shifted){
      for(int i = 0; i < N; ++i){
         if(!(i % 2))
            for(int j = 0; j < M; j += 2)
               imgDataReal[i * M + j] *= -1;
         else
            for(int j = 1; j < M; j += 2)
               imgDataReal[i * M + j] *= -1;
       }
    }

   fft2D(N, M, &imgDataReal[0], &imgDataImag[0], -1); //Forward 2DFFT

   // Combines real and imaginary into one vector
   for(int i = 0; i < N * M; ++i){
      px_val = std::sqrt(imgDataReal[i] * imgDataReal[i] +
                         imgDataImag[i] * imgDataImag[i]);
      //use log transformation for visualization purposes
      imgDataCombined.push_back(log(1 + px_val));
    }
```

```
   // Output magnitude data to file for reference
   outputMagnitude(imgDataCombined, dataFile);

   ImageType FFT2D_Magnitude_Image(N, M, Q);
   double max = *max_element(imgDataCombined.begin(),
                             imgDataCombined.end());
   for(int i = 0; i < N; ++i)
      for(int j = 0; j < M; ++j)
         // Rescales to 256 levels for better viewing
         FFT2D_Magnitude_Image.setPixelVal(i, j,
                           (imgDataCombined[i * M + j]) / max * 255);

   writeImage(imageFile.c_str(), FFT2D_Magnitude_Image);
}
```

(2a) Experiment 2, Part A

The code below was used to run part A of experiment 2 in shifted and unshifted form.

```
// Part A, 32x32 white square in center
ImageType generated_image_32x32(512, 512, 255);
generate_image(generated_image_32x32, 512, 512, 32);
writeImage("images/PA03/Experiment2/partA_original.pgm",
           generated_image_32x32);

// Unshifted magnitude
FFT2D_Main("images/PA03/Experiment2/partA_unshifted_magnitude.dat",
           "images/PA03/Experiment2/partA_unshifted.pgm",
           generated_image_32x32, 0);

// Magnitude shifted to center of frequency domain
FFT2D_Main("images/PA03/Experiment2/partA_shifted_magnitude.dat",
           "images/PA03/Experiment2/partA_shifted.pgm",
           generated_image_32x32, 1);
```

(2b) Experiment 2, Part B

The code below was used to run part B of experiment 2 in shifted and unshifted form.

```
// Part B, 64x64 white square in center
ImageType generated_image_64x64(512, 512, 255);
generate_image(generated_image_64x64, 512, 512, 64);
writeImage("images/PA03/Experiment2/partB_original.pgm",
           generated_image_64x64);

// Unshifted magnitude
FFT2D_Main("images/PA03/Experiment2/partB_unshifted_magnitude.dat",
           "images/PA03/Experiment2/partB_unshifted.pgm",
           generated_image_64x64, 0);

// Magnitude shifted to center of frequency domain
FFT2D_Main("images/PA03/Experiment2/partB_shifted_magnitude.dat",
           "images/PA03/Experiment2/partB_shifted.pgm",
            generated_image_64x64, 1);
```

(2c) Experiment 2, Part C

The code below was used to run part C of experiment 2 in shifted and unshifted form.

```
// Part C, 128x128 white square in center
ImageType generated_image_128x128(512, 512, 255);
generate_image(generated_image_128x128, 512, 512, 128);
writeImage("images/PA03/Experiment2/partC_original.pgm",
           generated_image_128x128);

// Unshifted magnitude
FFT2D_Main("images/PA03/Experiment2/partC_unshifted_magnitude.dat",
           "images/PA03/Experiment2/partC_unshifted.pgm",
           generated_image_128x128, 0);

// Magnitude shifted to center of frequency domain
FFT2D_Main("images/PA03/Experiment2/partC_shifted_magnitude.dat",
           "images/PA03/Experiment2/partC_shifted.pgm",
           generated_image_128x128, 1);
```