CS 674
Assignment 3

Bryson Lingenfelter Nate Thom Christopher Lewis

Submitted: November 13, 2019
Due: November 13, 2019

0.1 Division of Work

0.1.1 Coding

Bryson: fft.c wrapper, 1D FFT
Chris: - 2D FFT, Experiment 2
Nate: - 2D FFT, Experiment 3

0.1.2 Writing

Bryson: Experiment 1, Experiment 2
Chris: - Experiment 2
Nate: - Experiment 3

1 Technical Discussion

1.1 1D Fourier Transform

The Fourier transform is a method for converting a signal into its constituent frequencies. It does this by representing the signal as a weighted sum of sine and cosine waves using the following transformation to convert between the spatial domain and the frequency domain:

$$\mathcal{F}(f(x)) = \int_{-\infty}^{\infty} f(x)e^{-j2\pi ux}dx$$

$$\mathcal{F}^{-1}(F(u)) = \int_{-\infty}^{\infty} F(u)e^{j2\pi ux}du$$

For discrete signals, this is calculated with the Discrete Fourier Transform (DFT).A sum is needed to approximate the continuous Fourier transform.

$$\mathcal{F}(f(x)) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)e^{\frac{-j2\pi ux}{N}}$$

$$\mathcal{F}^{-1}(F(u)) = \sum_{u=0}^{N-1} F(u)e^{\frac{j2\pi ux}{N}}$$

The runtime of DFT can be improved through divide-and-conquer. The DFT can be rewritten as the sum of two DFTs, one working over the even indexes and the other working over the odd indexes. This reduces the runtime of DFT from $O(N^2)$ to $O(NlogN)$. The result is called the Fast Fourier Transform (FFT). In the results section we test the FFT on several different signals.

1.2 Fourier Transform on Images

Using the Fourier Transform on an image requires that the transform is extended into two dimensions. The Fourier Transform in two dimensions is:

$$\mathcal{F}(f(x,y)) = F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-j2\pi(ux+vy)}dxdy$$

$$\mathcal{F}^{-1}(F(u,v)) = f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u,v)e^{j2\pi(ux+vy)}dudv$$

The two dimensional Fourier Transform still converts between the spatial domain and the frequency domain. However, it works by applying the original transform on two dimensions. The DFT can also be used in two dimensions, as described by this transformation:

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

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

The 2D DFT cannot be directly translated to the FFT. However, we can rewrite the DFT as the sum of 1D DFTs by taking advantage of the fact that the kernel is seperable; that is, $e^{-j2\pi\frac{ux+vy}{N}} = e^{-j2\pi\frac{ux}{N}}e^{-j2\pi\frac{vy}{N}}$. This allows us to rewrite the DFT as the following:

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

Which is the Fourier transform of the columns of $F(u,v)$, where $F(u,v)$ is the Fourier transform of the rows of $f(x,y)$. Because these are both 1D Fourier transforms, we can use the FFT as discussed earlier to compute both transforms. This reduces the time complexity of the transform to $N\ O(NlogN)$ row transforms plus $N\ O(NlogN)$ column transforms, for a total time complexity of $O(N^2logN)$.

For our experiments we apply the 2D FFT to images. We can then plot the magnitude of the Fourier transform to see the magnitude of the different frequencies in the image.

1.3 Magnitude and Phase Experiments

The reconstruction of images from the frequency domain with only phase or only magnitude gives significantly different results. It is clear that both components are important, however reconstruction with only phase provides results of higher quality. The reconstruction of an image with only magnitude can be achieved by calculating the magnitude at each point in the frequency domain. Next, we set the real part of each point to the calculated magnitude and the imaginary part to zero. The general case and the case for an image are shown below. Note: $x = a + jb$ where $j = \sqrt{-1}$.

$$|x| = \sqrt{a^2 + b^2}$$

$$image(i,j) = \sqrt{image(i,j)^2 + image(i,j+1)^2}$$

and

$$image(i,j+1) = 0$$

Alternatively, the reconstruction of an image with only phase can be achieved by setting magnitude to 1. We can do this by setting the real part of the frequency to $cos(\theta)$ and the imaginary part to $sin(\theta)$ where $\theta = tan^{-1}(imaginary/real)$. To show this works, we write each point in the inverse transform as a real component plus an imaginary component: $F(u,v) = a_{u,v} + jb_{u,v}$ (For neatness, we write this as $p = a + jb$). The phase of this point is $\phi(p) = tan^{-1}\frac{b}{a}$ and the magnitude is $|p| = \sqrt{a^2+b^2}$. The transformation described above gives

$$\theta = tan^{-1}(\frac{b}{a})$$

$$a' = cos(\theta)$$

$$b' = sin(\theta)$$

Using this, we can directly calculate the new magnitude and phase, $\phi'(p)$ and $|p|'$.

$$\begin{aligned}\phi'(p) &= tan^{-1}(\frac{b'}{a'}) \\ &= tan^{-1}(\frac{cos\theta}{sin\theta}) \\ &= tan^{-1}(tan(\theta)) \\ &= \theta \\ &= tan^{-1}(\frac{b}{a}) \\ &= \phi(p)\end{aligned} \tag{2}$$

This shows that $\phi'(p) = \phi(p)$, so the transformation does not have any effect on the phase of the point.

$$\begin{aligned}|p|' &= \sqrt{a'^2 + b'^2} \\ &= \sqrt{cos^2(\theta) + \sin^2(\theta)} \\ &= \sqrt{1} \\ &= 1\end{aligned} \tag{3}$$

This shows that the transformation sets the magnitude to 1. The transformation therefore removes magnitude information without affecting phase information.

2 Implementation Details

2.1 1D Fourier Transform

To avoid writing the entire project in C, we created a python wrapper for the `fft.c` file. This was done by converting the C code to take data as input on the command line and print the results of the transformation as output. A python class, stored in `fft.py`, has functions for calling this code as a subprocess to compute the Fourier transform results. The class takes care of adding zeros as the imaginary component of a signal in the spatial domain as well as setting the imaginary component of the inverse transformation to zero to get rid of parasitic error.

For the first experiments, we directly use this class to compute the Fourier transform on different lists. Three helper functions, `gen_cos_wave`, `center_transform`, and `plot_transform_1d` are used. The first generates a cosine wave signal using the `cos` function from python's `math` library, the second centers a the transformation in the spatial domain by setting $f(x) = f(x)(-1)^x$, and the third plots the data from a transformation using `matplotlib`.

2.2 Fourier Transform on Images

For the second experiment, we use the separability property of a two dimensional FFT to use the one dimensional FFT and apply it along every row and column of an image. This gives us a large array that is double the size of the original image. This is due to the fact that we store both complex and real numbers in the same array, and there are both a complex and real number for any given pixel in the original image.

We then want to get the magnitude of the FFT. This is done by taking the square of the real component and imaginary component and adding them together, and then taking the square root of the whole sum as shown below.

$$|\mathcal{F}(u)| = \sqrt{R^2(u) + I^2(u)}$$

After this, scaling the magnitude is very important. To make the magnitude plot more visible, we scale the magnitude of every pixel using a log transformation. This makes the difference between the highest values and normal values less extreme.

$$D(u,v) = clog(1 + |\mathcal{F}(u,v)|$$

Where c is the highest value in the image.

Often, the image is considered to be off center and you can't see every aspect of it. This can then be rectified by applying the translation property to the FFT. This translation property for moving the transform into the center is:

$$f(x,y)(-1)^{x+y} <==> F(u - N/2, v - N/2)$$

This means that in code we can center the transform by multiplying each pixel by $(-1)^{x+y}$ in the spatial domain. The code is tested on three image. The images are a 32x32 square, a 64x64 square, and a 128x128 square, where the square is white and the background is black. We will talk about the results in section 3.2.

2.3 Magnitude and Phase Experiments

The purpose of this experiment is to showcase the importance of phase and magnitude in the reconstruction of an image from the frequency domain to the spatial domain. We use the tools developed in our second experiment to calculate the forward and inverse 2D Fourier transform of an image. It is important to remember that our implementation of the Fourier transform stores the real and imaginary parts adjacent to each other in the list. For example:

$$F(u,v)_{real} = list(i,j)$$

and

$$F(u,v)_{imaginary} = list(i,j+1)$$

For the first part of the experiment we remove phase and reconstruct the image. This is accomplished by iterating over each real and imaginary pair in our list and calculating magnitude with:

$$\sqrt{list(i,j)^2 + list(i,j+1)^2}$$

. We set the real part of the image (i.e. $list(i,j)$ to the calculated magnitude and the imaginary part (i.e. $list(i,j+1)$ to zero. Finally, we take the inverse transform of our updated list.

In the second portion of this experiment we set magnitude equal to one. This is accomplished by iterating over each real/imaginary pair and calculating θ, $sin(\theta)$, and $cos(\theta)$. θ is calculated as $tan^{-1}(imaginary/real)$. Next, the real part of the image is set to $cos(\theta)$ and the imaginary part is set to $sin(\theta)$. Last, we calculate the inverse Fourier transform to examine the resultant image.

3 Results and Discussion

3.1 1D Fourier Transform

We first test the Fourier transform using a toy dataset consisting of four discrete points; $f = [2,3,4,4]$. Figure 1 shows the real and imaginary components of the transformation, as well as the magnitude.

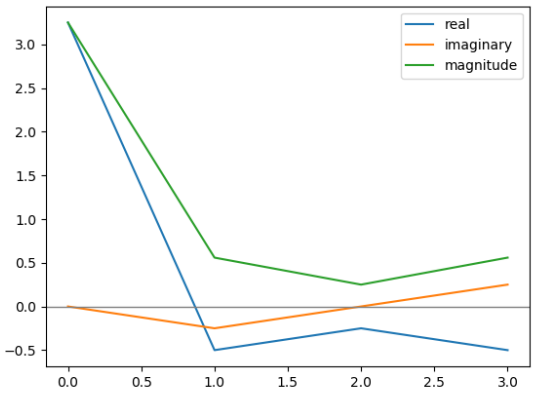

Figure 1: Real and imaginary components of the Fourier transform of f, as well as magnitude at each point.

The FFT was then tested on a cosine wave, with a frequency of 8 and an amplitude of 1. We take 128 samples of this wave from 0 to 128. In theory, the Fourier transform of a cosine function should result in two impulses at the positive and negative frequency of the cosine wave: $\mathcal{F}(cos(2\pi\mu_0 x)) = \frac{1}{2}(\delta(\mu-\mu_0)+\delta(\mu+\mu_0))$. Our results match up with this expectation: the transform has two points, both at a height of $\frac{1}{2}$, at -8 and 8. It is difficult to tell from the plot, but the imaginary value is 0 at all points, which is as expected because the left side of expression above contains no imaginary component. The results are plotted in Figure 2

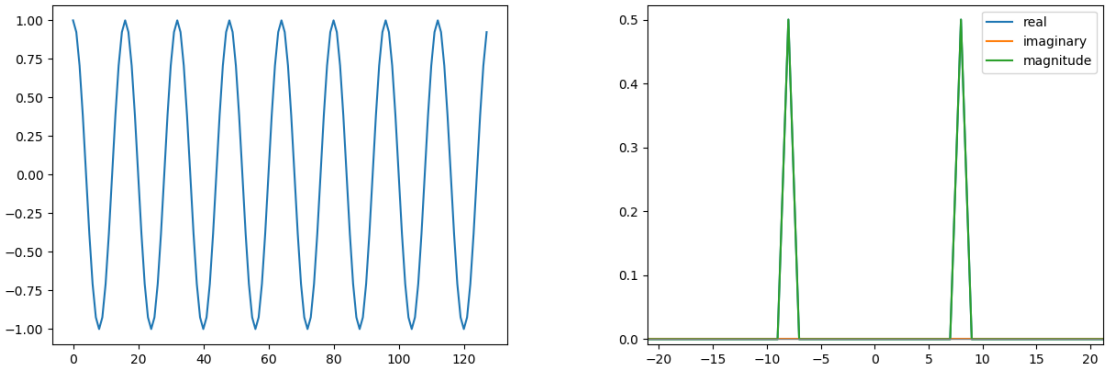

Figure 2: Plot of a cosine wave (left) and its associated Fourier transform (right).

To further experiment, we try the same experiment with a sine wave instead of a cosine wave. This should produce different results, because $\mathcal{F}(sin(2\pi\mu_0 x)) = \frac{1}{2j}(\delta(\mu-\mu_0)-\delta(\mu+\mu_0))$. Unlike the cosine transform, which is entirely real, the transform of the sine wave should be entirely imaginary because of the j in the denominator of the result. This is indeed what we observe in our results, as shown in Figure 3.

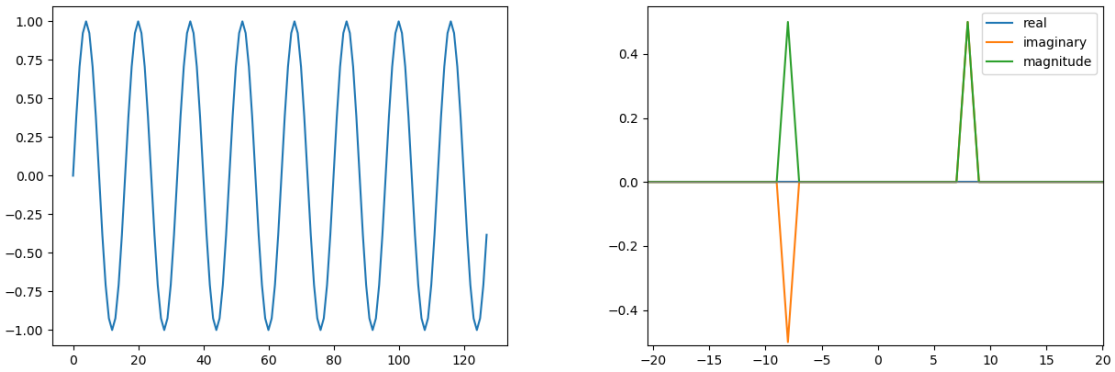

Figure 3: Plot of a sine wave (left) and its associated Fourier transform (right).

Next, we experimented with a rectangle function which has an amplitude of 1. The Fourier transform of this function should be $\mathcal{F}(Arect(x)) = Axe^{-j\pi ux}sinx(\pi ux)$. Again, our results match the theory; the real part of the transform is a sinc wave, and the magnitude slowly drops as the wave gets farther away from the center. The plot of these results is shown in Figure 4.

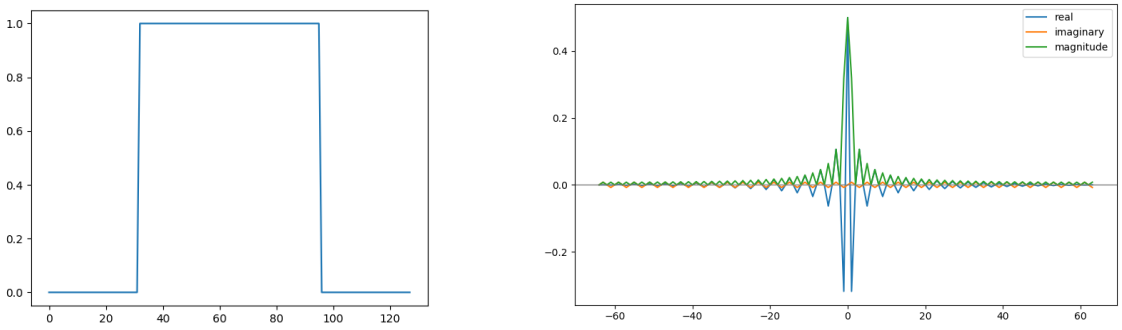

Figure 4: Plot of a rectangle function (left) and its associated Fourier transform (right).

Finally, we found an example for which our results do not align with the theory: the impulse function. Our discretization of this function was an array of 128 zeros with a 1 at array location 64. The Fourier transform of this function should be a line at 1 (the size of the impulse). However, as shown in Figure 5, this is not what our code generated.

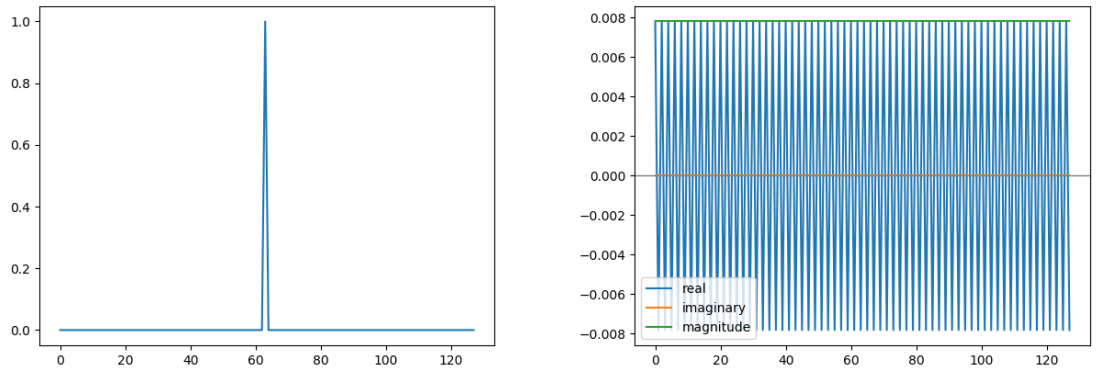

Figure 5: Plot of a discrete impulse function (left) and its associated Fourier transform (right).

There is a sine wave in the real part of the result, and the magnitude of the horizontal line is at 0.008 rather than 1. We note that this is not due to accidental division by N, because we get the correct scale on our other plots. To determine why the transform looked like this, we manually calculated the DFT of the signal:

$$\mathcal{F}(\delta(1)) = \frac{1}{128}\sum_{x=0}^{127} f(x)e^{\frac{-j2\pi ux}{128}}$$

Because our implementation of the impulse is a single 1 at x=64, this simplifies to:

$$\begin{aligned} F(\delta(1)) &= \frac{1}{128}1 * e^{\frac{-j2\pi u(64)}{128}} \\ &= \frac{1}{128}e^{-j\pi u} \\ &= \frac{1}{128}(cos(\pi u) - jsin(\pi u)) \end{aligned} \tag{4}$$

$jsin(\pi u)$ becomes zero because the transform is discrete and $sin(\pi u) = 0$ whenever u is an integer (which it always is), so the whole equation simplifies to $F(\delta(1)) = \frac{1}{128}cos(\pi u))$ which aligns with our results in Figure 5. $cos(\pi u)$ oscillates between -1 and 1, so the magnitude is always 1. The result is then multiplied by $\frac{1}{128}$, which is why the horizontal line is at .008 rather than 1. This shows how the translation from a continuous signal to a discrete one can cause anomalous results.

3.2 Fourier Transform on Images

For the 2D transform, we test our code on a black image with a white square in the center. We compute the Fourier transform on this image and plot the results in Figure 6. We also center the transform by multiplying by each pixel (x, y) by $(-1)^{x+y}$. Results for both the centered transform and non-centered transform are provided. For visualization purposes, we plot $log(1+F(u, v))$ to make the larger values smaller so that lower points in the result are more visible.

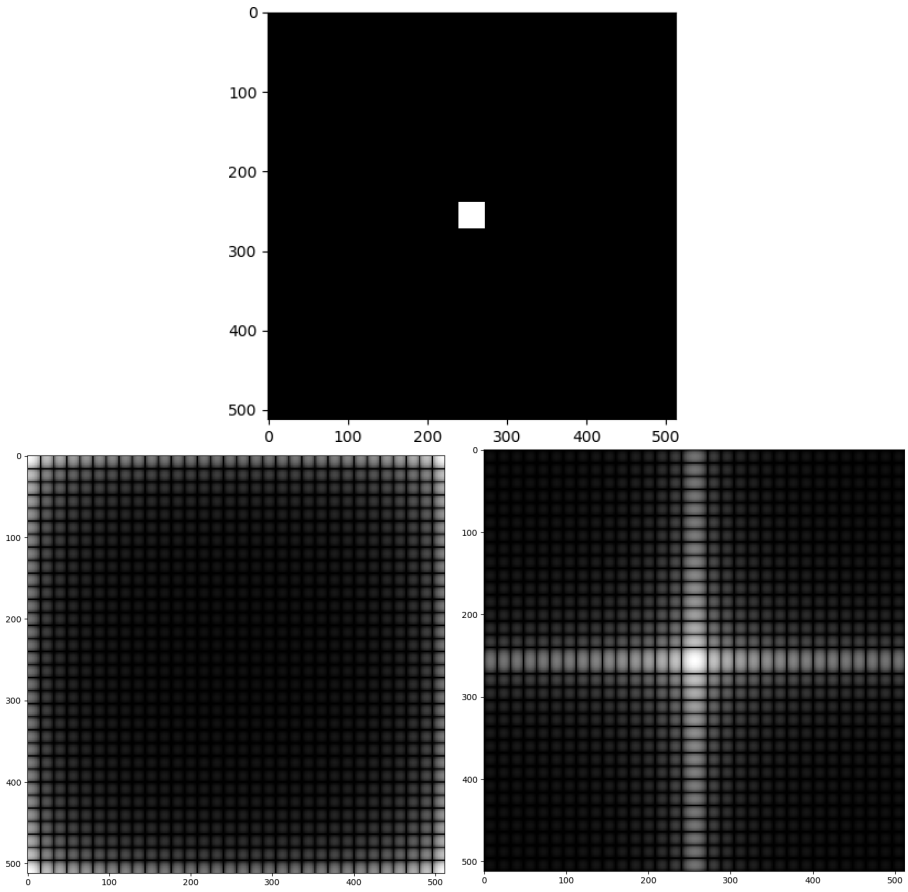

Figure 6: A 32px by 32px square (top) along with its associated Fourier transform (bottom). The left transform is not centered, the right transform is centered

The result of the transform is a 2D sinc function, similar to how the transform of the 1D rectangle function is a 1D sinc. In the non-centered image, the peak of the sinc is split between the corners because we are not capturing a full period. The right half of the period is captured on the left side of the plot, which then loops back around to the left half of the period on the right side of the plot. By centering the transform in the spatial domain, we obtain a full period with the peak of the sinc function in the center of the image.

In our next experiment, we change the size of the rectangle from 32 pixels by 32 pixels to 64 pixels by 64 pixels. The result of this is shown in Figure 7. As shown in the figure, the sinc function has a shorter frequency so the higher intensities are more concentrated along the x and y axes. The result makes sense, because the Fourier transform of a square (in 1D) is $\mathcal{F}(Arect(x)) = Axe^{-j\pi ux}sinx(\pi ux)$. x is the width of the *rect*, so a smaller square (lower x), results in a smaller frequency in $sinx(\pi ux)$. This can also be intuitively explained as the converse of the inverse Fourier transform for the ideal low-pass filter. In the frequency domain, a square with a smaller cutoff attenuates more frequencies and translates to a wider sinc in the spatial domain, resulting in more blurring. Similarly, in our results, a smaller square in the spatial domain translates to a flatter sinc in the frequency domain.

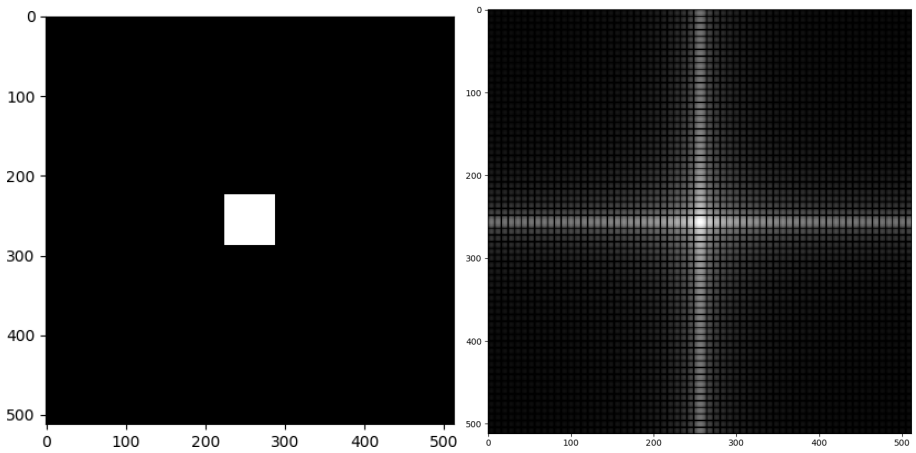

Figure 7: A 64px by 64px square along with its associated centered Fourier transform

When we grow the size of the square to 128 pixels by 128 pixels, we observe the same effect. The results of this are shown in Figure 8. As the square grows larger, the frequency of the Fourier transform gets smaller and the plot moves closer to being an impulse function.

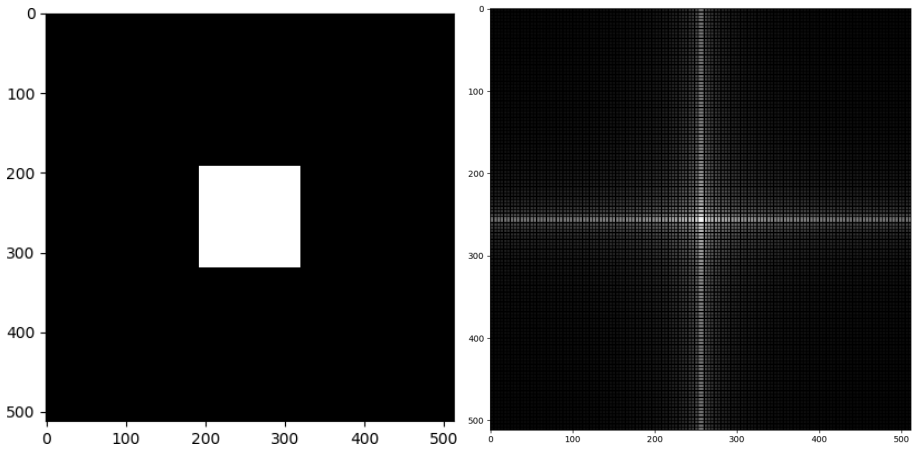

Figure 8: A 128px by 128px square along with its associated centered Fourier transform

3.3 Magnitude and Phase Experiments

The importance of magnitude and phase are clearly shown in this experiment. It can be clearly seen that phase carries most of the positional information. This is unexpected because magnitude is used when determining a filter size for removing noise.

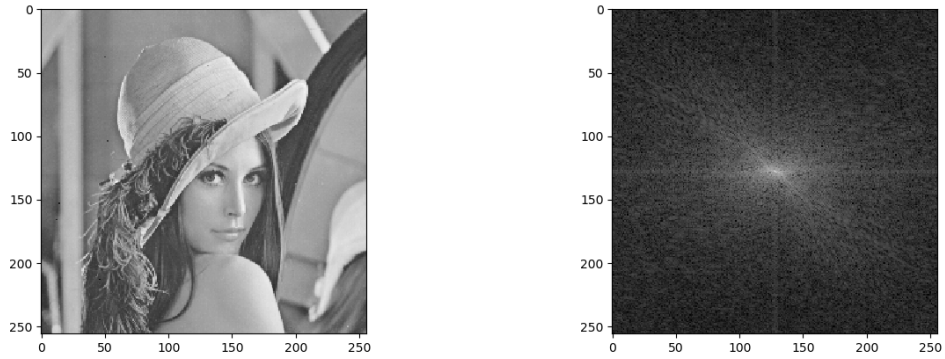

Figure 9: Plot of the original image (left) and the image's magnitude representation in the frequency domain (right)

In Figure 9 we see the original lenna image and the magnitude of the forward transform. The magnitude appears as we expect it to. We will reference this image later when we show the reconstruction of lenna after removing magnitude (setting the magnitude to 1). To improve visualization, we normalize the values from 0 to 255 and apply histogram equalization.

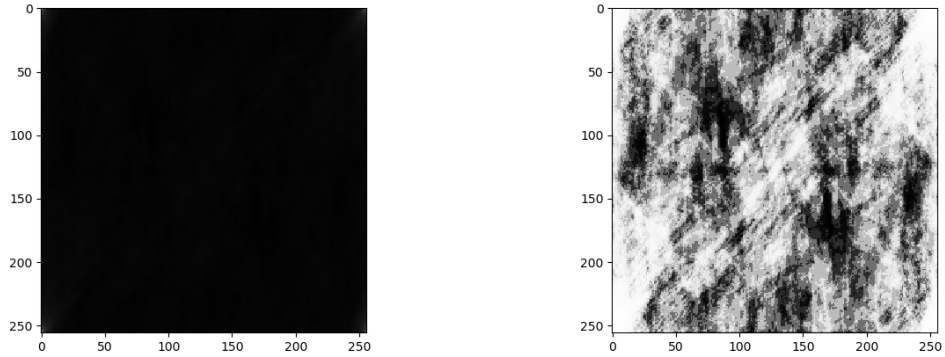

Figure 10: Plot of the image reconstruction with only magnitude. The normalized image is shown on the left, the normalized and equalized image is shown on the right.

We show the reconstruction of lenna from the frequency domain without phase in Figure 10. Although it was expected from in-class examples that magnitude would not result in a very meaningful image, our resultant image is much darker than we anticipated. Some lighter colored pixels appear in a dispersing fashion in each of the corners. Our interpretation of this is that the light colored pixels near the corners were drastically larger than the rest of the pixels in the image, so during normalization the rest of the shades were "washed out". There are no details in this image that would lead us to visually associate it with the original lenna image.

We show the reconstruction of lenna from the frequency domain without phase in Figure 10. Although it was expected from in-class examples that magnitude would not result in a very meaningful image, our resultant image was much darker than we anticipated. Some lighter colored pixels appeared in a dispersing fashion in each of the corners. Our interpretation of this is that the light colored pixels near the corners were drastically larger than the rest of the pixels in the image, so during normalization the rest of the shades were "washed out". In an attempt to restore the image quality we performed histogram equalization. This yielded the image on the right side of figure 10. This image aligns with our expectation. It should be noted

that there are no details in this image that would lead us to visually associate it with the original lenna image.

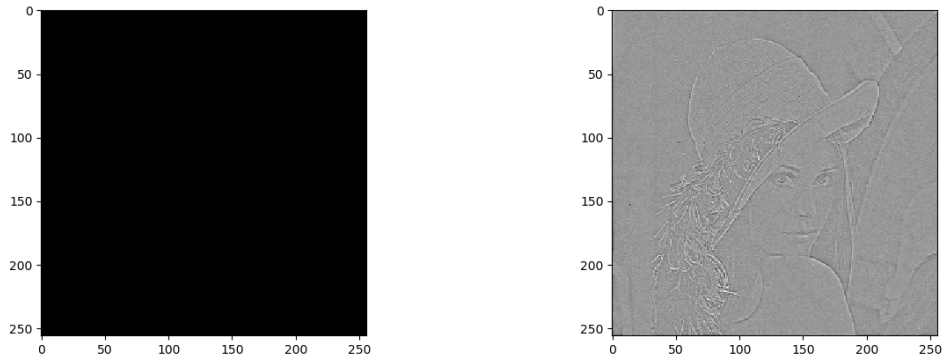

Figure 11: Plot of the magnitude in the frequency domain after setting to one (left) and the image's reconstruction with only phase (right)

Finally, we see the reconstruction of the image without magnitude. The importance of phase in determining the shape characteristics of the original image are clearly shown. It is interesting to note that all intensity information is gone because it is carried by the magnitude. Interestingly the result appears to be achieving something similar to edge detection.

4 Program Listing

4.1 FFT class

```
class FFT():
    def __init__(self):
        if not os.path.isfile("fft"):
            subprocess.call(["g++", "fft.c", "-o", "fft"])

    def fourier_transform(self, data, mode, extend=True):
        # If not otherwise specified, assume the input to
        # forward transform does not have zeros where the
        # imaginary values will be
        if extend == True and mode == 1:
            # add a zero after every real value in list
            data_complex = [0]*len(data)*2
            data_complex[::2] = data
        else:
            data_complex = data

        output = subprocess.check_output(["./fft"] + [str(d) for d in data_complex]
                                         + [str(mode)]).split()
        transform = [float(t) for t in output]

        # Get rid of imaginary values for inverse transform
        if mode == -1:
            for i in range(1, len(transform), 2):
                transform[i] = 0
```

```
        return transform
```

4.2 Experiment 1

```
def center_transform(data):
    new_data = []
    for i in range(len(data)):
        new_data.append(data[i]*(-1)**(i))
    return new_data

def plot_transform_1d(data, centered=True):
    if centered:
        x = range(-len(data)//4, len(data)//4)
    else:
        x = range(len(data)//2)
    mag = [(data[2*i]**2 + data[2*i+1]**2)**(1/2) for i in range(len(data)//2)]
    plt.plot(x, [data[2*i] for i in range(len(data)//2)], label="real")
    plt.plot(x, [data[2*i+1] for i in range(len(data)//2)], label="imaginary")
    plt.plot(x, mag, label="magnitude")
    plt.axhline(0, lw=1, color="grey")
    plt.legend()
    plt.show()

def gen_cos_wave(u, N):
    return [cos(2*pi*u*x/128) for x in range(N)]

# Part 1
data = [2, 3, 4, 4]
transform = fft_func.fourier_transform(data, 1)
plot_transform_1d(transform, centered=False)

inverse_transform = fft_func.fourier_transform(transform, -1)

# Part 2
cos_wave = gen_cos_wave(8, 128)
cos_transform = fft_func.fourier_transform(center_transform(cos_wave), 1)
plot_transform_1d(cos_transform)

# Part 3
rect_func = []
with open("Rect_128.dat") as rect_file:
    for line in rect_file:
        rect_func.append(float(line))

rect_transform = fft_func.fourier_transform(center_transform(rect_func), 1)
plot_transform_1d(rect_transform)

# Impulse testing
impulse_func = [0]*128
impulse_func[64] = 1
plt.plot(impulse_func)
plt.show()
```

```
impulse_transform = fft_func.fourier_transform(impulse_func, 1)
plot_transform_1d(impulse_transform, centered=False)
```

4.3 Experiment 2

```
def dft2d(data, mode):
    n = len(data[0])
    if mode is 1:
        data = add_zero(data)

    transform = []
    # dft of rows
    for i in range(0, len(data)):
        transform.append(fft_func.fourier_transform(data[i], mode, extend=False))

    # multiply by n
    if mode is 1:
        for i in range(len(transform)):
            for j in range(len(transform[i])):
                transform[i][j] *= n
    # get columns
    columns = []
    for i in range(len(transform[0]) // 2):
        columns.append([])
        for row in transform:
            columns[i].append(row[i*2])
            columns[i].append(row[i*2+1])
    # transform columns
    for x in range(0, len(columns)):
        columns[x] = fft_func.fourier_transform(columns[x], mode, extend=False)
    # convert columns to rows
    # get rows
    transform = []
    for i in range(len(columns[0]) // 2):
        transform.append([])
        for row in columns:
            transform[i].append(row[i * 2])
            transform[i].append(row[i * 2 + 1])

    return transform

def center_2d_transform(data):
    new_data = []
    for i in range(len(data)):
        new_data.append([])
        for j in range(len(data[i])):
            new_data[i].append(data[i][j]*(-1)**(i+j))
    return new_data

def plot_transform_2d(data, centered=True):
    if centered:
        x = range(-(len(data)*2)//4, (len(data)*2)//4)
```

```
        y = range(-len(data[0])//4, len(data[0])//4)
    else:
        x = range((len(data)*2)//2)
        y = range(len(data[0]) // 2)

    # magnitude
    mag = []
    for i in range(len(data)//2):
        mag.append([])
        for j in range(len(data[i])//2):
            mag[i].append(log((data[2*i][2*j]**2
                    + data[2*i+1][2*j+1]**2)**(1/2) + 1))

    plt.imshow(mag, label="magnitude", cmap=plt.get_cmap("gray"))
    plt.show()

def SquareImage(squareSize):
    size = squareSize/2
    size = int(size)
    im = Image.new("L", (512, 512), "black")
    im.paste("white", (256-size,256-size, 256+size,256+size))
    return im

#convert from square image
squareImage = SquareImage(32)
numpyArray = numpy.array(squareImage)

squareList = numpyArray.tolist()
plt.imshow(squareList, cmap=plt.get_cmap("gray"))
plt.show()

transform = dft2d(center_2d_transform(squareList), 1)
plot_transform_2d(transform)
plt.show()
```

4.4 Experiment 3

```
def remove_centering_2d_transform(data):
    data = remove_zeros(data)
    new_data = []
    for i in range(len(data)):
        new_data.append([])
        for j in range(len(data[i])):
            new_data[i].append(data[i][j] * (-1) ** (i + j))
    return new_data

def remove_zeros(data):
    new_data = []
    for i in range(len(data)):
        new_data.append([])
        for j in range(0, len(data[i]), 2):
            new_data[i].append(data[i][j])
    return new_data
```

```
lenna = Image.open("../images-pgm/lenna.pgm")
    width, height = lenna.size
    lenna_pixels = lenna.load()

    lenna_list = []
    for x in range(width):
        lenna_list.append([])
        for y in range(height):
            lenna_list[x].append(lenna.getpixel((y,x)))

# part 1
lenna_transform = dft2d(center_2d_transform(lenna_list), 1)

plot_transform_2d(lenna_transform)

# Set phase equal to zero
for i in range(len(lenna_transform)):
    for j in range(0, len(lenna_transform[i]), 2):
        lenna_transform[i][j] = sqrt((lenna_transform[i][j] ** 2) + (lenna_transform[i][j -
        lenna_transform[i][j+1] = 0

lenna_transform = dft2d(lenna_transform, -1)
lenna_transform = remove_centering_2d_transform(lenna_transform)
# lenna_inverse_transform = remove_zeros(lenna_inverse_transform)
plt.imshow(lenna_transform, cmap=plt.get_cmap("gray"))
plt.show()

# part 2
lenna_transform = dft2d(center_2d_transform(lenna_list), 1)
plot_transform_2d(lenna_transform)

# Set phase to original and magnitude to 1
for i in range(len(lenna_transform)):
    for j in range(0, len(lenna_transform[i]), 2):
        theta = (atan2(lenna_transform[i][j+1], lenna_transform[i][j]))
        lenna_transform[i][j] = cos(theta)
        lenna_transform[i][j+1] = sin(theta)

lenna_inverse_transform = dft2d(lenna_transform, -1)
lenna_inverse_transform = remove_centering_2d_transform(lenna_inverse_transform)
plt.imshow(lenna_inverse_transform, cmap=plt.get_cmap("gray"))
plt.show()
```