ECSE 316 – Signals and Networks

Assignment 2 Report

McGill University – Winter 2023

Members:

Rambod Azimi – 260911967

Saghar Sahebi – 260908343

Group #39

Date: 31/3/2023

Introduction

In this assignment, we implemented two different Discrete Fourier Transform (DFT) versions. DFT is a mathematical algorithm that transforms a finite sequence of samples into a sequence of complex numbers. It takes a discrete signal and rewrites it into a sum of complex exponential functions. DFT is widely used in many real-world examples, such as image processing, signal processing, and many more problems.

The first version of the DFT implemented in this assignment uses a popular technique called brute force which tries every possible solution.

The second version of the DFT implemented in this assignment uses another approach called the Fast Fourier Transform (FFT) which follows a divide-and-conquer approach. We can achieve this by splitting the problem into two smaller sub-problems and recursively repeating this process until we reach the initial condition and end the process. Although there are different algorithms for FFT, we use a popular algorithm called Cooley-Tukey FFT. After implementing the FFT algorithm successfully, we can extend it to a 2-dimensions case. Hence, we will be able to apply the algorithm to an image to manipulate it (i.e. denoise and compress the image).

We will discuss each algorithm design by providing an in depth explanation for each. In the end, we will talk about the complexity of each algorithm implemented and their advantages and disadvantages.

Program Design with Testing and Experiment

As mentioned in the introduction, we will use two different approaches to implement the Discrete Fourier Transform. The first approach uses brute force, which is simply using the formula in figure 1 below.

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi}{N} kn}, \qquad \text{for } k = 0, 1, 2, \ldots, N-1. \tag{1}$$

$$x_n = \frac{1}{N} \sum_{n=0}^{N-1} X_k e^{\frac{i2\pi}{N} kn}, \qquad \text{for } k = 0, 1, 2, \ldots, N-1. \tag{2}$$

Figure 1. The Discrete Fourier Transform (DFT) and its inverse

As illustrated in the two equations shown in figure 1, x denotes the signal, and X represents the Fourier transform of x. In the first equation, we simply decompose the signal x into a sum of its various frequency components. In the second equation, we can calculate the inverse Fourier transform.

In the second approach, we will use the Fast Fourier Transform (FFT) using a popular Cooley-Tukey FFT algorithm. It generally generates the Fourier Transform faster than the traditional approach because it takes advantage of symmetry to simplify the complexity of the traditional method. The detailed formula is shown in figure 2 below.

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

Figure 2. The Fast Fourier Transform (FFT) using the Cooley-Tukey algorithm

The algorithm simply splits the sum into two separate sums, one for even indices and one for odd indices. With this approach, we can calculate each sum separately and combine them by adding them together. Therefore, we can divide the original sums into many smaller and simpler sums and repeat the same process again and again until we reach the limit. We can apply the traditional algorithm to each small sum. The problem is that we must stop splitting the sum in the code. For example, let's say we have an extensive array of size 1024. In the first step, we divide the sum into two separate sub-arrays, each of size 512. Then, we repeat the process, and now we have four

sub-arrays, each of size 256. In the next iteration, we have eight sub-arrays, then 16. Sometimes, it is not necessary to reach many sub-arrays of size 2 in order to stop the process. We can choose a threshold to stop the splitting process.

After successfully implementing DFT and FTT, we can extend it to a two-dimensional case (2DDFT). As the name suggests, this time, instead of applying the Fourier Transform to a 1D array, we apply it to each row separately and then return the result. Let's assume we have a 2D array called f with N rows and M columns, and we want to compute the Fourier Transform for this 2D array. Figure 3 shows the process in mathematical form.

$$
\begin{aligned}
F[k,l] &= \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} f[m,n] e^{\frac{-i2\pi}{M}km} e^{\frac{-i2\pi}{N}ln} \\
&= \sum_{n=0}^{N-1}\Big(\sum_{m=0}^{M-1} f[m,n] e^{\frac{-i2\pi}{M}km}\Big) e^{\frac{-i2\pi}{N}ln} \\
&\text{for } k = 0, 1, \ldots, M-1 \text{ and } l = 0, 1, \ldots, N-1.
\end{aligned}
$$

Figure 3. 2DDFT algorithm with a two-dimensional array

The second line in Figure 3 means that we first apply the traditional DFT algorithm to each row of the 2D array. Then, we apply it to each column to calculate the Fourier Transform of the 2D array. The inverse Fourier transform of a 2D array is also shown in figure 4.

$$
\begin{aligned}
f[m,n] &= \frac{1}{NM}\sum_{l=0}^{N-1}\sum_{k=0}^{M-1} F[k,l] e^{\frac{i2\pi}{M}km} e^{\frac{i2\pi}{N}ln} \\
&= \frac{1}{NM}\sum_{l=0}^{N-1}\Big(\sum_{k=0}^{M-1} F[k,l] e^{\frac{i2\pi}{M}km}\Big) e^{\frac{i2\pi}{N}ln} \\
&\text{for } m = 0, 1, \ldots, M-1 \text{ and } n = 0, 1, \ldots, N-1.
\end{aligned}
$$

Figure 4. Inverse Fourier Transform algorithm with a two-dimensional array

Having completed the extension version of FTT, which supports a 2D array, we can apply this algorithm to images to manipulate them. The two main tasks that we implement in this assignment are the followings:

1. **Image denoising**
   The image is denoised by applying the FFT algorithm for the 2D array and truncating high frequencies. Finally, we display the original and the denoised images next to each other in one plot.

2. **Image Compression**
   Similar to image denoising, but in this case, we compress the image file given different values. For example, when we save 10% compression, it means that we remove 10% of the image and use the rest as the data. In this assignment, we use six different compression levels (0, 20%, 40%, 60%, 80%, and 90%) and plot them all next to each other in one plot.

Now, let's explain each mode separately and analyze the algorithms implemented in each mode.

Mode 1 (Convert the image into its FFT form and display the result)

Design

Figure 5 shows the algorithm implemented in the first mode.

```
def mode1(img):
    FFT_image = np.abs(FFT_2D(img)) # calling FFT_2D function and save the result in FFT_image

    # one by two subplot of the original image
    plot.subplot(1,2,1)
    plot.imshow(img, cmap= 'gray')
    plot.title("(Before FFT)")

    #one by two subplot of the FFT image
    plot.subplot(1,2,2)
    plot.imshow(FFT_image, norm=colors.LogNorm())
    plot.title("(After FFT)")

    plot.show()
```

Figure 5. Converting the image into its FFT form and displaying the result

First, we call the FFT_2D method on the image to generate the Fourier Transform (Fast version) for the 2D array (image) and save the result in FFT_image. Then, we create two subplots to plot both original and altered images in the same plot.

Testing and Experiment

Figure 6 shows the result of running the program on mode one. By comparing the generated plot with the plot generated using the built-in *np.fft.fft2()* method, we can conclude that the algorithm works correctly.

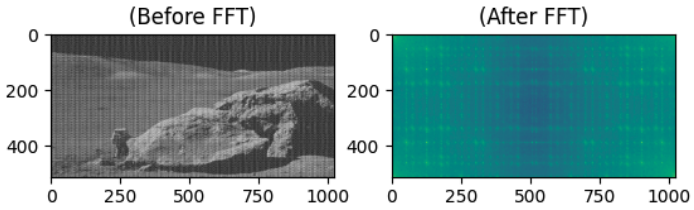

Figure 6. The result of running the program in the first mode

Mode 2 (Denoise the image by applying an FFT and display the result)

Design

Figure 7 shows the algorithm implemented in the second mode.

```
def mode2(img):
    print("Mode 2 is running...")
    # the denoise factor, we chose to go with 0.4
    denoise_factor = 0.4
    FFT_image = FFT_2D(img)
    # count the non zero for later when calculating the fraction
    before_zero = np.count_nonzero(FFT_image)
    # setting the high frequencies to 0
    # width
    FFT_image[:, int(denoise_factor * FFT_image.shape[1]) : int(FFT_image.shape[1] * (1-denoise_factor))] = 0
    # height
    FFT_image[int(denoise_factor * FFT_image.shape[0]) : int(FFT_image.shape[1] * (1-denoise_factor))] = 0

    # count the new non zero
    after_zero = np.count_nonzero(FFT_image)

    inverse_FFT_image = FFT_2D_inverse(FFT_image).real
    # as asked in the assignment printing the fraction and non-zeros
    fraction = (after_zero/before_zero)
    print(f"The number of non-zeros are: {after_zero}")
    print(f"The fraction they represent of the original Fourier coefficients: {fraction}")
    # before
    plot.subplot(1,2,1)
    plot.imshow(img, cmap= 'gray')
    plot.title("(Before denoising)")
    # after
    plot.subplot(1,2,2)
    plot.imshow(inverse_FFT_image, cmap= 'gray')
    plot.title("(After denoising)")
    plot.show()
```

Figure 7. Algorithm for denoising an image in the second mode

In the second mode, we have to denoise an image by applying FFT algorithms on the image. Firstly, we apply the FFT algorithm for the 2D array to find the image's Fourier Transform. Then, we set the output image's high frequencies to zero. Then, we calculate the inverse FFT of the output and save the result in the *inverse_FFT_image* variable. Finally, we plot the results in gray color.

Testing and Experiment

Figure 8 shows the result of running the program in the second mode. By comparing the generated plot with the generated plot using built-in methods in np, such as *np.fft.fft(), np.fft.fft2(), np.fft.ifft(),* we can conclude that our algorithm works correctly.

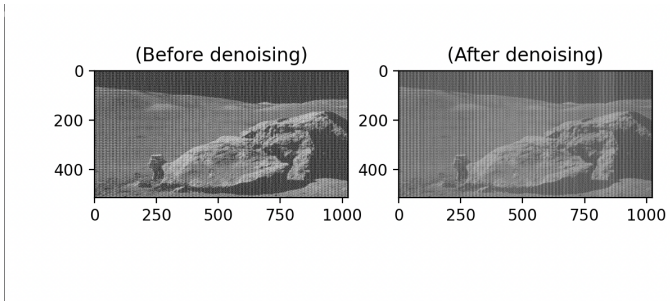

Figure 8. The result of running the program in the second mode

Mode 3 (Compress and save the image)

Design

Figure 9 shows the algorithm implemented in the third mode.

```
def mode3(img):
    print("Mode 3 is running...")
    FFT_image = FFT_2D(img)
    rate = [0, 0.2, 0.4, 0.6, 0.8, 0.95]
    # we need to perform compression
    # we will compress with 6 different levels and plot them
    plot.subplot(2,3,1)
    plot.imshow(np.real(FFT_2D_inverse(compression(FFT_image.copy(), rate[0]))), cmap= 'gray')
    plot.title("0%")

    plot.subplot(2,3,2)
    plot.imshow(np.real(FFT_2D_inverse(compression(FFT_image.copy(), rate[1]))), cmap= 'gray')
    plot.title("20%")

    plot.subplot(2,3,3)
    plot.imshow(np.real(FFT_2D_inverse(compression(FFT_image.copy(), rate[2]))), cmap= 'gray')
    plot.title("40%")

    plot.subplot(2,3,4)
    plot.imshow(np.real(FFT_2D_inverse(compression(FFT_image.copy(), rate[3]))), cmap= 'gray')
    plot.title("60%")

    plot.subplot(2,3,5)
    plot.imshow(np.real(FFT_2D_inverse(compression(FFT_image.copy(), rate[4]))), cmap= 'gray')
    plot.title("80%")

    plot.subplot(2,3,6)
    plot.imshow(np.real(FFT_2D_inverse(compression(FFT_image.copy(), rate[5]))), cmap= 'gray')
    plot.title("95%")
    plot.show()
```

Figure 9. The algorithm for compressing an image in the third mode

In the third mode, we have to compress a given image file with different rates by applying the FFT of the image. The compression comes from setting some Fourier coefficients to zero. We used six different percentages (0%, 20%, 40%, 60%, 80%, and 95%) for the rates.

Testing and Experiment

Finally, we display the result of the six sub-plots on a single-view page, as illustrated in figure 10. Comparing the final result with the implementation using built-in methods in the *NumPy* library shows they are similar. Thus, our algorithm works correctly.

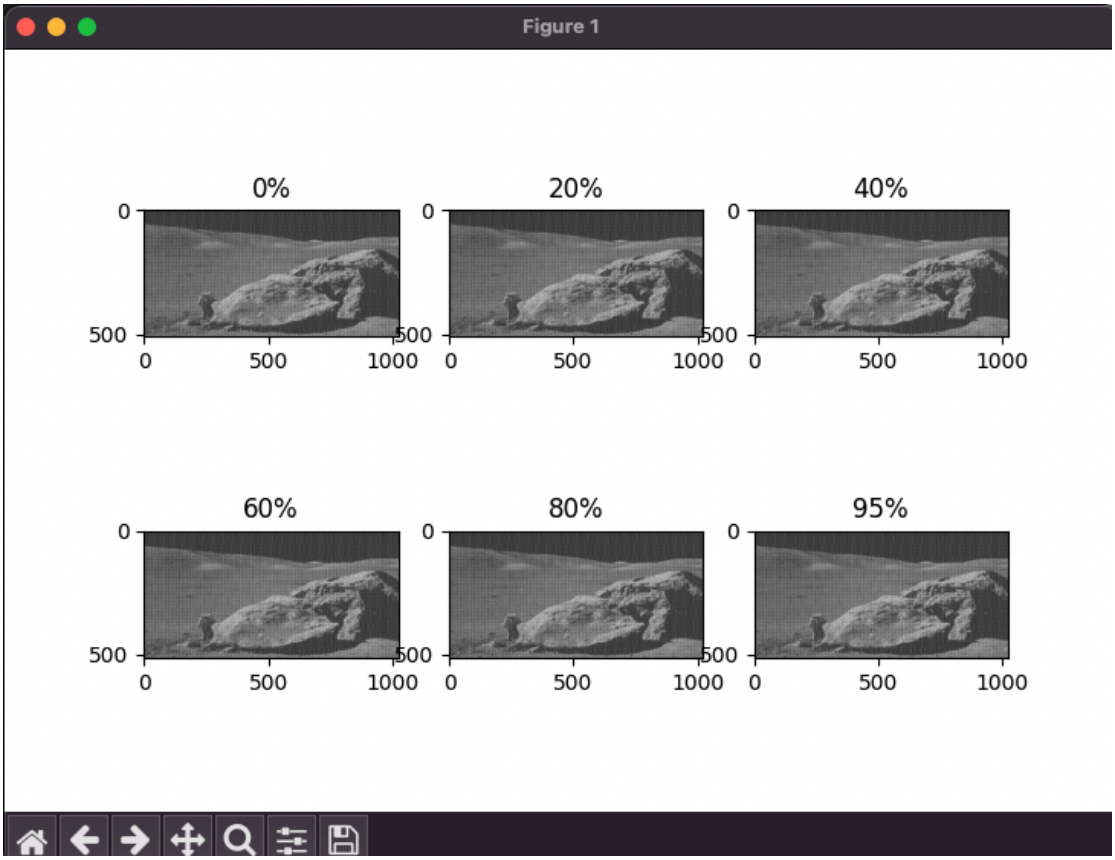

Figure 10. The result of running the program in the third mode

Mode 4 (Plot the runtime graphs for the report)

The last mode is simply the plotting mode, meaning we have to produce some plots that summarize the runtime complexity of the implemented algorithms in this assignment. We can achieve this by displaying the results in the command line. Figure 11.1 and 11.2 shows the results when running the program in the fourth mode.

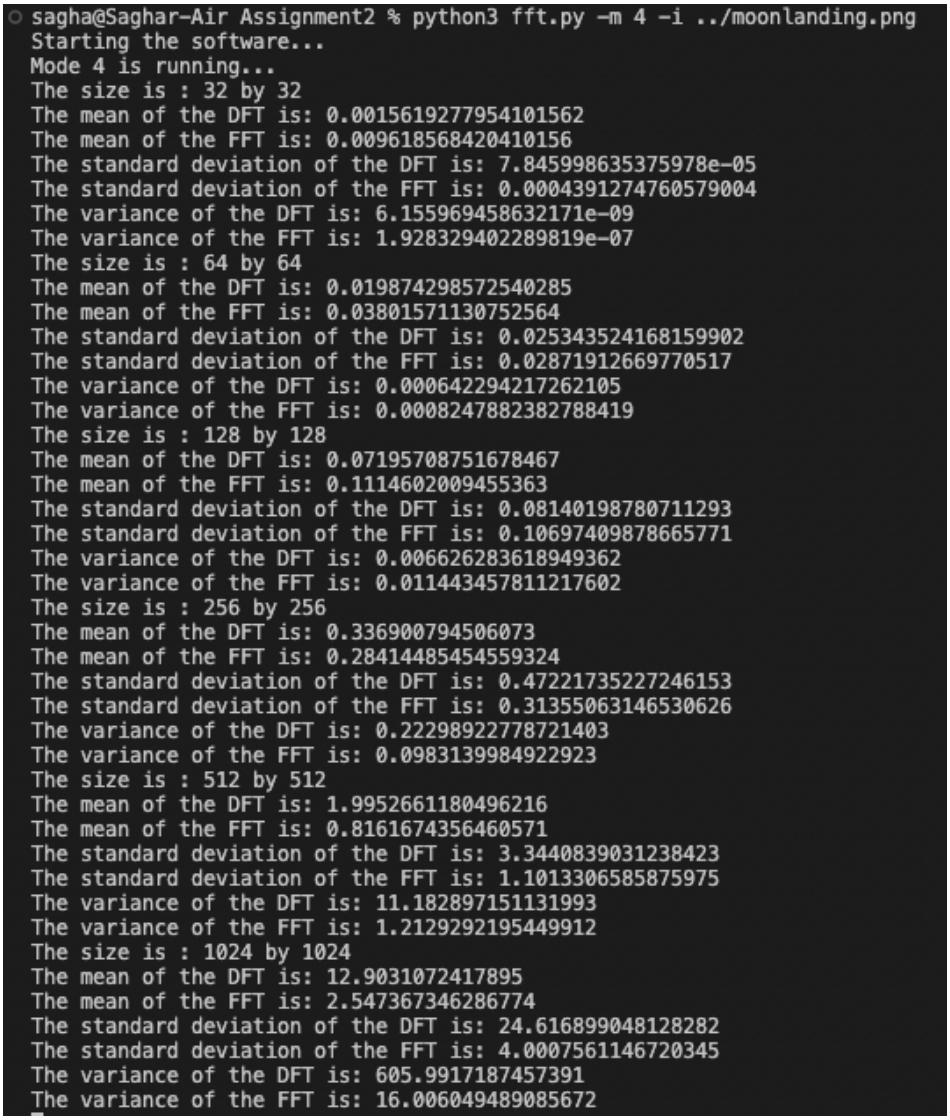

Figure 11.1 The mean, standard deviation and variance of mode 4

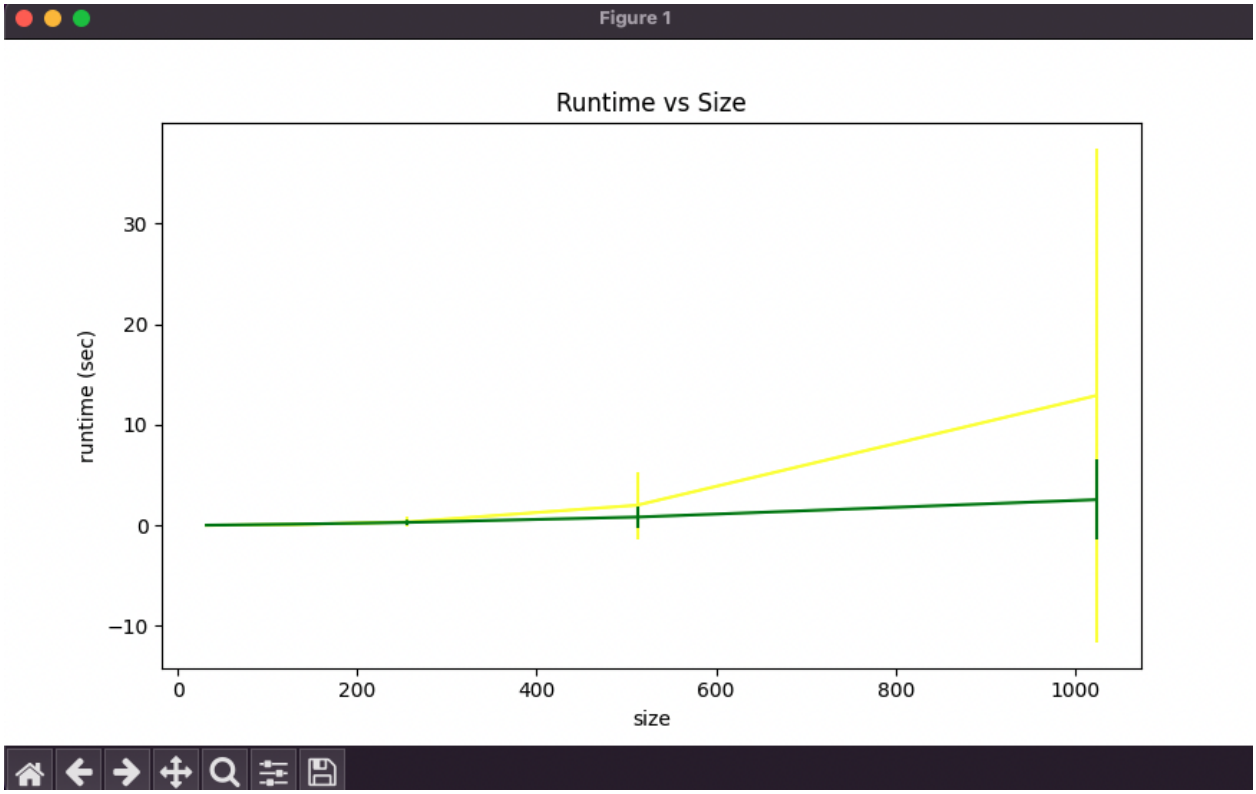

Figure 11.2 The Runtime vs Size graph generated from running mode 4

Figure 12 shows the result of our implementation in the first mode. As illustrated in the previous parts, we implemented the Cooley-Tukey algorithm, which breaks the original sum into smaller and simpler sums (even and od. Then, we try to calculate the Fourier Transform of each small sum and, finally, add them together. We could implement the same algorithm using the NumPy library in Python. NumPy library has built-in methods for calculating the Fourier Transform of many different types of signals. For example, np.fft.fft() method implements the Fast Fourier Transform algorithm for 1-dimensional arrays. In addition, np.fft.fft2() implements the same algorithm with the support for 2-dimensional arrays (images). We could have also used the np.fft.ifft() methods which calculate the Inverse Fast Fourier Transform of a discrete signal.

Now, in order to compare our algorithm with the built-in methods in the NumPy library, we can compare Figure 12 (our implementation of FTT) with Figure 13 (using NumPy to generate the result). We can clearly observe that they look almost identical.

```python
# FFT of a 2D array
def FFT_2D(image: np.ndarray):
    M = len(image)
    N = len(image[0])
    X = np.zeros([M,N], dtype='complex_')

    for i, row in enumerate(image): # call the FFT or
        X[i] = FFT(row)

    X2 = np.zeros([N,M], dtype='complex_')
    for i, col in enumerate(X.T): # call the FFT on e
        X2[i] = FFT(col)

    return X2.T
```

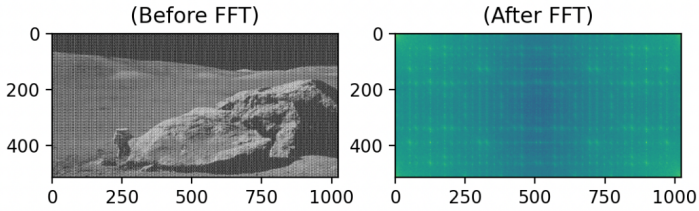

Figure 12. The result of FTT with our implementation

```python
# FFT of a 2D array
def FFT_2D(image: np.ndarray):

    return np.fft.fft2(image)


# inverse FFT of a 2D array
def FFT_2D_inverse(y):
```

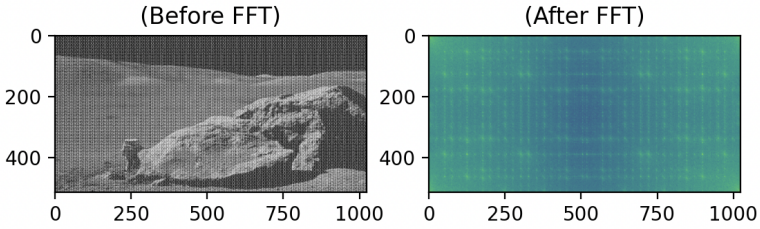

Figure 13. The result of FTT using the built-in method in the NumPy library in Python

Analysis

DFT Time Complexity

The time complexity of the traditional 1D DFT algorithm is O(N^2). This time complexity is because each term requires a sum of N terms O(N) and contains N terms itself. Therefore, the time complexity will be **O(N^2)**.

In the 2D DFT algorithm, the time complexity is O(N^4) because, as shown in figure 3, each row consumes a time complexity of O(N^2) and because we have M rows and N column in the 2D array, and also because N = M, thus we have a total time complexity of O(N^4) in this algorithm.

FFT Time Complexity

The time complexity of the Cooley-Tukey algorithm, which uses a popular technique called divide-and-conquer, is given by the following general expression.

$$\mathbf{T(n) = aT(n/b) + O(n)}$$

Where n denotes the input size, a represents the number of sub-problems created at each level, and n/b means the size of each sub-problem. In the Cooley-Tukey algorithm, we have decomposed the original sum into two sub-sums, which were called Even and Odd. Therefore, a = 2. So far, we have:

$$\mathbf{T(n) = 2\ T(n/b) + O(n)}$$

Next, we've decomposed the original sum into two equally-sized sub-sums. In other words, if we have a sum of 1024 terms, we have two 512 sub-sums after decomposing them. Therefore, b = 2. Hence, the T(n) will be the following:

$$\mathbf{T(n) = 2\ T(n/2) + O(n)}$$

Therefore, because a = b = 2, the log a (b) will be 1. Hence, we are going to reach the second case in the Master's theorem. So, by finding the recurrence relation as well as the time complexity in terms of Theta, we can prove that the time complexity of this algorithm is **O(n log n)**.