Fast Fourier Transform

Analysis, Denoising, Compression, and Runtime

November 28, 2024

1 Design

The Fast Fourier Transform program is organized across 8 different files (fft.py, fftQuery.py, FourierTransform.py, FastMode.py, DenoiseMode.py, CompressMode.py, PlottingMode.py, QueryMode.py). The fft.py file is the entry point of the program. It contains the main() function which defines the program's core logic. It is responsible for calling subroutines from the other files to parse the command line arguments (fftQuery.py), perform the Discrete Fourier Transform algorithms (FourierTransform.py), and execute/plot different modes (FastMode.py, DenoiseMode.py, CompressMode.py, PlottingMode.py). The QueryMode.py file contains the *Mode* Enum Class which is used by other files to parse and handle the flow execution of the program modes, and the *TestingFourierTransform.py* file contains code that was used for outputting test plots in Section 2.

It is important to note that the program will convert all given images to grayscale before plotting the results. Additionally, if the given image's dimensions are not powers of 2 then they are scaled up to the nearest power of 2 using the *cv2.resize()* function for all modes of the program. This is necessary for the implemented FFT algorithm to work.

The *fftQuery.py* file contains the *fftQuery* class that is responsible for parsing the command line arguments (mode and image name) and packaging their values in an object. It also performs input validation for the arguments (making sure that the mode is not out of range and the image file exists).

The *FourierTransform.py* file contains functions that are responsible for performing the naive 1 Dimensional Discrete Fourier Transform (naive 1D DFT), 1 Dimensional Fast Fourier Transform (1D FFT), 2 Dimensional naive Discrete Fourier Transforms (naive 2D DFT), 2 Dimensional Fast Fourier Transform (2D FFT), and all the inverses of the above algorithms (naive 1D IDFT, 1D IFFT, naive 2D IDFT, 2D IFFT) with numpy arrays. The naive 1D DFT and its inverse operation are computed by the functions *dft_naive_1D()* and *idft_naive_1D()* using matrix multiplication.

If we consider the equation for the naive 1D DFT:

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

We can define the discrete signal $\mathbf{x}$ and it's transform $\mathbf{X}$ as 1×N matrices

$$\mathbf{x} \coloneqq \begin{bmatrix} x_0 & x_1 & \cdots & x_{N-1} \end{bmatrix}, \quad \mathbf{X} \coloneqq \begin{bmatrix} X_0 & X_1 & \cdots & X_{N-1} \end{bmatrix} \tag{2}$$

and the following weight matrix:

$$W_{k,n} \coloneqq e^{-i\frac{2\pi kn}{N}}, \quad k, n = 0, 1, \ldots, N-1 \tag{3}$$

$$\mathbf{W} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & e^{-i\frac{2\pi}{N}} & e^{-i\frac{4\pi}{N}} & \cdots & e^{-i\frac{2\pi(N-1)}{N}} \\ 1 & e^{-i\frac{4\pi}{N}} & e^{-i\frac{8\pi}{N}} & \cdots & e^{-i\frac{4\pi(N-1)}{N}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{-i\frac{2\pi(N-1)}{N}} & e^{-i\frac{4\pi(N-1)}{N}} & \cdots & e^{-i\frac{2\pi(N-1)^2}{N}} \end{bmatrix} \tag{4}$$

Then, we can re-write the Discrete Fourier Transform as the result of this matrix multiplication:

$$\mathbf{X} \coloneqq \mathbf{x} \cdot \mathbf{W} \tag{5}$$

and thus the following system of equation

$$\begin{bmatrix} X_0 & X_1 & \cdots & X_{N-1} \end{bmatrix} \coloneqq \begin{bmatrix} x_0 & x_1 & \cdots & x_{N-1} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & e^{-i\frac{2\pi}{N}} & e^{-i\frac{4\pi}{N}} & \cdots & e^{-i\frac{2\pi(N-1)}{N}} \\ 1 & e^{-i\frac{4\pi}{N}} & e^{-i\frac{8\pi}{N}} & \cdots & e^{-i\frac{4\pi(N-1)}{N}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{-i\frac{2\pi(N-1)}{N}} & e^{-i\frac{4\pi(N-1)}{N}} & \cdots & e^{-i\frac{2\pi(N-1)^2}{N}} \end{bmatrix} \tag{6}$$

Similarly for the naive 1D IDFT:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i\frac{2\pi kn}{N}}, \quad \text{for } n = 0, 1, 2, \ldots, N-1 \tag{7}$$

It can be re-written as the result of this matrix multiplication:

$$\mathbf{x} \coloneqq \frac{1}{N} \mathbf{X} \cdot \mathbf{W'} \tag{8}$$

where **W'** is defined as

$$\mathbf{W'} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & e^{i\frac{2\pi}{N}} & e^{i\frac{4\pi}{N}} & \cdots & e^{i\frac{2\pi(N-1)}{N}} \\ 1 & e^{i\frac{4\pi}{N}} & e^{i\frac{8\pi}{N}} & \cdots & e^{i\frac{4\pi(N-1)}{N}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{i\frac{2\pi(N-1)}{N}} & e^{i\frac{4\pi(N-1)}{N}} & \cdots & e^{i\frac{2\pi(N-1)^2}{N}} \end{bmatrix} \tag{9}$$

The 1D FFT and its inverse are computed by the recursive functions *fft_1D()* and *ifft_1D()* using the Cooley-Tukey algorithm (note that the length of the input signal N needs to be a power of 2 for the FFT algorithm to work). In addition, the following two optimizations were applied to the algorithm:

- Once the signal has been divided recursively to a length of 4, precomputed weighted matrices **W** and **W'** are used to perform the naive 1D DFT (base case of the recursion).
- After recursively solving the FFT for the even and odd parts of the signal, the first half of the transformed signal is obtained by the given equation in the assignment handout:

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{i2\pi km}{N/2}} \quad + \quad e^{-\frac{i2\pi k}{N}} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{i2\pi km}{N/2}}, \quad \text{for } k = 0, 1, 2, \ldots, N/2-1 \tag{10}$$

the second half of the signal is obtained in a similar manner using the periodicity of the complex exponential [1]:

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{i2\pi km}{N/2}} \quad - \quad e^{-\frac{i2\pi k}{N}} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{i2\pi km}{N/2}}, \quad \text{for } k = N/2, N/2+1, \ldots, N-1 \tag{11}$$

Similar equations are also used for the 1D IFFT, exploiting the periodicity of the complex exponential.

The naive 2D DFT and 2D FFT are implemented by performing the naive 1D DFT and 1D FFT respectively for all the rows first, then for all the transformed columns. Row and column 1D Fourier Transformations are applied using the *numpy.apply_along_axis()* function.

The *FastMode.py* file contains a function to plot the magnitude of the FFT complex coefficients in a 2D log-scaled gray-scale image.

The *DenoiseMode.py* file contains functions responsible to denoise a given image and plot the results. The denoising of the image is done by setting the coefficients of frequencies outside of the range $[-\frac{\pi}{7}, \frac{\pi}{7}]$ to 0. This is done by shifting the low frequencies of the 2D FFT array to the center with *numpy.fft.fftshift()*, and setting to zero complex coefficients that are outside of an inner rectangle representing frequencies in the range $[-\frac{\pi}{7}, \frac{\pi}{7}]$. Then the 2D FFT is unshifted (*numpy.fft.ifftshift()*) and 2D IFFT is performed to obtain the final denoised image.

The *CompressMode.py* file contains functions responsible to compress a given image and plot the results. The compression of the image is done by only keeping the coefficients of frequencies with a magnitude above the 40th, 75th, 90th, 99th, and 99.9th percentile (5 compressed images plotted alongside the original image).

The *PlottingMode.py* file contains the class *PlottingMode* that is responsible for running the naive 2D DFT and the 2D FFT on different randomly generated 2D square arrays and plotting the average runtime along with associated confidence intervals. The array sizes chosen to perform the runs are $[2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}]$. For each array size, 10 runs are conducted for both the naive 2D DFT and the 2D FFT algorithms.

2 Testing

2.1 Benchmark Testing

To verify the correctness of our 2D Fourier Transform algorithms (naive 2D DFT and 2D FFT) implementations, we benchmarked our results against a known valid implementation from the numpy library: *numpy.fft.fft2()*. We generated a matrix of random values, and we computed the 2D Fourier Transform algorithm on this array using both our custom models (naive 2D DFT and 2D FFT) and numpy's model. We then computed the difference in magnitude between corresponding cells with the reference model (shown in Figure 1).

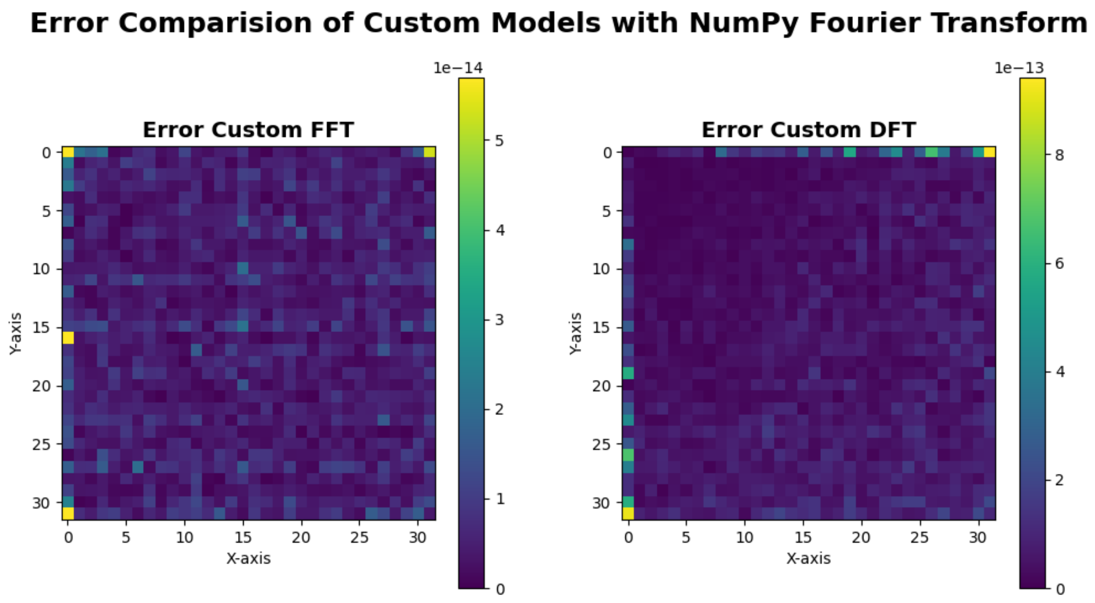

Figure 1: Absolute Cell-Wise Differences Between NumPy's FFT and Our Custom Implementations

As the absolute cell-wise differences for both models (2D naive DFT and 2D FFT) are strictly smaller than 1×10^{-12} for every cell with respect to the numpy implementation, we conclude that our naive 2D DFT and 2D FFT produce expected results.

Similarly, to verify that our naive 2D IDFT and 2D IFFT behaved as expected, we benchmarked it against the Inverse Fourier Transform from numpy: *numpy.fft.ifft2()*. We also created a random matrix, computed its transform, and we then computed the 2D Inverse Fourier Transform algorithms on this transformed matrix using both our custom models (naive 2D DFT and 2D FFT) and numpy's model. Finally, we calculated the differences between numpy's results and our custom models' results (shown in Figure 2).

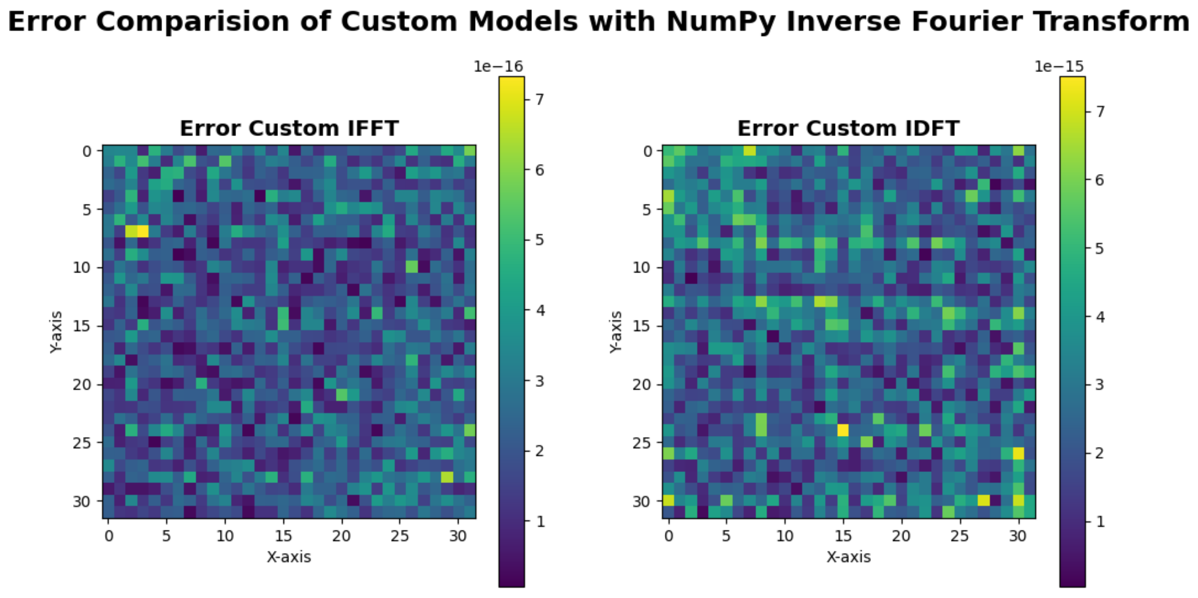

Figure 2: Absolute Cell-Wise Differences Between NumPy's IFFT and Our Custom Implementations

As the absolute cell-wise differences for both models (2D naive IDFT and 2D IFFT) are strictly smaller than 1×10^{-14} for every cell with respect to the numpy implementation, we conclude that our naive 2D IDFT and 2D IFFT produce expected results.

For the simpler 1D versions of the algorithms tested above (naive 1D DFT, 1D FFT, naive 1D IDFT, and 1D IFFT), the test consisted of running the 1D algorithm on arrays of randomly generated data and cross-checking with the 1D numpy Fourier transform algorithms (*numpy.fft.fft()* and *numpy.fft.ifft()*).

2.2 Fast Mode

In order to verify that our custom implementation of the 2D FFT worked as expected for an image signal in the Fast Mode, we compared it to Numpy's implementation of the 2D FFT. Figure 4 shows the log-scaled 2D FFT magnitudes of our custom implementation and numpy's implementation side by side (this comparison plot is implemented in the function *plot_testing_fast_mode()* under the *FastMode.py* file). The two log-scaled 2D FFT plots are visibly the same, showing that our implementation of the 2D FFT outputs expected values.

2.3 Denoising Mode

For denoising, the testing involved comparing our own denoising mode with scipy's gaussian denoising function: *scipy.ndimage.gaussian_filter()*. Figure 3 shows the results of denoising with our custom denoising model (removing frequencies outside the range $[-\frac{\pi}{7}, \frac{\pi}{7}]$) and scipy's gaussian denoising model with $sigma = 2.3$. The two denoised images are visibly similar apart from a slight difference in lighting (the gaussian filter shows a slightly lighter image), showing that our implementation of the denoising mode seems to output a correctly denoised image.

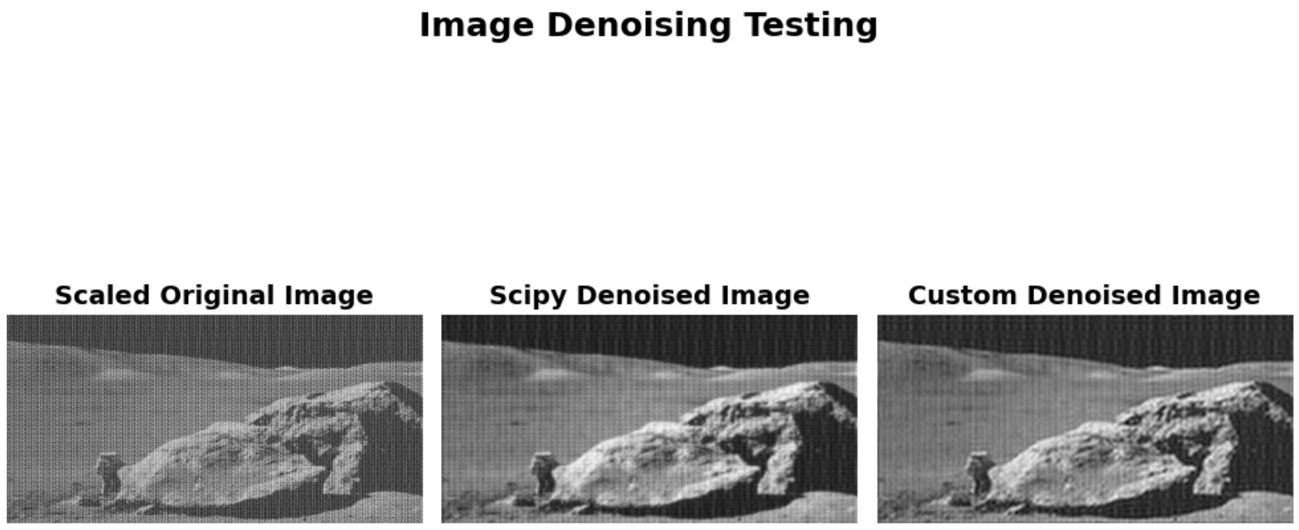

Figure 3: Comparing the fft.py Denoise mode with Scipy's gaussian filter denoising

2.4 Compression Mode

We tested the compression mode by simply making sure that the correct number of FFT coefficients were set to 0 for different compressed images at multiple levels of compression (40%, 75%, 90%, 99%, and 99.9%).

2.5 Plotting Mode

For runtime analysis, we made sure that the plot showing the runtime with respect to the 2D array sizes justified the expected runtime complexities of the 2D naive DFT and 2D FFT as explained in Section 3. Namely, the 2D naive DFT and 2D FFT should take $\mathcal{O}(N^3)$ and $\mathcal{O}(N^2 \log N)$ time to complete respectively where N is the size of the 2D array (length of row or column).

In Figure 10, we observe that the 2D FFT execution is much faster at large sizes than the naive 2D DFT. We can also notice that the naive 2D DFT exhibits cubic growth whereas the 2D FFT grows much slower ($\mathcal{O}(N^2 \log N)$), confirming the time complexity of our implemented Fourier Transform algorithms.

3 Runtime Analysis

3.1 Discrete Fourier Transform - 1D

From equation 1, we observe that for a discrete signal in the time-domain containing **N** discrete values, the Discrete Fourier Transform requires computing a sum of complex exponential terms for each of the **N** frequencies. Specifically, for each frequency coefficient X_k (where $k = 0, 1, 2, \dots, N-1$), one must perform a summation over all the time-domain samples x_n (where $n = 0, 1, 2, \dots, N-1$). This is shown more explicitly in equation 6.

As such, to obtain a single frequency coefficient:

$$X_k \quad \text{for some } k \in \{0, 1, 2, \dots, N-1\} \tag{12}$$

one must perform N operations. From this, it follows that to compute all N frequency coefficients, one must repeat this N times. Hence, the time complexity of the naive 1D DFT is $\mathcal{O}(N^2)$.

3.2 Discrete Fourier Transform - 2D

Given a two-dimensional input signal of $N \times M$ size and the following equation:

$$X_{k,l} = \sum_{m=0}^{N-1} \sum_{n=0}^{M-1} x_{m,n} \cdot e^{-i\frac{2\pi}{M}km} \cdot e^{-i\frac{2\pi}{N}ln} = \sum_{m=0}^{N-1} \left(\sum_{n=0}^{M-1} x_{m,n} \cdot e^{-i\frac{2\pi}{M}km} \right) \cdot e^{-i\frac{2\pi}{N}ln} \tag{13}$$

for $k = 0, 1, 2, \dots, M-1$ and $l = 0, 1, 2, \dots, N-1$.

From Equation 13, we observe that to compute the naive 2D DFT, one must apply the naive 1D DFT twice: once along the rows and once along the columns. Hence, we can derive from Section 3.1 that the time complexity can be expressed as:

$$M \times \mathcal{O}(N^2) + N \times \mathcal{O}(M^2) = \mathcal{O}(MN^2) + \mathcal{O}(NM^2) = \mathcal{O}(MN^2 + NM^2)$$

And if we define $\mathbf{P} = \max(N, M)$, then the time complexity can be approximated as:

$$\mathcal{O}(MN^2 + NM^2) \approx \mathcal{O}(P^3)$$

Thus, the time complexity of the 2D naive DFT is $\mathcal{O}(P^3)$.

3.3 Fast Fourier Transform - 1D

Let $t(x)$ be the time or number of instructions to execute the 1D FFT, and N be the length of the input signal. To determine the time complexity of the recursive 1D FFT algorithm, the following recurrence relation must be solved:

$$\begin{aligned} &t(N) = c_1 N + 2t(N/2), \quad c_1 \in \mathbb{N} \\ &t(4) = c_2, \quad c_2 \in \mathbb{N} \end{aligned} \tag{14}$$

The term $2t(N/2)$ in the recurrence is obtained from the two recursive calls in the 1D FFT algorithm (for even and odd sub-signals), and the $c_1 N$ term is obtained from the step of merging odd and even results from the recursive calls to obtain the final transformed signal (Equation 10 represents the merging step of odd and even recursive calls to get the transformed signal for $k \in [0, N/2-1]$, and Equation 11 represents the merging step to get the transformed signal for $k \in [N/2, N-1]$). $t(4)$ is the base case in the *fft_1D()* function. We chose to stop the recursion when the length of the sub-signals reaches 4 at which point, the FFT is obtained using the precomputed $\mathbf{W}$ matrix from Equation 4 (which takes constant time c_2).

The recurrence can be solved by back substitution:

$$\begin{aligned} t(N) &= c_1 N + 2t(N/2) \\ t(N) &= c_1 N + 2(c_1 N/2 + 2t(N/4)) \\ t(N) &= 2c_1 N + 4t(N/4) \\ t(N) &= rc_1 N + 2^r t(\frac{N}{2^r}) \\ t(N) &= (\log_2(N) - 2)c_1 N + 2^{\log_2(N)-2} t(\frac{N}{2^{\log_2(N)-2)}}), \quad \text{for } r = \log_2(N) - 2 \\ t(N) &= (\log_2(N) - 2)c_1 N + \frac{N}{4} t(4) \\ t(N) &= (\log_2(N) - 2)c_1 N + \frac{c_2}{4} N \end{aligned} \tag{15}$$

Therefore, the time complexity of the 1D FFT $\mathcal{O}(t(N)) = \mathcal{O}(N \log N)$.

3.4 Fast Fourier Transform - 2D

Given a two-dimensional input signal of N $\times$ M size (N rows and M columns). The 2D FFT performs the 1D FFT across all rows first, then across all columns. So the time complexity for the first set of transformations over the rows would be $\mathcal{O}(NM \log M)$, and the time complexity for the second set of transformations over the columns would be $\mathcal{O}(MN \log N)$. Therefore the time complexity of the 2D FFT is $\mathcal{O}(NM \log M + MN \log N)$ or $\mathcal{O}(P^2 \log P)$, for $P = \max(N, M)$.

4 Experimentation

4.1 FFT Transform

Figure 4 shows the 2D FFT implementation of the *fft.py* program alongside the *numpy.fft.fft2()* function results in log-scaled plots. Both 2D FFT implementations seem to give the same complex-valued coefficients visually. A more rigorous numerical comparison done in Section 2.1 shows that the cell-wise difference in magnitude for all the coefficients is less than 1×10^{-12}.

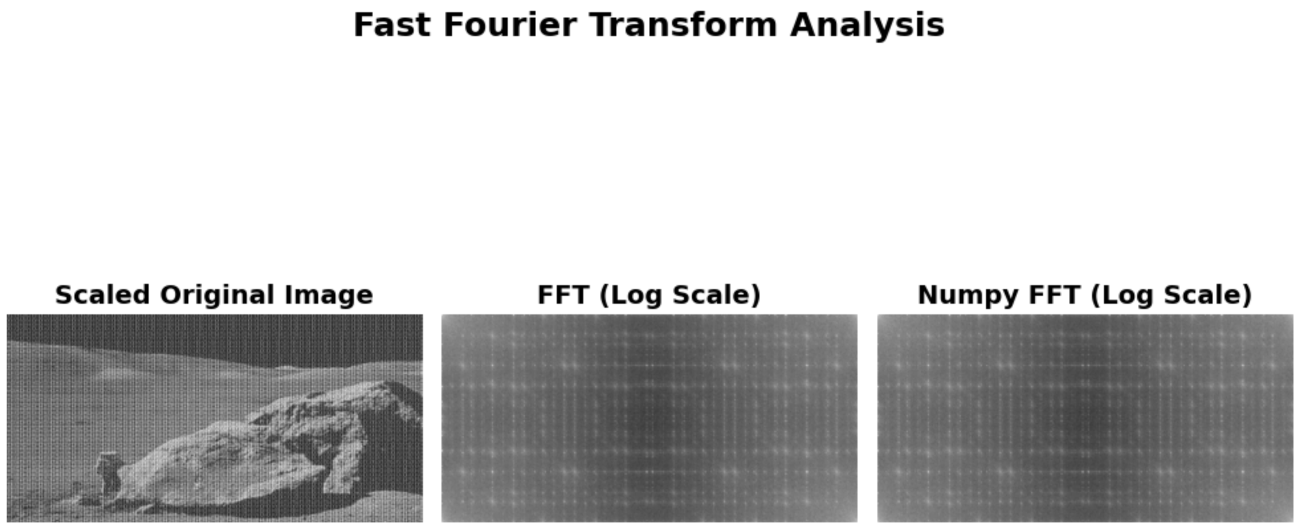

Figure 4: Comparing Custom FFT with Numpy's version

4.2 Denoising

Denoising aims to remove noise from a signal. This is done by removing high frequencies from the signal. As a result, the denoised signal exhibits smoother features.

Let the matrix $\mathbf{X_{FFT}}$ be a complex coefficient matrix obtained by taking the 2D FFT of a 5×5 matrix and then applying *numpy.fft.fftshift()* to shift low frequencies to the center.

$$\mathbf{X_{FFT}} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} \\ x_{51} & x_{52} & x_{53} & x_{54} & x_{55} \end{bmatrix}$$

To denoise, we need to set to 0 some high frequency coefficients of the signal. Since $\mathbf{X_{FFT}}$ is shifted to have the high frequencies on the outside, we can denoise by setting to 0 all coefficients outside a rectangular region in the middle of the matrix.

$$\mathbf{X_{FFT}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & x_{22} & x_{23} & x_{24} & 0 \\ 0 & x_{32} & x_{33} & x_{34} & 0 \\ 0 & x_{42} & x_{43} & x_{44} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The smaller the centered rectangle of non-zero coefficients (equivalent to retaining a smaller percentage of the signal), the smoother the image becomes, with fewer distinct features remaining. Figures 5,6, and 7, 8 show a progressively increasing level of denoising (i.e., a decrease in the percentage of the original frequencies retained). Table 2 summarizes the ranges of frequencies kept and the percentage of original coefficients for Figures 5,6, 7, and 8. By trial and error, we found that by keeping a range of $[-\frac{\pi}{7}, \frac{\pi}{7}]$ and setting the rest of the coefficients for frequencies outside of the range to 0 gave the best result for denoising. We also tried thresholding the magnitude of the coefficients (similarly to what is done for compression). Still, it didn't seem to give denoised images with features as smooth as when considering ranges of frequencies to keep/set to 0.

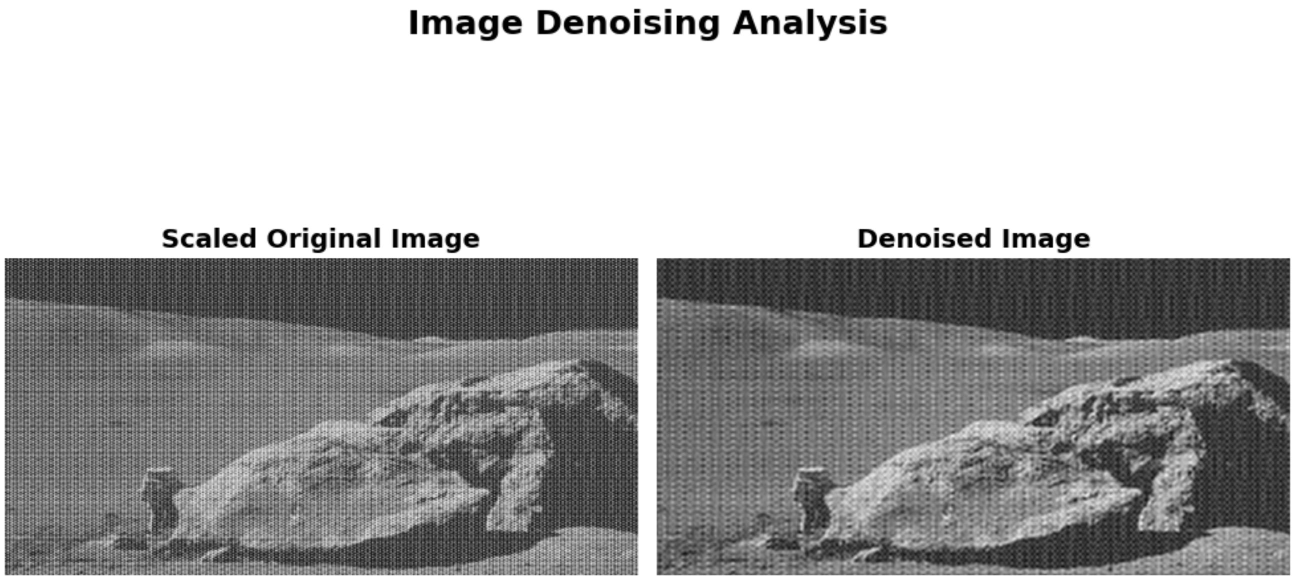

Figure 5: Denoised Image keeping frequencies in $[-\frac{\pi}{4}, \frac{\pi}{4}]$ (6.25% non-zero coefficients)

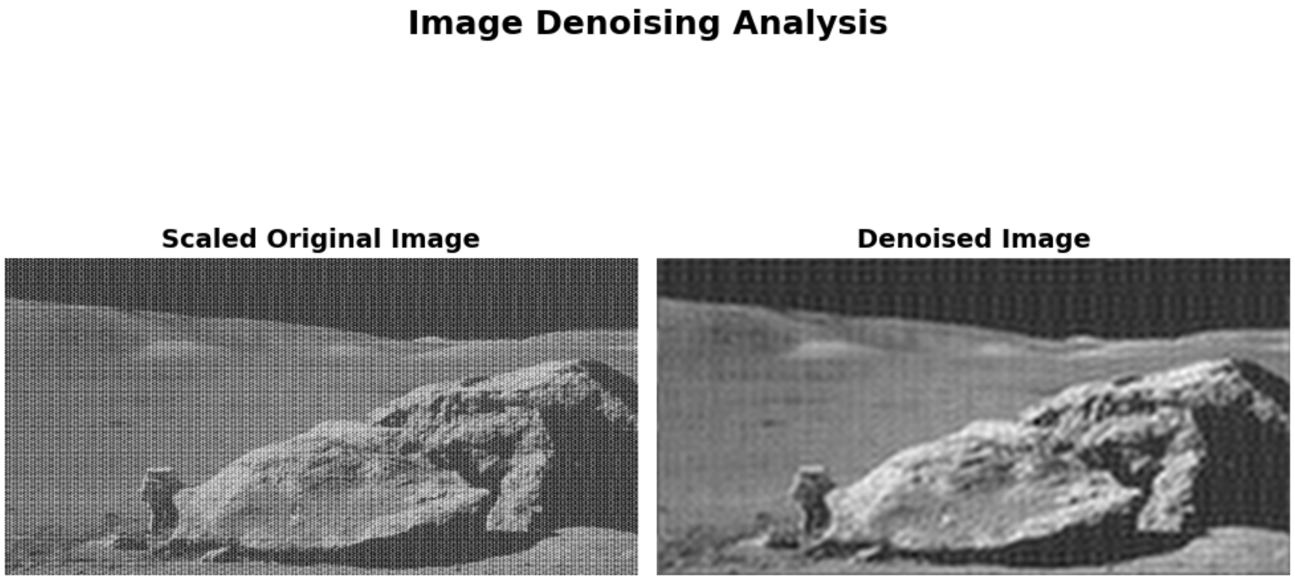

Figure 6: Denoised Image keeping frequencies in $[-\frac{\pi}{8}, \frac{\pi}{8}]$ (1.56% non-zero coefficients)

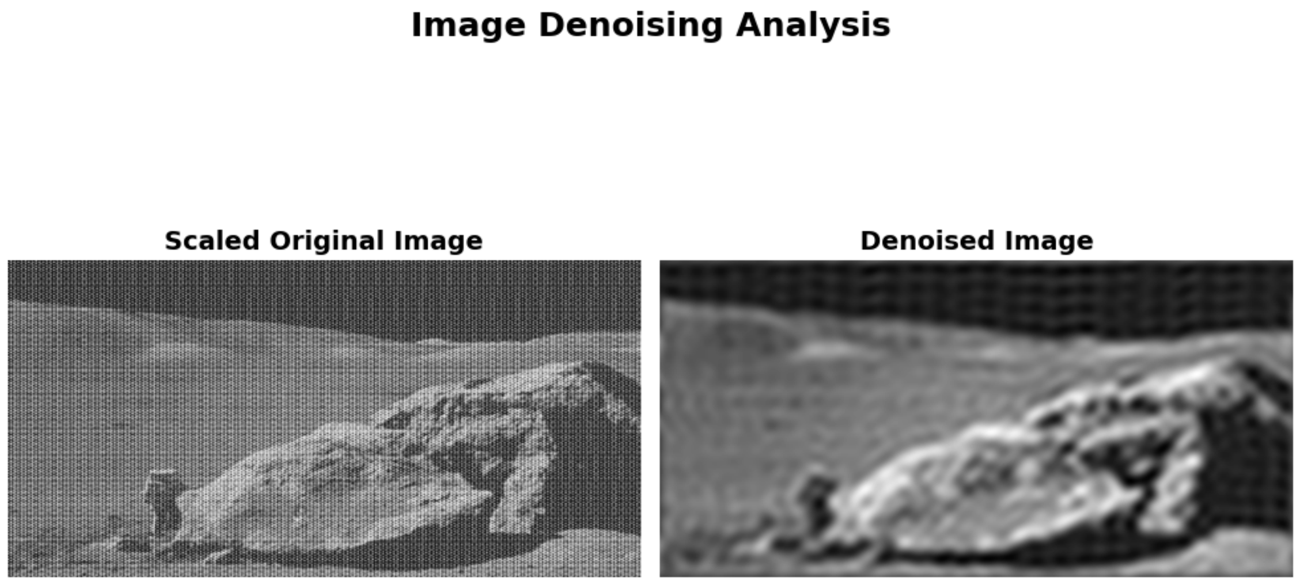

Figure 7: Denoised Image keeping frequencies in $[-\frac{\pi}{16}, \frac{\pi}{16}]$ (0.39% non-zero coefficients)

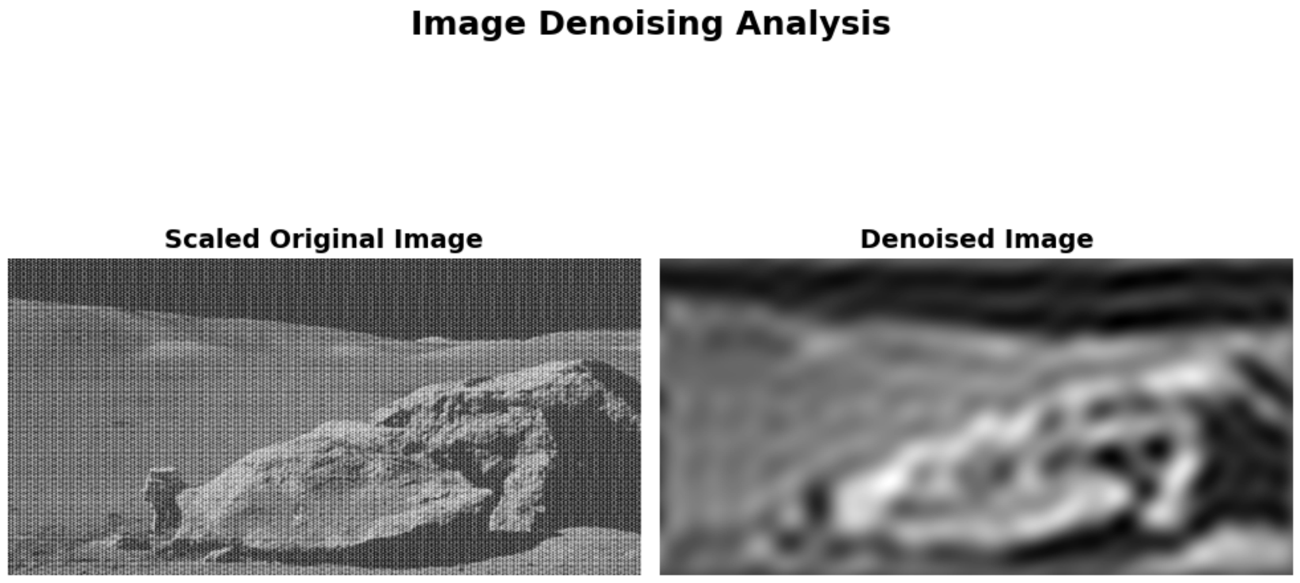

Figure 8: Denoised Image keeping frequencies in $[-\frac{\pi}{32}, \frac{\pi}{32}]$ (0.10% non-zero coefficients)

Range of Frequencies Kept	**Percentage of Original Coefficients (%)**
$[-\frac{\pi}{4}, \frac{\pi}{4}]$	6.25
$[-\frac{\pi}{8}, \frac{\pi}{8}]$	1.56
$[-\frac{\pi}{16}, \frac{\pi}{16}]$	0.39
$[-\frac{\pi}{32}, \frac{\pi}{32}]$	0.10

Table 1: Ranges of Non-Zero Frequency Coefficients and their Corresponding Percentages

4.3 Compression

The two techniques mentioned in the assignment handout for compression were experimented (thresholding the coefficients' magnitude to discard coefficients with low magnitude, and keeping low-frequency coefficients with a fraction of high-frequency coefficients). We ended up choosing to apply only the thresholding of the coefficients' magnitude without filtering out the high frequencies (as it is done in denoising) to have a separation of concerns for the two operations of compression and denoising. Compression would be solely concerned with reducing the size of the image without filtering. If a user desires to apply both compression and denoising, then they would have the ability to perform both modes on the image. However, we believe that they should also be able to compress the image without necessarily removing the noise.

Figure 9 shows the Moon landing picture compressed at different percentages. Table 2 shows the corresponding size for each compressed image in terms of non-zero coefficients left after compression. It is interesting to notice that the quality of the reconstruction is almost as good as the uncompressed image up until 90% compression levels. Up until 90% compression, the blur of the compression is hardly noticeable. There is only a (slight) loss of contrast between very light and very dark areas in the image that is visible at 75% and 90% compression. The 99% and 99.9% compressed images however show a clear blur (which is so extreme for 99.9% compression that the original content of the image is hard to decipher).

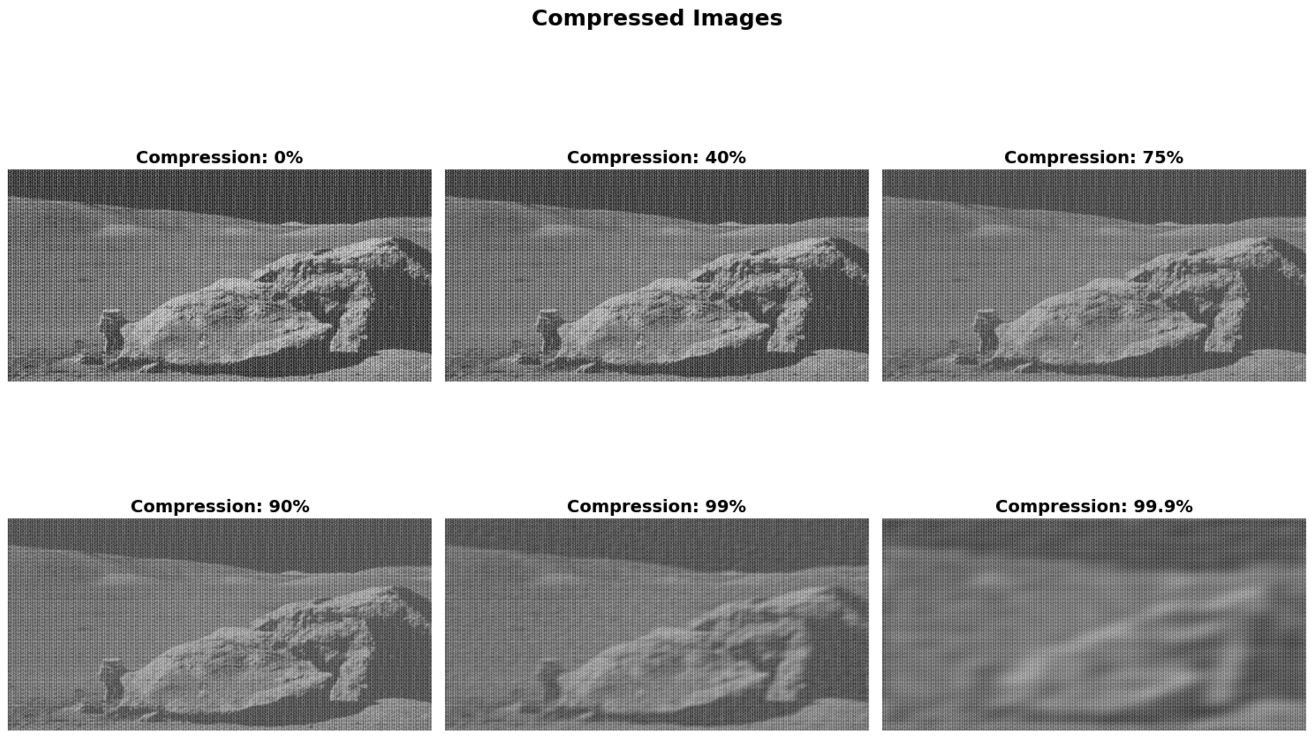

Figure 9: Compressed images of the Moon landing picture at 0%, 40%, 75%, 90%, 99%, and 99.9% levels

Compression Level (%)	Number of Non-Zero Coefficients
0.00	524,288
40.00	314,573
75.00	131,072
90.00	52,429
99.00	5,243
99.90	525

Table 2: Number of Non-Zero Coefficients per Compression Level

4.4 Plotting runtime

Figure 10 shows that the Naive Fourier Transform (naive 2D DFT, red trend) runtime increases more rapidly than the Fast Fourier Transform (2D FFT, blue trend) runtime with respect to 2D array size. The plot is consistent with our findings in Sections 3.1 and 3.2, where we determined that the runtime of the naive 2D DFT is $\mathcal{O}(N^3)$ as the red curve in the figure seems to have a cubic growth trend.

Similarly, in Sections 3.3 and 3.4, we found that the 2D FFT has a time complexity of $\mathcal{O}(N^2 \log N)$, which is consistent with the growth trend of the blue curve.

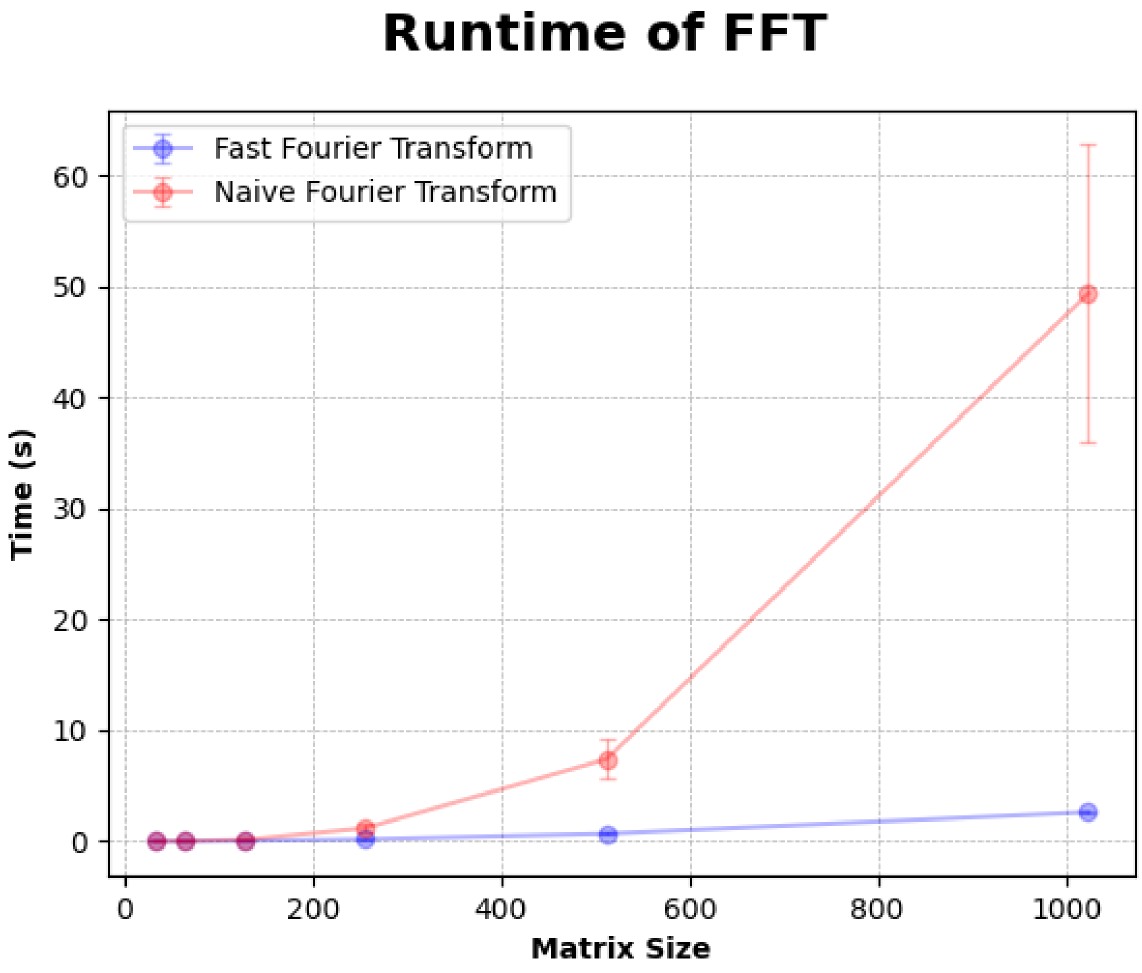

Figure 10: Runtime Analysis of DFT and FFT for Increasing Array Sizes

References

[1] Steven G. Johnson and Matteo Frigo. Implementing ffts in practice, 2008.