# ECSE 316 Assignment 2 Report

Group 6
Jason Shao 261113937
Ben Boudana 261119289
March 30th, 2025

## 1.Introduction

Fourier analysis enables the decomposition of signals into sinusoidal components, providing a powerful framework for image processing. In this project, we implement both the naive Discrete Fourier Transform (DFT) and the efficient Cooley-Tukey Fast Fourier Transform (FFT) for one-dimensional signals, and extend these methods to two dimensions for image applications. By transforming images into the frequency domain, we can isolate low-frequency information—which represents the overall structure—from high-frequency details, often corresponding to noise. This separation facilitates effective denoising and compression techniques. The report outlines the design, implementation, and evaluation of these methods, highlighting the performance advantages of the FFT over the brute-force approach.

## 2.Program Design

The program is organized in a modular fashion by first defining all the necessary helper functions to compute the Fourier Transformation and put together later in the 4 image processing modes function. The design includes several key components:

1. Core Fourier Transform Functions

- Naive DFT and Inverse DFT:
  naive_dft() and naive_idft() are the two functions we defined to compute the Fourier transform directly using the mathematical definition. They both have a running time complexity of O(N^2) and they are later used in Mode 4 as a baseline for correctness and performance comparisons.

  Code Implementation:

```python
def naive_dft(x):
    """
    Compute the Discrete Fourier Transform (DFT) of 1D array x using the naive method.
    """
    N = x.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(-2j * np.pi * k * n / N)
    return np.dot(M, x)

def naive_idft(X):
    """
    Compute the inverse DFT of 1D array X using the naive method.
    """
    N = X.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(2j * np.pi * k * n / N)
    return np.dot(M, X) / N
```

Figure 2.1: Code Implementation of naive_dft() and naive_idft()

- Cooley-Tukey FFT and Inverse FFT:
  fft() and ifft() are the two functions we defined that uses the divide-and-conquer strategy to reduce the time complexity to O(NlogN). This method works by splitting the signal into even and odd indexed parts and then combining the results with precomputed twiddle factors.

  Code Implementation:

```python
def fft(x):
    """
    Compute the FFT of 1D array x using the Cooley-Tukey algorithm.
    Assumes the length of x is a power of 2.
    """
    N = x.shape[0]
    if N == 1:
        return x
    if N % 2:
        return naive_dft(x)
    even = fft(x[::2])
    odd  = fft(x[1::2])
    factor = np.exp(-2j * np.pi * np.arange(N) / N)
    return np.concatenate([even + factor[:N//2] * odd,
                           even - factor[:N//2] * odd])

def ifft(X):
    """
    Compute the inverse FFT of 1D array X.
    """
    return np.conjugate(fft(np.conjugate(X))) / X.shape[0]
```

Figure 2.2: Code Implementation of fft() and ifft()

- 2D Fourier Transform:
  fft2d() and ifft2d() are defined by applying the 1D FFT routines first along the rows and then along the columns of the input image. This approach enables the transformation of 2D signals (images) into the frequency domain.

  Code Implementation:

```python
def fft2d(img):
    """
    Compute the 2D FFT of a 2D array (image) by applying the 1D FFT
    to each row then each column.
    """
    M, N = img.shape
    F = np.zeros((M, N), dtype=complex)
    for m in range(M):
        F[m, :] = fft(img[m, :])
    F2 = np.zeros((M, N), dtype=complex)
    for n in range(N):
        F2[:, n] = fft(F[:, n])
    return F2

def ifft2d(F):
    """
    Compute the inverse 2D FFT of a 2D array by applying the 1D IFFT
    to each column then each row.
    """
    M, N = F.shape
    f = np.zeros((M, N), dtype=complex)
    for n in range(N):
        f[:, n] = ifft(F[:, n])
    f2 = np.zeros((M, N), dtype=complex)
    for m in range(M):
        f2[m, :] = ifft(f[m, :])
    return f2
```

Figure 2.3: Code Implementation of fft3d() and ifft2d()

2. Image Processing Modes

The program supports multiple operational modes, selectable via command-line arguments, to showcase different applications of the Fourier transform:

- Mode 1 – Fast Display:
  After the image is loaded a padding is added to the image so that the image has height and width that are both a power of 2. Then its 2D FFT is computed. The output displays the original image alongside its logarithmically scaled Fourier transform. This mode demonstrates how frequency components are distributed within the image.

  Code Implementation:

```python
def run_mode1(image_path):
    """
    Mode 1: Fast mode – Convert the image to FFT form and display
    the original image alongside its Fourier transform (log-scaled).
    """
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if img is None:
        print("Error loading image.")
        return
    img_padded = pad_to_power_of_two(img)
    F = fft2d(img_padded)
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.imshow(img_padded, cmap='gray')
    plt.title("Original Image")
    plt.axis('off')
    plt.subplot(1, 2, 2)
    # Using logarithm of the magnitude for better visibility
    plt.imshow(np.log(np.abs(F) + 1), cmap='gray', norm=LogNorm())
    plt.title("FFT (log-scaled)")
    plt.axis('off')
    plt.tight_layout()      You, 6 days ago • finish fft
    plt.show()
```

Figure 2.4: Code Implementation of run_mode1()

- Mode 2 – Denoising:
  After loading the image, the padding is added and then its FFT is computed. The high-frequency components are suppressed by applying a low-pass filter (masking out coefficients beyond a defined cutoff). The inverse FFT is then computed to reconstruct the denoised image, which is displayed next to the original. This mode highlights how noise can be reduced by filtering in the frequency domain.

  Code Implementation:

```python
def run_mode2(image_path):
    """
    Mode 2: Denoise – Apply FFT, zero high frequencies, and display
    the original image alongside its denoised version.
    """
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if img is None:
        print("Error loading image.")
        return
    img_padded = pad_to_power_of_two(img)
    denoised, nonzeros, total = denoise_image(img_padded, cutoff_ratio=0.1)
    print("Denoising: Keeping", nonzeros, "out of", total, "coefficients (",
          f"{nonzeros/total*100:.2f}%)")
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.imshow(img_padded, cmap='gray')
    plt.title("Original Image")
    plt.axis('off')
    plt.subplot(1, 2, 2)
    plt.imshow(denoised, cmap='gray')      You, 6 days ago • finish fft
    plt.title("Denoised Image")
    plt.axis('off')
    plt.tight_layout()
    plt.show()
```

Figure 2.5: Code Implementation of run_mode2()

- Mode 3 – Compression:
  The image undergoes FFT-based compression by thresholding the Fourier coefficients. For various compression levels, a percentage of the smallest coefficients are set to zero, and the image is reconstructed using the inverse FFT. A subplot displays the reconstructed images at different compression ratios along with the corresponding counts of nonzero coefficients.

  Code Implementation:

```python
def run_mode3(image_path):
    """
    Mode 3: Compress - Compress the image at various levels by zeroing
    Fourier coefficients and display the 6 reconstructed images.
    Also print the number of nonzero coefficients for each level.
    """
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if img is None:
        print("Error loading image.")
        return
    img_padded = pad_to_power_of_two(img)
    levels = [0, 50, 70, 90, 95, 99.9]
    compressed_images, nonzero_counts = compress_image(img_padded, levels)

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))
    for i, ax in enumerate(axes.flatten()):
        ax.imshow(compressed_images[i], cmap='gray')
        ax.set_title(f"{levels[i]}% comp. | Nonzeros: {nonzero_counts[i]}")
        ax.axis('off')
    plt.tight_layout()
    plt.show()
```

Figure 2.6: Code Implementation of run_mode3()

- Mode 4 – Runtime Benchmarking:
  To compare the performance of the naive DFT and the FFT, the program generates random square arrays of various sizes (e.g., 8×8, 16×16, etc.) and measures the execution time of both methods over multiple runs. The average runtimes and standard deviations are plotted, providing visual insight into the efficiency improvements offered by the FFT.

  Code Implementation:

```python
def run_mode4():
    """
    Mode 4: Plot runtime graphs for the naive DFT and FFT algorithms.
    For various square array sizes (powers of 2), run each method 10 times,
    compute mean runtimes and standard deviations, and plot with error bars.
    Note: Since the naive 2D DFT is very slow for large sizes, we restrict
    sizes to small arrays.
    """
    sizes = [8, 16, 32, 64, 128, 256, 512]
    num_runs = 10
    naive_times = []
    fft_times = []
    size_list = []

    def naive_2d(arr):
        M, N = arr.shape
        F = np.zeros((M, N), dtype=complex)
        for m in range(M):
            F[m, :] = naive_dft(arr[m, :])
        F2 = np.zeros((M, N), dtype=complex)
        for n in range(N):
            F2[:, n] = naive_dft(F[:, n])
        return F2

    for n in sizes:
        arr = np.random.rand(n, n)
        t_naive = []     You, 6 days ago • finish fft
        t_fft = []
        for _ in range(num_runs):
            start = time.time()
            naive_2d(arr)
            t_naive.append(time.time() - start)
            start = time.time()
            fft2d(arr)
            t_fft.append(time.time() - start)
        mean_naive, std_naive = np.mean(t_naive), np.std(t_naive)
        mean_fft, std_fft = np.mean(t_fft), np.std(t_fft)
        naive_times.append((mean_naive, std_naive))
        fft_times.append((mean_fft, std_fft))
        size_list.append(n)
        print(f"Size {n}x{n}: Naive DFT = {mean_naive:.6f}s ± {std_naive:.6f}s, FFT = {mean_fft:.6f}s ± {std_fft:.6f}s")

    plt.errorbar(size_list, [t[0] for t in naive_times], yerr=[t[1] for t in naive_times],
                 fmt='o-', label='Naive DFT')
    plt.errorbar(size_list, [t[0] for t in fft_times], yerr=[t[1] for t in fft_times],
                 fmt='o-', label='FFT')
    plt.xlabel("Array Size (n x n)")
    plt.ylabel("Runtime (seconds)")
    plt.title("Runtime Comparison: Naive DFT vs. FFT")
    plt.legend()
    plt.show()
```

Figure 2.7: Code Implementation of run_mode4()

## 3. Command-Line Interface

The program uses Python's argparse module to allow users to specify the mode of operation and the input image file path. This design choice provides flexibility for testing different functionalities without modifying the code.

Code Implementation:

```python
def main():
    parser = argparse.ArgumentParser(description="FFT Assignment Implementation")
    parser.add_argument("-m", "--mode", type=int, default=1,
                        help="Mode: 1-Fast display, 2-Denoise, 3-Compress, 4-Runtime plots")
    parser.add_argument("-i", "--image", type=str, default="default_image.png",
                        help="Image file path (default: default_image.png)")
    args = parser.parse_args()

    if args.mode == 1:
        run_mode1(args.image)
    elif args.mode == 2:
        run_mode2(args.image)
    elif args.mode == 3:
        run_mode3(args.image)
    elif args.mode == 4:
        run_mode4()
    else:
        print("Invalid mode selected. Choose a mode from 1 to 4.")

if __name__ == "__main__":
    main()
```

Figure 2.8: Code Implementation of main()

## 3.Testing

To validate the correctness of our implemented Fourier algorithms, we compared their output against the well-established Fast Fourier Transform (FFT) functions provided by the NumPy Python library.

Our testing procedure involved generating diverse input data to challenge our implementations. For the 1D algorithms (DFT and FFT), we tested with a constant signal (e.g., an array of all ones) and a single-frequency sine wave, comparing the resulting frequency spectra from our code and NumPy's. For the 2D algorithms (2D DFT and 2D FFT), we used a simple gradient image and an image with a single bright pixel, again comparing the transformed outputs from our functions against NumPy's fft2 function.

For each comparison, we computed the Fourier transform using both our implemented algorithm and the corresponding NumPy function. To account for potential minor differences due to floating-point arithmetic, we employed the numpy.allclose() function with a defined tolerance. By examining the frequency spectra for the 1D tests and the 2D transformed images, and by confirming numerical similarity using numpy.allclose(), we aimed to ensure our algorithms captured the expected frequency domain representations and transformations in a manner consistent with NumPy's established accuracy. The consistent agreement within the set tolerance across these varied test cases provided confidence in the adequate performance of our implementations.

## 4.Analysis

The 1D Discrete Fourier Transform (DFT) and its inverse (IDFT) perform O(n) operations for each of the n output points, resulting in a quadratic time complexity of $O(n^2)$. This direct computation becomes increasingly inefficient for larger signals. Extending this to a 2D DFT on an n×n image, by applying the 1D DFT separably along rows and then columns, leads to a cubic runtime complexity of $O(n^3)$.

The Fast Fourier Transform (FFT) and its inverse (IFFT) leverage a recursive approach to reduce the computational expense. By breaking down the problem into smaller, more manageable subproblems, the 1D FFT achieves a runtime of O(n*log(n)). This logarithmic scaling makes it significantly faster for larger inputs compared to the naive DFT. Similarly, the 2D FFT, by applying the 1D FFT along both rows and columns of an n×n image, attains a runtime complexity of $O(n^2*\log(n))$. This represents a substantial improvement over the $O(n^3)$ complexity of the naive 2D DFT, highlighting the efficiency gains from the FFT's divide-and-conquer strategy in both one and two dimensions.

# 5.Experiment

## a) FFT

For the first experiment step, we loaded the provided "moonlanding.png" image (python fft.py -i moonlanding.png in default fast mode) and computed its FFT using our implementation. This was visualized as a log-scaled magnitude plot next to the original image.

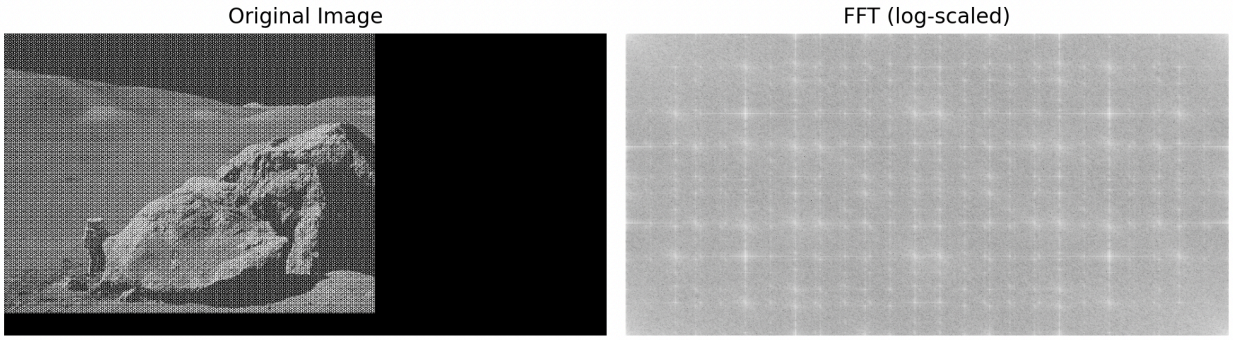


Figure 5.1: moonlanding.png and its log scaled FFT side by side

We then compared this visual representation of the frequency spectrum to the expected output of NumPy's built-in np.fft.fft2 function. This can be seen in figure 5.2 below.

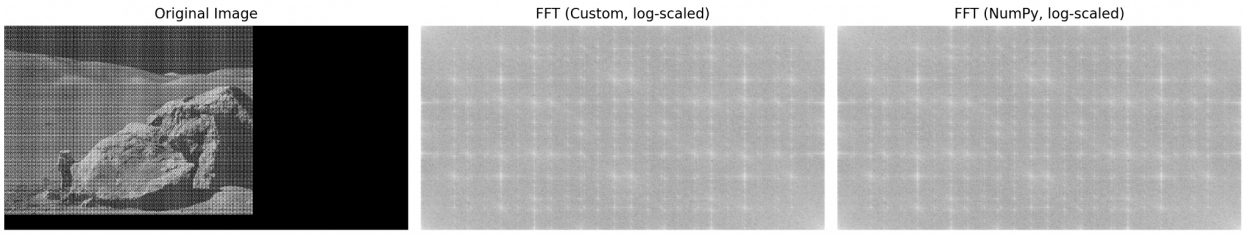


Figure 5.2: moonlanding.png log-scaled and using NumPy FFT built in function side by side with the original image and our implementation of FFT

The slight visual difference between your FFT and NumPy's likely stems from minor variations in their algorithms or floating-point precision. NumPy's highly optimized implementation might handle calculations or edge cases a bit differently, leading to subtly cleaner or more defined frequency magnitudes. Despite these small differences, the strong similarity confirms your implementation correctly captures the image's core frequency information.

## b) Denoise

fft.py mode 2 loads the "moonlanding.png" image and pads it to a power of two. It then computes the 2D FFT and filters it by retaining only the low-frequency components located in the top-left corner of the unshifted frequency spectrum, effectively removing high frequencies.

Finally, it calculates the inverse FFT of this filtered spectrum to produce a denoised image, which is displayed side-by-side with the original padded image for visual comparison of the noise reduction and any loss of detail in figure 5.3.

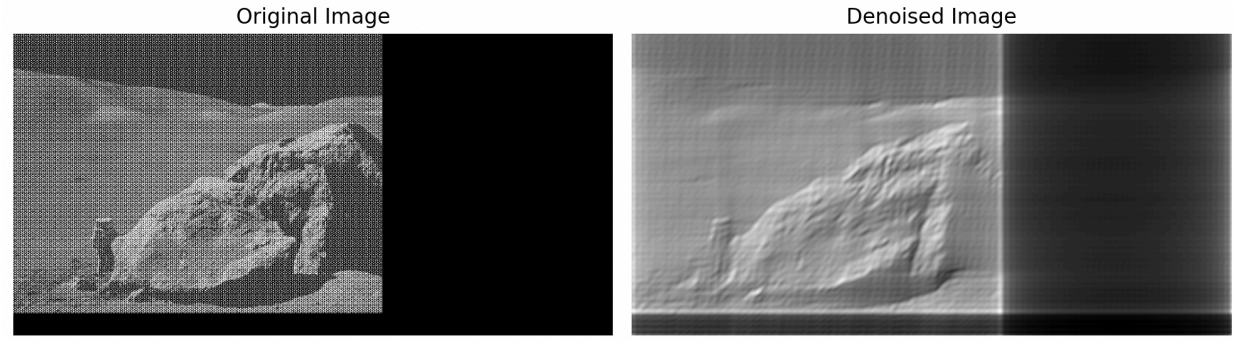


Figure 5.3: moonlanding.png denoised by removing all high frequencies

To remove low frequencies, the denoise_image function was modified to create a mask that is False in the top-left corner of the unshifted FFT spectrum (representing low frequencies) and True elsewhere. When this mask is applied to the FFT coefficients, the low-frequency components are set to zero, while the high-frequency components are retained. The inverse FFT of this modified spectrum then produces an image where the overall structure and smooth variations are suppressed, and edges, fine details, and noise might be emphasized.

The comparison will now show the original image next to this high-pass filtered version, illustrating the impact of removing the image's foundational low-frequency information, as seen in figure 5.4

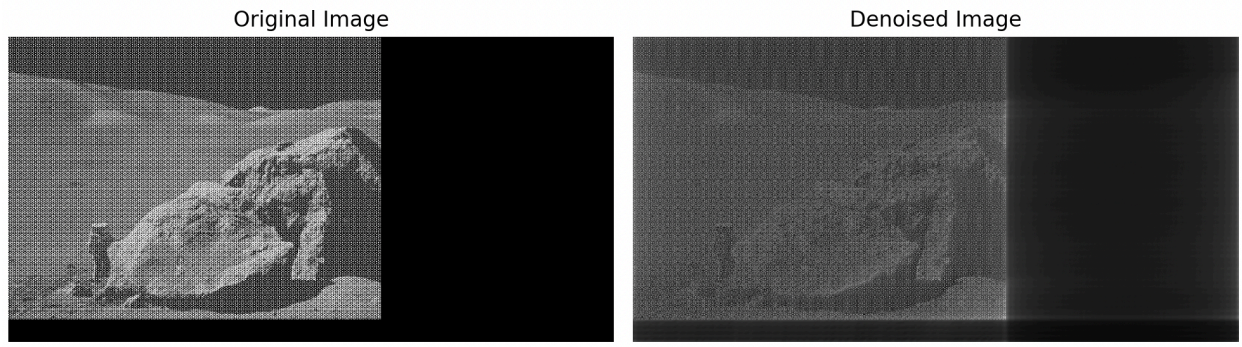

Figure 5.4: moonlanding.png denoised by removing all low frequencies

With the denoise_image function now modified, when run_mode2 is executed, it calculates the 2D FFT of the padded input image. It then identifies the magnitudes of all the frequency coefficients and determines a threshold such that only the top 5% of coefficients with the largest magnitudes are retained. All other coefficients, whose magnitudes fall below this threshold, are set to zero. Finally, the inverse FFT is computed on this sparse frequency spectrum, resulting in a denoised image that emphasizes the most prominent features while attempting to suppress weaker signals, including potential noise.

Figure 5.5 shows the original image alongside this denoised version, highlighting the effect of keeping only the strongest frequency components

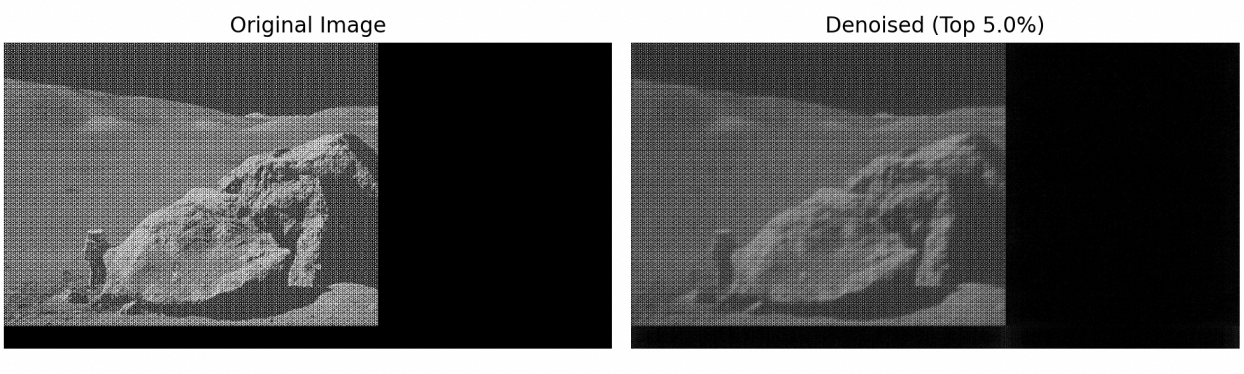


Figure 5.5: moonlanding.png denoised and keeping top 5% of coeffiecients

After some further modification, The denoise_image function now implements a two-step denoising process. First, it applies a frequency cutoff, retaining only the lowest 10% of the frequency components (in the top-left corner of the unshifted FFT spectrum) and discarding the rest. Second, from these remaining frequency coefficients, it calculates their magnitudes and keeps only the top 5% with the highest magnitudes, setting all other coefficients to zero. The inverse FFT is then computed on this heavily filtered frequency spectrum to produce the final denoised image. These results are shown in figure 5.6

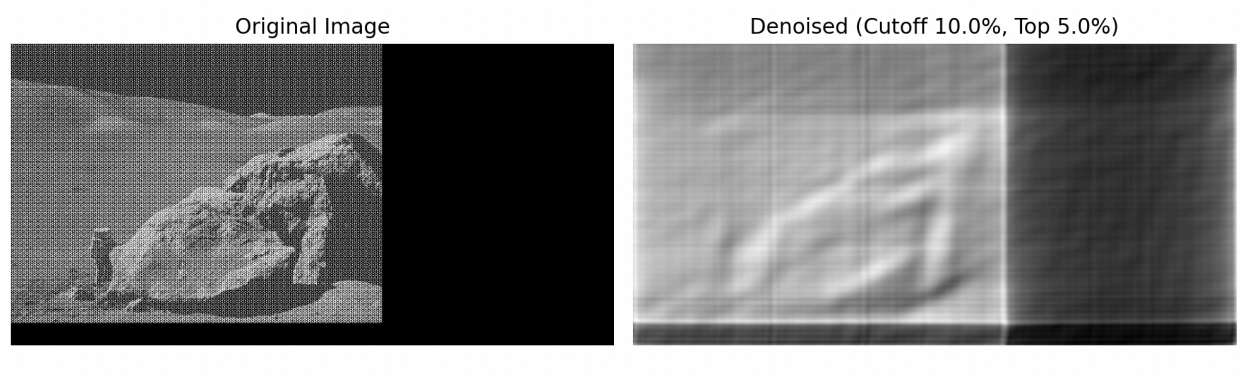


Figure 5.6: moonlanding.png denoised after a cutoff of 10% keeping top 5% of coefficients.

Keeping only the top 5% worked best because it aggressively filtered out low-magnitude frequencies, effectively targeting distributed noise while retaining the high-magnitude coefficients that represented the image's core, strong features. This resulted in significant noise reduction with minimal loss of essential image information, outperforming methods like broad frequency removal or range-based filtering in this specific case.

## c) Compress

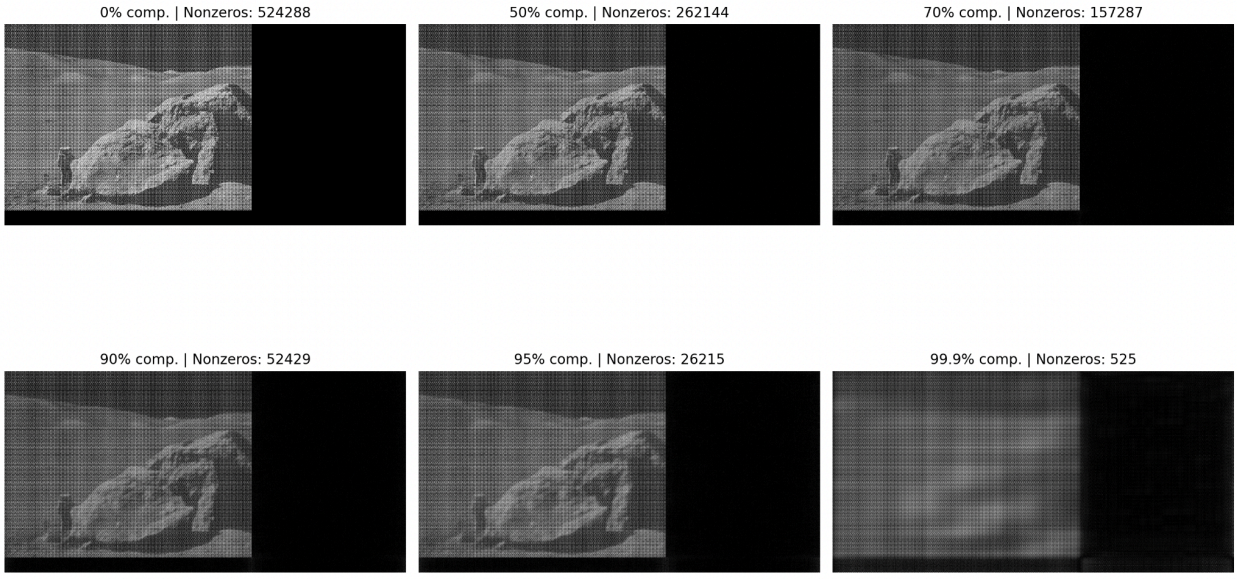


Figure 5.7: moonlanding.png at 0%, 50%, 70%, 90% 95% and 99.9% compression

At 0% compression, the file is largest, and the image is perfect, as seen in figure 5.7. As we compress more (to 90%), the file gets much smaller, and the image still looks okay with some detail missing. When we compress a lot (to 95-99.9%), the file becomes very small, but the image quality becomes very bad with many distortions. This shows that while high compression saves space, it loses a lot of image quality.

**d) Plotting for Runtime**

Using the problem sizes 8, 16, 32, 64, 128, 256, 512, we ran mode 4, the plotting functionality of fft.py 10 times to find the average runtimes of the naive DFT and FFT. Running the plotting function just once yielded the results of figure 5.8.

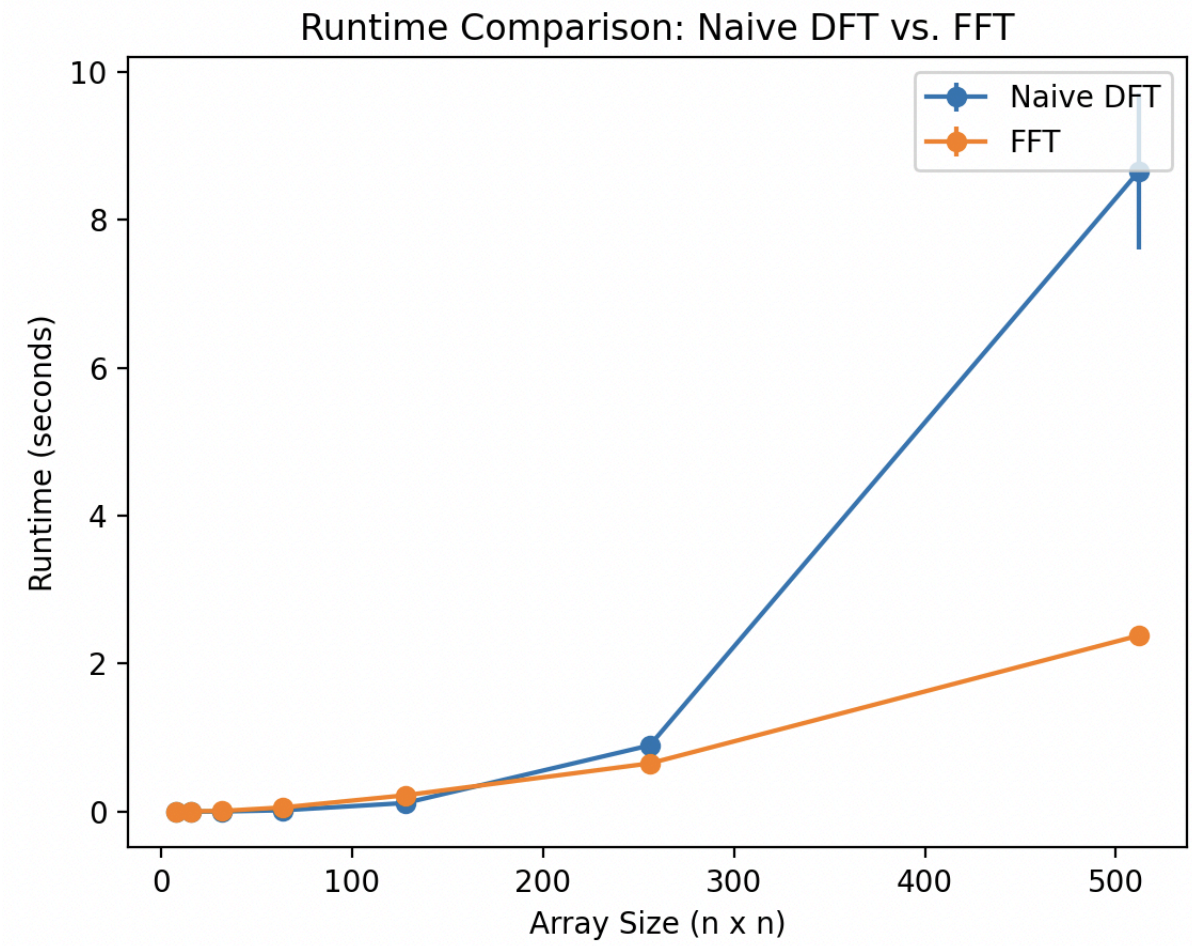


Figure 5.8: Naive DFT vs FFT initial execution

Based on figure 5.8, we can observe that FFT is much faster than DFT as the values increase. After running the same execution ten times, we achieved the following in figure 5.9 below.

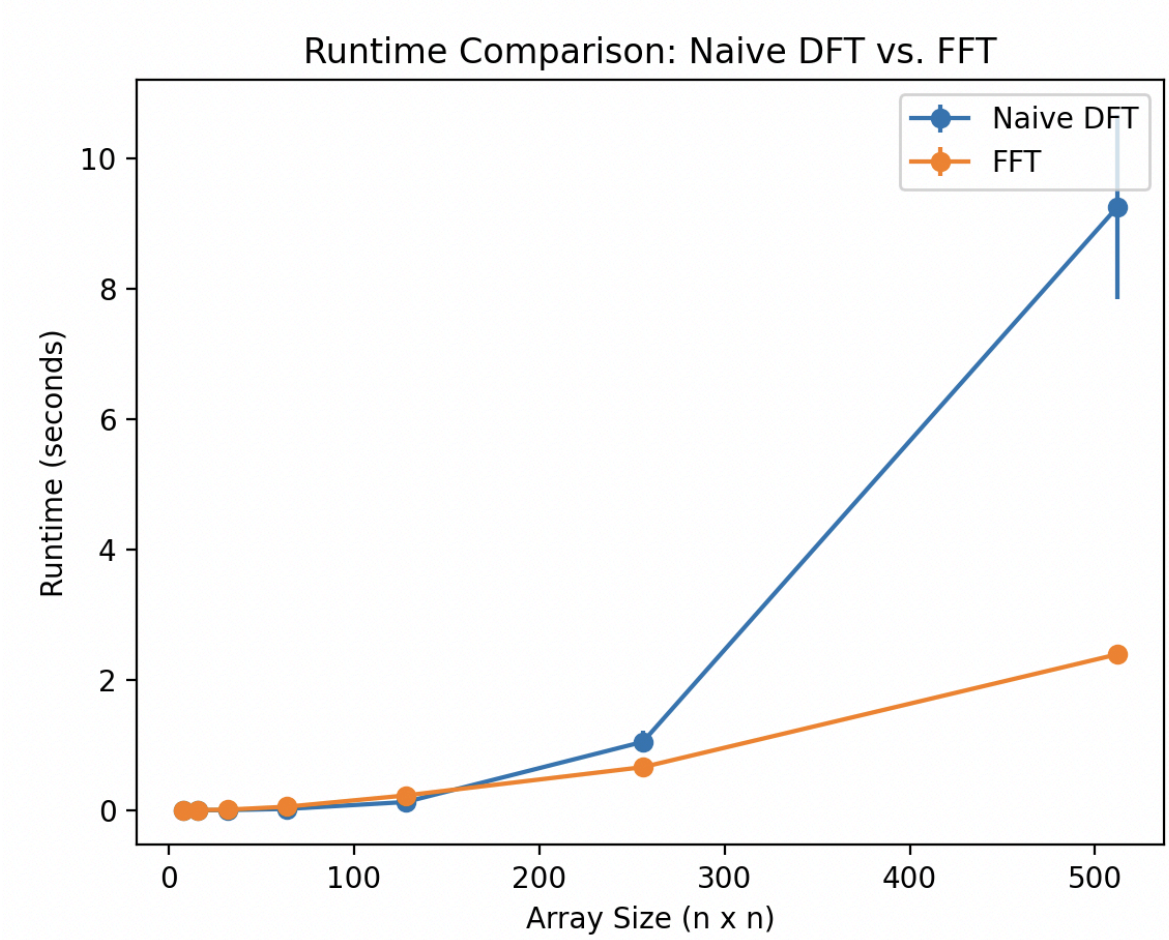


Figure 5.9: Naive DFT runtime vs FFT runtime after 10 runs

We ran the runtime experiment ten times for each problem size to obtain the mean running time and standard deviation, as presented in the table below. Similar to what might be observed in a single run, the data clearly indicates that the FFT algorithm consistently exhibits a faster runtime compared to the naive DFT across all tested problem sizes.

Table 1: Average runtimes of Naive DFT and FFT

| **Size n x n** | **Naive DFT Mean (s)** | **Naive DFT Std. Dev (s)** | **FFT Mean (s)** | **FFT Std. Dev (s)** |
|---|---|---|---|---|
| 8 x 8 | 0.000140 | 0.000163 | 0.000481 | 0.000008 |
| 16 x 16 | 0.000495 | 0.000119 | 0.003770 | 0.000778 |

| | | | | |
|---|---|---|---|---|
| 32 x 32 | 0.001557 | 0.000244 | 0.008950 | 0.001069 |
| 64 x 64 | 0.017531 | 0.004434 | 0.056247 | 0.004341 |
| 128 x 128 | 0.116200 | 0.012463 | 0.220388 | 0.004642 |
| 256 x 256 | 0.895086 | 0.041636s | 0.651744 | 0.004119 |
| 512 x 512 | 8.654022 | 1.055404 | 2.379261 | 0.060460 |

While the difference in execution time appears relatively small for the initial, smaller array dimensions, it becomes increasingly pronounced as the problem size grows. This widening gap in performance highlights the superior scalability of the FFT algorithm for larger datasets, a trend further supported by the standard deviation values, which suggest a degree of consistency in the runtimes for each method at a given size.