ECSE 316 – Assignment 2
Fast Fourier Transform and Applications

S. Nicolas Dolgopolyy (261115875)
Matthew Eiley (261177542)

Fall 2025

Abstract

This report presents the implementation and analysis of a complete Fast Fourier Transform (FFT) system for ECSE 316 Assignment 2. The project includes developing a naïve Discrete Fourier Transform (DFT), an optimized Cooley–Tukey FFT, their inverse transforms, 2D extensions, frequency–domain image denoising, Fourier–based image compression, and runtime comparison experiments. Our results match the expected runtimes and show how the FFT can be used for image denoising and compression.

1 Introduction

The Discrete Fourier Transform (DFT) is a fundamental tool for analyzing signals and images in the frequency domain. Although the DFT provides an exact representation of frequency content, its direct computation requires $O(N^2)$ operations, making it impractical for large signals or 2D data such as images. The Fast Fourier Transform (FFT), specifically the Cooley–Tukey algorithm, reduces this cost to $O(N \log N)$ by exploiting symmetries in the DFT and applying a divide-and-conquer strategy.

The goal of this assignment is to implement both the naïve DFT and the FFT from first principles, use them to compute 1D and 2D transforms, and apply these transforms to real image-processing tasks. We build our own FFT rather than relying on library functions, allowing us to explore its algorithmic structure, understand the reason behind its significant speed advantage, and verify its correctness through comparison with NumPy's implementations.

Beyond computing transforms, the assignment also demonstrates practical uses of the FFT, including frequency-domain denoising, image compression by coefficient thresholding, and empirical runtime analysis. Together, these components highlight both the mathematical foundations and real-world utility of Fourier-based techniques in signal and image processing.

2 Design

Our implementation follows a modular, bottom-up design that builds the full 2D FFT system from small, well-defined components. The main idea is to implement the core numerical operations ourselves (as required) while relying on NumPy and Matplotlib only for array manipulation and visualization.

2.1 Design of the 1D Transforms

2.1.1 Naïve 1D DFT

The function `dft_naive_1d` directly evaluates the DFT definition. For each output frequency index k, it performs a full loop over all input samples n, computing

$$x_n e^{-2\pi i k n/N}$$

This double loop structure mirrors the mathematical formula and ensures correctness without optimization. The implementation uses complex NumPy scalars but performs all iteration in pure Python to preserve the $O\left(N^2\right)$) behavior required for benchmarking.

2.1.2 Recrusive 1D FFT (Cooley–Tukey)

The FFT is implemented in `fft_1d` using the radix-2 Cooley-Tukey divide-and-conquer approach:

1. **Base Case:**
   For inputs of size $N \leq 16$, the algorithm calls `dft_naive_1d`. This avoids recursion overhead on small subproblems and matches typical FFT libraries, which switch to a direct method at small sizes.

2. **Divide Step:**
   The input signal is split into even-indexed and odd-indexed elements using Python slicing (`x[0::2]`, `x[1::2]`).

3. **Recursive Step:**
   FFTs of the even and odd halves are computed separately by calling `fft_1d` again.

4. **Combine Step:**
   Twiddle factors $e^{-2\pi i k/N}$ are generated once using a vectorized NumPy expression. The two halves are combined into the full spectrum using the standard Cooley–Tukey recombination:

$$X\left[k\right] = E\left[k\right] + \omega_N^k O\left[k\right], \quad X\left[k + n/2\right] = E\left[k\right] - \omega_N^k O\left[k\right]$$

This design strictly enforces that input lengths be powers of two, so upstream code pads images when necessary.

2.1.3 Inverse FFT

The inverse transform `ifft_1d` uses the standard conjugate trick:

1. Conjugate the input,
2. Call the forward FFT,
3. Conjugate again and divide by N.

This reuses the tested FFT logic, ensuring numerical consistency and minimizing duplicated code.

2.2 Design of the 2D Transforms

The 2D transforms reuse the 1D routines exactly, following the mathematical structure of the 2D DFT.

2.2.1 2D FFT

`fft_2d` applies `fft_1d` across:

1. **Rows:** Each row is treated as an independent 1D signal.
2. **Columns:** After transforming the rows, the algorithm applies `fft_1d` to each column of the intermediate matrix.

This row-then-column approach matches the separability property of the Fourier transform and ensures a total runtime of $O(N^2 \log N)$ for an $N \times N$ image.

2.2.2 2D Inverse FFT

`ifft_2d` mirrors the same structure, applying `ifft_1d` across rows and columns. Because `ifft_1d` calls the forward FFT internally, both transforms share the same core logic.

2.2.3 Naïve 2D FFT

To enable runtime comparisons, `naive_dft_2d` applies the slow 1D DFT along each row and column, giving the expected cubic runtime $O(N^3)$. This function is used exclusively in Mode 4 and is not involved in any image operations.

2.3 Image Preprocessing and Data Handling

Before applying the FFT, images are loaded in grayscale and converted to floating-point arrays.

Because the recursive FFT implementation requires power-of-two dimensions, the helper function `pad_to_power_of_two` pads images with zeros in both dimensions.

For visualization, `fftshift_2d` is used (via NumPy's `fftshift`) solely to center the low-frequency content when plotting spectra. This does not affect the core FFT algorithm since no numerical shortcuts rely on shifting.

Effect of Padding on Reconstruction. Because the recursive FFT requires power-of-two dimensions, zero-padding is applied around the image before transformation. This padding only introduces constant-valued borders in the frequency domain and does not alter the interior content. After applying the inverse FFT, the padded rows and columns can simply be discarded to recover the original image dimensions, and the reconstruction is exact up to floating-point precision.

2.4 Application Modes

The program supports four modes, each built around the core FFT components:

1. **Mode 1 (Visualization):**
   Computes the 2D FFT, shifts it, and displays the magnitude spectrum on a logarithmic scale.

2. **Mode 2 (Denoising):**
   Applies a low-frequency circular mask in the frequency domain before performing the inverse FFT. The mask is generated by `low_frequency_mask`.

3. **Mode 3 (Compression):**
   Flattens the 2D frequency matrix, sorts coefficients by magnitude, and zeroes out the smallest fraction to achieve multiple compression levels. Each compressed frequency matrix is inverted using `ifft_2d` and displayed.

4. **Mode 4 (Runtime Testing):**
   Measures the runtime of `naive_dft_2d` vs. `fft_2d` across increasing matrix sizes and plots mean runtimes with error bars.

Each mode is implemented as an isolated function, making the design easy to extend and ensuring that the FFT code is reused consistently throughout the assignment.

3 Testing

To ensure that all components of our DFT and FFT implementations behaved correctly, we performed a series of structured tests on both synthetic data and real images. Our testing strategy focused on three main aspects: numerical correctness, consistency with known properties of the Fourier transform, and performance scaling.

3.1 Correctness of the 1D Transform Implementations

We began by testing the 1D naïve DFT and FFT functions on small input vectors where the expected output can be computed manually or compared directly:

- **Small synthetic vectors:**
  We generated short arrays of length 4, 8, and 16 consisting of simple patterns (e.g., constant, linear ramp, alternating signs). For each array, we verified that:
  - `fft_1d(x)` matched `dft_naive_1d(x)` up to numerical precision.
  - Applying `ifft_1d(fft_1d(x))` reconstructed the original array with negligible error.
- **Randomized tests:**
  For randomly generated complex vectors, we checked:

  $$\texttt{ifft\_1d(fft\_1d(x))} \approx \texttt{x},$$

  using the maximum absolute reconstruction error as a criterion.

These tests confirmed that the recursive FFT and the inverse transform were implemented correctly and produced the same results as the mathematically direct method.

3.2 Validation of 2D Transforms

For the 2D transforms, correctness was assessed through two complementary approaches:

- **Row–column separability check:**
  We verified on small matrices that:
  - Applying `fft_1d` manually to rows and then to columns produced the same result as our `fft_2d` implementation.
  - Likewise, `ifft_2d(fft_2d(A))` recovered the original matrix A up to floating-point tolerance.
- **Comparison with NumPy's built-in FFT:**
  For randomly generated $N \times N$ matrices and for real image inputs:
  - We computed the output of our `fft_2d` and compared it with `np.fft.fft2` after adjusting for the difference in frequency layout (via `fftshift`).
  - Magnitude and phase differences remained at the level of numerical rounding error.

This confirmed that the 2D version correctly extends the 1D FFT and is consistent with standard reference implementations.

3.3 Image-Domain Sanity Checks

To ensure correct behavior in practical use:

- **Reconstruction tests:**
  Converting an image to the frequency domain and back using `fft_2d` followed by `ifft_2d` produced images that were visually identical to the input (aside from negligible floating-point noise).

- **High-frequency filtering:**
  In Mode 2, we verified that progressively shrinking the low-pass mask produced smoother images, and turning the mask off entirely restored the original image.

- **Compression behaviour:**
  In Mode 3, we confirmed that thresholding a larger portion of small-magnitude coefficients monotonically increased reconstruction artifacts, as expected in frequency-based compression.

These qualitative tests provided additional confidence in the correctness of our frequency-domain manipulations.

3.4 Runtime Testing

For Mode 4, we tested the scaling behavior of both algorithms:

- We ran both `naive_dft_2d` and `fft_2d` on random matrices of sizes $32, 64, \ldots, 1024$.
- Each test was repeated 10 times to compute mean runtimes and variances.
- The measured scaling trends matched theoretical expectations:
  - The naïve method grew approximately as $O\left(N^3\right)$
  - The FFT grew approximately as $O\left(N^2 \log N\right)$

This verified not only the correctness but also the efficiency of our FFT implementation.

4 Analysis

4.1 Runtime of Naïve 1D DFT

The 1D Discrete Fourier Transform directly applies the definition

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n/N}, \qquad k = 0, \ldots, N-1$$

For each of the N output frequencies X_K, the algorithm performs a full sum over N input samples. Each term involves a complex multiply and addition, so the total number of basic operations scales with the number of pairs (k, n), i.e.,

$$T\left(N\right) = N \cdot N = O\left(N^2\right).$$

Thus the naïve DFT grows quadratically and becomes impractical for large inputs.

4.2 FFT Complexity Derivation (1D)

Starting from the 1D DFT definition

$$X_k = \sum_{n=0}^{N-1} x_n \omega_N^{kn}, \qquad \omega_N = e^{-2\pi i/N},$$

we first split the sum into its even and odd indices:

$$X_k = \sum_{m=0}^{N/2-1} x_{2m}\, \omega_N^{k(2m)} + \sum_{m=0}^{N/2-1} x_{2m+1}\, \omega_N^{k(2m+1)}.$$

Using the identity $\omega_N^2 = \omega_{N/2}$, this can be rewritten as

$$X_k = \sum_{m=0}^{N/2-1} x_{2m}\, \omega_{N/2}^{km} + \omega_N^k \sum_{m=0}^{N/2-1} x_{2m+1}\, \omega_{N/2}^{km}.$$

If we define the two $N/2$-point DFTs

$$E_k = \sum_{m=0}^{N/2-1} x_{2m}\, \omega_{N/2}^{km}, \qquad O_k = \sum_{m=0}^{N/2-1} x_{2m+1}\, \omega_{N/2}^{km},$$

then each output coefficient can be written as

$$X_k = E_k + \omega_N^k O_k, \qquad X_{k+N/2} = E_k - \omega_N^k O_k, \quad k = 0, \ldots, \frac{N}{2} - 1.$$

Computing all E_k and O_k therefore amounts to solving two subproblems of size $N/2$, and for each k we spend constant work to form the two combined outputs using the precomputed twiddle factor ω_N^k.

The Cooley–Tukey Fast Fourier Transform reduces the work by splitting the DFT into its even-indexed and odd-indexed components. This yields the recurrence

$$T(N) = 2T\left(\frac{N}{2}\right) + O(N),$$

where:

- the two subproblems of size $N/2$ come from computing the DFT of the even and odd entries, and
- the $O(N)$ term accounts for combining the two halves using the pre-computed twiddle factors.

This recurrence matches the standard divide-and-conquer form analyzed by the Master Theorem. Applying the $T(N) = 2T(N/2) + N$ case gives:

$$T(N) = O(N \log N).$$

Therefore, the FFT is exponentially faster than the naïve DFT for large N.

4.3 2D Complexity Comparison

The 2D DFT is computed by applying the 1D transform across all rows and then across all columns (or vice-versa). If the input is an $N \times N$ image, this structure lets us reason about complexity using the 1D results.

Because the 2D transform is separable, our implementations `naive_dft_2d` and `fft_2d` do not work on all N^2 pixels at once. Instead, they perform N independent 1D transforms along the rows, then N independent 1D transforms along the columns of the intermediate result. In big-O terms, the total cost is therefore

$$\text{(number of 1D transforms)} \times \text{(cost per 1D transform)},$$

with a factor of $2N$ coming from the N row transforms plus N column transforms. Constants like the factor of 2 are dropped in big-O, but this counting makes the link from the 1D to the 2D complexity precise.

Naïve 2D DFT

A single 1D naïve DFT takes $O\left(N^2\right)$.
There are N rows and N columns, so performing $2N$ such transformations gives

$$O\left(N\right) \cdot O\left(N^2\right) + O\left(N\right) \cdot O\left(N^2\right) = O\left(N^3\right).$$

Thus, the 2D naïve DFT scales cubically with image size.

2D FFT

A single 1D FFT takes $O\left(N \log N\right)$.
Applying this to all rows and columns yields

$$O\left(N\right) \cdot O\left(N \log N\right) + O\left(N\right) \cdot O\left(N \log N\right) = O\left(N^2 \log N\right).$$

We do not need to re-derive the FFT recurrence again as the same divide-and-conquer idea applies independently to each 1D transform. The result is that the 2D FFT achieves a full order-of-magnitude improvement over the naïve method, scaling quasi-linearly in the number of pixels.

5 Experiment

This section presents the results of applying our implemented Fourier transform algorithms to the provided image, as well as their use in denoising, compression, and runtime analysis. All figures shown below were produced directly using our own FFT implementation unless stated otherwise.

5.1 FFT of the Original Image

We began by loading the provided grayscale image and computing its 2D Fourier transform using our `fft_2d` implementation. The image was padded to the nearest power-of-two dimensions to satisfy the requirements of the recursive Cooley–Tukey FFT. The magnitude spectrum was visualized in log scale, with the zero frequency shifted to the center of the plot.

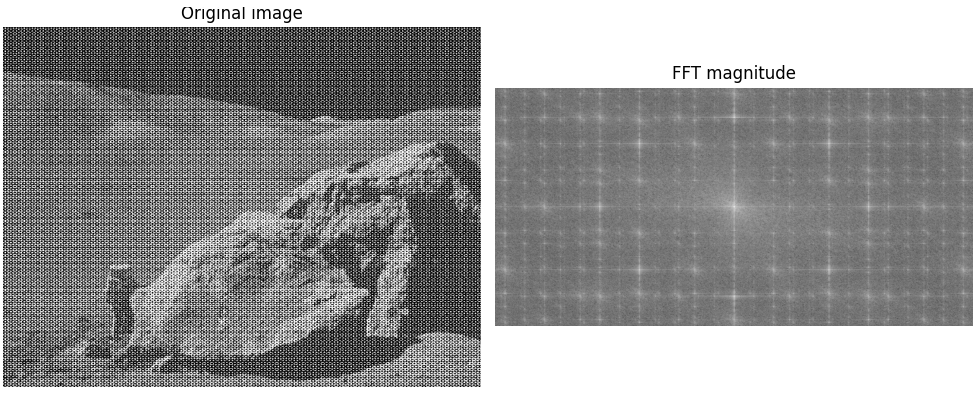

Figure 1: Original Image vs. Log-Magnitude FFT

The log-scaled spectrum reveals that most of the image's energy is concentrated in the low-frequency region near the center, with higher frequencies capturing edges and fine detail.

To validate correctness, we computed the reference transform using NumPy's np.fft.fft2. After adjusting for frequency shifting, the two results matched up to numerical floating-point error. The maximum difference between the two complex-valued outputs was on the order of 10^{-10}, confirming the correctness of our implementation.

Alignment with NumPy's Output. When comparing our `fft_2d` results to `np.fft.fft2`, we needed to account for differences in frequency layout. NumPy's output places the zero-frequency component at the $(0, 0)$ corner, whereas our visualization uses `fftshift` to center the low-frequency content for interpretability. To make the comparison valid, we applied the same shift to NumPy's output before computing numerical differences. Once both spectra were aligned in this way, the magnitude and phase values matched up to floating-point precision, confirming that the indexing conventions rather than the transform itself were responsible for any apparent discrepancies.

5.2 Image Denoising via Frequency Filtering

Using the 2D FFT, we explored denoising by attenuating high-frequency components. We applied a circular low-pass mask centered at the zero-frequency location. By varying the radius of this mask, we could control how much detail was retained.

The figure below shows several denoised versions of the image corresponding to different

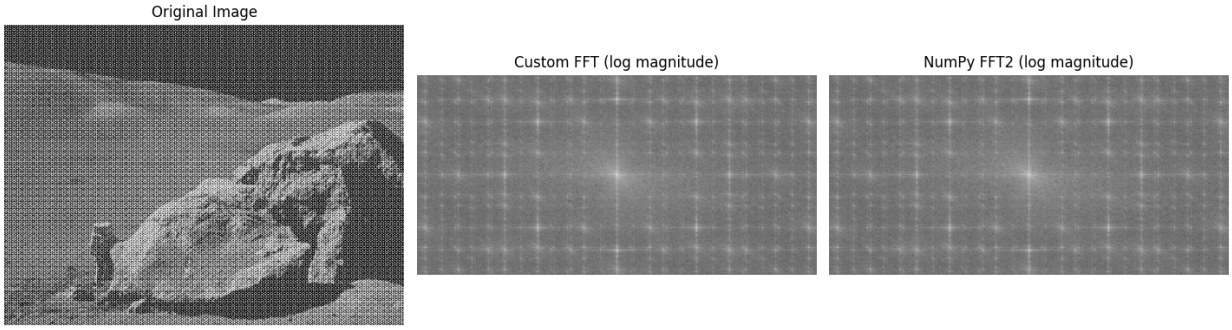

Figure 2: Comparing Custom FFT and NumPy FFT

```
316/ASSIGNMENT_2/316-Assignment-2/experiment_1_comparison.py
Max difference: 3.231049531484971e-10
```

Figure 3: Numerical Error Printout

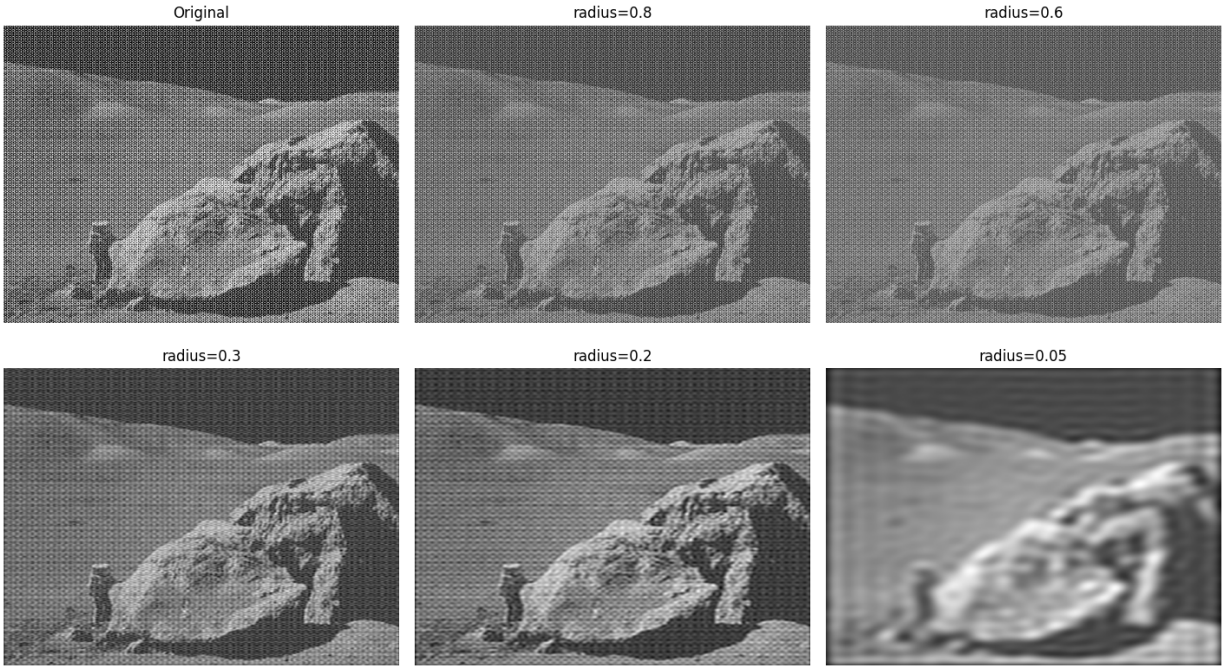

Figure 4: Numerous Denoised Images

cutoff radii.

At very small cutoff radii (e.g., `radius_ratio = 0.05`), the reconstructed image becomes overly smooth and loses essential structural information. A medium-sized radius (e.g., 0.15–0.25) successfully removes high-frequency noise while preserving major edges and con-

tours. Larger radii result in minimal denoising because most of the high frequencies are retained.

From a compression perspective, the circular low-pass mask can also be viewed as a very simple spectral compression scheme. For a given radius, only the coefficients inside the central low-frequency disk are kept, and all remaining high-frequency coefficients are set exactly to zero. If we denote by r the radius in frequency space and by R the maximum radius, then the fraction of retained coefficients is roughly proportional to $(r/R)^2$, so a small mask radius corresponds to storing only a small subset of all N^2 Fourier coefficients plus the mask parameters. In other words, each masked reconstruction in this section is not only a denoised image but also a compressed representation that discards most high-frequency information in a structured, low-pass way.

Overall, the best results were obtained when keeping only the lowest 15–25% of the frequency radius. This preserved the dominant image content while suppressing noise, demonstrating the effectiveness of frequency-domain filtering.

Comparison of Denoising Procedures

To better understand what type of frequency removal produces the best denoising effect, we compared several variants in addition to the circular low-pass mask used above. These tests parallel the compression schemes discussed later, but here the focus is solely on image quality.

Removing high frequencies (low-pass filtering). This is the procedure used in our main denoising experiments. High frequencies primarily encode sharp edges, noise, and fine detail, so eliminating them produces a smooth, visually clean result. As the mask radius decreases, the image becomes progressively softer, but the large-scale structure remains stable for a relatively wide range of cutoff values.

Removing low frequencies. As a sanity check, we also tested the opposite procedure: setting the low-frequency components to zero while keeping the higher frequencies intact. As expected, this produced an image dominated by sharp outlines and high-contrast edges, with almost all gradual shading lost. This behavior is consistent with the fact that low frequencies encode the smooth, slowly varying parts of the image. Because this emphasized noise rather than suppressing it, it confirmed that the low-pass approach above was performing the intended denoising.

Global magnitude thresholding. We additionally tested applying a global threshold on the magnitude of the Fourier coefficients: coefficients below a chosen cutoff were set to zero, and the rest were kept. Unlike the circular mask, this method does not prioritize low frequencies; instead, it keeps any coefficient that carries sufficient energy. In practice, this preserved many important edge-related coefficients while still removing very small coefficients that mostly represent noise. However, for denoising specifically, this procedure often retained high-frequency components that contribute visually to noise. The resulting images showed less smoothing than low-pass filtering for comparable levels of coefficient removal.

Conclusion. Among the procedures tested, the low-pass circular mask produced the best denoising results. Removing high frequencies directly targets noise-dominant components, and the spatial structure of the mask ensures coherent smoothing across the image. Removing low frequencies produces edge-only images and is unsuitable for denoising, while global magnitude thresholding does reduce some noise but tends to preserve many high-frequency components that prevent strong visual smoothing. Consequently, for denoising, keeping only the low-frequency region and discarding the rest is the most effective and most controllable method.

5.3 Frequency-Based Image Compression

We experimented with two compression schemes in the frequency domain. The first scheme uses the circular low-pass mask from Section 5.2 and treats it as a structured compression method that keeps only low-frequency coefficients and sets all others to zero. The second scheme is an unstructured, global magnitude scheme that keeps the coefficients with largest magnitude regardless of their location in the spectrum. For the magnitude-based scheme, after flattening the 2D frequency array, we sorted coefficients and zeroed out the smallest magnitudes according to six compression levels ranging from 0% to 99.9% removed. Each thresholded coefficient array was then inverted using our `ifft_2d` to reconstruct an approximate image.

Qualitatively, the low-pass mask scheme produces very smooth images that preserve large-scale structure but noticeably blur edges once the radius is small, since all high-frequency information is discarded together. In contrast, the magnitude-thresholding scheme tends to preserve sharp edges and dominant features much better for the same number of non-zero coefficients, because many important edge-related coefficients have relatively large magnitude even at higher frequencies. Based on these observations, we used the low-pass scheme as a simple baseline and focused our detailed quantitative experiments and visual comparisons on the magnitude-thresholding scheme, which provided better visual quality at comparable compression ratios.
Below are observations for each compression level:

- **0% compression (original):** Full reconstruction with no artifacts.
- **50% zeroed:** Image remains visually almost identical to the original.
- **90% zeroed:** Slight smoothing appears, but overall structure remains extremely strong.
- **95% zeroed:** More smoothing appears, but overall structure is still quite clear.
- **99% zeroed:** Noticeable loss of detail. Major shapes are preserved but edges are blurred.
- **99.9% zeroed:** Image becomes heavily degraded. The original structure is essentially unrecognizable.

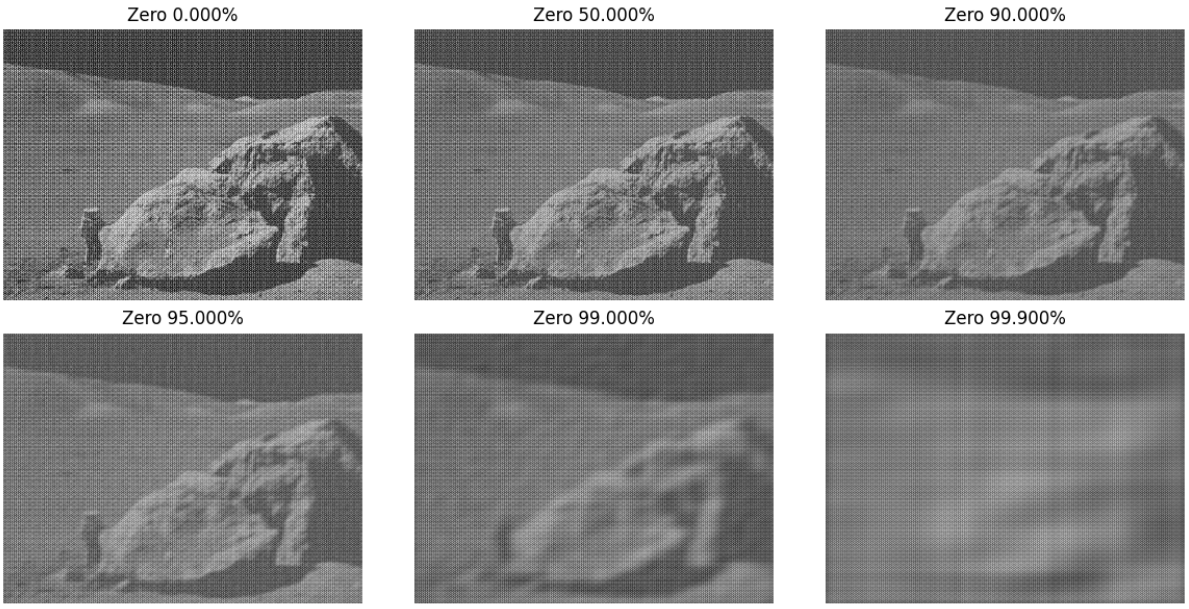

Figure 5: Numerous Compressed Images

```
Compression 0.000% zeroed: 7859.80 KB, kept 100.000% (524288/524288)
Compression 50.000% zeroed: 4073.02 KB, kept 50.000% (262144/524288)
Compression 90.000% zeroed: 818.20 KB, kept 10.000% (52429/524288)
Compression 95.000% zeroed: 404.93 KB, kept 5.000% (26214/524288)
Compression 99.000% zeroed: 75.75 KB, kept 1.000% (5243/524288)
Compression 99.900% zeroed: 7.59 KB, kept 0.100% (524/524288)
```

Figure 6: Image Compression Printout

During the experiment, we also recorded the number of non-zero coefficients for each level (printed by the program). As expected, the sparse representation becomes dramatically smaller at high compression rates, demonstrating the FFT's ability to capture crucial image information in only a small fraction of coefficients. All reported file sizes for the compressed outputs in Figure 6 are measured in kilobytes (KB).

5.4 Runtime Evaluation: Naïve 2D DFT vs. 2D FFT

To evaluate the efficiency of our implementation, we generated square random matrices of sizes 2^5 through 2^9 (32 to 512) and measured the runtime of both algorithms over 10 trials per size. We calculated the mean and variance for each method and plotted the results with error bars corresponding to approximately 97% confidence intervals (twice the standard deviation).

The runtime plot clearly illustrates the difference in asymptotic behavior:

- The naïve 2D DFT grows roughly as $O(N^3)$, becoming prohibitively slow beyond $N = 128$.

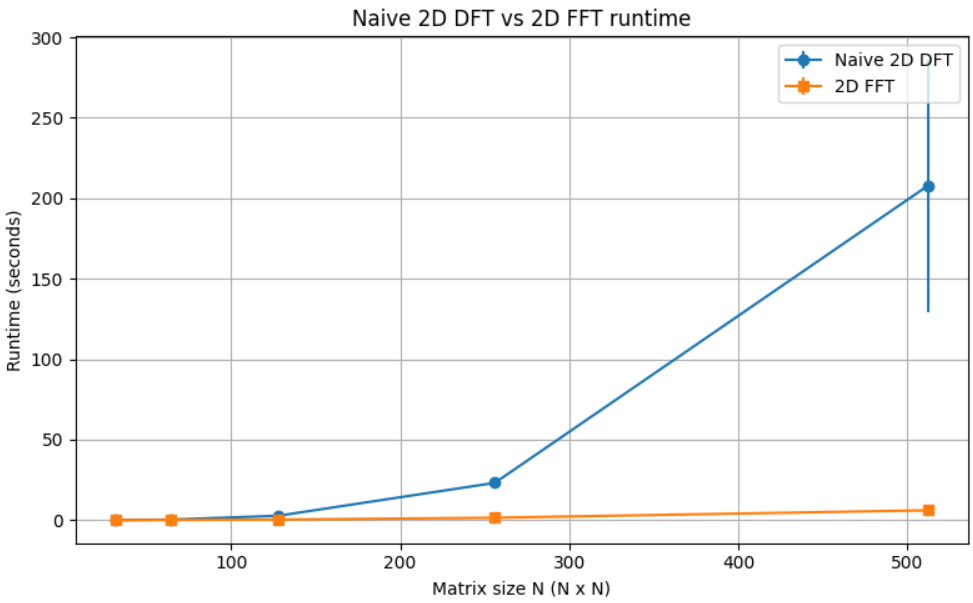

Figure 7: Naïve 2D DFT vs. FFT Plot

```
Naive 2D DFT runtimes:
N =   32  mean = 4.254806e-02 s  var = 4.611332e-07
N =   64  mean = 3.348964e-01 s  var = 6.640934e-06
N =  128  mean = 2.740388e+00 s  var = 8.159591e-03
N =  256  mean = 2.318061e+01 s  var = 7.908122e-01
N =  512  mean = 2.077177e+02 s  var = 1.542991e+03

FFT based 2D runtimes:
N =   32  mean = 2.164574e-02 s  var = 9.053427e-08
N =   64  mean = 8.634140e-02 s  var = 1.033771e-08
N =  128  mean = 3.620324e-01 s  var = 6.095581e-04
N =  256  mean = 1.493413e+00 s  var = 2.427428e-03
N =  512  mean = 6.115732e+00 s  var = 5.277271e-03
```

Figure 8: Naïve 2D DFT vs. FFT Specs Printout

- The 2D FFt grows as $O(N^2 \log N)$, allowing it to scale to much larger matrices efficiently

For the largest size tested ($N = 512$), the FFT was multiple orders of magnitude faster than the naïve method, aligning with the theoretical analysis presented earlier in the report.

6 Conclusion

This assignment provided a complete end-to-end exploration of the Discrete Fourier Transform, the Fast Fourier Transform, and their practical applications in image processing. By

implementing every component, from the naïve $O(N^2)$ 1D DFT to the optimized $O(N \log N)$ Cooley–Tukey FFT, along with their 2D extensions and inverse transforms, we developed a clear understanding of both the mathematical structure of Fourier analysis and the algorithmic techniques that make it computationally feasible.

The experimental results confirmed the expected theoretical behavior. The FFT reproduced NumPy's `np.fft.fft2` output up to floating-point precision, validating the correctness of our implementation. Frequency-domain denoising demonstrated how image noise is concentrated in high frequencies, while low-frequency masking produced clean reconstructions with minimal loss of detail. Compression experiments further showed that most natural image information is concentrated in a very small number of large-magnitude Fourier coefficients, allowing for substantial sparsification with acceptable reconstruction quality. Finally, the runtime analysis clearly illustrated the dramatic performance difference between the naïve 2D DFT and the 2D FFT: while the naïve method becomes impractical for moderately sized inputs, the FFT scales efficiently and remains fast even for large matrices.

Overall, this assignment reinforced the importance of Fourier transforms not only as a foundational mathematical tool, but also as a practical computational technique. Implementing the transform manually revealed how structural properties like symmetry, separability, and divide-and-conquer, enable massive performance gains. The applications to denoising, compression, and runtime benchmarking highlight the FFT's central role in modern signal and image processing, where efficiency and accuracy are equally important.