ECSE 316 - Signals and Networks

Assignment 2 Report

Samantha Handal 260983914
Amélie Barsoum 260988658

November 24th, 2023

I. Design

FFT 1D Implementation (fft_1d)

The Fast Fourier Transform (FFT) is a fundamental tool in signal processing, allowing efficient computation of the Discrete Fourier Transform (DFT). Our implementation of fft_1d leverages the Cooley-Tukey algorithm, a paradigmatic example of the divide-and-conquer technique. This approach begins by recursively dividing the signal into even and odd components, continuing until the signal length is reduced to one or less - the recursion's base case. Each recursive call applies the FFT on these smaller segments. Subsequently, the algorithm combines these results, employing complex exponential terms to integrate the even and odd parts, by adding the complex exponential term to the even part for the first half of the frequency bins and subtracting the complex exponential term from the even part for the second half of the frequency bins. This method results in a complete FFT computation, transforming the time-domain signal into its frequency-domain representation.

FFT 2D Implementation (fft_2d)

Extending this to two dimensions, the fft_2d function adapts the 1D FFT for image processing applications. This is achieved by first applying the 1D FFT across each row of the image matrix, followed by a subsequent application down each column. This row-and-column methodology efficiently computes the 2D FFT, essential for converting images between the spatial and frequency domains. Inverse FFT follows a similar pattern, ensuring a reversible transformation crucial for various image-processing tasks.

Logarithmic Scaling in Visualization (fast_mode)

In the fast_mode function, logarithmic scaling plays a critical role in visualizing the frequency spectrum of an image. The Fourier transform of an image often yields a broad range of magnitudes. Direct linear representation of these values might obscure finer details, especially for components with lower magnitudes. Logarithmic scaling, as implemented (log_scalled = np.log(np.abs(f_transform) + 1)), compresses this wide range, enhancing visibility and interpretability. By converting the spectrum into a logarithmic scale, both small and large-magnitude components are effectively visualized. The addition of one before applying the logarithm prevents undefined values, ensuring a smooth transformation across the entire spectrum.

Low Pass Filtering in Image Denoising (low_pass_filter)

This function operates by shifting the zero-frequency component to the center of the image's Fourier transform and then selectively retaining low-frequency components. The extent of frequencies kept is governed by the keep_fraction parameter, which dictates the size of the central frequency window in the mask applied to the Fourier transform. High frequencies, typically representing noise, are set to zero by this mask, leaving only the desired low frequencies. This selective process effectively reduces noise when the image is transformed back to the spatial domain using the inverse FFT.

Threshold-Based Compression (compress_image_keep_high)

The compress_image_keep_high function implements a threshold-based compression algorithm using the FFT. This method transforms the image into the frequency domain, where a threshold is applied to retain only the most significant frequency components. The threshold is determined by the keep_fraction parameter, which decides the percentile of the highest magnitude coefficients to keep. By setting a percentage of the lowest magnitude frequencies to zero, this technique effectively compresses the image. The remaining non-zero frequency components are then inverse-transformed back to the spatial domain to reconstruct the compressed image. This approach is advantageous for its simplicity and effectiveness in reducing image size while maintaining a high level of detail, especially in areas with significant frequency components.

Balanced Frequency Retention (compress_image_keep_middle)

The compress_image_keep_middle function offers a more nuanced approach to image compression. Unlike the threshold-based method, this function retains both very low and a fraction of the highest frequencies. It achieves this by calculating two cutoff values based on keep_fraction: one for the lower half and one for the upper half of the frequency spectrum. This method creates masks to isolate these frequency ranges, combining them to form a comprehensive frequency retention strategy. The result is an image where essential low-frequency components are preserved alongside a selected range of high-frequency details. This dual approach ensures that the core image structure (low frequencies) and finer details (high frequencies) are maintained, leading to a more balanced compression outcome.

Runtime Analysis of DFT vs FFT (plot_runtime_graphs)

The plot_runtime_graphs function was developed to evaluate the runtime performance of two-dimensional Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT) across varying image sizes, ranging from 2^5 to 2^9. This analysis involved conducting ten experimental runs for each image size, where the runtimes of both DFT and FFT were recorded and averaged, with their standard deviations calculated to assess performance consistency. The results were visualized on a log-log plot with error bars representing a 97% confidence interval, effectively highlighting the exponential relationship between image size and runtime. This visualization and quantitative analysis demonstrated FFT's superior efficiency over DFT, especially for larger images, underscoring its practical applicability in large-scale and real-time image processing scenarios.

II. Testing

Our testing approach for the FFT and DFT algorithms involved a direct comparison with NumPy's built-in functions. By applying both our implementation and NumPy’s functions to the same datasets, we could cross-reference the outputs. This method was particularly effective for the Fourier Transform algorithms, where we compared the spectral images generated from both implementations for visual consistency. This straightforward comparison between our results and those obtained from a well-established library provided a solid benchmark for accuracy.

In the case of image denoising and compression, our testing strategy was primarily visual. We compared the original images with the outputs of our denoising and compression algorithms. For

denoising, the key indicator of success was a noticeable reduction in visual noise, such as scratchy patterns, without significant loss of image detail. In image compression, the focus was on maintaining the visual integrity of the original image while increasing the number of zeroed coefficients. We conducted a detailed visual examination of the compressed images at various levels of compression to evaluate the balance between image quality and data reduction. This visual analysis was crucial in assessing the practical effectiveness of these algorithms in image processing tasks.

III. Analysis

Runtime of Naïve DFT for 1D

The Discrete Fourier Transform (DFT) for a one-dimensional signal, by its definition, requires the computation of each frequency component of the signal by summing over all signal samples multiplied by a complex exponential factor. For a signal of length *N,* this process involves *N* summations for each of the *N* frequency components, leading to a total of *N^2* operations. Therefore, the runtime complexity of the naïve 1D DFT is O(N^2).

Runtime Complexity of FFT for 1D

In contrast, the Fast Fourier Transform (FFT) significantly optimizes the computational process. The Cooley-Tukey algorithm employs a divide-and-conquer strategy; It recursively divides the signal into smaller even and odd components until reaching a base case of size 1. The recurrence relation for the FFT can be expressed as:

$T(N) = 2T(N/2) + O(N)$

Here, T(N) is the runtime for a signal of size N, 2T(N/2) represents the two recursive calls on half-sized signals, and O(N) accounts for the combining step. Solving this recurrence relation, we find that the FFt has a runtime complexity of O(NlogN).

2D Complexity Argument for DFT and FFT

Extending these algorithms to two dimensions, the complexity implications become more pronounced. For a 2D DFT on an M x N image, the first step (applying DFT to each row) requires M x O(N^2) operations. The second step (applying DFT to each column of the result) requires N×O(M^2) operations. Therefore, the total number of operations is approximately M×N^2+N×M^2, which simplifies to O(M^2N+MN^2)

In the case of 2D FFT, the efficiency is retained with a complexity of O(MNlogN + NMlogM). Applying the same logic as before, applying FFT to all rows collectively requires M x O(NlogN) = O(MNlogN), and applying FFT to all columns requires N x O(MlogM) = O(NMlogM).

IV. Experiment

A. FFT Implementation

Below is a comparison of the custom FFT implementation (Figure 1) versus the built-in np.fft.fft2 function in NumPy (Figure 2). A correspondence in results between the implemented algorithm and NumPy's built-in function would serve as an indication of the correctness of the approach.

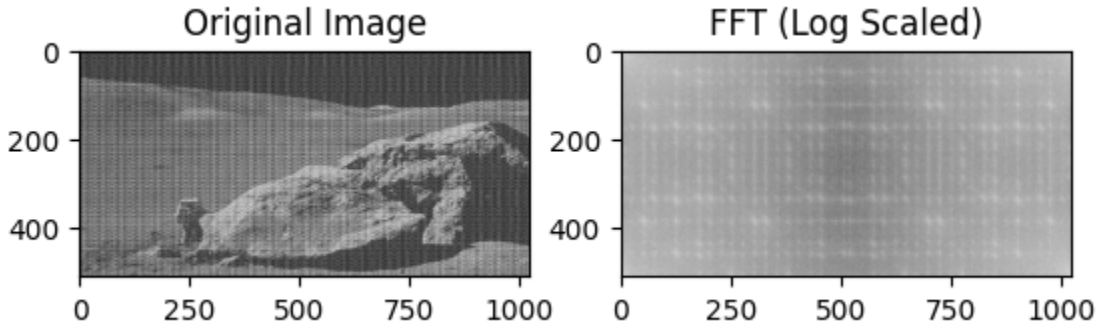

Figure 1: Original image vs FFT (fft_2d)

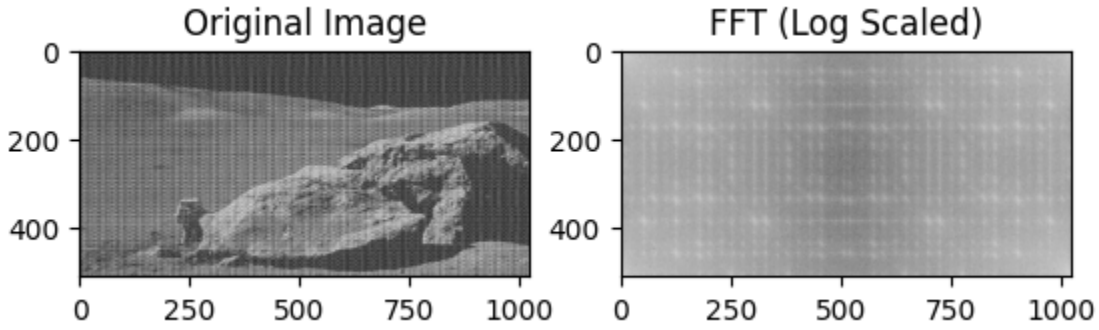

Figure 2: Original image vs FFT (np.fft.fft2)

B. Denoising an image

The following figures represent the results of different denoising techniques using Fast Fourier Transform (FFT) based filtering on an original image. The techniques applied are high pass filtering, low pass filtering, threshold filtering, and bandpass filtering.

High-pass filtering (Figure 3) preserves high-frequency components while removing low-frequency components, often used to emphasize edges and details. However, this did not denoise the image effectively, instead, it resulted in an almost blank image, indicating that the essential information in the original image is primarily in the low-frequency range.

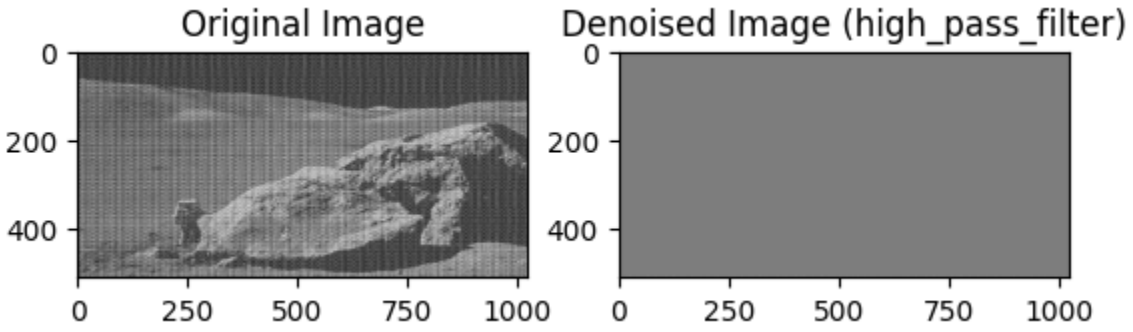

Figure 3: Original image vs High Pass Filtering

Retaining only low-frequency components (Figure 4) smooths out the image, removing high-frequency noise. This approach was the most effective in denoising the image, as it kept the core visual content intact while reducing noise.

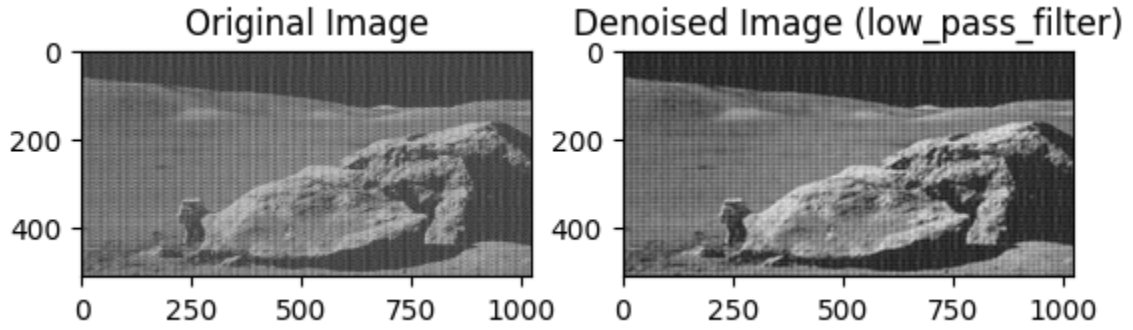

Figure 4: Original image vs Low Pass Filtering

By thresholding the FFT magnitudes (Figure 5), small fluctuations likely representing noise are removed. The resulting denoised image appears similar to the original but with less granular noise, suggesting that this method also preserves the image's integrity quite well.

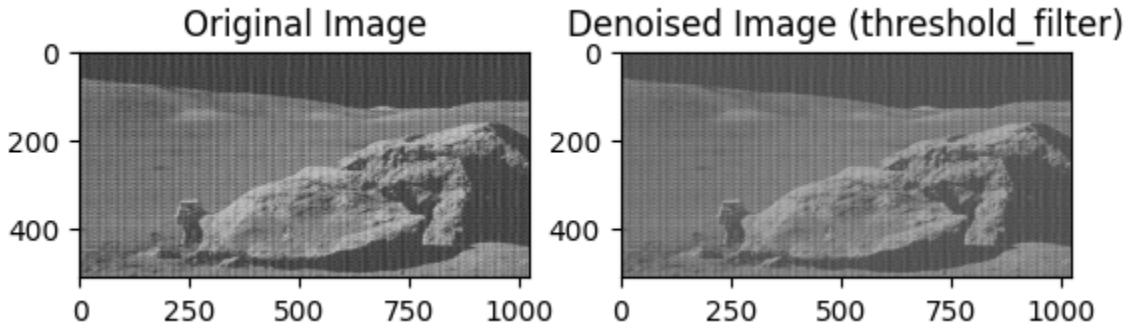

Figure 5: Original image vs Threshold Filtering

The bandpass filtering technique targets a specific range of frequencies, removing everything else. The resultant image became blurry because while it removes noise, it also discards some essential frequencies that contribute to the sharpness and detail of the image.

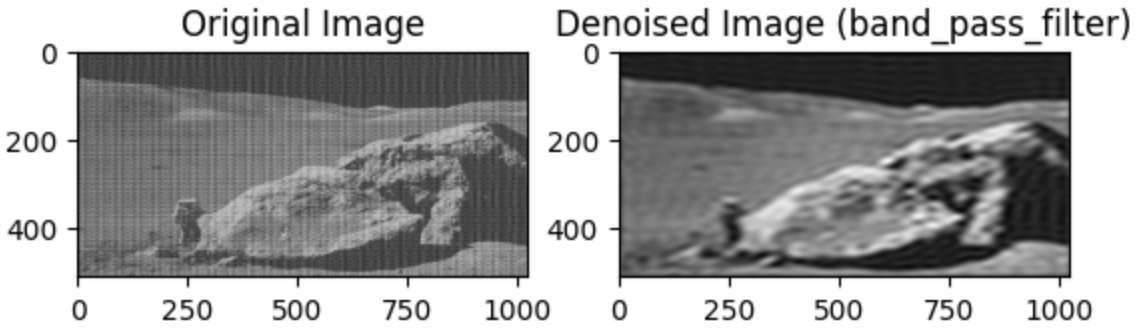

Figure 6: Original image vs Band Pass Filtering

Based on the results, the low pass filter provided the best denoising performance, effectively removing noise while retaining the important features of the image. The high pass filter was not suitable for this particular image, as it removed too much of the vital low-frequency content. The threshold filter performed decently but could potentially remove important details if not carefully calibrated. Lastly, the band pass filter's effectiveness would heavily depend on the chosen frequency range and might require fine-tuning to achieve the desired balance between noise removal and detail preservation.

C. Image compression

Two approaches were explored when implementing image compression; threshold-based (Figure 7) and balanced frequency retention (Figure 8). In the threshold-based approach, the magnitude of the FFT coefficients is used to determine which coefficients are retained; only the largest percentile, as determined by keep_fraction, is kept. This method aims to preserve the most significant features of the image while reducing file size, as evidenced by the non-zero coefficients reported for each compression level.

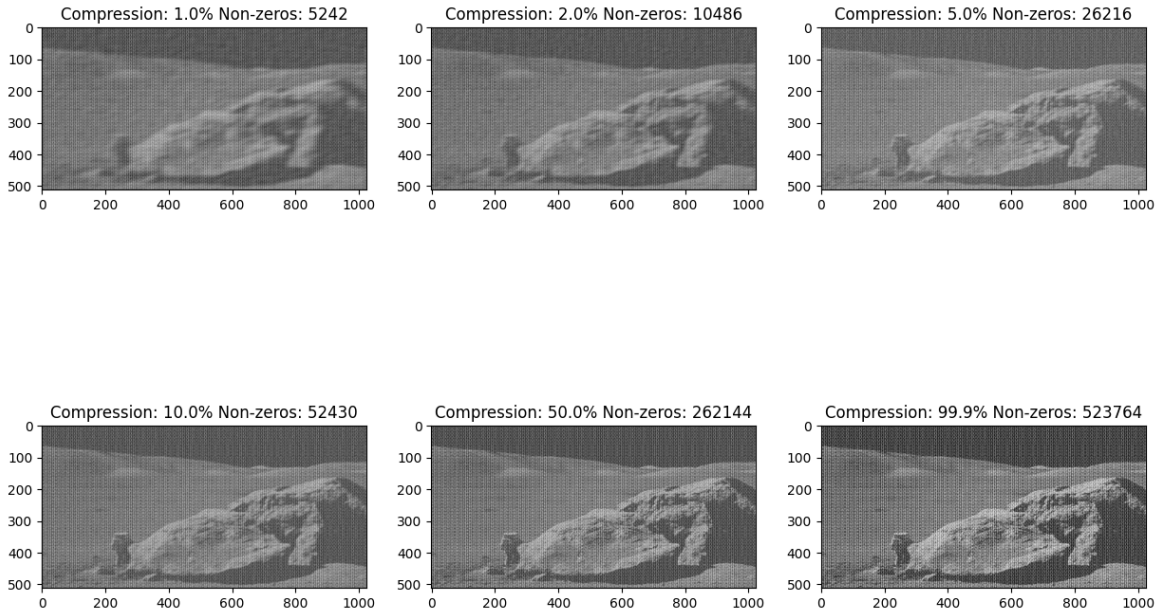

Figure 7: Compression results for threshold-based approach

The balanced frequency retention approach is more selective, retaining all very low-frequency coefficients and a fraction of the highest-frequency coefficients. This method seeks to maintain both the essential characteristics of the image and some of the finer details that might be lost with a more aggressive threshold-based approach. The non-zero coefficients indicate how much of the image information is preserved at each compression level.

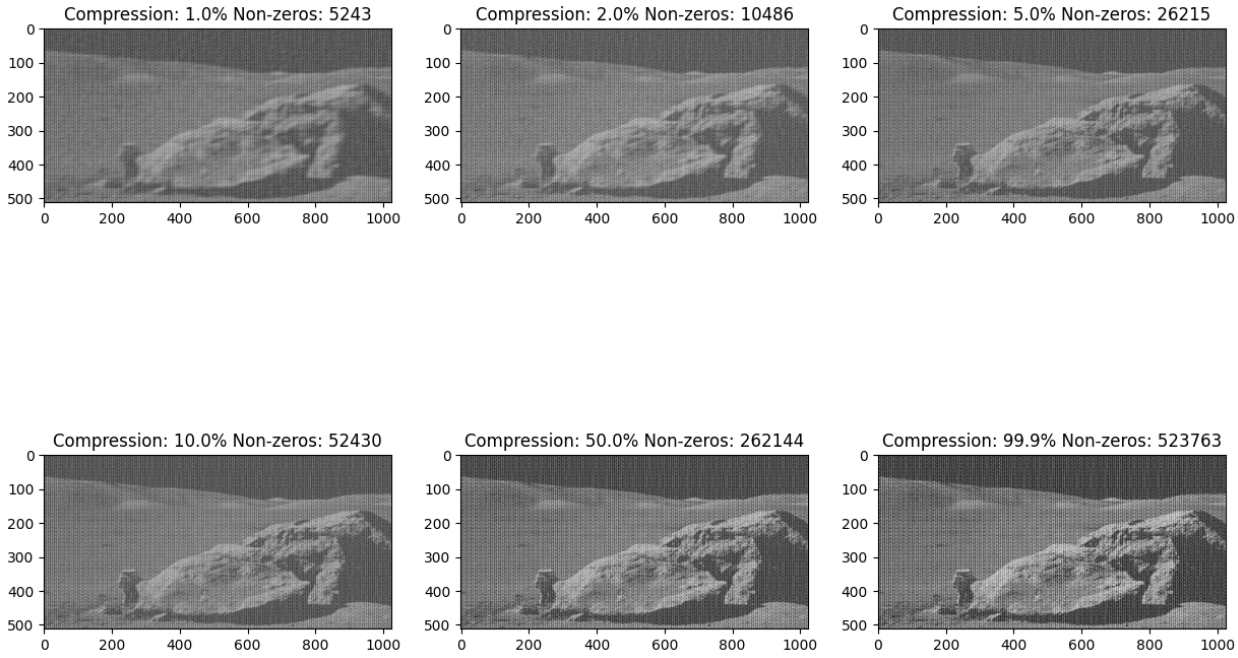

Figure 8: Compression results for balanced frequency retention approach

By comparing the subplots, it's evident that the balanced frequency retention method provides a more nuanced compression, potentially preserving more detail as the compression level increases, as opposed to the threshold-based method which may result in a greater loss of detail with higher compression ratios. The quality of the reconstruction for both methods can be visually assessed by comparing the compressed images to the original. It is expected that the images with higher percentages of non-zero coefficients will more closely resemble the original image, whereas those with fewer non-zero coefficients may show more degradation.

The reconstructed images at varying levels of compression demonstrate the trade-offs between compression ratio and image quality. With increasing compression (higher percentages of zeroed coefficients), the image quality typically decreases, resulting in a loss of detail and clarity. The goal is to achieve a balance where the compressed image retains as much of the original image's quality as possible while significantly reducing the file size.

D. Runtime plotting and analysis

In the runtime analysis between the Discrete Fourier Transform (DFT) and the Fast Fourier Transform (FFT), a clear distinction in computational efficiency is observed. The analysis plotted on the logarithmic scale below (Figure 9), demonstrates that for smaller image sizes, the runtime of DFT and FFT are similar. However, as the image size increases, the DFT's runtime grows exponentially, reflecting the theoretical computational complexity discussed in part III of O(M^2N+MN^2). In contrast, the FFT displays a much smoother increase in runtime, consistent with its O(MNlogN + NMlogM) complexity.

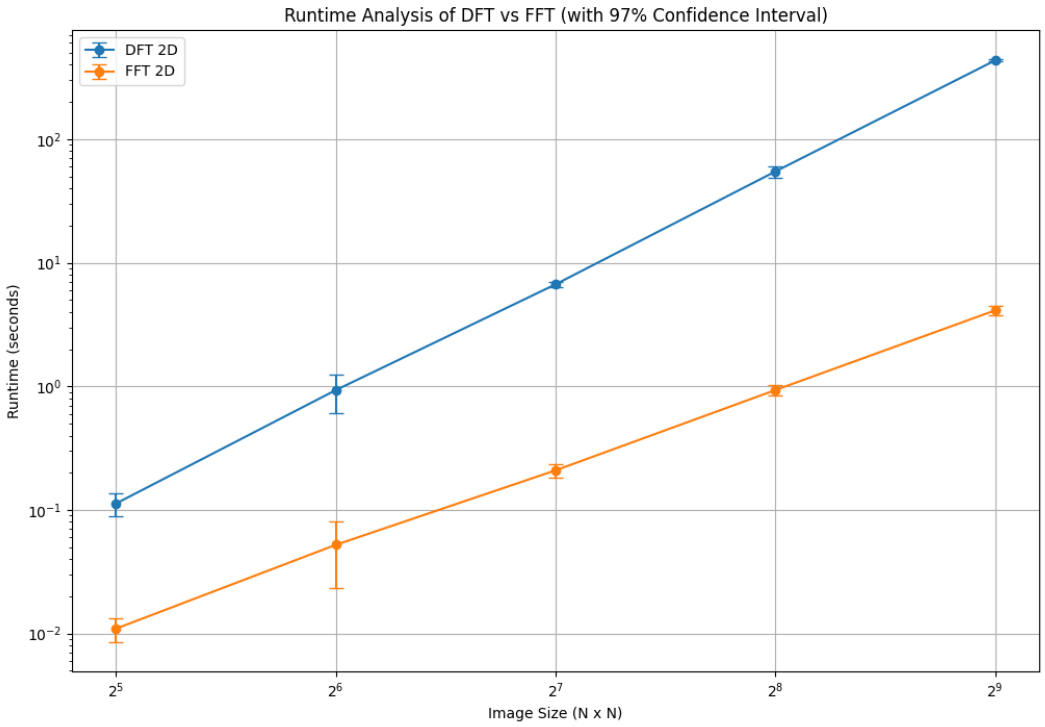

Figure 9: Runtime performance between DFT & FFT

This effect is particularly notable when the runtime is observed at an image size of 2^9, where the DFT's performance exceeds 100 seconds, which is approximately 100x slower than FFT for the same image size. The error bars, representing a 97% confidence interval, are calculated as twice the standard deviation, indicating the reliability and consistency of the measurements across multiple trials.

The findings from this analysis reinforce the practicality of FFT in image processing tasks, especially when dealing with large datasets or requiring real-time analysis. The DFT's computational demands make it impractical for extensive image sizes or time-sensitive applications. Consequently, FFT is the better choice for efficient image analysis.

V. Resources

[1] https://sbme-tutorials.github.io/2021/cv/notes/2_week2.html#:~:text=Discrete%20Fourier%20Transform%20(DFT),-Fourier%20transform%20is&text=Low%20frequency%20components%20are%20found,frequency%20components%20are%20in%20peripherals
[2] https://www.youtube.com/watch?v=gGEBUdM0PVc&ab_channel=SteveBrunton
[3] https://www.youtube.com/watch?v=E8HeD-MUrjY&ab_channel=SteveBrunton
[4] https://www.youtube.com/watch?v=nl9TZanwbBk&ab_channel=SteveBrunton
[5] http://databookuw.com/databook.pdf
[6] https://www.dspguide.com/ch27/6.htm