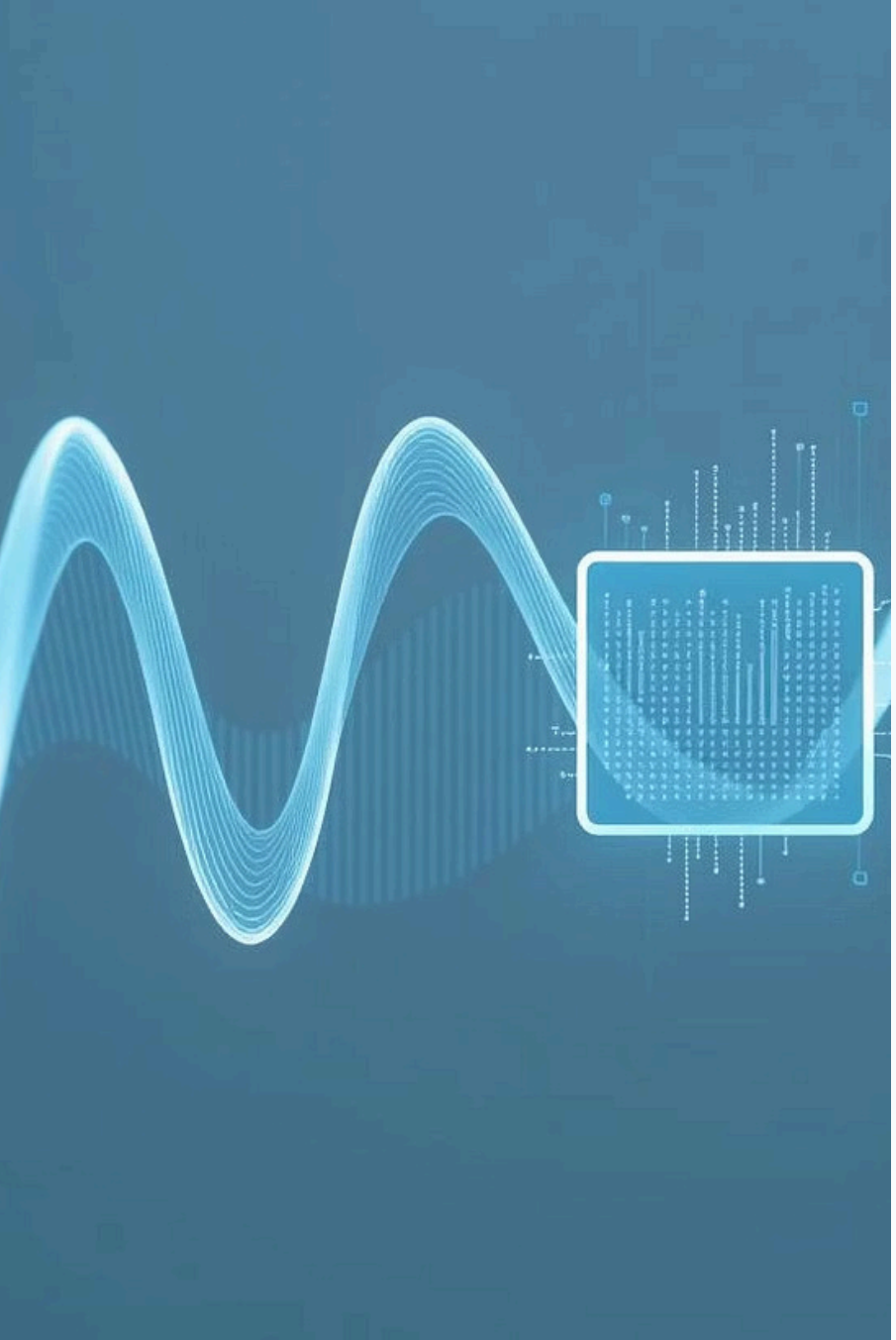


Fast Fourier Transform (FFT) & Its Application in Image Compression

An Engineering Mathematics Perspective

Presented by:

Parsa Bukani
Parsa Saeednia
Mohammadreza Pirhadi

From the University of Tehran

Why Frequency Matters — A Quick Look at Fourier Series

1 Decomposition of Signals

The Fourier Series shows how any complex periodic signal can be broken down into a sum of simple sine and cosine waves, each with a specific frequency, amplitude, and phase.

2 Enhanced Signal Analysis

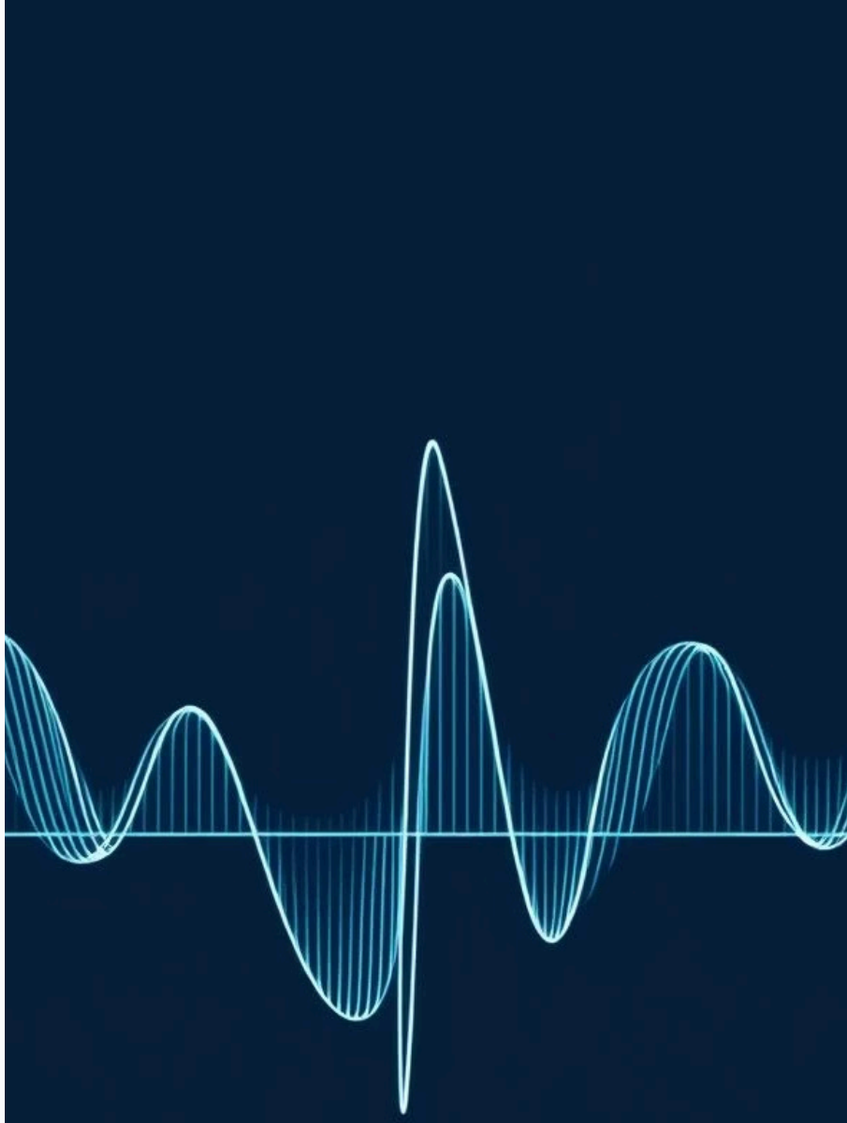
Viewing signals in the frequency domain helps reveal their true nature, making it easier to analyze patterns, periodicity, and noise that are not obvious in the time domain.

3 Identifying Key Information

In the frequency domain, low frequencies represent broad structures (like overall image brightness), while high frequencies capture fine details and sharp edges, allowing us to pinpoint important data.

4 Enabling Compression & Filtering

This frequency understanding is vital for techniques like image and audio compression (by discarding less significant frequencies) and filtering out unwanted components.



From Fourier Series to Discrete Fourier Transform (DFT)

While the Fourier Series provides a powerful tool for analyzing continuous periodic signals, real-world applications in fields like image processing and audio analysis deal with discrete, sampled data. This necessitated the development of the Discrete Fourier Transform (DFT), an adaptation of Fourier analysis specifically designed for finite sequences of sampled signals.

The DFT takes a sequence of N discrete samples from a time-domain signal and transforms it into a sequence of N discrete frequency-domain components. Each component represents the amplitude and phase of a specific frequency present in the original discrete signal. This transformation is crucial for understanding the frequency content of digital signals.

The core of the Discrete Fourier Transform is represented by the following formula:

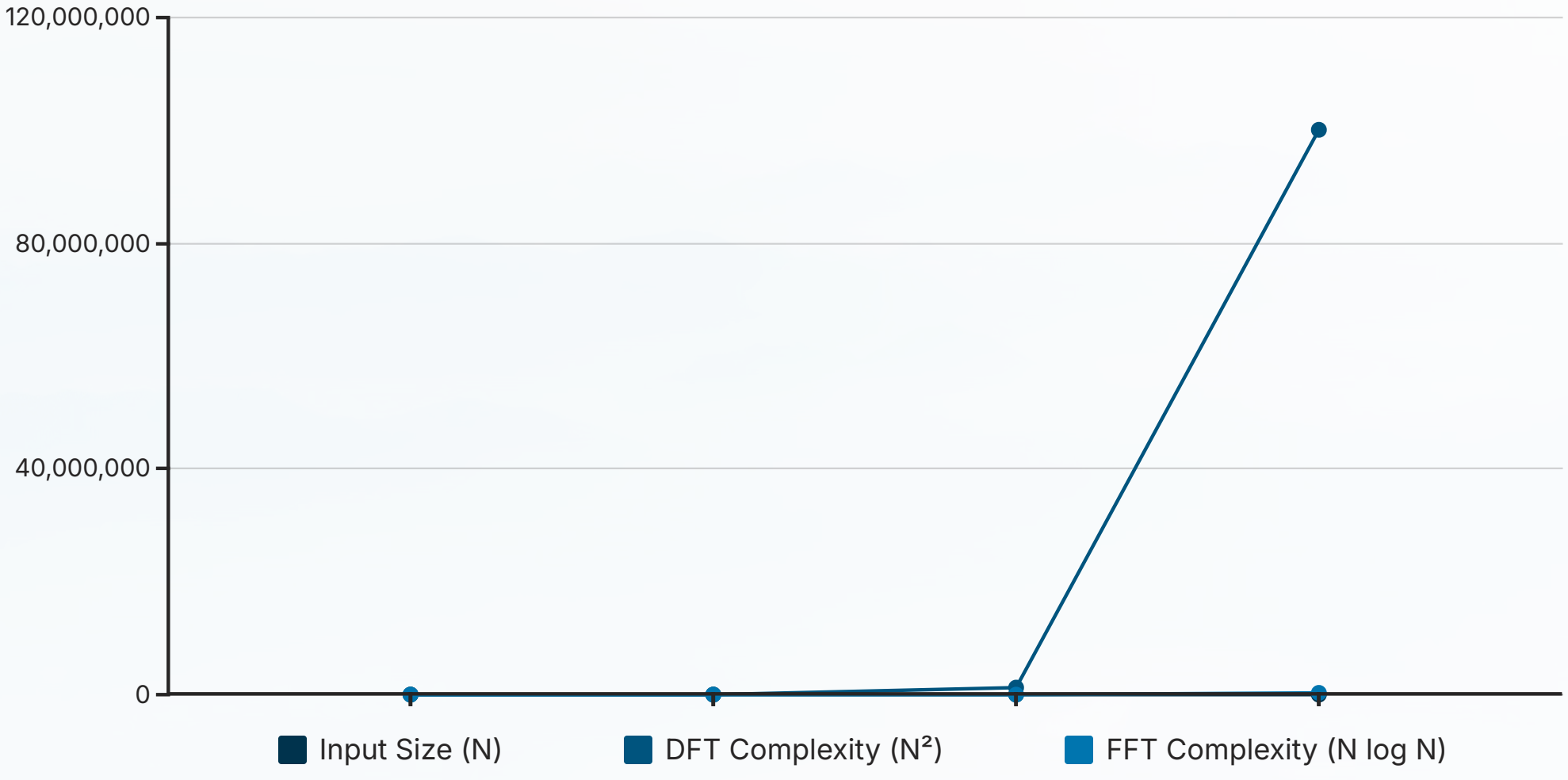
$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i \frac{2\pi}{N} kn}$$

The DFT is a fundamental tool in digital signal processing, enabling efficient analysis, compression, and filtering of digital signals in a wide array of applications, including image processing, audio engineering, and telecommunications.

Why Do We Need FFT?

While the Discrete Fourier Transform (DFT) is a powerful analytical tool, its direct computation for a signal of size N requires approximately **$O(N^2)$** operations. This high computational cost makes the DFT impractical for processing large datasets, such as high-resolution images or long audio signals.

The **Fast Fourier Transform (FFT)** algorithm significantly reduces this computational burden. By employing a divide-and-conquer strategy, FFT performs the same transformation with a complexity of just **O(N log N)**. This drastic reduction in computational time makes frequency analysis fast, efficient, and scalable for real-world applications.



As the graph illustrates, the computational cost of DFT grows **quadratically** with input size, while FFT grows **logarithmically**, making it much more efficient for large datasets.

How FFT Works

The Core Idea: Decomposing the DFT

Instead of directly calculating the N-point DFT, the Fast Fourier Transform (FFT) leverages the inherent symmetry and periodicity of the complex exponential. It ingeniously re-expresses the sum by splitting the input sequence x_n into two interleaved parts: its even-indexed and odd-indexed elements.

The original Discrete Fourier Transform (DFT) is defined as:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i \frac{2\pi}{N} kn}$$

FFT then rewrites this sum by separating the terms for even ($n = 2m$) and odd ($n = 2m + 1$) indices, effectively breaking down one large DFT into two smaller ones:

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i \frac{2\pi}{N} k(2m)} + \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i \frac{2\pi}{N} k(2m+1)}$$

Recursive Breakdown and Twiddle Factors

This separation naturally leads to two distinct DFTs, each of half the original size (N/2 points):

- **E_k**: The DFT of the even-indexed elements.
- **O_k**: The DFT of the odd-indexed elements.

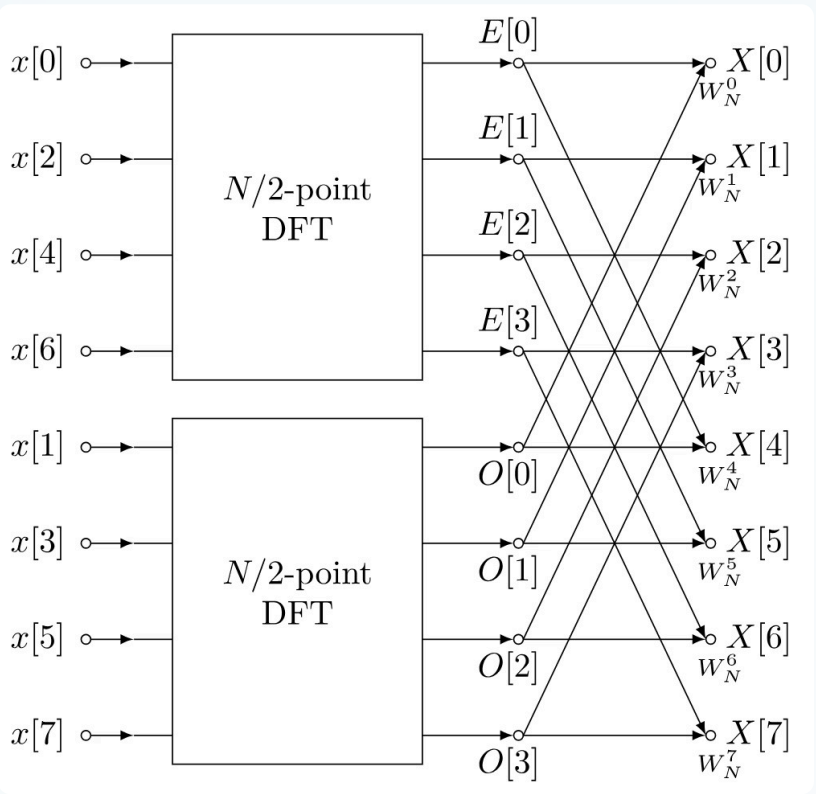
These two smaller DFTs are then combined using the following relationship:

$$X_k = E_k + W_N^k \cdot O_k$$

Where (**twiddle factor**):

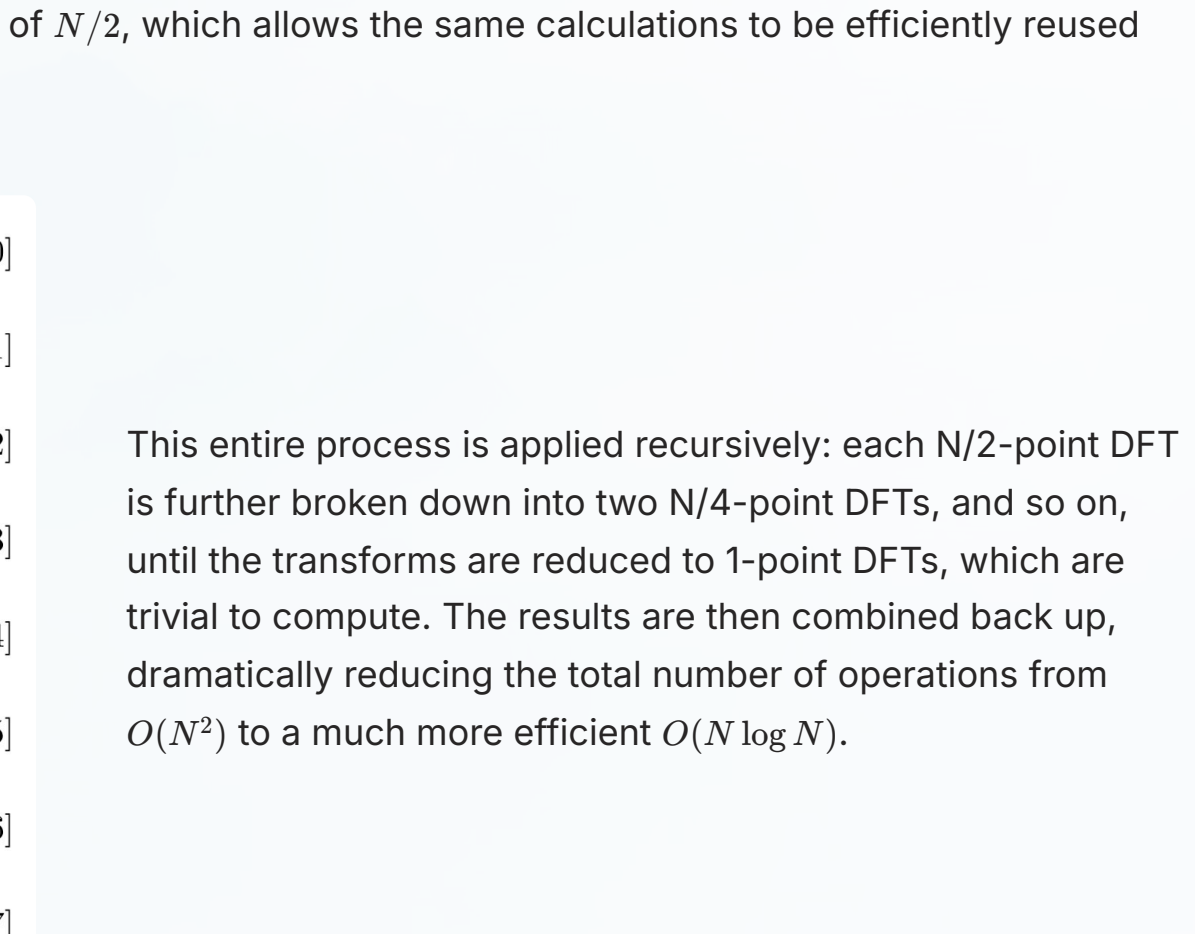
$$W_N^k = e^{-i \frac{2\pi k}{N}}$$

This factor accounts for the necessary phase shift when combining the results from the two smaller DFTs. A crucial property of the twiddle factor is its periodicity of $N/2$, which allows the same calculations to be efficiently reused for $X_{k+N/2}$.



This entire process is applied recursively: each N/2-point DFT is further broken down into two N/4-point DFTs, and so on, until the transforms are reduced to 1-point DFTs, which are trivial to compute. The results are then combined back up, dramatically reducing the total number of operations from $O(N^2)$ to a much more efficient $O(N \log N)$.

Time-based representation (above) and frequency-based representation (below) of the same signal, where the lower representation can be obtained from the upper one by Fourier transformation.



FFT Properties & Applications

The Fast Fourier Transform (FFT) significantly reduces computation by exploiting the inherent symmetry and periodicity of complex exponentials. For optimal performance, FFT works most efficiently when the input size (N) is a power of 2.

Audio Processing

Enhancing, compressing, and analyzing sound signals.

Image Processing

Filtering, compression, and feature extraction in images.

Data Compression

Reducing file sizes for efficient storage and transmission.

Signal Analysis

Identifying frequency components in complex signals.

Communications

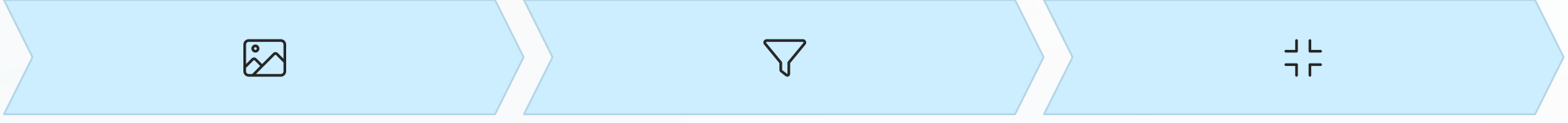
Modulation, demodulation, and channel equalization.

Radar Systems

Detecting objects, measuring speed, and mapping environments.

Image Compression Using FFT

The Fast Fourier Transform (FFT) plays a crucial role in image compression by allowing us to analyze and manipulate an image's frequency components. This process enables significant data reduction while preserving visual quality.



1. Spatial to Frequency Domain

Images are initially represented as pixel values in the spatial domain. A 2D FFT converts this into the frequency domain, mapping spatial variations to frequencies.

2. Filter High Frequencies

Low frequencies hold key image features (edges, textures). High frequencies often represent fine details or noise. We discard or reduce these high-frequency components to remove redundancy.

3. Reconstruct Compressed Image

An Inverse FFT (IFFT) transforms the filtered frequency data back to the spatial domain. The result is a smaller, yet visually acceptable, compressed image.

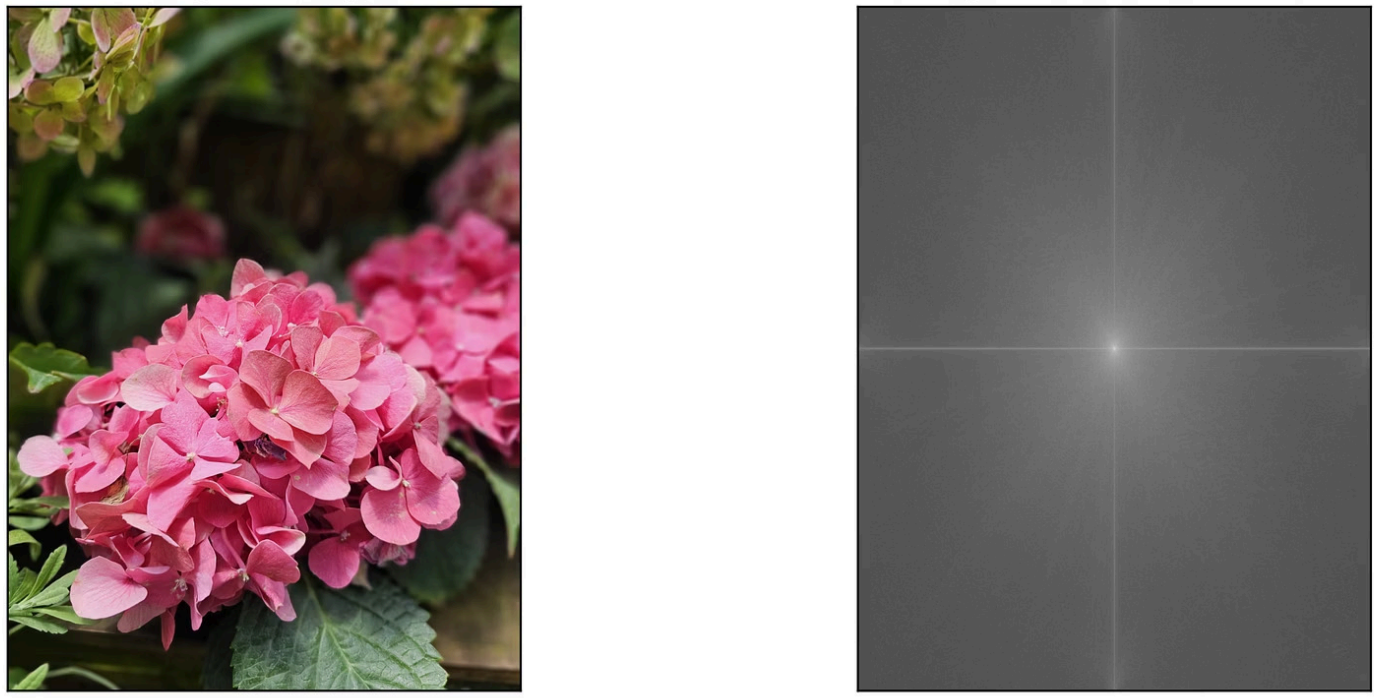
This method reduces data by removing parts of the image that people notice the least, making the file smaller without losing important details.

FFT Image Compression — Python Code

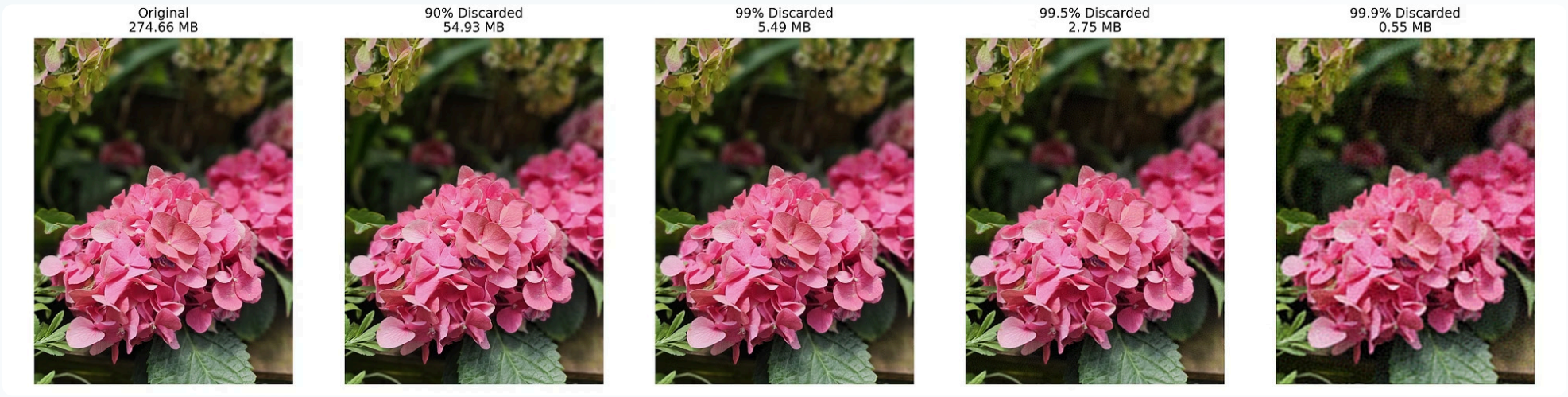
- Image is converted to frequency domain using 2D FFT (per color channel).
- Small-magnitude frequency coefficients are discarded based on a threshold.
- Inverse FFT is applied to reconstruct the image.
- Estimated size is computed from remaining non-zero coefficients.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

# ---------------------- Compression Functions ---------------------- #

def discard_less_important(img: np.ndarray, threshold) -> np.ndarray:
    """Discard 'threshold'% of small-magnitude coefficients."""
    sorted_coefficients: np.ndarray = np.sort(np.abs(img.flatten()))
    threshold_index = int((threshold / 100.0) * sorted_coefficients.shape[0])
    threshold_value = sorted_coefficients[threshold_index]
    mask = np.abs(img) > threshold_value
    new_coefficients = img * mask
    return new_coefficients

def discard_less_important_multi_channel(img, threshold):
    new_coefficients = np.zeros(img.shape, dtype=np.complex128)
    for channel in range(img.shape[2]):
        new_coefficients[:, :, channel] = discard_less_important(img[:, :, channel], threshold)
    return new_coefficients

# ---------------------- Load and Preprocess Image ---------------------- #

# Load color image in BGR, convert to RGB
image_bgr = cv2.imread('20240323_133304.jpg')
image = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)
image = image.astype(np.float64)

# Apply FFT to each channel
f_transform = np.zeros(image.shape, dtype=np.complex128)
for c in range(3):
    f_transform[:, :, c] = np.fft.fftshift(np.fft.fft2(image[:, :, c]))

# Show FFT magnitude spectrum of one channel (e.g., Red)
magnitude_spectrum = np.log(np.abs(f_transform[:, :, 0]) + 1)

plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1), plt.imshow(image.astype(np.uint8))
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(1, 2, 2), plt.imshow(magnitude_spectrum, cmap='gray')
plt.title('FFT Spectrum (Red Channel)'), plt.xticks([]), plt.yticks([])
plt.tight_layout()
plt.savefig("fft_spectrum_comparison.png", dpi=300)
plt.show()

# ---------------------- Compression and Reconstruction ---------------------- #

thresholds = [90, 99, 99.5, 99.9]
compressed_images = [np.clip(image, 0, 255).astype(np.uint8)]
estimated_sizes = []

original_size_mb = image.nbytes / (1024 * 1024)
estimated_sizes.append(original_size_mb)

for threshold_percent in thresholds:
    # Apply compression
    f_compressed = discard_less_important_multi_channel(f_transform, threshold_percent)

    # Estimate storage
    nonzero_coeffs = np.count_nonzero(f_compressed)
    estimated_size_bytes = nonzero_coeffs * 16
    estimated_size_mb = estimated_size_bytes / (1024 * 1024)
    estimated_sizes.append(estimated_size_mb)

    # Reconstruct image
    img_compressed = np.zeros(image.shape, dtype=np.float64)
    for c in range(3):
        f_ishift = np.fft.ifftshift(f_compressed[:, :, c])
        img_back = np.fft.ifft2(f_ishift)
        img_compressed[:, :, c] = np.abs(img_back)

    img_compressed = np.clip(img_compressed, 0, 255).astype(np.uint8)
    compressed_images.append(img_compressed)

# ---------------------- Plot All Results ---------------------- #

titles = [
    f'Original\n{estimated_sizes[0]:.2f} MB'
] + [
    f'{t}% Discarded\n{sz:.2f} MB' for t, sz in zip(thresholds, estimated_sizes[1:])
]

plt.figure(figsize=(20, 5))
for i, (img, title) in enumerate(zip(compressed_images, titles)):
    plt.subplot(1, 5, i + 1)
    plt.imshow(img)
    plt.title(title)
    plt.axis('off')

plt.tight_layout()
plt.savefig("compressed_image_comparison.png", dpi=300)
plt.show()
```

Compression Results



FFT Spectrum Visualization: Original image vs. its FFT magnitude spectrum



Compression Results with Estimated Sizes: Compressed images at increasing thresholds (90%, 99%, 99.5%, 99.9%)

As compression increases, the file size significantly drops, while visual quality experiences only a slight, often imperceptible, degradation.

Summary & Conclusion

- Fourier-based methods are essential tools in engineering mathematics for analyzing signals and systems.
- FFT allows efficient frequency-domain computation, reducing complexity from O(N^2) to O(NlogN).
- In our project, we used FFT to perform image compression by discarding high-frequency components.
- Results showed that significant data reduction is possible while preserving key image features.
- FFT has wide applications in signal processing, compression, filtering, and AI.

FFT bridges theory and practice — enabling real-world solutions through mathematical insight.