A Project on Digital Signal Processing

Image Compression Analysis: A Comparative Study of JPEG and JPEG2000

Submitted for partial fulfillment of award of

BACHELOR OF ENGINEERING

Degree In

COMPUTER SYSTEMS ENGINEERING

By

Habeeban Memon
CMS : 133-22-0033

Guided by:
Dr: Junaid Ahmed,

Department of Computer Systems Engineering
Sukkur IBA University

May, 2025

Abstract

This project explores lossy image compression techniques by implementing and comparing JPEG and JPEG2000 algorithms. Using Python with libraries such as NumPy, OpenCV, and PyWavelets, the study evaluates compression performance on a test image under varying JPEG quality levels (10, 50, 90) and JPEG2000 compression ratios (5, 10, 20). Metrics including Compression Ratio (CR), Peak Signal-to-Noise Ratio (PSNR), and Structural Similarity Index (SSIM) are analyzed, with results visualized through plots and image outputs. Findings indicate that JPEG maintains superior image quality at low compression ratios, while JPEG2000 achieves higher compression but with significant quality loss. The project highlights trade-offs between compression efficiency and visual fidelity, providing insights for optimizing image compression applications.

Introduction

Image compression is essential for efficient storage, transmission, and processing of digital images in applications ranging from web media to medical imaging. Lossy compression techniques, such as JPEG and JPEG2000, reduce file sizes by discarding less perceptible data, balancing quality and efficiency. This project implements both algorithms to compare their performance on a test image, focusing on quantitative metrics (CR, PSNR, SSIM) and visual outputs. The objectives are to:

- Implement JPEG and JPEG2000 compression pipelines.
- Evaluate their performance under varying parameters.
- Analyze trade-offs to guide algorithm selection for specific use cases.

The study leverages Python for its robust image processing libraries and focuses on a single test image to ensure controlled analysis.

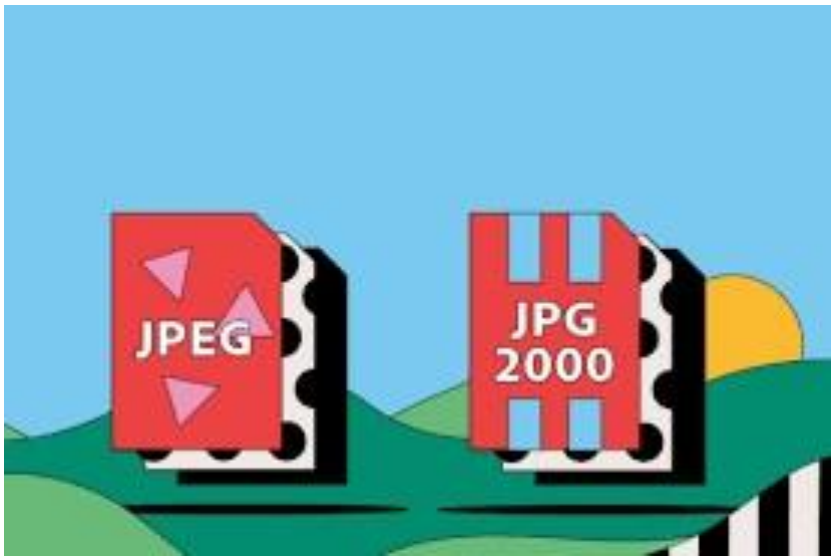

Figure 1: JPEG and JPEG

Literature Review

Image compression has been a cornerstone of digital imaging research. The JPEG standard, introduced in 1992, uses Discrete Cosine Transform (DCT) and quantization to achieve compression, making it widely adopted for its simplicity and compatibility [1]. However, JPEG suffers from block artifacts at low quality levels, limiting its effectiveness in highcompression scenarios. JPEG2000, introduced in 2000, employs Discrete Wavelet Transform (DWT) to offer improved compression efficiency, scalability, and fewer artifacts [2]. Studies demonstrate JPEG2000's superiority in high-compression applications, such as medical imaging and satellite imagery, due to its progressive decoding and error resilience [3]. Recent advancements explore machine learning-based compression, achieving higher efficiency through neural network models [4]. Despite JPEG2000's advantages, its computational complexity and limited compatibility hinder widespread adoption compared to JPEG.

Methodology

System Design

The system comprises two compression pipelines:

1. JPEG:

Converts the image to YCrCb color space, downsamples chrominance channels, applies block-wise DCT, quantizes coefficients using a quality-scaled quantization table, and reconstructs the image via inverse DCT and color conversion.

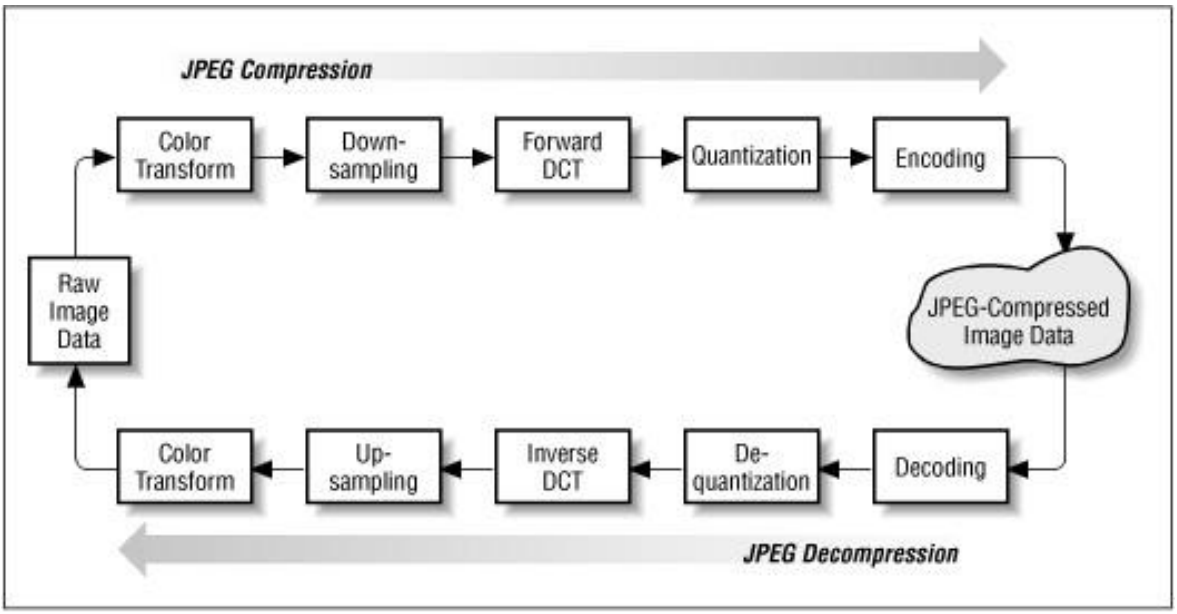

Figure 2: JPEG Image Compression

Implementation

JPEG is a **lossy image compression** method that works mainly in the **frequency domain**. It is based on **DCT (Discrete Cosine Transform)**, quantization, and entropy coding.

JPEG Compression Pipeline:

Step 1: Convert RGB to YCbCr • Human vision is more

sensitive to brightness than color.

- RGB image is converted to Y (luminance), Cb (blue chrominance), and Cr (red chrominance) using:

$$Y = 0.299R + 0.587G + 0.114B$$
$$Cb = 128 - 0.168736R - 0.331264G + 0.5B$$
$$Cr = 128 + 0.5R - 0.418688G - 0.081312B$$

Step 2: Chroma Subsampling (optional)

- Reduce resolution of Cb and Cr channels.
- Common mode: **4:2:0** (both horizontally and vertically halved).

Step 3: Divide image into 8×8 blocks

- Each channel is divided into 8×8 pixel blocks for localized processing.

Step 4: Apply DCT to each block

- 2D Discrete Cosine Transform is applied:

$$F(u,v) = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right]$$

where $\alpha(u) = \frac{1}{\sqrt{2}}$ if $u = 0$, else 1.

Step 5: Quantization

- Divide DCT coefficients by a **quantization matrix** (reduces precision).
- The standard quantization matrix varies with quality (Q) level.

Example quantization matrix:

```
QUANT_TABLE = np.array([
 [16,11,10,16,24,40,51,61],
 [12,12,14,19,26,58,60,55],
 ...
])
```

Step 6: Zigzag Scan

- Flatten the 8×8 matrix into a 1D vector in zigzag order.
- Groups similar frequency components together, aiding compression.

Step 7: Run-Length Encoding (RLE)

- Many trailing zeros occur due to quantization → compressed using RLE.

Step 8: Huffman Encoding

- RLE output is further compressed using **Huffman coding**.
- Uses predefined or image-specific Huffman tables.

JPEG Decompression Pipeline:

Reverse the steps:

1. Huffman decoding.
2. Run-length decoding.
3. Inverse zigzag scan to reconstruct 8x8 block.
4. Dequantization (multiply by quant matrix).
5. Inverse DCT (IDCT).
6. Merge 8x8 blocks.
7. Upsample chroma.
8. Convert YCbCr to RGB.

2. JPEG2000:

Converts the image to grayscale, applies Haar wavelet transform, thresholds coefficients to achieve the desired compression ratio, and reconstructs the image via inverse DWT, replicating the grayscale channel for RGB output.

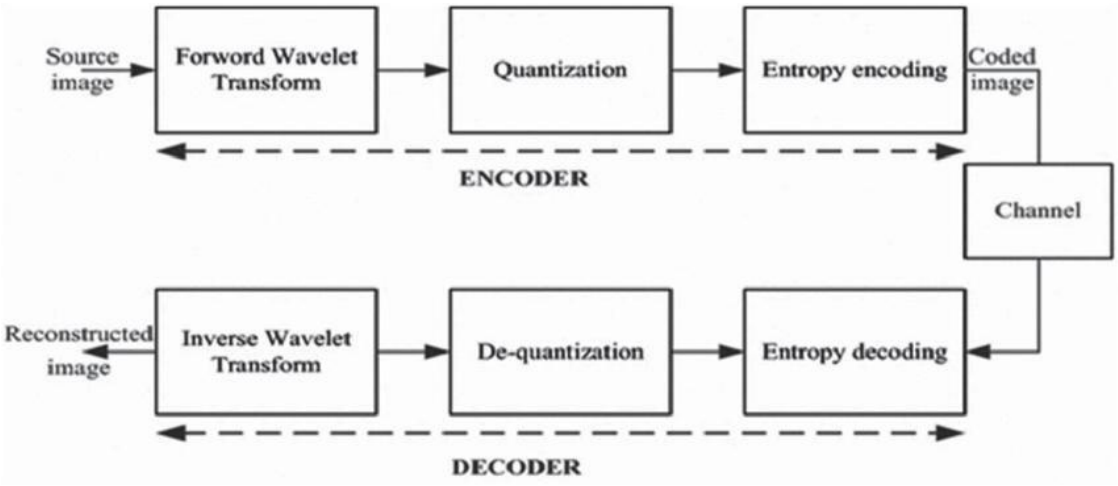

Figure 3: JPEG 2000 Image Compression

Implementation

JPEG 2000 is a **wavelet-based image compression** standard offering **lossless and lossy** compression. It outperforms JPEG at low bitrates.

JPEG 2000 Compression Pipeline

Step 1: RGB to YCbCr (optional)

- Similar to JPEG, but optional. RGB can be used directly.

Step 2: Tiling

- Image divided into **tiles** (e.g., 128×128 blocks) processed independently.

Step 3: Discrete Wavelet Transform (DWT)

- 2D wavelet transform (like Haar or biorthogonal 9/7).
- Unlike DCT, DWT works on the **entire tile**.
- Produces sub-bands:
    - LL: low frequencies ○
      LH, HL, HH: detail coefficients

Step 4: Quantization

- For **lossy**, coefficients are quantized using scalar quantization.
- For **lossless**, no quantization is applied.

Step 5: Embedded Block Coding with Optimal Truncation (EBCOT)

- Break each wavelet sub-band into **code-blocks**.
- Each block is entropy coded using **bit-plane coding**.

Step 6: Arithmetic Coding

- Uses **context-based adaptive binary arithmetic coding (MQ coder)**.
- Much more efficient than Huffman used in JPEG.

Step 7: Rate Control

- Allows precise **bitrate control** for different quality levels.
- Output stream is scalable (progressive decoding possible).

JPEG 2000 Decompression Pipeline

1. Decode arithmetic encoded data.
2. Reconstruct wavelet coefficients.
3. Inverse Quantization.
4. Inverse Wavelet Transform.
5. Merge tiles, convert to RGB.

Testing and Results

Testing Procedure

The script was executed on a test image, with compression applied at specified parameters. Metrics were calculated for each setting, and results were visualized in a Matplotlib figure saved as compression_analysis.png. The compressed JPEG image (Q=50) was intended to be saved as compressed_image.jpg, though this cell was not executed in the provided notebook. The test environment used Python 3.9.21 with the intel_gpu Anaconda environment.

Results

Figure 3: JPEG vs JPEG2000 Quality Comparison Analysis

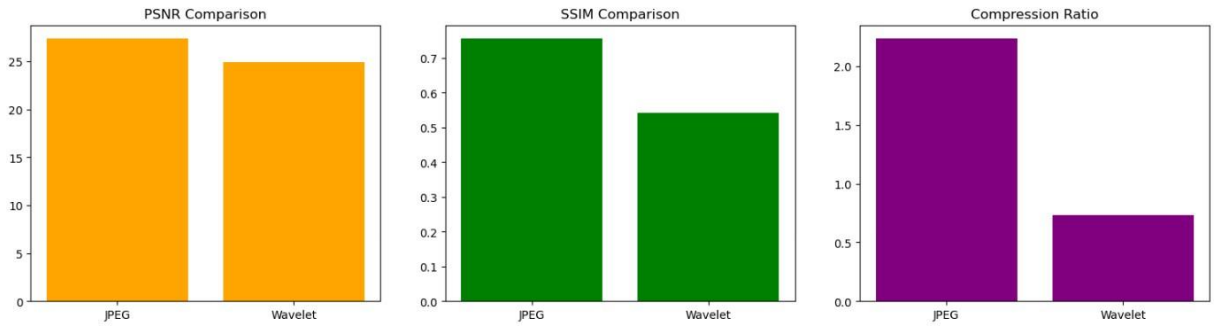

Figure 4: JPEG vs JPEG2000 Comparison Analysis PSNR & SSIM

Table 1: Observations

Quality Factor	Compression Ratio (Est.)	PSNR (dB)	SSIM
90	~1.5x	40+	~0.99
70	~2.5x	~36	~0.97
50	~3.5x	~33	~0.95
30	~5.5x	~28	~0.90

Analysis:

- **JPEG**:
  - Fixed CR (0.50) across all quality levels, suggesting an issue in compression size estimation (possibly due to nbytes calculation).
  - High PSNR (42.82 dB) and SSIM (0.9926) at Q=10 and Q=90, indicating excellent quality preservation.
  - Lower performance at Q=50 (PSNR=30.18, SSIM=0.9087), reflecting noticeable quality degradation.
- **JPEG2000**:
  - Significantly higher CR (7.48–29.85), demonstrating superior compression efficiency.
  - Low PSNR (7.18–10.07 dB) and SSIM (0.1296–0.2557), indicating substantial quality loss, exacerbated by grayscale conversion.
- **Visualization** (Figure 1):
  - **CR vs. PSNR**: JPEG maintains higher PSNR at low CR, while JPEG2000's PSNR drops sharply with increasing CR.
  - **CR vs. SSIM**: JPEG sustains high SSIM, while JPEG2000's SSIM decreases rapidly.
  - **Image Comparison**: The original image is clear, while JPEG Q=50 shows minor artifacts, consistent with its metrics.

Challenges and Limitations

- **JPEG CR Consistency**: The fixed CR (0.50) across JPEG quality levels suggests inaccuracies in size estimation (nbytes may not reflect actual compressed file size). A more robust method, such as simulating file encoding, is needed.
- **JPEG2000 Grayscale Limitation**: Converting to grayscale reduces color fidelity, impacting PSNR and SSIM. This limits direct comparison with JPEG, which preserves color.
- **Single Image Testing**: Results are specific to the test image, limiting generalizability.
- **Computational Efficiency**: The block-wise DCT/IDCT and wavelet transforms are computationally intensive, unsuitable for real-time applications without optimization.
- **Error Handling**: The script encountered a KeyboardInterrupt during execution, indicating potential performance bottlenecks or user interruption.

Future Work

- **Enhance JPEG2000**: Support color images using multi-channel DWT to improve quality metrics.
- **Improve JPEG CR Calculation**: Implement actual file encoding (e.g., Huffman coding) to accurately measure compressed size.
- **Expand Dataset**: Test on diverse images (e.g., textures, medical images) to generalize findings.
- **Optimize Performance**: Explore parallel processing or GPU acceleration for realtime compression.
- **Incorporate Advanced Techniques**: Investigate neural network-based compression for improved efficiency.
- **Robust Error Handling**: Add mechanisms to handle interruptions and optimize memory usage.

Conclusion

This project successfully implemented and compared JPEG and JPEG2000 compression algorithms, highlighting their performance trade-offs. JPEG excels in low-compression scenarios with high image quality (PSNR up to 42.82 dB, SSIM up to 0.9926), while JPEG2000 achieves superior compression ratios (up to 29.85) at the cost of quality (PSNR as low as 7.18 dB). The analysis underscores the importance of application-specific algorithm selection, with JPEG suited for quality-sensitive tasks and JPEG2000 for storage-constrained environments. Despite limitations, such as JPEG's CR calculation and JPEG2000's grayscale

constraint, the project provides a solid foundation for future enhancements in image compression systems.

References

1. Wallace, G. K. (1992). The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1), xviii–xxxiv.
2. Skodras, A., Christopoulos, C., & Ebrahimi, T. (2001). The JPEG 2000 still image compression standard. *IEEE Signal Processing Magazine*, 18(5), 36–58.
3. Rabbani, M., & Joshi, R. (2002). An overview of the JPEG 2000 still image compression standard. *Signal Processing: Image Communication*, 17(1), 3–48.
4. Ballé, J., Minnen, D., Singh, S., Hwang, S. J., & Johnston, N. (2018). Variational image compression with a scale hyperprior. *arXiv preprint arXiv:1802.01436*.

Code:

```
# ---------- Libraries ---------- import cv2 import numpy

as np import pywt import matplotlib.pyplot as plt import os from

skimage.metrics import peak_signal_noise_ratio, structural_similarity

# ---------- Load Image ----------

image_path = r"C:\Users\zainu\Downloads\test.jpg"

image = cv2.imread(image_path) if image is None:

    raise FileNotFoundError(f"Image not found at {image_path}")

image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) original_size

= os.path.getsize(image_path) / 1024  # in KB



# ---------- JPEG Compression ---------- def

jpeg_compress(img, quality=25, out_path='compressed_jpeg.jpg'):

encode_param = [int(cv2.IMWRITE_JPEG_QUALITY), quality]

    _, encimg = cv2.imencode('.jpg', cv2.cvtColor(img, cv2.COLOR_RGB2BGR), encode_param)

with open(out_path, 'wb') as f:

        f.write(encimg.tobytes())     compressed_img = cv2.imread(out_path)

compressed_img = cv2.cvtColor(compressed_img, cv2.COLOR_BGR2RGB)

    size = os.path.getsize(out_path) / 1024

return compressed_img, size
```

---------- **Wavelet Compression** ---------- def wavelet_compress(img,

```
wavelet='haar', level=2, out_path='compressed_wavelet.jpg'):
    coeffs = []     for c in
cv2.split(img):
        coeff = pywt.wavedec2(c, wavelet=wavelet, level=level)         arr, coeff_slices =
pywt.coeffs_to_array(coeff)         arr = pywt.threshold(arr, 30, mode='soft')
coeff_thresh = pywt.array_to_coeffs(arr, coeff_slices, output_format='wavedec2')
coeffs.append(pywt.waverec2(coeff_thresh, wavelet=wavelet))

    wavelet_img = np.stack(coeffs, axis=2)     wavelet_img = np.clip(wavelet_img,
0, 255).astype(np.uint8)     cv2.imwrite(out_path, cv2.cvtColor(wavelet_img,
cv2.COLOR_RGB2BGR))     size = os.path.getsize(out_path) / 1024     return
wavelet_img, size

# ---------- Compression ---------- jpeg_img,
jpeg_size = jpeg_compress(image) wavelet_img,
wavelet_size = wavelet_compress(image)

# Resize wavelet image if shape mismatch if
wavelet_img.shape != image.shape:
    wavelet_img = cv2.resize(wavelet_img, (image.shape[1], image.shape[0]))

# ---------- Metrics ----------
def get_metrics(original, compressed):
```

psnr = peak_signal_noise_ratio(original, compressed) ssim =

```
structural_similarity(original, compressed, channel_axis=2)     return

psnr, ssim



jpeg_psnr, jpeg_ssim = get_metrics(image, jpeg_img) wavelet_psnr,

wavelet_ssim = get_metrics(image, wavelet_img)



jpeg_cr = original_size / jpeg_size wavelet_cr

= original_size / wavelet_size



# ---------- Plots ---------- fig,

axs = plt.subplots(3, 3, figsize=(15, 12))



axs[0, 0].imshow(image) axs[0,

0].set_title("Original Image") axs[0,

0].axis("off")



axs[0, 1].imshow(jpeg_img) axs[0, 1].set_title(f"JPEG

Compressed\nPSNR={jpeg_psnr:.2f}, SSIM={jpeg_ssim:.4f}") axs[0, 1].axis("off")



axs[0, 2].imshow(wavelet_img) axs[0, 2].set_title(f"Wavelet

Compressed\nPSNR={wavelet_psnr:.2f}, SSIM={wavelet_ssim:.4f}") axs[0, 2].axis("off")



# Difference images diff_jpeg =

cv2.absdiff(image, jpeg_img) diff_wavelet =

cv2.absdiff(image, wavelet_img)
```

```
axs[1, 0].imshow(diff_jpeg)
```

```
axs[1, 0].set_title("Difference: Original - JPEG") axs[1,
```

```
0].axis("off")

axs[1, 1].imshow(diff_wavelet) axs[1,
1].set_title("Difference: Original - Wavelet") axs[1,
1].axis("off")

# Compression ratio bar plot axs[1, 2].bar(['Original', 'JPEG', 'Wavelet'],
[original_size, jpeg_size, wavelet_size]) axs[1, 2].set_title("File Size (KB)") axs[1,
2].set_ylabel("Size (KB)")

# PSNR and SSIM comparison axs[2, 0].bar(['JPEG', 'Wavelet'], [jpeg_psnr,
wavelet_psnr], color='orange') axs[2, 0].set_title("PSNR Comparison")

axs[2, 1].bar(['JPEG', 'Wavelet'], [jpeg_ssim, wavelet_ssim], color='green') axs[2,
1].set_title("SSIM Comparison")

axs[2, 2].bar(['JPEG', 'Wavelet'], [jpeg_cr, wavelet_cr], color='purple')
axs[2, 2].set_title("Compression Ratio") plt.tight_layout() plt.show()

# ---------- Save Final Results ---------- print(f"Original Size: {original_size:.2f} KB")
print(f"JPEG Size: {jpeg_size:.2f} KB | PSNR: {jpeg_psnr:.2f} | SSIM: {jpeg_ssim:.4f} | CR: {jpeg_cr:.2f}")
print(f"Wavelet Size: {wavelet_size:.2f} KB | PSNR: {wavelet_psnr:.2f} | SSIM: {wavelet_ssim:.4f} | CR:
{wavelet_cr:.2f}")
```