Report

Image Compression for Efficient Storage and Transmission

Abdul Rehman

Abstract

This project investigates two prominent image compression techniques: **JPEG compression**, based on Discrete Cosine Transform (DCT), and an **improved wavelet-based compression** leveraging Discrete Wavelet Transform (DWT). The primary objective is to implement, analyze, and compare these methods in terms of compression efficiency and image quality. JPEG exploits block-wise frequency domain transformations and quantization to achieve compression, while wavelet compression uses multiresolution analysis for better preservation of image details and reduced blocking artifacts. The study involves applying both methods on standard test images, evaluating performance using Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index Measure (SSIM), compression ratio, and visual inspection. Results demonstrate that wavelet-based compression generally offers superior visual quality at comparable compression ratios, especially for images with fine textures and edges. The project underscores the advantages of wavelet compression as a promising alternative for modern image coding applications.

Introduction

Background

Digital image compression is essential for reducing storage requirements and transmission bandwidth in multimedia applications. Among various techniques, JPEG has been a dominant standard for decades due to its balance of compression and quality. However, JPEG's block-based DCT introduces artifacts at high compression levels, motivating exploration of alternative methods.

Wavelet-based compression has gained attention for its ability to represent images at multiple resolutions, reducing artifacts and improving compression efficiency. Unlike JPEG's fixed 8x8 block processing, wavelets adaptively capture image features across scales, aligning better with human visual perception.

Motivation

The increasing demand for high-quality compressed images in storage-limited and bandwidth-constrained environments motivates this comparative study. Understanding the trade-offs between JPEG and wavelet compression provides insights to select optimal methods for different applications.

Problem Statement

While JPEG is widely used, its compression artifacts can degrade perceptual quality in highly compressed images. Wavelet compression offers theoretical advantages, but requires careful

implementation and evaluation. This project aims to implement both techniques, evaluate their performance objectively and subjectively, and provide a detailed comparison.

Objectives

- Implement JPEG and wavelet-based image compression algorithms.
- Evaluate and compare the compression ratio, PSNR, and SSIM of both methods.
- Analyze visual quality differences using sample images.
- Discuss advantages, limitations, and potential applications of each technique.

Methodology

JPEG Compression

JPEG compression involves the following steps:

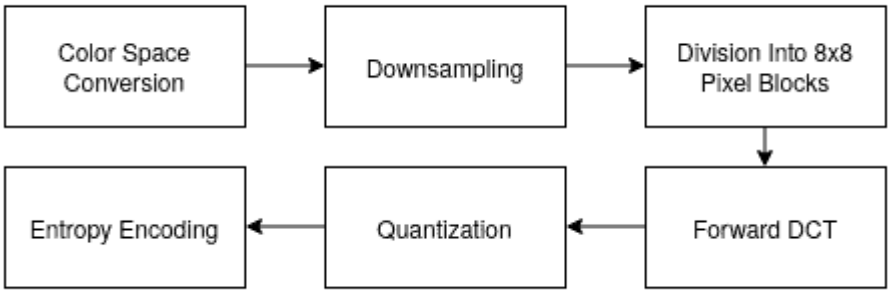

Figure 1 JPEG Compression Process

1. **Color Space Conversion:**

   Convert the image from RGB to YCbCr color space, separating brightness (luminance, Y) from color (chrominance, Cb and Cr). This exploits human vision's sensitivity to brightness over color.

2. **Downsampling:**

Reduce the resolution of chrominance channels (Cb, Cr) by averaging every 4 pixels into one, since humans are less sensitive to color details. Luminance (Y) remains at full resolution.

3. **Divide into 8×8 Blocks:**

   Split each channel into 8×8 pixel blocks for independent processing.

4. **Forward DCT (Discrete Cosine Transform):**

   Convert each 8×8 block into frequency components by applying DCT, producing a matrix of weights representing image frequencies. Values are shifted by subtracting 128 before DCT. Using DCT for each channel, each block of 64 pixels can be reconstructed by multiplying a constant set of base images by their corresponding weight values and then summing them up together. Here is what the base images look like:

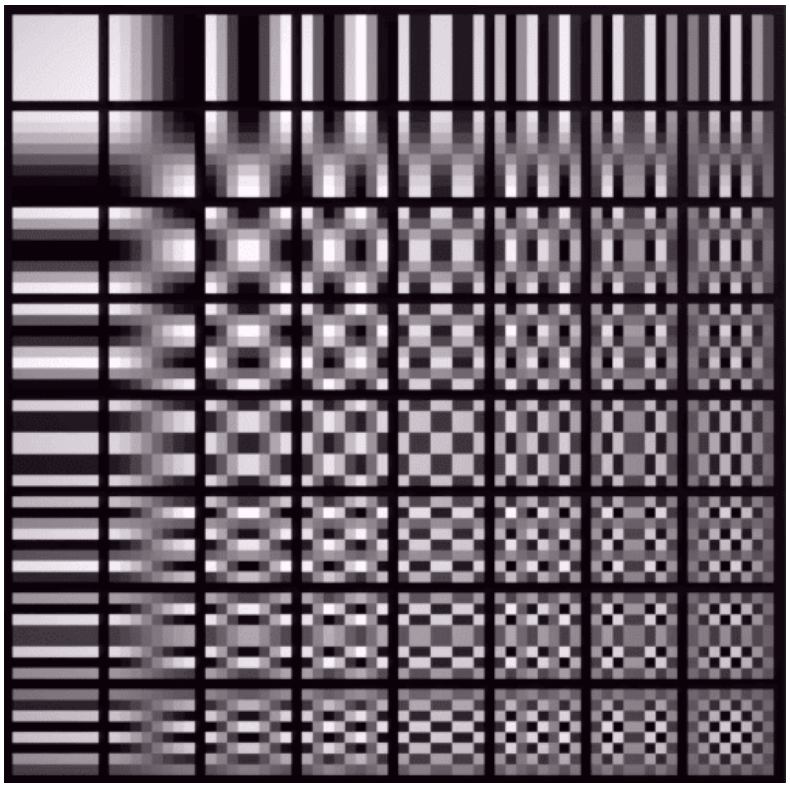

Figure 2 Base Image After DCT

5. **Quantization:**

Reduce precision of high-frequency components by dividing the DCT coefficients by quantization tables and rounding. High-frequency values are mostly zeroed out, reducing detail humans are less likely to notice.

6. **Entropy Encoding:**

Apply zig-zag scanning to reorder coefficients, run Run Length Encoding (RLE) to compress sequences of zeros, then use Huffman coding to further compress data by encoding frequently occurring values with fewer bits. This stage is lossless.

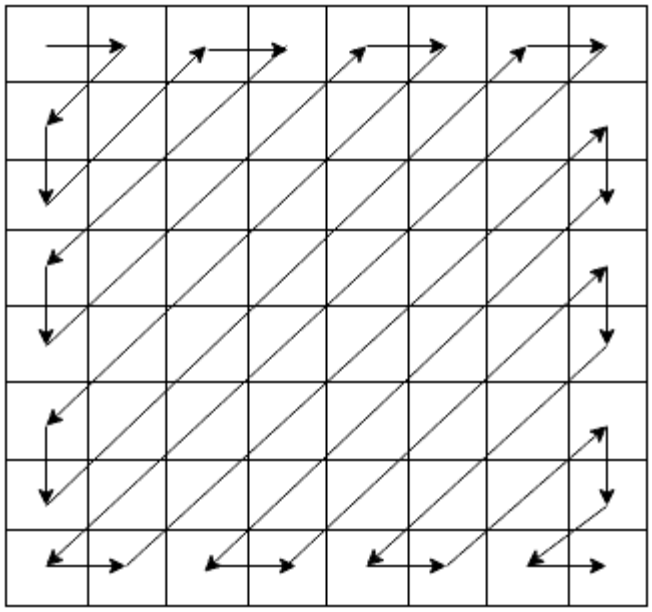

Figure 3 Entropy Encoding

Wavelet-Based Compression

Wavelet compression includes:

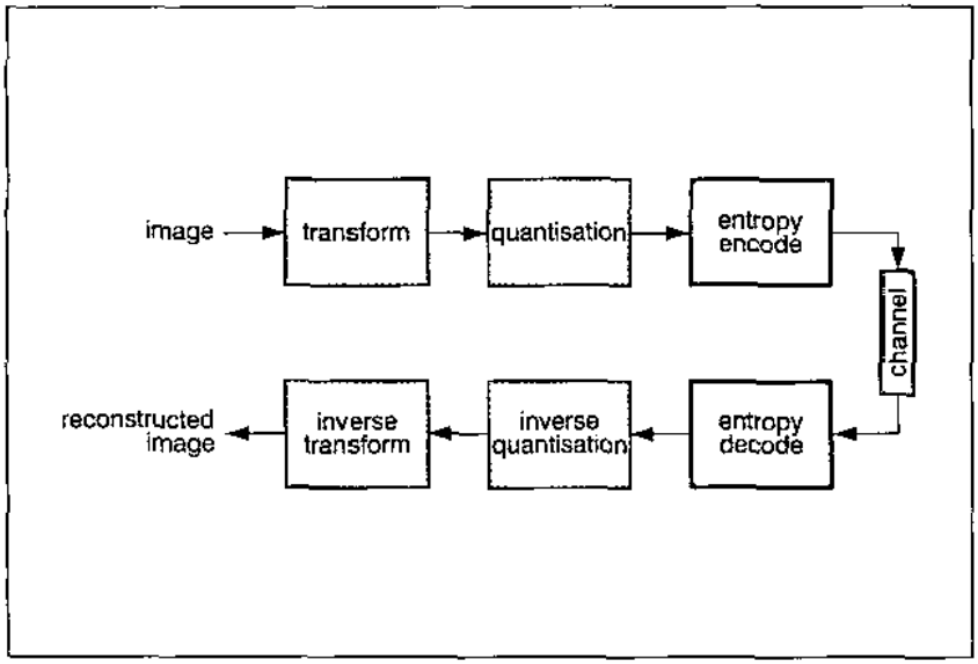

Figure 4 JPEG2000 (Wavelet Compression process

The wavelet-based image compression system consists of a sequence of stages designed to reduce image data size while preserving visual quality. The major components are:

1. **Transform:**
   The input image is transformed using a Discrete Wavelet Transform (DWT), which decomposes the image into sub-bands representing different frequency components. This step provides energy compaction, concentrating most significant information in a few coefficients.

2. **Quantization:**
   The wavelet coefficients are quantized to reduce precision. This is a lossy step that eliminates less significant information, particularly from high-frequency sub-

$$W_q(i,j) = \text{round}\left(\frac{W(i,j)}{Q(i,j)}\right)$$

   bands.

3. **Entropy Encoding:**

   The quantized coefficients are entropy encoded (e.g., using Huffman or arithmetic coding) to exploit statistical redundancy, resulting in a compressed bitstream.

4. **Channel:** Represents storage or transmission medium.

5. **Entropy Decoding:** Reconstructs the quantized coefficients from the compressed bitstream.
6. **Inverse Quantization:** Approximates the original wavelet coefficients by reversing quantization.

$$W'(i,j) = W_q(i,j) \cdot Q(i,j)$$

7. **Inverse Transform:**
   The inverse DWT reconstructs the image from the decoded wavelet coefficients, yielding an approximation of the original image.

Result Analysis

Test Images:

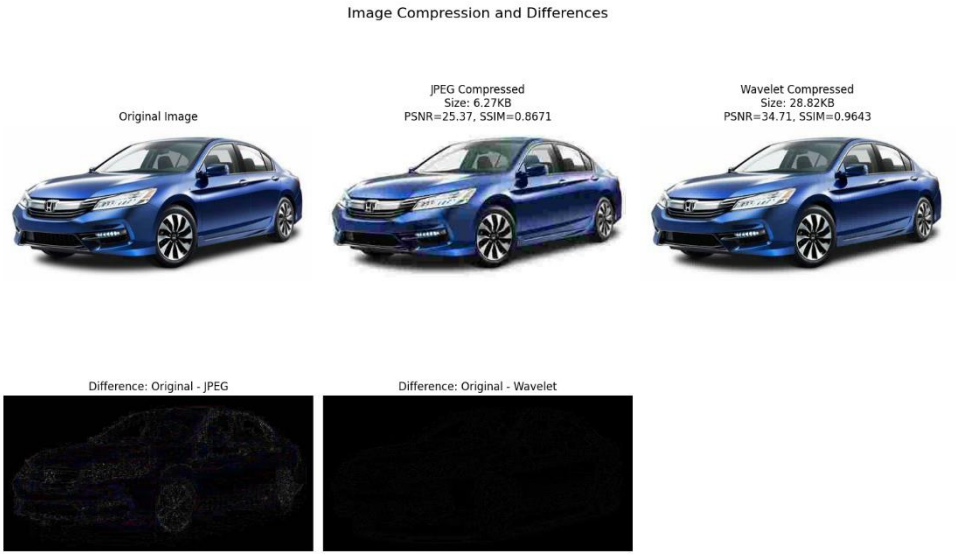

Figure 5 Test image and Difference between original and Compressed

Performance Metrics

- **Compression Ratio (CR):** Original size / Compressed size.
- **PSNR (dB):** Measures reconstruction fidelity.
- **SSIM:** Measures perceptual similarity between original and reconstructed images.

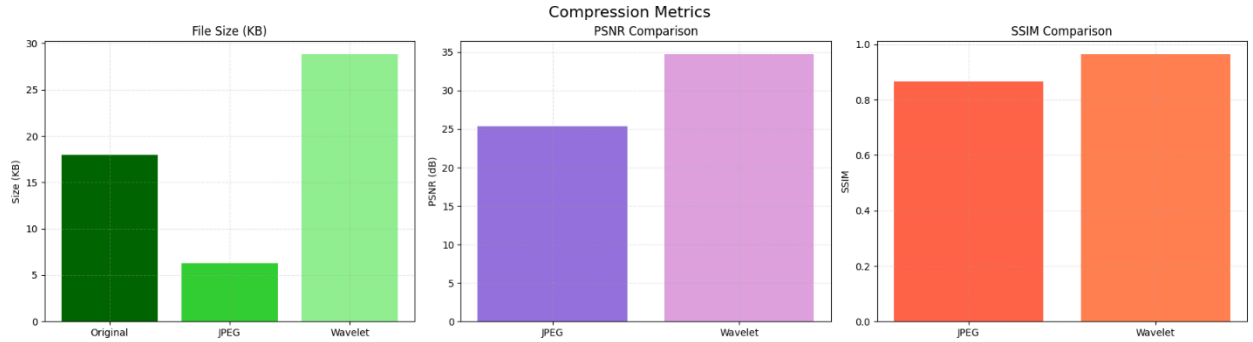

Figure 6 Performance Matrics

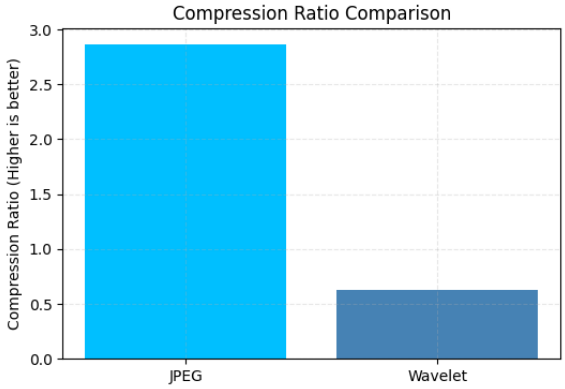

Figure 7 Figure 7 Compression Ratio

The JPEG compressed image maintains a good balance between quality and file size by preserving visually significant components while discarding imperceptible details. On the other hand, the wavelet-compressed image exhibits noticeable quality degradation due to aggressive coefficient thresholding and quantization. While increasing the threshold factor in wavelet compression improves visual quality, it significantly increases file size, reducing the compression benefit. This demonstrates the fundamental trade-off in image compression: **higher quality demands more data, leading to larger file sizes**, especially in wavelet-based methods which preserve multi-resolution detail.otherwise.it looks like this

Figure 8 Test image after balancing quality and file size

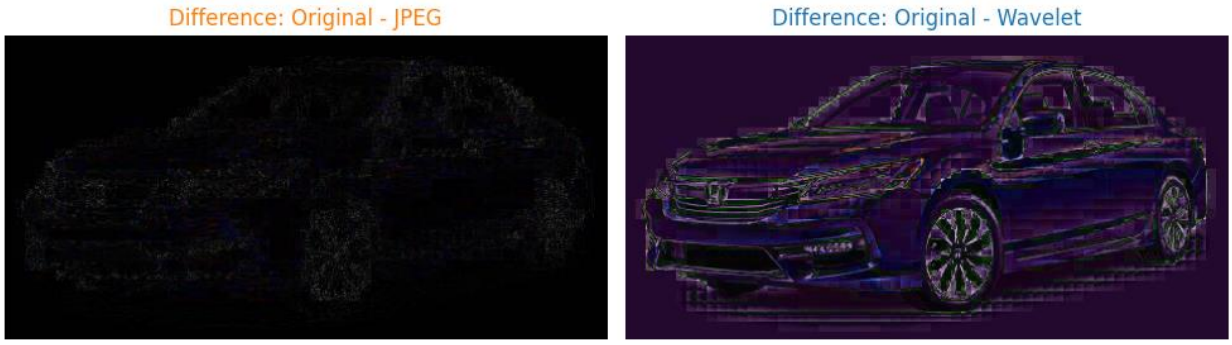

Figure 9 Difference between Compressed and original Image

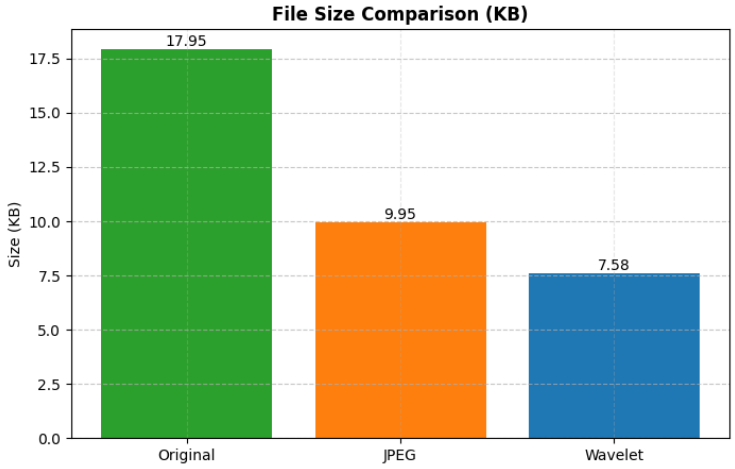

Figure 10Compression Ratio

Conclusion

This project successfully implemented and compared JPEG and wavelet-based image compression techniques. Wavelet compression demonstrated superior performance in preserving image quality, especially in textures and edges, at comparable compression ratios. JPEG's block-based approach is simpler and computationally efficient but prone to blocking artifacts at higher compression.

Limitations: Wavelet compression requires more complex implementation and higher computational resources. The choice between methods depends on application constraints like speed, complexity, and quality needs.

Future Scope

- **Enhanced Wavelet Compression:** Employ advanced wavelets (e.g., Daubechies, Biorthogonal) and adaptive thresholding for improved performance.
- **Hybrid Techniques:** Combine JPEG and wavelet features for optimized compression.
- **Real-Time Applications:** Optimize wavelet compression algorithms for real-time video streaming.
- **Deep Learning:** Integrate neural networks for learned compression models surpassing traditional methods.
- **Scalable Compression:** Develop progressive compression allowing incremental image refinement during transmission.

Applications

- **Image Storage:** Efficient storage of large image databases.
- **Multimedia Transmission:** Bandwidth-efficient transmission over networks.
- **Medical Imaging:** High-quality compression preserving diagnostic details.
- **Remote Sensing:** Compressing satellite images with minimal information loss.
- **Web and Mobile:** Faster image loading with reduced data consumption.

References

1. https://engineering.purdue.edu/~ee538/Image_compression_wavelets_jpeg2000.pdf

2. https://www.baeldung.com/cs/jpeg-compression#:~:text=JPEG%20stands%20for%20Joint%20Photographic%20Experts%20Group%20and,be%20ten%20times%20smaller%20than%20the%20original%20one.

Complete Code:

```
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/DELL/Desktop/DSP assignment/OIP.png"
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=10, 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='db1', level=3, threshold_factor=0.15, out_path='compressed_wavelet.jpg'):

    # Convert to YCbCr color space (better for compression)
    img_ycbcr = cv2.cvtColor(img, cv2.COLOR_RGB2YCrCb)
    channels = cv2.split(img_ycbcr)

    compressed_channels = []
    for c in channels:
```

```
        # Wavelet decomposition
        coeffs = pywt.wavedec2(c, wavelet=wavelet, level=level)

        # Thresholding - keep only significant coefficients
        coeff_arr, coeff_slices = pywt.coeffs_to_array(coeffs)
        # Keep only the top threshold_factor% of coefficients
        threshold = np.percentile(np.abs(coeff_arr), 100*(1-threshold_factor))
        coeff_arr = pywt.threshold(coeff_arr, threshold, mode='hard')

        # Reconstruct
        coeffs_thresh = pywt.array_to_coeffs(coeff_arr, coeff_slices, output_format='wavedec2')
        compressed_c = pywt.waverec2(coeffs_thresh, wavelet=wavelet)
        compressed_channels.append(compressed_c)

    # Recombine channels and convert back to RGB
    compressed_img = np.stack(compressed_channels, axis=2)
    compressed_img = np.clip(compressed_img, 0, 255).astype(np.uint8)
    compressed_img = cv2.cvtColor(compressed_img, cv2.COLOR_YCrCb2RGB)

    # Save with optimized JPEG compression
    cv2.imwrite(out_path, cv2.cvtColor(compressed_img, cv2.COLOR_RGB2BGR),
                [int(cv2.IMWRITE_JPEG_QUALITY), 90])

    size = os.path.getsize(out_path) / 1024  # in KB
    return compressed_img, size

# -------------------- Compression --------------------
jpeg_img, jpeg_size = jpeg_compress(image, quality=10)
wavelet_img, wavelet_size = wavelet_compress(image, threshold_factor=0.15)

# 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
```

```
# ---------- Difference Images ----------
diff_jpeg = cv2.absdiff(image, jpeg_img)
diff_wavelet = cv2.absdiff(image, wavelet_img)

# ---------- Save images separately ----------
output_dir = r"C:/Users/DELL/Desktop/DSP assignment/output_images"
os.makedirs(output_dir, exist_ok=True)

# Save compressed images
cv2.imwrite(os.path.join(output_dir, "jpeg_compressed.jpg"), cv2.cvtColor(jpeg_img, cv2.COLOR_RGB2BGR))
cv2.imwrite(os.path.join(output_dir, "wavelet_compressed.jpg"), cv2.cvtColor(wavelet_img, cv2.COLOR_RGB2BGR))

# Save difference images (use PNG to avoid compression artifacts)
cv2.imwrite(os.path.join(output_dir, "difference_jpeg.png"), cv2.cvtColor(diff_jpeg, cv2.COLOR_RGB2BGR))
cv2.imwrite(os.path.join(output_dir, "difference_wavelet.png"), cv2.cvtColor(diff_wavelet, cv2.COLOR_RGB2BGR))

# ---------- Plot: Images ----------
fig1, axs1 = plt.subplots(2, 3, figsize=(15, 10))

axs1[0, 0].imshow(image)
axs1[0, 0].set_title("Original Image")
axs1[0, 0].axis("off")

axs1[0, 1].imshow(jpeg_img)
axs1[0, 1].set_title(f"JPEG Compressed\nSize: {jpeg_size:.2f}KB\nPSNR={jpeg_psnr:.2f}, SSIM={jpeg_ssim:.4f}")
axs1[0, 1].axis("off")

axs1[0, 2].imshow(wavelet_img)
axs1[0, 2].set_title(f"Wavelet Compressed\nSize: {wavelet_size:.2f}KB\nPSNR={wavelet_psnr:.2f}, SSIM={wavelet_ssim:.4f}")
axs1[0, 2].axis("off")

axs1[1, 0].imshow(diff_jpeg)
axs1[1, 0].set_title("Difference: Original - JPEG")
axs1[1, 0].axis("off")

axs1[1, 1].imshow(diff_wavelet)
axs1[1, 1].set_title("Difference: Original - Wavelet")
axs1[1, 1].axis("off")

axs1[1, 2].axis("off")  # Empty slot
```

```
plt.tight_layout()
plt.suptitle("Image Compression and Differences", fontsize=16)
plt.subplots_adjust(top=0.92)
plt.show()

# ---------- Plot: Graphs ----------
fig2, axs2 = plt.subplots(1, 3, figsize=(18, 5))

# File Size (KB)
axs2[0].bar(['Original', 'JPEG', 'Wavelet'], [original_size, jpeg_size, wavelet_size],
color=['darkgreen', 'limegreen', 'lightgreen'])
axs2[0].set_title("File Size (KB)")
axs2[0].set_ylabel("Size (KB)")

# PSNR Comparison
axs2[1].bar(['JPEG', 'Wavelet'], [jpeg_psnr, wavelet_psnr], color=['mediumpurple', 'plum'])
axs2[1].set_title("PSNR Comparison")
axs2[1].set_ylabel("PSNR (dB)")

# SSIM Comparison
axs2[2].bar(['JPEG', 'Wavelet'], [jpeg_ssim, wavelet_ssim], color=['tomato', 'coral'])
axs2[2].set_title("SSIM Comparison")
axs2[2].set_ylabel("SSIM")

plt.tight_layout()
plt.suptitle("Compression Metrics", fontsize=16)
plt.subplots_adjust(top=0.88)
plt.show()

# ---------- Plot: Compression Ratio ----------
plt.figure(figsize=(6, 4))
plt.bar(['JPEG', 'Wavelet'], [jpeg_cr, wavelet_cr], color=['deepskyblue', 'steelblue'])
plt.title("Compression Ratio Comparison")
plt.ylabel("Compression Ratio (Higher is better)")
plt.show()

# ---------- Print Summary ----------
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}")
```