Assignment No. 7

Title

Implementation and Analysis of Image Compression using Discrete Cosine Transform (DCT) and Huffman Coding

1. Problem Statement

Image data consumes large amounts of storage and bandwidth. The aim of this assignment is to implement an image compression pipeline that combines block-based Discrete Cosine Transform (DCT) with quantization (lossy stage) and Huffman coding (lossless stage). Using an open-source image dataset, you will measure and compare the effects of lossy compression (DCT + quantization) and lossless compression (Huffman coding on residuals or transformed coefficients) in terms of visual quality and compression ratio. Based on the experimental results, recommend suitable compression techniques for different application scenarios.

2. Aim

To design, implement, and analyze an image compression system that uses DCT for energy compaction and Huffman coding for entropy coding; and to evaluate the trade-offs between lossy and lossless compression on an open-source image dataset.

3. Objectives

1. Implement block-wise 2D DCT and inverse DCT (IDCT) for grayscale images.
2. Apply quantization on DCT coefficients to create a lossy compressed representation.
3. Implement Huffman coding for the quantized coefficients (or for run-length encoded zig-zag sequences) to accomplish lossless entropy coding.
5. Analyze the visual effects and quantitative trade-offs between different quantization strengths (i.e., varying quality levels) and lossless encoding choices.
6. Suggest suitable compression techniques based on results and application needs.

4. Expected Outcomes

- Working implementation of DCT-based lossy compression and Huffman-based lossless coding.
- Tables and plots showing compression ratio vs PSNR/SSIM for sample images.
- Visual examples (original vs compressed at different quality levels) to illustrate perceptual changes.
- A reasoned recommendation for when to use lossy-only, lossless-only, or combined approaches.

5. Theory / Background

5.1 Discrete Cosine Transform (DCT)

DCT is a transform that converts spatial domain image blocks into frequency domain coefficients. For 8×8 blocks (as in JPEG), the DCT concentrates most of the signal energy in a few low-frequency coefficients. The 2D DCT for an N×N block f(x,y) is:

$$DCT(i,j) = 1/\sqrt{2n}\ \mathrm{C(i)C(j)} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} Pixel(x,y) Cos[(2x+1)i\pi/2N]\ Cos[(2y+1)i\pi/2N]$$

$$C(x) = 1/\sqrt{2} \text{ if x is 0, else 1 if x>0}$$

5.2 Quantization (Lossy step)

Quantization reduces precision of DCT coefficients by dividing them by a quantization matrix and rounding to integers. Coarser quantization increases compression (more zeros) but reduces reconstructed image quality. Standard JPEG uses an 8×8 luminance quantization matrix scaled by a quality factor.

5.3 Zig-Zag Scan and Run-Length Encoding

A zig-zag scan orders coefficients from low to high frequency so that long runs of zeros (after quantization) are contiguous and can be efficiently run-length encoded (RLE). This is commonly used before entropy coding.

5.4 Huffman Coding (Lossless step)

Huffman coding is a variable-length, prefix-free coding algorithm that maps symbols (e.g., quantized coefficient values or RLE tokens) to bit sequences based on symbol frequencies. More frequent symbols get shorter codes.

5.5 Metrics

- **Compression Ratio (CR)** = (original size) / (compressed size)
- **Bits per pixel (bpp)** = (compressed bits) / (number of pixels)
- **PSNR** (Peak Signal-to-Noise Ratio): higher means closer to original.
- **SSIM** (Structural Similarity Index): perceptual similarity in [0,1].

6. Dataset

Pick any open-source image dataset. Examples: - Kodak PhotoCD / Kodak Lossless True Color Image Suite—24 high-quality images commonly used for image coding evaluation. - BSDS500 (Berkeley Segmentation Dataset)-good for natural images. - CIFAR-10 / CIFAR-100 — smaller images (32×32), useful for testing algorithm correctness quickly.

Recommendation: Use Kodak or BSDS500 for perceptual-quality experiments; CIFAR for quick functional tests.

7. Implementation Details

7.1 Preprocessing

- Convert color images to grayscale (or process each RGB channel independently).
- Pad image so that both dimensions are multiples of block size (e.g., 8).
- Cast to float and shift pixel range to [-128, 127] (JPEG style) before DCT.

7.2 Compression Pipeline (high-level)

1. Break image into non-overlapping 8×8 blocks.
2. For each block, compute 2D DCT.
3. Quantize DCT coefficients using a quantization matrix scaled by a quality factor Q.
4. Zig-zag scan the quantized 8×8 block into a 1D sequence.
5. Apply run-length encoding (RLE) on sequences of zeros (optional but recommended).
6. Build frequency table for RLE symbols and construct Huffman codes.
7. Encode symbols with the Huffman tree to produce compressed bitstream.

7.3 Decompression Pipeline

1. Decode Huffman bitstream to RLE or coefficient symbols.
2. Reconstruct the zig-zag sequence and inverse zig-zag to 8×8 blocks.
3. Multiply by quantization matrix (de-quantization) to get coefficient estimates.
4. Apply IDCT on each block and reconstruct the image (add 128 shift back).
5. Clip pixel values to [0,255] and convert to uint8.

8. Pseudocode

8.1 Compression (single-channel)

Input: image I (grayscale), block_size = 8, quality Q
Output: compressed_bitstream, huffman_table, image_shape

```
FUNCTION CompressImage(image, block_size=8, quality):
    padded_image = PadImage(image, block_size)
    shifted_image = padded_image - 128
    quant_matrix = ScaleQuantMatrix(quality)
    symbol_stream = []

    FOR each block in shifted_image (size = block_size x block_size):
        dct_block = Apply2DDCT(block)
        quant_block = Round(dct_block / quant_matrix)
        zigzag_seq = ZigZagScan(quant_block)
        rle_seq = RunLengthEncode(zigzag_seq)
        Append rle_seq to symbol_stream
    END FOR

    huffman_table = BuildHuffmanTable(symbol_stream)
    compressed_bits = HuffmanEncode(symbol_stream, huffman_table)

    RETURN compressed_bits, huffman_table, OriginalImageShape(image)
```

8.2 Decompression

Input: compressed_bits, codebook, image_shape, block_size=8, quality Q
Output: reconstructed_image

```
FUNCTION DecompressImage(compressed_bits, huffman_table, image_shape, block_size=8,
quality):
    quant_matrix = ScaleQuantMatrix(quality)
    symbol_stream = HuffmanDecode(compressed_bits, huffman_table)
    block_sequences = SplitIntoBlocks(symbol_stream)

    reconstructed_image = EmptyMatrix(image_shape)

    FOR each sequence in block_sequences:
        quant_block = InverseZigZag(sequence, block_size)
        dequant_block = quant_block * quant_matrix
        idct_block = Apply2DIDCT(dequant_block)
        Place idct_block into reconstructed_image
```

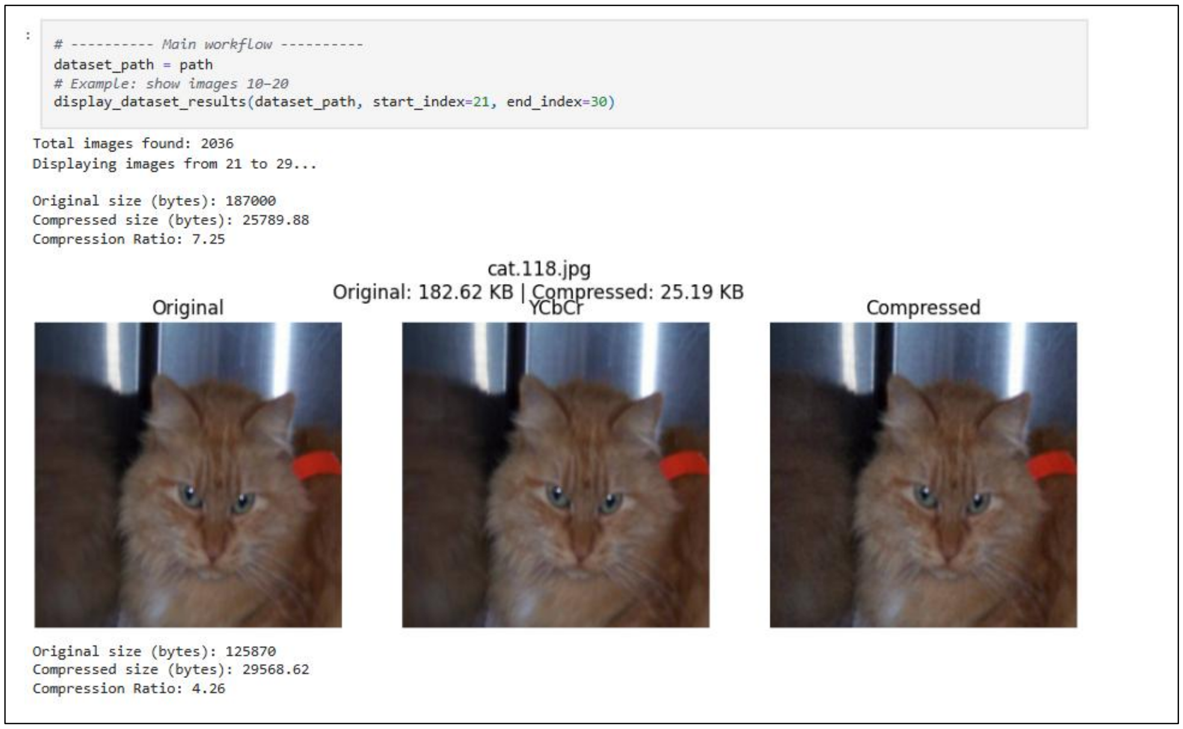
```
 END FOR

reconstructed_image = reconstructed_image + 128
Clip reconstructed_image to [0,255]

RETURN reconstructed_image
```

9. Experimental Steps

1. Choose several representative images from the dataset (e.g., 10 images covering smooth and textured regions).
2. Pick a range of quality factors Q (e.g., 10, 30, 50, 70, 90) to sweep quantization strength.
3. For each image and Q:
    a. Compress using the pipeline above.
    b. Record compressed size (in bits), compute CR and bpp.
    c. Reconstruct image and compute PSNR and SSIM versus original.
    d. Save side-by-side visual comparisons (original, compressed for each Q).
4. Optionally, test applying Huffman coding directly to pixel values (lossless) to compare with DCT+Huffman.

5. Plot: PSNR vs bpp, SSIM vs bpp, and Compression Ratio vs Q.

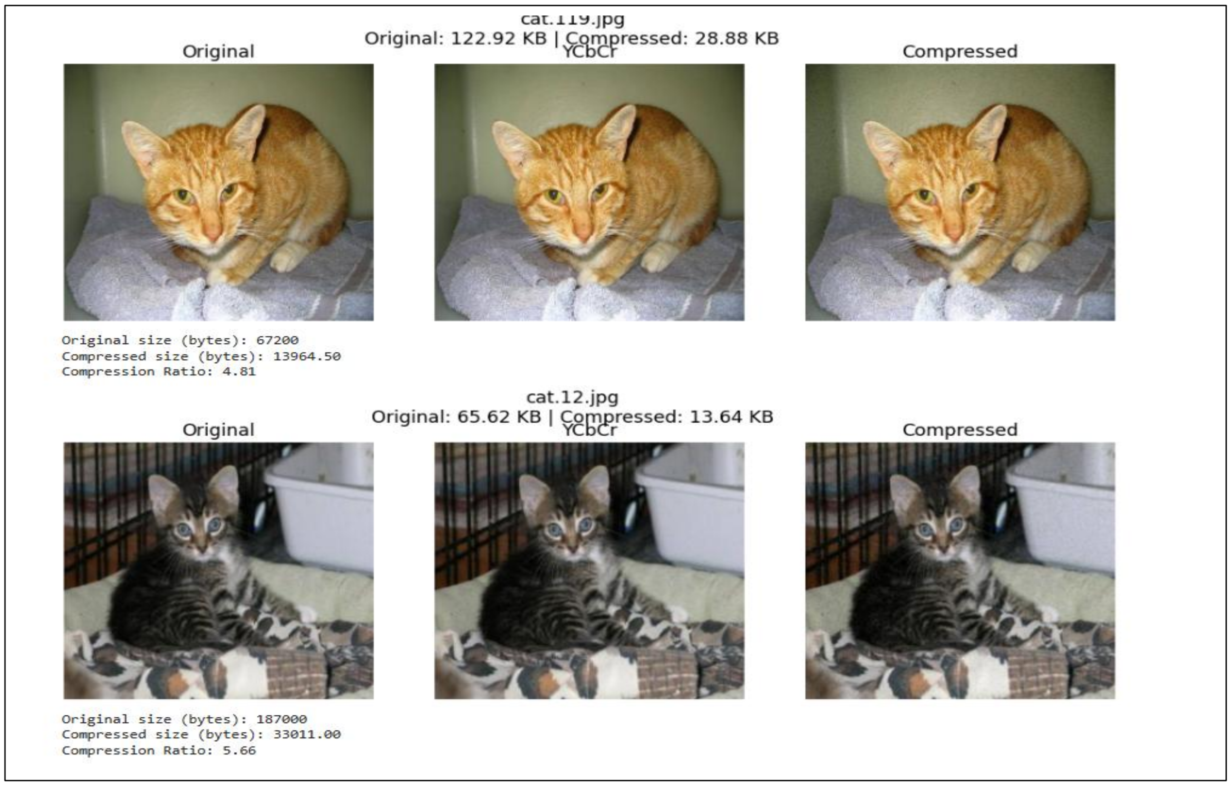
10. Result



```
# ---------- Main workflow ----------
dataset_path = path
# Example: show images 10-20
display_dataset_results(dataset_path, start_index=21, end_index=30)
```

```
Total images found: 2036
Displaying images from 21 to 29...

Original size (bytes): 187000
Compressed size (bytes): 25789.88
Compression Ratio: 7.25
```

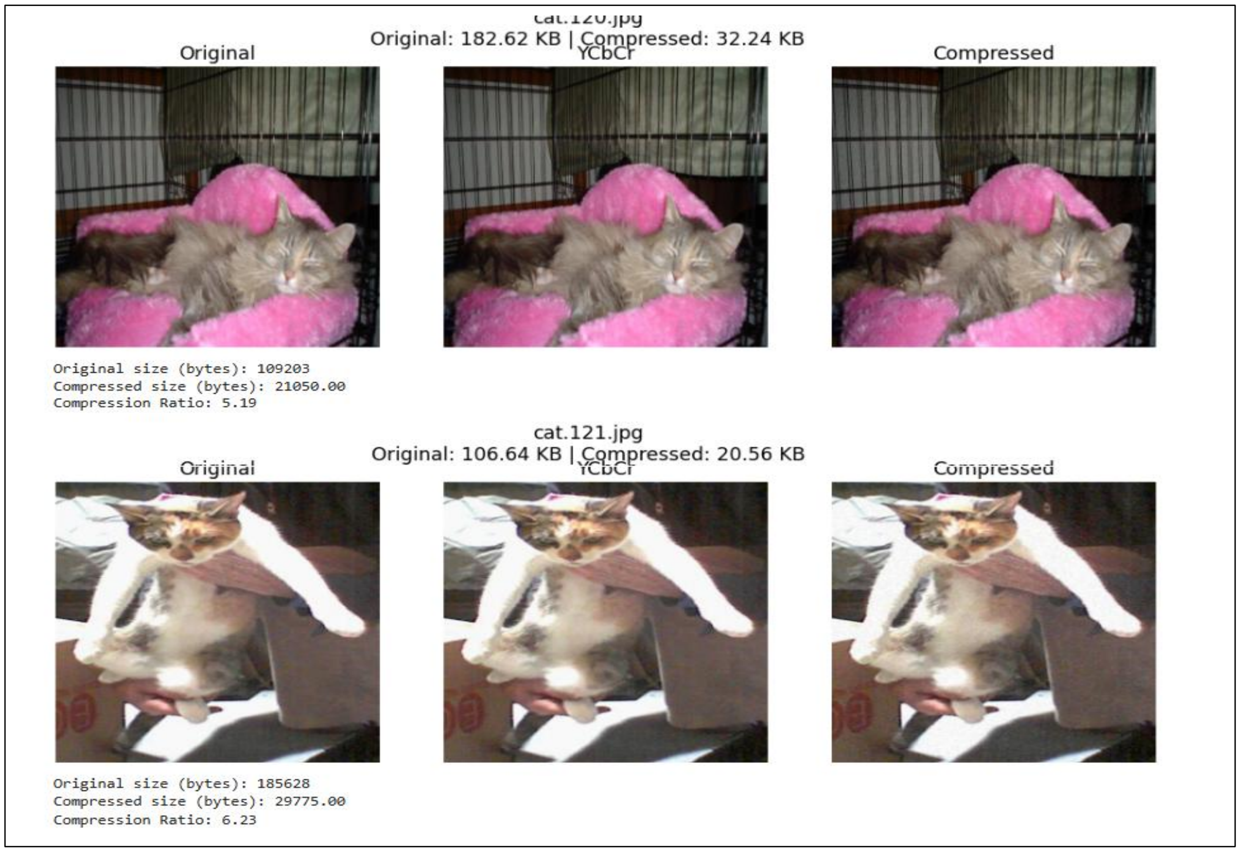
```
Original size (bytes): 125870
Compressed size (bytes): 29568.62
Compression Ratio: 4.26
```


cat.119.jpg
Original: 122.92 KB | Compressed: 28.88 KB
Original
YCbCr
Compressed

```
Original size (bytes): 67200
Compressed size (bytes): 13964.50
Compression Ratio: 4.81
```

cat.12.jpg
Original: 65.62 KB | Compressed: 13.64 KB
Original
YCbCr
Compressed

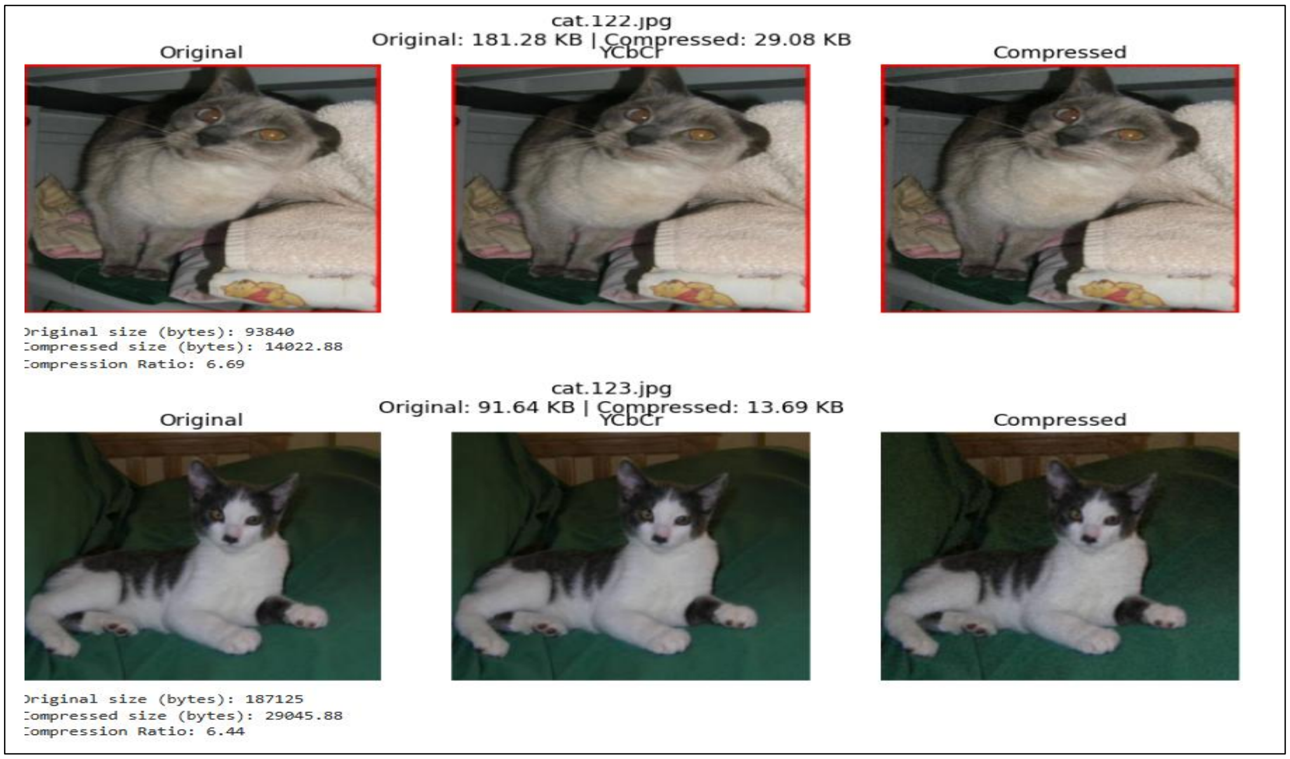
```
Original size (bytes): 187000
Compressed size (bytes): 33011.00
Compression Ratio: 5.66
```


cat.120.jpg
Original: 182.62 KB | Compressed: 32.24 KB
Original
YCbCr
Compressed

```
Original size (bytes): 109203
Compressed size (bytes): 21050.00
Compression Ratio: 5.19
```

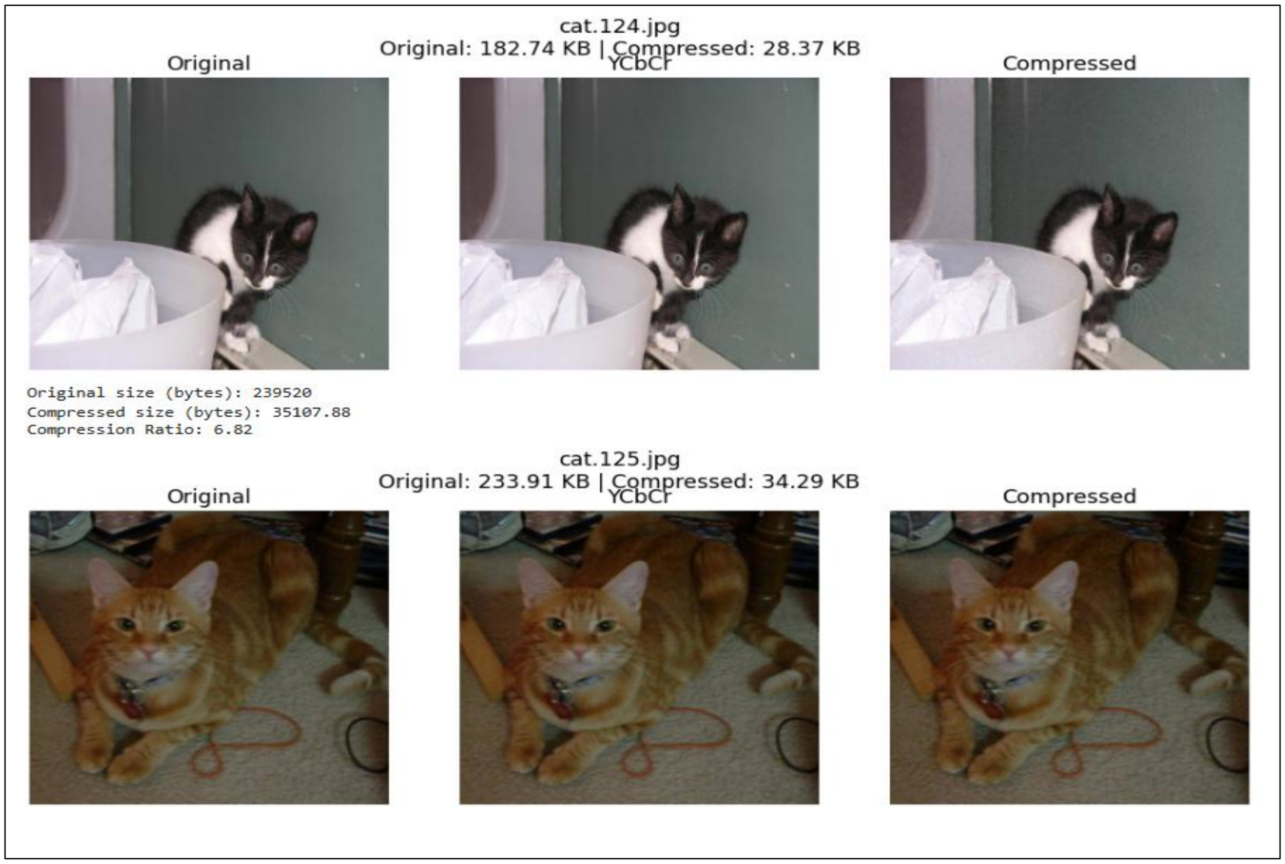
cat.121.jpg
Original: 106.64 KB | Compressed: 20.56 KB
Original
YCbCr
Compressed

```
Original size (bytes): 185628
Compressed size (bytes): 29775.00
Compression Ratio: 6.23
```


cat.122.jpg
Original: 181.28 KB | Compressed: 29.08 KB
Original
YCbCr
Compressed
cat.123.jpg
Original: 91.64 KB | Compressed: 13.69 KB
Original
YCbCr
Compressed

```
)riginal size (bytes): 93840
:ompressed size (bytes): 14022.88
:ompression Ratio: 6.69
```

```
)riginal size (bytes): 187125
:ompressed size (bytes): 29045.88
:ompression Ratio: 6.44
```


cat.124.jpg
Original: 182.74 KB | Compressed: 28.37 KB
Original
YCbCr
Compressed
cat.125.jpg
Original: 233.91 KB | Compressed: 34.29 KB
Original
YCbCr
Compressed

```
Original size (bytes): 239520
Compressed size (bytes): 35107.88
Compression Ratio: 6.82
```

11. Conclusion

The study demonstrates that DCT combined with Huffman coding achieves significant compression while maintaining acceptable image quality at medium quantization levels. Lossless methods guarantee fidelity but at the cost of lower compression efficiency. Therefore, lossy compression with controlled quantization is best suited for general applications, whereas lossless compression remains essential for domains requiring absolute accuracy.

1. For storage where fidelity is critical (medical, archival): Use lossless compression (PNG, lossless JPEG, or Huffman-based schemes) despite low compression ratios.
2. For general photographic images where storage/bandwidth is limited (web, social media): Use combined approach (DCT + quantization + entropy coding) with moderate quality ($Q \approx 50$–70) — it offers a good trade-off between size and perceived quality.
3. For highly textured imagery or where details matter (satellite, microscopy): Use higher quality (less quantization) or use wavelet-based codecs (e.g., JPEG2000) which reduce blocking artifacts.
4. If extreme compression is required at the expense of detail: Lower Q values yield small files but notable loss of high-frequency detail and blocking artifacts.

12. Github Link:

https://github.com/rachanadixit/FDIP/blob/main/123B5F138_FDIP_Assignment_7.ipynb