JPEG: Image Compression in Python

Irsh Vijay Aditya Raj
Dept. of Electronics & Electrical Comm. Engg.
EC69211: Image & Video Processing Laboratory

November 3, 2024

Abstract

JPEG is one of the most widely-used picture formats on the internet. We present the `ImageCompressor` class implemented in python to orchestrate complete JPEG compression pipeline using `compress` and `decompress` class functions. Quantitative analyses of compression over ranges of Quality Factor and Time Complexity are included in Results.

Keywords: JPEG, Lossy Compression, DCT, Run Length Encoding

1 Introduction

JPEG (Joint Photographic Experts Group) is a lossy image compression algorithm that results in significantly smaller file sizes with little to no perceptible impact on picture quality and resolution. A JPEG-compressed image can be ten times smaller than the original one. JPEG works by removing information that is not easily visible to the human eye while keeping the information that the human eye is good at perceiving. [1].

2 Methodology

First, let's take a look at an overview of the steps that JPEG takes when compressing an image.

2.1 Color Space Conversion

The human eye is more sensitive to brightness than color. Therefore, the algorithm can exploit this by making significant changes to the picture's color information without affecting the picture's perceived quality. To do this, JPEG must first change the picture's color space from RGB to YCbCr. YCbCr consists of 3 different channels: Y is the luminance channel containing brightness information, and Cb and Cr are chrominance blue and chrominance red channels containing color information.
No information lost in this step.

2.1.1 Transformation Matrices

1. `transform_matrix`: Converts RGB values to YCbCr values by applying a weighted sum based on how humans perceive brightness and color:

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}$$

2. `inv_transform_matrix`: Converts YCbCr back to RGB by inverting the transformatio:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.0 & 0.0 & 1.402 \\ 1.0 & -0.344136 & -0.714136 \\ 1.0 & 1.772 & 0.0 \end{bmatrix} \begin{bmatrix} Y \\ Cb - 128 \\ Cr - 128 \end{bmatrix}$$

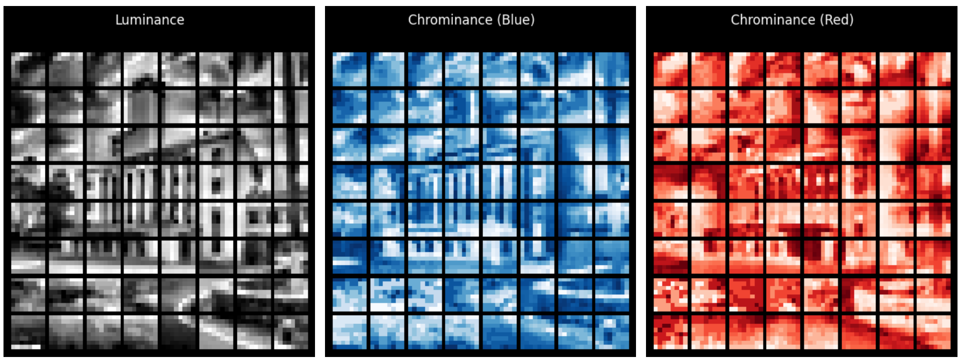
2.2 Downsampling

This step, chroma subsampling technique reduces the resolution of color information. Common downsampling formats include 4:4:4, 4:2:2, and 4:2:0:

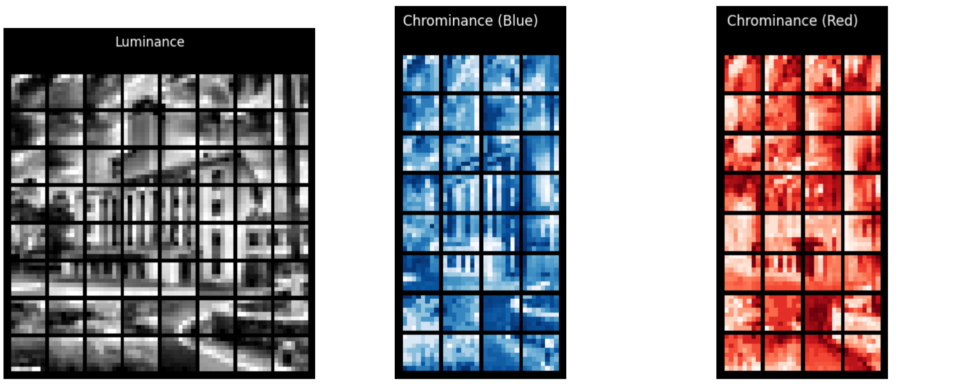
1. **4:4:4**: In this format, both luminance (Y) and chrominance (Cb and Cr) channels have the same full resolution. Each pixel has its unique color information, leading to the highest color fidelity but with a larger file size.

2. **4:2:2**: Here, the luminance channel retains full resolution, but chrominance channels are halved in the horizontal direction. For every two luminance samples, there is one sample for Cb and Cr, effectively reducing the color data by 50% while preserving horizontal detail.

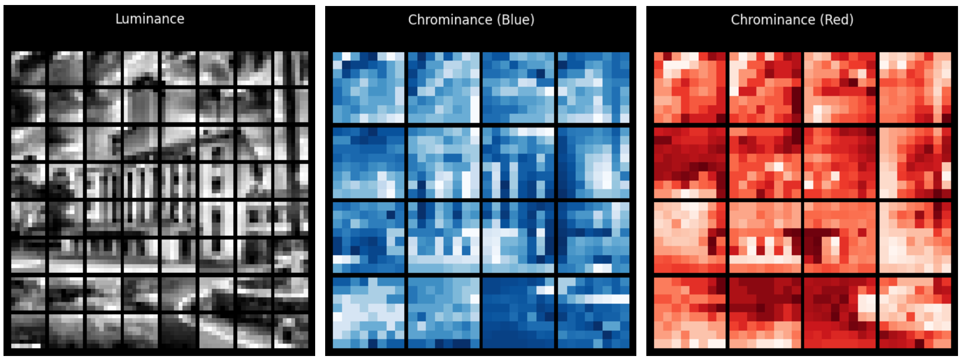
3. **4:2:0**: In 4:2:0 subsampling, both horizontal and vertical chrominance resolutions are halved. This format provides a 75% reduction in chroma data.



Luminance Chrominance(Blue) Chrominance(Red)
Example Image with '4:4:4' subsampling.



Luminance Chrominance(Blue) Chrominance(Red)
Example Image with '4:2:2' subsampling.



Luminance Chrominance(Blue) Chrominance(Red)
Example Image with '4:2:0' subsampling.

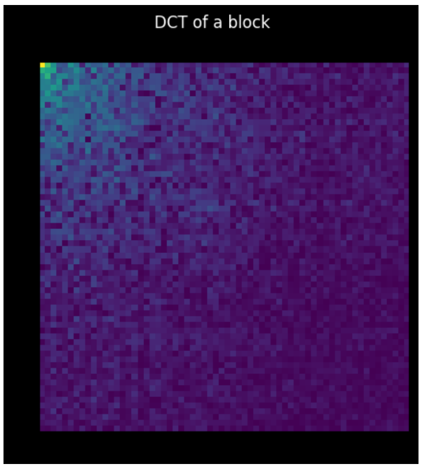
It's important to note that downsampling is only applied to the chrominance channels, and the luminance channel keeps its original size.

2.3 Division Into 8×8 Pixel Blocks

After downsampling, the pixel data of each channel is divided into 8×8 blocks of 64 pixels. From now on, the algorithm processes each block of pixels independently.

2.4 Forward DCT (Discrete Cosine Transform)

First, each pixel value for each channel is subtracted by 128 to make the value range from -128 to +127.
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.
This step is lossless.



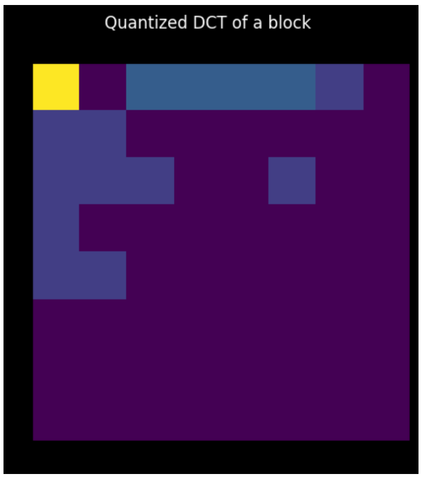
Discrete Cosine Transform (Log) of a block.

2.5 Quantization

The human eye is not very good at perceiving high-frequency elements in an image. In this step, JPEG removes some of this high-frequency information without affecting the perceived quality of the image.
To do this, the weight matrix is divided by a precalculated quantization table, and the results are rounded to the closest integer. Here is what a quantization table for chrominance channels looks like:

$$\text{Luminance Table} = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

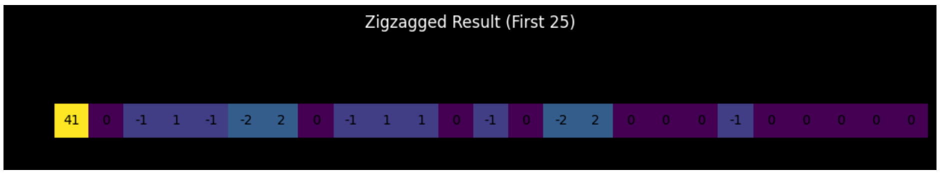
$$\text{Chrominance Table} = \begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix}$$



Quantized Discrete Cosine Transform (Log) of a 64x64 block.

2.6 Entropy Encoding

Now all that's left to do is to list the values that are inside each matrix, run RLE (Run Length Encoding) since we have a lot of zeroes, and then run the Huffman Coding algorithm before storing the data.
RLE and Huffman Coding are lossless compression algorithms that reduce the space needed to store information without removing any data.



Zigzag List (First 25).

As we can see, the result of quantization has many 0s toward the bottom left corner. To keep the 0 redundancy, the JPEG algorithm does a zig-zag scanning pattern that keeps all zeroes together.

3 JPEG Decoding

Whenever we open a .jpg file using an image viewer, the file needs to be decoded before it can be displayed. steps in reverse order:

1. Run Huffman Decoding on the file data.
2. Deconstruct RLE (Run-Length Encoding).

3. Put the list of numbers into an 8×8 matrix.
4. Multiply the integer values by their corresponding quantization table.
5. Multiply the weight values by their corresponding base images, then sum them up to get the pixel values.
6. Upscale the chrominance channels.
7. Change the color space from YCbCr to RGB.
8. Display the image.

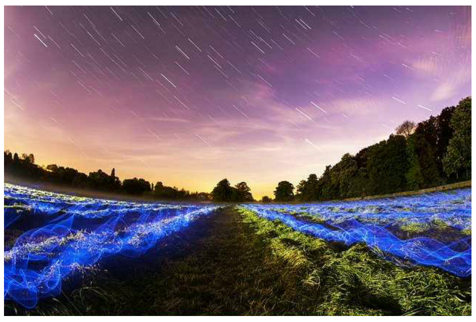
4 Experimental Setup

This JPEG compression pipeline is composed of several Python classes. First, `ColorConverter` transforms the image from RGB to YCbCr color space to separate luminance and chrominance information. Next, `Subsampler` applies chroma subsampling to reduce color data resolution, exploiting human sensitivity to brightness over color. `BlockProcessor` divides the image into 8x8 blocks for frequency transformation. `Quantizer` compresses the frequency data by reducing precision of high-frequency components. `ZigZag` reorders the quantized data to prioritize low-frequency components. The `EntropyEncoder` and `huff` modules then apply Huffman coding to compactly store these values. Utility functions `save_to_file` and `load_from_file` manage file storage, allowing compressed data to be saved and retrieved as needed. Bitmap support is included using the `BMP` class.

5 Results

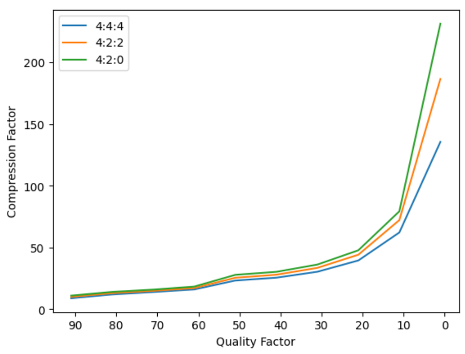
A few key findings of our experiments.

(a) Original Image

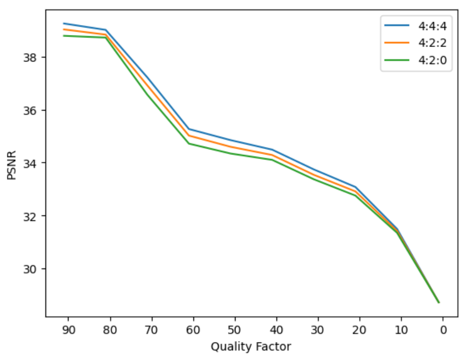


(b) Our Compressed Image

Figure 1: Our Compression Results



(a) Compression vs. Quality Factor



(b) PSNR vs. Quality Factor

Figure 2: Analysis by varying Quality Factor.

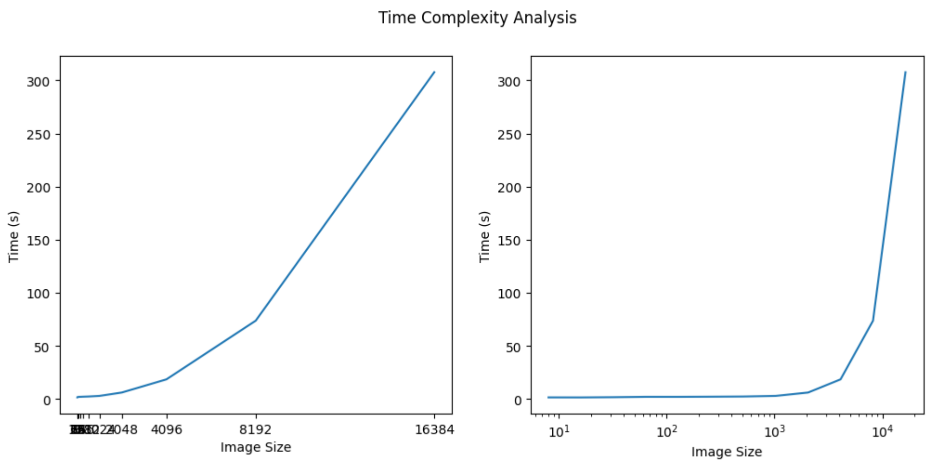


Figure 3: Time Complexity Analysis

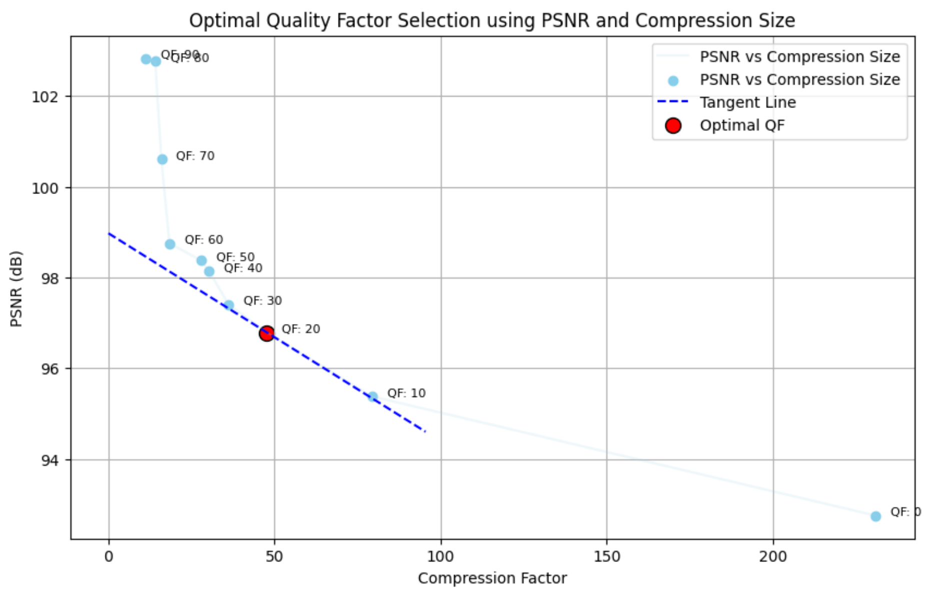


Figure 4: Optimal Quality Factor

6 Discussion

- We observed that as the quality factor, which controls the degree of quantization, decreases, the compression factor increases at an exponential rate. The compression factor is defined as:

$$\text{compression\_factor} = \frac{\text{byte\_size(original)}}{\text{byte\_size(compressed)}}$$

  This inverse relationship allows higher compression ratios at lower quality factors.

- Similarly, the Peak Signal-to-Noise Ratio (PSNR) improves with an increase in quality factor. The PSNR is given by:

$$\text{PSNR} = 10\log_{10}\left(\frac{255^2}{\frac{1}{M\times N}\sum_{i=1}^{M}\sum_{j=1}^{N}[\text{compressed}(i,j) - \text{original}(i,j)]^2}\right)$$

  This metric reflects the balance between compression and image fidelity, with higher PSNR values indicating less loss in image quality.

- When analyzing the time complexity of the compression process with increasing image size, it appears to grow between $O(n\log(n))$ and

$O(n^2)$. This observation suggests that the algorithm's complexity scales significantly with larger images, likely due to computational demands in stages such as Discrete Cosine Transform (DCT) and quantization.

- We defined an optimal quality factor by examining the slope of the PSNR versus compression factor curve at the points corresponding to maximum and minimum quality factors. A tangent line with this slope is plotted to identify the point of optimal balance between compression efficiency and image quality inspired by the elbow method while choosing optimal 'k' in k-means clustering.

References

[1] G. Wallace, "The jpeg still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992.