Implementing the JPEG Compression Algorithm

Michael Mente[01634435], Lukas Pezzei[11834075], and Julian Saria[01608505]

University of Vienna, Universitätsring 1, 1010 Vienna, Austria
https://www.univie.ac.at/

Abstract. In this paper we implement the JPEG compression algorithm step-by-step from scratch to convey the mathematical basis in a comprehensible way. We focus on the progressive processing of the image, showing the intermediate results using an example image to illustrate the comprehensibility of the algorithm. Subsequently, we explore the possibilities of different parameters and their influence on the image quality. In addition, we use quantitative measurement techniques to objectively assess the image quality.

Keywords: Image Compression · JPEG · DCT.

1 Introduction

Over the years the quality and the resolution of pictures drastically increased. To store these pictures effectively a good compression algorithm is required. Compression can be lossy or lossless. Lossless means that the exact same picture can be reconstructed, whereas for lossy that information in the picture is lost and can not be restored. When losing information certain aspects, like higher frequencies, are removed that are not perceivable to humans. In the end the pictures do differ in numbers but not for the human eye. There are plenty of lossless compression algorithms such as PNG (Portable Network Graphics), TIFF (Tagged Image File Format) and GIF (Graphics Interchange Format). On the other hand the best known lossy compression algorithm is JPEG (Joint Photographic Experts Group), but there are also others like HEIF (High Efficiency Image Format). All these file formats are specialised for different applications. For example PNG is used to display detailed web-graphics with high contrasts while JPEG is just an universal format that is supported on most operating systems that cannot display details that well due to the lossy compression, however it has the advantage of a reduced file size. In this paper the conversion from a PNG image, in this case the Lenna image (Figure 1), to JPEG is described in detail and the performance is analyzed in the end.

2 Implementation

In the following section we will go through the steps of the JPEG compression algorithm in the order they are executed. To convey the inner workings of the algorithm in an understandable way, the well-known Lenna image (Figure 1), is used to show the outcome of every step of the algorithm exemplary.

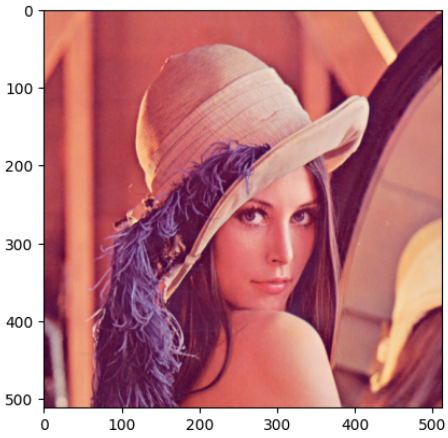


Fig. 1. Original Standard Test Image of Lenna [1]

2.1 Colorspace Transformation

In the first implementation step the color components of an image need to be transferred. Usually an image is visually represented by its RGB (Red, Green and Blue) values. This is due to the nature of the human eyes since they perceive these colors best. In the RGB representation, each pixel consists of a triplet of integers, in the range from 0 to 255, where each value corresponds to one color. However there is more than just one possibility to represent an image. In order to do some compression later on we need to first describe the image in a different color space.
As contrast information has a higher impact on our visual perception than color information, a fitting approach is to transcode the image to the YUV color space, where Y is the luminance and U and V are responsible for the chrominance. This color-space conversion can be archived with the following formulas.

$$Y = 0.299 * R + 0.587 * G + 0.114 * B \tag{1}$$
$$U = 0.493 + (B - Y) \tag{2}$$
$$V = 0.877 + (R - Y) \tag{3}$$

Note that at first U and V are the divergence from Y to Blue and Red and in order to obtain the desired chrominance, we need to add 128 to both values. This color space can now also be referred to as Y,Cb,Cr. Where the C is an indicator for the chrominance. This step will come in handy later on. The visual representation of the color space transformation is shown in figure 2.

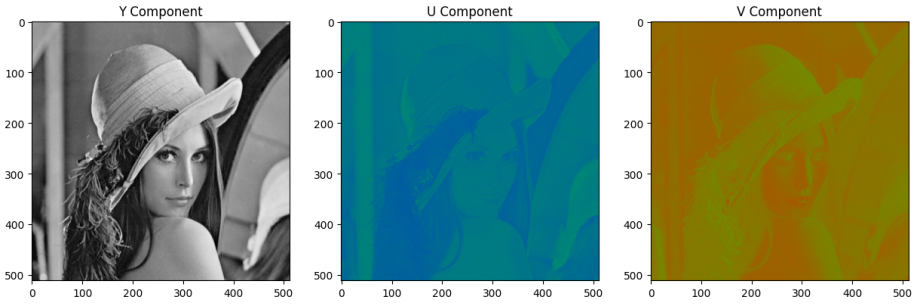


Fig. 2. Original Image decomposed in Y, U, V components

2.2 Sub-sampling of Color Components

The next step is to sub-sample each of the color components individually. As previously stated the color components are now separated in luminance and chrominance. This substitution of color space opens up following concept. If the contrast information is more relevant than the color information, then we could smooth out the color values between some of the pixel while keeping the original luminance.

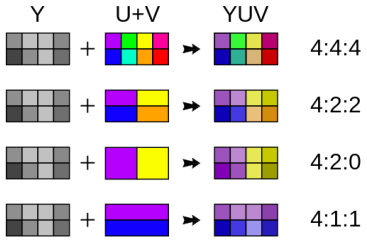


Fig. 3. Ratios of sub-sampling [3]

Sub-sampling is a process of loosing mostly very little to non visible data through adjusting values to their neighbors. The effectiveness of the compression is based on the fact that the resolution of certain data can now be decreased significantly. There are a few ratios to choose from for the desired sub-sampling. All of them below follow the same basic principle they remain Y untouched and only change the U and V component. The first parameter of the ratio is most commonly 4. It represents the amount of pixels taken as the width of the block to be sub-sampled. Each block consists of two rows resulting in a block size of 4x2 Pixels. The following parameter takes into account how much values will be considered for the first row and the last does the exact same but for the second row as shown in figure 3. As an example the ratio of 4:2:0 that we used, consists

of an 4x2 block where the first and the third pixel will determine the color for the second and fourth. And in the second line, since the last parameter is zero, the previously taken pixel samples of the upper line are responsible for color values of the lower. Sub-sampling effectively reduces the resolution of the U and V component which is an irreversible lossy process.
In the figure 4 the effect of sub-sampling the Y, U, V components with the ratio of 4:2:0 can be seen. Only minimal differences are visible. This also shows as the resolution of the U and V component is reduced to a quarter of Y component.

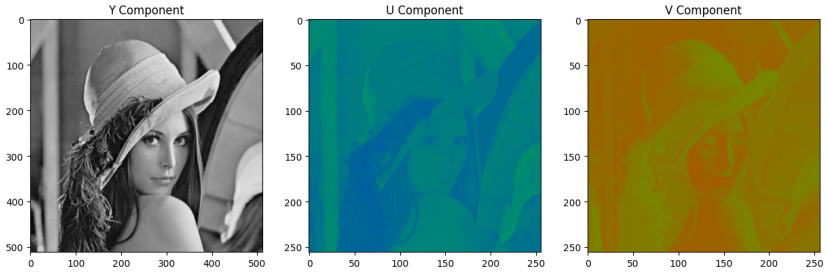


Fig. 4. Sub-sampled Y, U and V components of the Image

2.3 Splitting into 8x8 Blocks

In order to archive better performance for the following actions and to match the resolution of the quantization matrix, the image needs to be split in smaller blocks of size is 8x8. If the resolution of the original image does not perfectly divide by the factor of 8, then there is padding applied, which can also be seen in figure 5 where the first lines are repeated. At the end of this operation every component of the image is flattened in a list of eight by eight blocks.

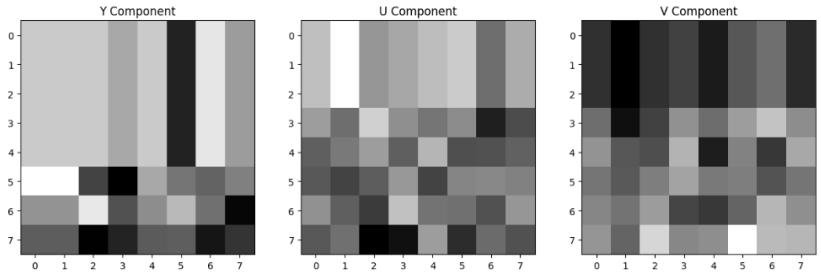


Fig. 5. Sub-sampled 8x8 Blocks of the Y,U and V Components

2.4 Discrete Cosine Transform

In this step the Discrete Cosine Transform (DCT) is applied to every single 8x8 block. The DCT is a transformation like the Discrete Fourier Transform (DFT), with the difference that it only has real values. Therefore it transforms the image from the spatial domain to the frequency domain. We do this since a lot of information in the higher frequencies is not needed, due to the fact that humans do not perceive them as well. In the next step these higher frequency components will be removed. The DCT is defined as

$$D(i,j) = \frac{1}{\sqrt{2N}} C(i)C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} p(x,y) cos\Big[\frac{(2x+1)i\pi}{2N}\Big] cos\Big[\frac{(2y+1)j\pi}{2N}\Big] \quad (4)$$

$$\mathrm{C(u)} = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } u = 0 \\ 1, & \text{if } u > 0 \end{cases}$$

To make this more clear, as an example a 8x8 Block is here shown where the DCT is applied to. Before applying we have to remind that the DCT is a cosine transform which is symmetric around the origin, so it is designed to work on the values in the range -128 to 127, but we have values from 0 to 256. So we simply subtract 128 from all the pixel values in the first step and then apply the DCT. The example matrix A_{image} is the grayscaled first block of the Lenna picture.

$$A_{image} = \begin{pmatrix} 158 & 156 & 165 & 147 & 98 & 106 & 110 & 124 \\ 157 & 159 & 164 & 144 & 95 & 105 & 108 & 122 \\ 161 & 163 & 161 & 142 & 94 & 104 & 108 & 121 \\ 165 & 139 & 161 & 145 & 90 & 101 & 108 & 121 \\ 152 & 106 & 164 & 142 & 85 & 100 & 105 & 116 \\ 109 & 105 & 164 & 141 & 87 & 100 & 105 & 117 \\ 91 & 105 & 162 & 142 & 86 & 99 & 104 & 116 \\ 91 & 103 & 161 & 142 & 83 & 98 & 104 & 122 \end{pmatrix}$$

After subtracting 128 from every entry we get

$$A_{image-128} = \begin{pmatrix} 30 & 28 & 37 & 19 & -30 & -22 & -18 & -4 \\ 29 & 31 & 36 & 16 & -33 & -23 & -20 & -6 \\ 33 & 35 & 33 & 14 & -34 & -24 & -20 & -7 \\ 37 & 11 & 33 & 17 & -38 & -27 & -20 & -7 \\ 24 & -22 & 36 & 14 & -43 & -28 & -23 & -12 \\ -19 & -23 & 36 & 13 & -41 & -28 & -23 & -11 \\ -37 & -23 & 34 & 14 & -42 & -29 & -24 & -12 \\ -37 & -25 & 33 & 14 & -45 & -30 & -24 & -6 \end{pmatrix}$$

To easier calculate the DCT, we can use the its basis vectors and create a basis transformation matrix T, which we then can apply to the blocks of the color

components of the original image. T is then defined with the help of equation 4

$$T(i,j) = \begin{cases} \frac{1}{\sqrt{N}}, & \text{if } i = 0 \\ \sqrt{\frac{2}{N}} cos\left[\frac{(2j+1)i\pi}{2N}\right], & \text{if } i > 0 \end{cases} \tag{5}$$

Therefore T is

$$T = \begin{pmatrix} 0.354 & 0.354 & 0.354 & 0.354 & 0.354 & 0.354 & 0.354 & 0.354 \\ 0.49 & 0.416 & 0.278 & 0.098 & -0.098 & -0.278 & -0.416 & -0.49 \\ 0.462 & 0.191 & -0.191 & -0.462 & -0.462 & -0.191 & 0.191 & 0.462 \\ 0.416 & -0.098 & -0.49 & -0.278 & 0.278 & 0.49 & 0.098 & -0.416 \\ 0.354 & -0.354 & -0.354 & 0.354 & 0.354 & -0.354 & -0.354 & 0.354 \\ 0.278 & -0.49 & 0.098 & 0.416 & -0.416 & -0.098 & 0.49 & -0.278 \\ 0.191 & -0.462 & 0.462 & -0.191 & -0.191 & 0.462 & -0.462 & 0.191 \\ 0.098 & -0.278 & 0.416 & -0.49 & 0.49 & -0.416 & 0.278 & -0.098 \end{pmatrix}$$

T is an orthonormal matrix since it is a basis transformation matrix, which was constructed by the basis vectors of the DCT. So the inverse of T is the same as the transpose of T. To get the DCT of the image we need to do a simple basis transformation that is defined as

$$A_{DCT} = T * A_{image} * T^{-1} \tag{6}$$

By applying the DCT as shown in equation 6, the values in the resulting 8x8 block are

$$A_{DCT} = \begin{pmatrix} 22.716 & 29.113 & 26.342 & 11.589 & -44.815 & -28.815 & -9.226 & 4.734 \\ 20.53 & 30.835 & 25.829 & 7.534 & -48.093 & -29.547 & -10.505 & 3.943 \\ 22.656 & 35.618 & 23.6 & 3.84 & -48.818 & -30.466 & -10.235 & 3.203 \\ 31.889 & 16.015 & 14.804 & 11.468 & -52.689 & -31.663 & -8.349 & 4.183 \\ 28.108 & -18.873 & 8.607 & 16.374 & -58.148 & -29.66 & -10.229 & 2.02 \\ -11.483 & -33.224 & 15.697 & 17.038 & -53.131 & -27.986 & -8.906 & 3.49 \\ -28.961 & -38.285 & 16.25 & 17.834 & -52.622 & -28.813 & 8.845 & 3.296 \\ -28.803 & -39.434 & 14.458 & 18.613 & -54.557 & -27.923 & -6.082 & 9.098 \end{pmatrix}$$

Now we have 64 DCT coefficients, where the ones in the upper left corner are from the lower frequencies and the ones in the lower right are the higher frequencies. The top left DCT coefficient is also called DC component which is the average value of the whole block, the other components are called AC components. As already stated the human eye is most sensitive to lower frequencies and that is why some of the higher frequency components get removed in the quantization process subsequently.

2.5 Quantization

Quantization is a lossy step in JPEG, where a range of values is mapped to a single quantum (discrete) value. Depending on the quality of the picture different quantization tables can be used. For our example we use the quality of 50%, where the quantization table looks as follows

$$Q_{50} = \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix}$$

This table is the JPEG standard and from this table every other quantization quality can be calculated like shown in listing 1.

```
1  def get_quantization_matrix_for_quality_percent(quality_percentage: int):
2      assert 0 <= quality_percentage <= 100
3      if quality_percentage == 0:
4          return np.zeros((8, 8)) + 255
5      elif quality_percentage > 50:
6         q = np.rint(quantization_50*((100-quality_percentage)/50))
7         return np.clip(q, a_min=1, a_max=255)
8      elif quality_percentage < 50:
9          q = np.rint(quantization_50*(50/quality_percentage))
10         return np.clip(q, a_min=1, a_max=255)
11     elif quality_percentage == 50:
12         return quantization_50
```

Listing 1: Implementation of quantization table calculation

If the quality percentage is above 50% the formula from the first elif statement is used. If it is below that threshold then we use it from the second elif statement. When looking at the Q_{50} matrix we can see that the lower right has higher values and therefore the values from A in the lower right corner will be almost 0, because Quantization is defined for every value i,j in the matrix as followed

$$A_{quant}(i,j) = round\Big(\frac{A_{DCT}(i,j)}{Q_{50}(i,j)}\Big) \tag{7}$$

This means for example for the first entry i = 1, j = 1

$$A_{quant}(1,1) = round\Big(\frac{22.716}{16}\Big) = round(1,41975) = 1$$

So our resulting matrix A_{quant} is

$$A_{quant} = \begin{pmatrix} 1 & 3 & 3 & 1 & -2 & -1 & 0 & 0 \\ 2 & 3 & 2 & 0 & -2 & -1 & 0 & 0 \\ 2 & 3 & 1 & 0 & -1 & -1 & 0 & 0 \\ 2 & 1 & 1 & 0 & -1 & 0 & 0 & 0 \\ 2 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

As there are a lot of entries containing the value 0, for high frequency components, therefore not that much perceivable information is lost. In addition they are located in the bottom right of the matrix so we can exploit this in the next step.

2.6 Reordering of values

The goal of this step is to reorder the matrix so that similar values can be grouped together, since this will achieve better results during the run length encoding step. As all zeros are located in the bottom right, we make use of Zig-Zag reordering which traverses the matrix as shown in figure 6.

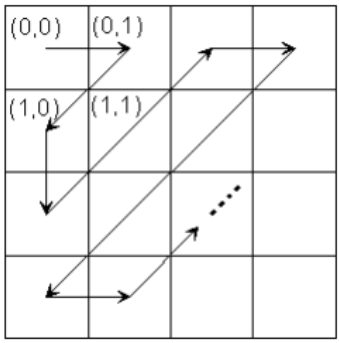


Fig. 6. Zig-Zag traversal [2]

For the implementation we initially require that the values of a 8x8 matrix are ordered in row major order.

$$M_{row-major} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \\ 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \\ 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 \\ 48 & 49 & 50 & 51 & 52 & 53 & 54 & 55 \\ 56 & 57 & 58 & 59 & 60 & 61 & 62 & 63 \end{pmatrix}$$

We traverse the matrix $M_{row-major}$ in the order shown in figure 6 to obtain the following list
$resort\_indices$ = [0, 1, 8, 16, 9, 2, 3, 10, 17, 24, 32, 25, 18, 11, 4, 5, 12, 19, 26, 33, 40, 48, 41, 34, 27, 20, 13, 6, 7, 14, 21, 28, 35, 42, 49, 56, 57, 50, 43, 36, 29, 22, 15, 23, 30, 37, 44, 51, 58, 59, 52, 45, 38, 31, 39, 46, 53, 60, 61, 54, 47, 55, 62, 63]
To accomplish a Zig-Zag ordering we access the blocks in the order as shown in $resort\_indices$.

```
def resort_values_zig_zag(block, block_size):
    global resort_indices
    #get indices for reqired block size
    if block_size not in resort_indices:
        resort_indices[block_size] = get_zig_zag_indices(block_size)
    indices = resort_indices[block_size]
    new_block = []
    #flatten the matrix to a 1-dimensional list that is traversed like M
    block = np.reshape(block, (1, block_size ** 2))[0]
    #create a list in zig-zag order
    return [block[index] for index in indices ]
```

Listing 2: Zig-Zag ordering implementation

If we resort our values from A_{quant} according to $resort\_indices$ using the function shown in listing 2 we compute the list $A_{reordered}$.
$A_{reordered}$ = [1, 3, 2, 2, 3, 3, 1, 2, 3, 2, 2, 1, 1, 0, -2, -1, -2, 0, 1, -1, -0, -1, -1, 0, 0, -1, -1, 0, 0, 0, -1, -1, 0, 0, -1, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Here we can observe that there are many zeros in the end which benefits the next step.

2.7 Run Length Encoding

Run length encoding performs best if the same data occurs multiple times after each other. It decodes the symbol and the how often the symbol is going to be repeated. That means the more often a symbol is repeated in a message the better the algorithm performs. In our case we have many zeros and therefore run length encoding will compress our data. From $A_{reordered}$ we obtain a list of tuples which consists of a symbol and the occurrence of this symbol that would look like this
$A_{runlength}$ = [(1, 1), (3, 1), (2, 2), (3, 2), (1, 1), (2, 1), (3, 1), (2, 2), (1, 2), (0, 1), (-2, 1), (-1, 1), (-2, 1), (0, 1), (1, 1), (-1, 1), (0, 1), (-1, 2), (0, 2), (-1, 2), (0, 3), (-1, 2), (0, 2), (-1, 1), (0, 4), (-1, 1), (0, 6), (-1, 1), (0, 3), (-1, 1), (0, 13)]
(x, y) ... (symbol, occurrence)

2.8 Huffman Encoding

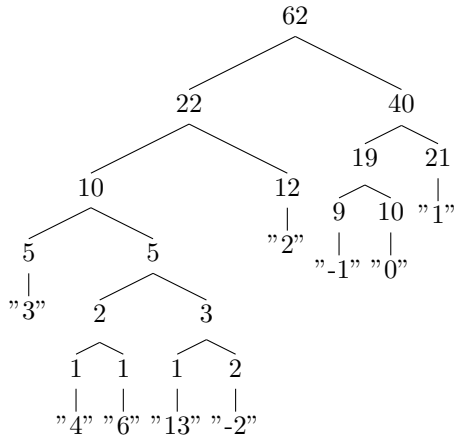
This is the most popular method for lossless compression which produces optimal prefix code to reduce the data volume even further. The idea is to generate a tree where the symbols that are very frequent are going to have the shortest paths and the rarely occurring symbols have the longest path. The path describes the amount of bits needed to encode a symbol. The first step is to count the probability of each symbol. We do not differentiate between the symbol "1" and the occurrence "1", as they are stored as the same bit, so therefore we iterate over all tuples and create a dictionary with the symbols and save how often they occur in the code. This results in the following symbol-occurrence table (Table 1).

symbol	**occurrence**
1	21
2	12
0	10
-1	9
3	5
-2	2
4	1
6	1
13	1

Table 1. Occurrence of symbols

Using this table we create a tree, where we merge in every step the two lowest occurrences to a new node. If the values are the same and there are more possibilities for the lowest occurrence value we simply take the first one in the list. That means that the Huffman tree is not unique. There are many correct

Huffman trees to a problem, it really does not matter which one is created, as long it is created correctly. For our table the following Huffman tree was created.



Here we can see that the symbol "6" could easily swap places with the symbol "13" and we still would produce a correct Huffman tree, but with different bit codes for "6" and "13". In the end this does not matter because the symbols "6" and "13" have the same occurrence. They are equally important as swapping their bit codes does not have any effect, as the length of the bit codes is the same. In the last step we create a code-word for every single symbol by traversing the tree. Going left means adding "1", going right means adding "0", so we end up with a Huffman lookup table (Table 2).

symbol	**bit code**
1	00
2	10
0	010
-1	011
3	111
-2	11000
4	11011
6	11010
13	11001

Table 2. Bit code lookup table

After exchanging every symbol from $A_{runlength}$ with the bit code from the table we generate a long bit string that looks like this

$A_{hufmann}$ = 000011100101011110000010001110010100010010001100000011001100000010000000011000100001110010100111001011101110010100110001011011011000101101001100010111011000101100101

2.9 Serialization

Finally, to test the effectiveness of our compression, it was necessary to pack all the data into one file. After all, the actual effectiveness of the compression can only be measured if we also take the data that is required to reverse the compression into account. The following data must be stored as shown in table 3.

Data	**Length (Bytes)**
Bits prepended to y	8
Bits prepended to u	8
Bits prepended to v	8
Sub-sampling Setting J	8
Sub-sampling Setting A	8
Sub-sampling Setting B	8
Resolution Width	16
Resolution Height	16
Huffman Table y	variable
Huffman Table u	variable
Huffman Table v	variable
Encoded Data y	variable
Encoded Data u	variable
Encoded Data v	variable
Quantization Table	64

Table 3. Data Stored in Compressed Image File

We have to store the Huffman coded data sets of each color component and the corresponding Huffman tables. Furthermore we need to store the quantization table, as well as the sub-sampling settings. The last required information is the resolution of the original image.

To actually store the Huffman encoded data, it needs a conversion from a bit string to bytes, in case the length of the bit string is not divisible by 8, null bytes are prepended to the byte string as shown in the implementation in listing 3.

The next step is to encode the Huffman tables to bytes. Here we used a simple encoding method. First, we transformed the data type "Dictionary" into a string, and then we stripped superfluous information. Since the Huffman table contains only numbers, it is possible to additionally use letters for encoding. We took advantage of this fact by replacing consecutive sequences with letters, on the one hand we have sequences of consecutive repeating numbers (For example: five consecutive "0" are replaced with the letter "e", if 5 digits with the value "1" follow each other they are replaced with the capital letter "E"). In addition, we also considered alternating sequences and replaced them with letters, again

```python
def bitstring_to_bytes(bits: str):
    prepended = 0
    if len(bits) % 8 != 0:
        prepended = 8 - (len(bits) % 8)
    data = int(bits, 2).to_bytes((len(bits) + 7) // 8, byteorder='big')
    return data, prepended
```

Listing 3: Conversion of Bit-String to Bytes

depending on the order - i.e. whether the alternating sequence starts with 1 or 0 - we replaced it with an uppercase or lowercase letter. By this simple method the size of the Huffman tables could be reduced by about half, for the shown example image the cumulative size of all 3 Huffman-tables could be reduced from 2333 bytes to 1028 bytes.

Finally, the encoding of the quantization table was still a challenge. Experiments with Huffman encoding under prior reordering and run length encoding (such as the individual 8x8 blocks), have shown that this mechanism is only suitable for extremely low-quality compression tables, which are rarely used. This is because the quantization table contains only 8-bit values, and also has just 64 elements. Therefore, the quantization table is stored directly without any encoding mechanisms since it takes up only 64 bytes anyway. The sub-sampling information can be stored in 24 bytes.

However, to store the resolution we have a maximum allowable image resolution of $2^{16} * 2^{16}$, as the height and the width are each stored in a 16-bit field.

To separate this data, we had two possibilities. Either we define a fixed header in which the positions in the file are specified, i.e., at which byte this data starts (and ends), or we use a separator that is unique and can be used to separate the different data within the file. We decided to use a unique separator for the following reasons: On the one hand, the use of a separator does not limit the maximum file size (because when storing addresses, it is necessary to specify a certain length by choosing the data type) and on the other hand, this method requires less memory. The separator we use is inserted between each field. Surprisingly, the separator we use can be only 3 bytes long with incrementing values from the null byte (Listing 4).

```python
separator = b"\x00\x01\x02"
```

Listing 4: Separator used to determine sections in Binary File

Unlike the actual JPEG standard used, our file format cannot contain additional metadata, our implementation is focused on evaluating only the compression mechanisms used in the JPEG standard and therefore only the minimal set of data necessary for decompression is stored.

2.10 Deserialization

In the next step, the deserialization, in principle only the steps of the serialization are undone. Since the order of the fields is fixed in our file format, it is sufficient to divide the binary data based on the separator (Listing 4). In this step the biggest challenge is to reverse the encoding to bytes. First the encoding of the Huffman tables has to be reversed, then it is necessary to generate a bit string from the image data of the single color channels. Here it is important to consider that during the serialization bits have been added in front which now have to be cut off as shown in listing 5.

```
1 def bytes_to_bitstring(bytestring, prepended_bits=0):
2     return ''.join(f'{byte:08b}' for byte in bytestring)[prepended_bits:]
```

Listing 5: Conversion of binary data to bit string

2.11 Decode Huffman

When encoding with Huffman we created a long bit string $A_{huffman}$ with a lookup table (Table 4). Now we use both this information to recreate the symbols of the bit string. So we have

symbol	bit code
1	00
2	10
0	010
-1	011
3	111
-2	11000
4	11011
6	11010
13	11001

Table 4. Colored bit code lookup table

$A_{huffman}$ = 0000111001010111100000100011100101000100100011000000
1100110000001000000001100010000111001010011100101110111001010011000
0101101101100010110100110001011101100010110001
Now we just do a longest prefix match, here shown for the first few symbols
$A_{huffman}$ =00 00 111 00 10 10 111 10 00 00 10 00 111 00 10 10 00 10 010 00
11000 00 011 00 1100000010000000011000100001110010100111001011101110
0101001100010110110110001011010011000101110110001011001
By replacing the bits with the symbols we recreate the string we $A_{runlength}$ that we encoded previously, so that we end up with
$A_{decoded} = A_{runlength}$ = [(1, 1), (3, 1), (2, 2), (3, 2), (1, 1), (2, 1), (3, 1), (2, 2), (1, 2), (0, 1), (-2, 1), (-1, 1), (-2, 1), (0, 1), (1, 1), (-1, 1), (0, 1), (-1, 2), (0, 2), (-1, 2), (0, 3), (-1, 2), (0, 2), (-1, 1), (0, 4), (-1, 1), (0, 6), (-1, 1), (0, 3), (-1, 1), (0, 13)]

2.12 Inverse Run Length Encoding

This step is also quite simple, we have our string $A_{runlength}$ which are tuples of the symbol and the occurrence of that symbol. So instead of having tuples we simply save the symbol as often as it occurs. So we end up with the same result that we had from $A_{reordered}$ that is

$A_{reordered}$ = [1, 3, 2, 2, 3, 3, 1, 2, 3, 2, 2, 1, 1, 0, -2, -1, -2, 0, 1, -1, -0, -1, -1, 0, 0, -1, -1, 0, 0, 0, -1, -1, 0, 0, -1, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

2.13 Inverse Reordering of Values

The values are still ordered with the Zig-Zag algorithm. The next step is to reverse that as well. Since we know how we reordered the values in the first place, we know at which index position which element has to be. This is computed by finding the indices of
resort_indices = [0, 1, 8, 16, 9, 2, 3, 10, 17, 24, 32, 25, 18, 11, 4, 5, 12, 19, 26, 33, 40, 48, 41, 34, 27, 20, 13, 6, 7, 14, 21, 28, 35, 42, 49, 56, 57, 50, 43, 36, 29, 22, 15, 23, 30, 37, 44, 51, 58, 59, 52, 45, 38, 31, 39, 46, 53, 60, 61, 54, 47, 55, 62, 63]
to generate $M_{zig-zag-inverse}$.
E.g. this means for the first few values of the first row of the $M_{zig-zig-inverse}$ that the value "0" in *resort_indices* is at index 0, the value "1" is at position 1, the value "3" is at index 5, the value "4" is at index 6, the value "5" is at index

14, and this is done analogously for the others.

$$M_{zig-zag-inverse} = \begin{pmatrix} 0 & 1 & 5 & 6 & 14 & 15 & 27 & 28 \\ 2 & 4 & 7 & 13 & 16 & 26 & 29 & 42 \\ 3 & 8 & 12 & 17 & 25 & 30 & 41 & 43 \\ 9 & 11 & 18 & 24 & 31 & 40 & 44 & 53 \\ 10 & 19 & 23 & 32 & 39 & 45 & 52 & 54 \\ 20 & 22 & 33 & 38 & 46 & 51 & 55 & 60 \\ 21 & 34 & 37 & 47 & 50 & 56 & 59 & 61 \\ 35 & 36 & 48 & 49 & 57 & 58 & 62 & 63 \end{pmatrix}$$

That means to reconstruct the original matrix we need to pick the values of the list $A_{reordered}$ as they appear in $M_{zig-zag-inverse}$ as shown in listing 6.

```python
def resort_values_zig_zag_reverse(block, block_size):
    global resort_indices_inverse
    # to reverse the process of zigzag iteration, give incremental list to zigzag iteration
    # then find indices of incremental values
    # then use indices to reverse reordering

    # store in variable so that it is not calculated again for every block
    if block_size not in resort_indices:
        resort_indices[block_size] = get_zig_zag_indices(block_size)
    if block_size not in resort_indices_inverse:
        resort_indices_inverse[block_size] = [resort_indices[block_size].index(i)
            for i in range(block_size ** 2)]

    indices = resort_indices_inverse[block_size]

    new_block = [block[index] for index in indices]
    return np.reshape(new_block, (block_size, block_size))
```

Listing 6: Inverse Zig-Zag ordering

After inverse Zig-Zag traversal we end up with the A_{quant} matrix which is

$$A_{quant} = \begin{pmatrix} 1 & 3 & 3\,1 & -2 & -1\,0\,0 \\ 2 & 3 & 2\,0 & -2 & -1\,0\,0 \\ 2 & 3 & 1\,0 & -1 & -1\,0\,0 \\ 2 & 1 & 1\,0 & -1 & 0 & 0\,0 \\ 2 & -1 & 0\,0 & -1 & 0 & 0\,0 \\ 0 & -1 & 0\,0 & -1 & 0 & 0\,0 \\ -1 & -1 & 0\,0 & -1 & 0 & 0\,0 \\ 0 & 0 & 0\,0 & 0 & 0 & 0\,0 \end{pmatrix}$$

2.14 Inverse Quantization

In this step it becomes apparent that the quantization process is lossy, since the recreated values are not the same anymore. This is done by applying the inverse quantization formula that is

$$A_{invQuantized}(i,j) = A(i,j) * Q_{50}(i,j) \tag{8}$$

Which results in

$$A_{invQuantized} = \begin{pmatrix} 16 & 33 & 30\,16 & -48 & -40\,0\,0 \\ 24 & 36 & 28\,0 & -52 & -58\,0\,0 \\ 28 & 39 & 16\,0 & -40 & -57\,0\,0 \\ 28 & 17 & 22\,0 & -51 & 0 & 0\,0 \\ 36 & -22 & 0\,0 & -68 & 0 & 0\,0 \\ 0 & -35 & 0\,0 & -81 & 0 & 0\,0 \\ -49 & -64 & 0\,0 & -103 & 0 & 0\,0 \\ 0 & 0 & 0\,0 & 0 & 0 & 0\,0 \end{pmatrix} \approx A_{DCT}$$

$$A_{DCT} = \begin{pmatrix} 22.716 & 29.113 & 26.342 & 11.589 & -44.815 & -28.815 & -9.226 & 4.734 \\ 20.53 & 30.835 & 25.829 & 7.534 & -48.093 & -29.547 & -10.505 & 3.943 \\ 22.656 & 35.618 & 23.6 & 3.84 & -48.818 & -30.466 & -10.235 & 3.203 \\ 31.889 & 16.015 & 14.804 & 11.468 & -52.689 & -31.663 & -8.349 & 4.183 \\ 28.108 & -18.873 & 8.607 & 16.374 & -58.148 & -29.66 & -10.229 & 2.02 \\ -11.483 & -33.224 & 15.697 & 17.038 & -53.131 & -27.986 & -8.906 & 3.49 \\ -28.961 & -38.285 & 16.25 & 17.834 & -52.622 & -28.813 & 8.845 & 3.296 \\ -28.803 & -39.434 & 14.458 & 18.613 & -54.557 & -27.923 & -6.082 & 9.098 \end{pmatrix}$$

As we can see in the matrices the values changed slightly since most of the higher frequencies where removed and could not have been reconstructed. Other frequencies are just approximated and close to their original value.
It is especially interesting to note that at this step the information of the high frequency components is lost, which can be observed by comparing the bottom lines of the matrix $A_{invQuantized}$ to A_{DCT}, which is the original image data.

2.15 Apply inverse DCT

In this step the inverse DCT is applied to convert from the frequency domain to the spatial domain of the picture, so that we can actually visualize it. The inverse DCT is easily derived from equation 6

$$\begin{aligned} A_{DCT} &= T * A_{image} * T^{-1} | T^{-1} * \\ T^{-1} * A_{DCT} &= A_{image} * T^{-1} | * T \\ T^{-1} * A_{DCT} * T &= A_{image} \end{aligned} \tag{9}$$

When applying equation 9 the result is

$$A_{reconstructedImage-128} = \begin{pmatrix} 24.53 & 34.15 & 39.21 & 26.91 & -33.91 & -32.57 & -10.26 & -7.7 \\ 34.58 & 39.37 & 36.9 & 16.87 & -34.82 & -52.31 & -14.33 & -13.32 \\ 40.25 & 37.3 & 23.18 & 18.64 & -25.98 & -50.28 & -12.78 & -12.13 \\ 31.52 & 14.94 & 35.14 & -1.89 & -40.76 & -1.42 & -5.78 & -2.43 \\ 31.7 & -28 & 29.01 & -9.49 & -58.52 & -7.47 & -11.83 & -7.92 \\ -4.99 & -30.25 & 27.66 & -10.26 & -73.96 & -9.55 & -16.97 & -11.23 \\ -58.1 & -43.31 & 29.19 & -14.02 & -98.46 & -14.08 & -25.02 & -16.98 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Before applying the DCT we subtracted 128 from every pixel previously, therefore we have to undo this as well as to round to the nearest integer and clip values that are not in our color range from [0,255]

$$A_{reconstructedImage} = \begin{pmatrix} 153 & 162 & 167 & 155 & 94 & 95 & 118 & 120 \\ 163 & 167 & 165 & 145 & 93 & 76 & 114 & 115 \\ 168 & 165 & 151 & 147 & 102 & 78 & 115 & 116 \\ 160 & 143 & 163 & 126 & 87 & 127 & 122 & 126 \\ 160 & 100 & 157 & 119 & 69 & 121 & 116 & 120 \\ 123 & 98 & 156 & 118 & 54 & 118 & 111 & 117 \\ 70 & 85 & 157 & 114 & 30 & 114 & 103 & 111 \\ 128 & 128 & 128 & 128 & 128 & 128 & 128 & 128 \end{pmatrix} \approx A_{image}$$

$$A_{image} = \begin{pmatrix} 158 & 156 & 165 & 147 & 98 & 106 & 110 & 124 \\ 157 & 159 & 164 & 144 & 95 & 105 & 108 & 122 \\ 161 & 163 & 161 & 142 & 94 & 104 & 108 & 121 \\ 165 & 139 & 161 & 145 & 90 & 101 & 108 & 121 \\ 152 & 106 & 164 & 142 & 85 & 100 & 105 & 116 \\ 109 & 105 & 164 & 141 & 87 & 100 & 105 & 117 \\ 91 & 105 & 162 & 142 & 86 & 99 & 104 & 116 \\ 91 & 103 & 161 & 142 & 83 & 98 & 104 & 122 \end{pmatrix}$$

We can see that the reconstructed image is quite similar but not exactly the same, but when comparing the actual pictures the perceived difference is smaller than the numbers would indicate.

2.16 Merge Blocks

At the moment all our information is still split up into smaller blocks, in order to reconstruct the image, we need to combine them as one piece. It is necessary to keep the information of the original resolution for this step. We calculated the down-sampled resolution with the original one and later used this for the merging process. The merging of the individual blocks happens for each of the Y,U and V component similarly. For each element the initial added padding blocks are now cropped so that only the original image resolution remains.

2.17 Up-Sampling

Up-sampling is necessary to recreate the original shape of the image. In order to do that we take our decompressed color components that were computed in the previous step and use the original resolution by also considering the sub-sampling settings of each sub-sampled component as shown in listing 7. It should be mentioned that our method repeats the information so that the true data resolution is still lower. As we can see in figure 7 we can already observe small artifacts in the compressed color components of the image.

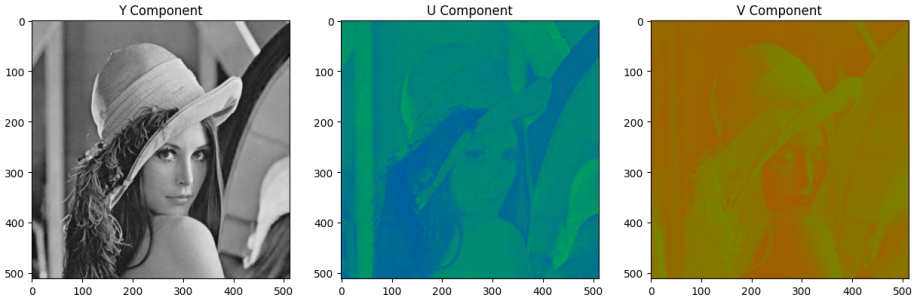


Fig. 7. Compressed Y,U and V Components

2.18 Inverse Color Space Transformation

As a last step the individual color components need to be transferred back to their original color space. Currently our elements are converted into the YUV Space. RGB is the preferred default representation which is also easier to grasp because it only contains a mixture of colors. Therefore we take our existing parts and change their base. This is achieved by the inverse operation of the formula stated in the beginning.
Starting with the transformation from Y,Cb,Cr into Y,U,V, the Cb and Cr components get each subtracted with the value of 128. Now again U and V are

```python
def up_sample_u_v(u, v, j, a, b, resolution=(32, 32)):
    if j != 4 or a not in [4, 2, 1] or b not in [a, 0] or j // a != j / a:
        raise RuntimeError("Invalid value for subsampling settings")
    if j == 4 and a == 4 and b == 4:
        return u, v
    if b != 0:
        u_up = u.repeat(1, axis=0).repeat(j // a, axis=1)
        v_up = v.repeat(1, axis=0).repeat(j // a, axis=1)
    else:
        u_up = u.repeat(2, axis=0).repeat(j // a, axis=1)
        v_up = v.repeat(2, axis=0).repeat(j // a, axis=1)
    u_up = u_up[:resolution[1], :resolution[0]]
    v_up = v_up[:resolution[1], :resolution[0]]
    return u_up, v_up
```

Listing 7: Up-sampling Implementation

the divergence of blue and red from the luminance. In order to get back to its original color space the following formulas need to be applied.

$$R = Y + 1.13983 * V \tag{10}$$
$$G = Y - 0.39465 * U - 0.58060 * V \tag{11}$$
$$B = Y + 2.03211 * U \tag{12}$$

After these steps the compressed image can be represented as before but with the benefit of greatly reduced file size with a quantization quality of 50% as can be seen in figure 8.

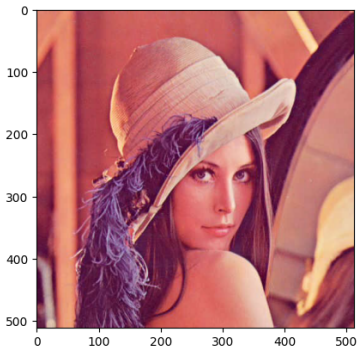


Fig. 8. Reconstructed original image with a quantization quality of 50%

3 Evaluation

In order to be able to measure the quality quantitatively apart from the optical impression, it was necessary to find quantitative measurement methods. After all, the applied compression method loses information that influences the optical impression and the quality.

3.1 PSNR

Finally, to evaluate the quality of our compression methods, we utilized the Peak Signal to Noise Ratio. We chose the PSNR because it is a commonly used measurement method to evaluate the image quality of compression algorithms. One of the big advantages of this measurement is that it correlates well with the human perception of image quality especially when the image is affected by noise as the human eye perceives changes logarithmically.

Of course it must be taken into account that PSNR is not a perfect method to measure image quality because not all types of distortion are taken into account equally. Ringing or block artifacts are not taken into account as much even though they are more perceptible to the human eye than other artifacts. In addition, PSNR does not give any information about the content of the image. Therefore it is possible that two different images have the same PSNR value even though they have different levels of detail or sharpness.

The PSNR is a metric to calculate the quality of a compressed or reconstructed image. It is calculated as the ratio of the maximum pixel value to the mean squared error between two images - the original image and the compressed image, as shown in the mathematical formula in equation 13 and 14.

$$MSE = \frac{1}{m \cdot n} \sum_{m-1}^{i=0} \sum_{n-1}^{j=0} [I(i,j) - K(i,j)]^2 \tag{13}$$

$$PSNR = 10 \cdot log_{10} \left(\frac{MAX_I^2}{MSE} \right) = 20 \cdot log_{10} \left(\frac{MAX_I}{\sqrt{MSE}} \right) \tag{14}$$

Due to the fact that both the original image and the compressed image are required to calculate a score, there is a disadvantage that the quality cannot be determined by only examining the compressed image as a human intuitively could. In addition, the original image may already not have optimal quality characteristics and thus the PSNR does not correlate with the actual image quality since the calculation of the PSNR score is influenced from the beginning. Therefore the PSNR is solely a measure to compare the quality of a compressed image to the original one, indicating the similarity.

We have implemented this formula for our calculations as shown in listing 8, which is also compatible for RGB images. In the case of RGB images, the

```python
def get_peak_signal_to_noise_ratio(self):
    mse = np.mean((self.original_image - self.decompressed_image) ** 2)
    if mse == 0:
        return np.inf
    psnr = 10 * np.log10((255 ** 2) / mse)
    return psnr
```

Listing 8: PSNR Calculation Implementation

average mean square error for all individual colors must be taken, which is done already as "self.original_image" and "self.decompressed_image" are lists of three color-components, corresponding to R, G and B.

In summary, the PSNR means that a high value indicates better image quality while low values mean poorer image quality.

3.2 Analyzing the Image Quality

First of all, it was important for us to see how the influence of the quantization tables affects the image quality (Figure 9). A very interesting picture emerges

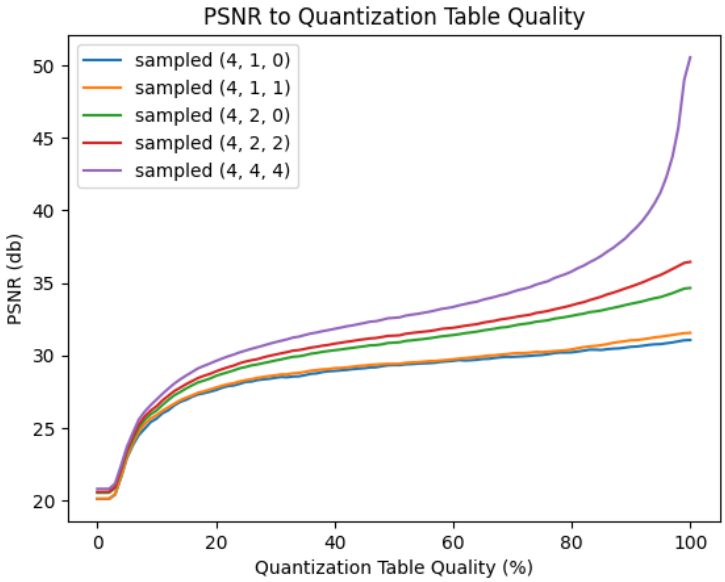


Fig. 9. Quantization to PSNR Relation

for all sub-sampling methods. For low quantization table qualities, the image quality changes very strongly at first as indicated by the PSNR-score, but already from about 20% (quality of the quantization table) the PSNR-score increases significantly less than before.

It should be noted that for the sub-sampling settings (4:4:4) the PSNR for high qualities still increases strongly for higher quantization qualities, this is due to the fact that for this setting no sub-sampling is performed and thus the image can be restored without any significant loss in comparison to other sub-sampling configurations. Furthermore it is very interesting to highlight that for the sub-sampling settings (4:2:2) and (4:2:0) very similar behavior of image quality to quantization table quality can be observed, although the number of samples for (4:2:2) is twice as high as for (4:2:0). And that behaves similarly with the settings (4:1:1) and (4:1:0).

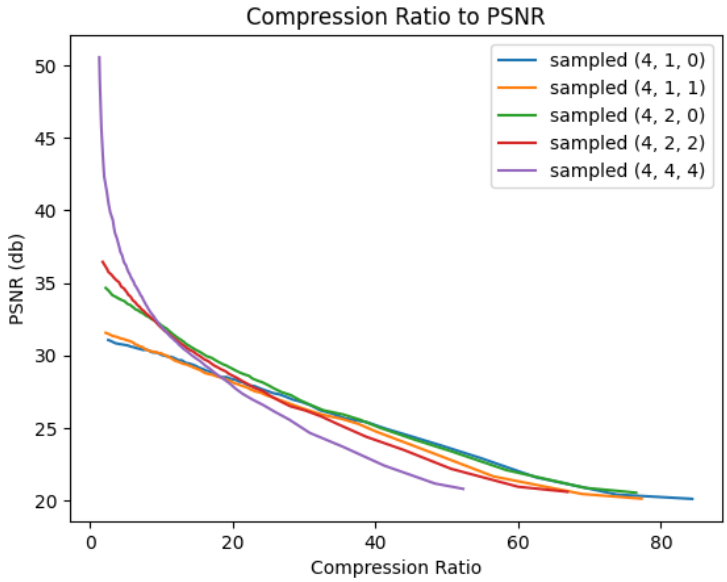


Fig. 10. Rate-Distortion Curve

In the course of our evaluation, a clear correlation between the compression rate and the PSNR could also be observed. As can be seen in figure 10, the image quality decreases as the compression rate increases. We can clearly see that there is no linear relationship and the curve rather has a convex shape, indicating that there is a diminishing return in terms of image quality as the compression ratio increases. This relationship is also called the rate-distortion curve.

Finally, it was interesting for us to find out how the quantization table quality affects the compression rate. As can be seen in figure 11, the effectiveness of the

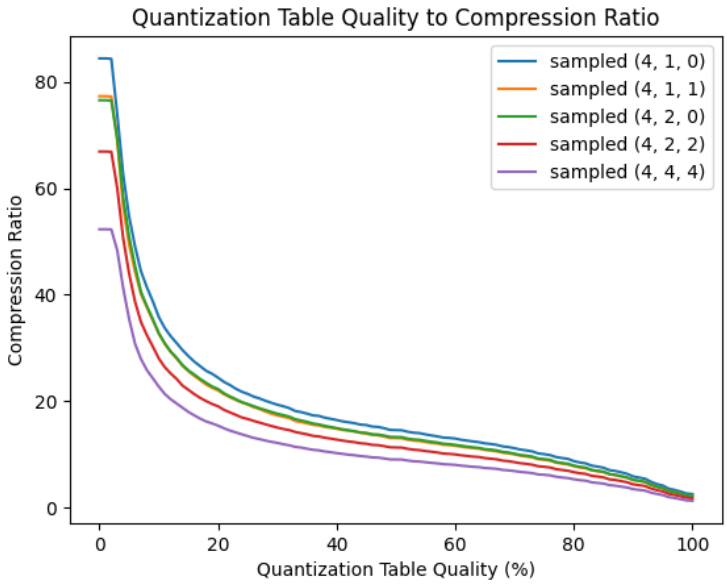


Fig. 11. Quantization to Compression Ratio Relation

compression rate is highest for very low qualities however the compression rate differs less for qualities in the center bracket. Based on this combination of graphs we could conclude that the optical image quality changes the most for low qualities and when higher qualities of quantization tables are used mainly the compression rate changes (i.e. the file gets bigger) but the image quality does not increase as much.

3.3 Visualizing Assumptions

To verify this, we have compressed the image for very low qualities, from 2.5 to 15%, and plotted it visually in figure 12. It can be clearly seen that already at a quality of 10% the artifacts are comparatively minor, and are not easily detectable without direct comparison to the original image. Furthermore the perceived quality increase is less from 10% to 15%, than from 2.5% to 7.5%. This matches the the steep increase of the PSNR score for low quantization qualities as shown in figure 10.

These graphs (Figure 9, 10, 11) suggest that a quality of about 20% can already achieve good optical results while maintaining good compression, which is also confirmed in Figure 13 where those settings were applied (Of course this is only a subjective measure and strongly dependant on the individual as well as the input image).

Fig. 12. Visualisation of Quantization Qualities

However even when comparing the compressed side by side to the original one (Figure 13), hardly any visual differences can be made out, but only with the help of the difference shown in addition. Here it can be seen quite clearly that predominantly information from the high-frequency range is lost, these are details on the hat and the sharpness of the contours.

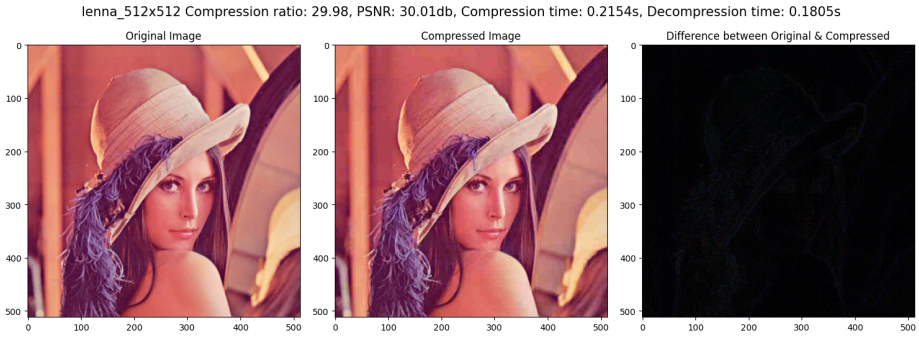


Fig. 13. Visualisation of Quantization Quality of 20% (Sub-sampled (4,2,0))

To further strengthen this assumption we also applied the same settings (Quantization quality of 20%, Sub-sampling: (4:2:0)) to a second image (Figure 14) which is higher in resolution and which has greater detail/high frequency

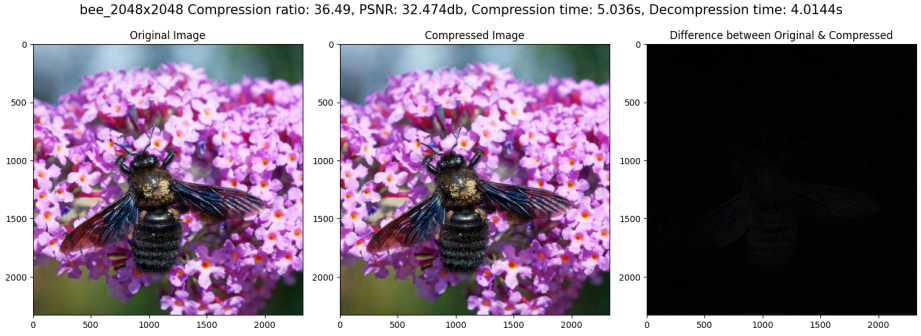


Fig. 14. Visualisation of Quantization Quality for higher resolution Image of 20% (Sub-sampled (4,2,0))

information in the image. Also this image can be resembled very close to the original image at 20% quantization quality. It can be seen that most of the lost information comes from the difference in the lower body and the wings of the bee, as shown in the difference. Also it is interesting to note that due to the higher image resolution the compression ratio is more effective as the algorithm can work more efficiently, while maintaining a sufficient PSNR score.

4 Conclusion

We have implemented JPEG from scratch without using any additional image-processing libraries, just with the mathematical basics. The parameters like the quality and the sub-sampling settings are modifiable and were compared in the evaluation section. Here the result showed that with 50% compression we can hardly see any difference to the original image, but even 20% were sufficient enough to get good results. As we can see from the evaluation we have diminishing returns in image quality indicated by the PSNR when the quality is set higher than 20%. Furthermore the compression ratio is superior for images with higher resolution because of the use of Huffman coding that performs better on data that has a lot of similar values.

References

1. Hooker, D.: Lenna. Signal and Image Processing Institute (Jul 1973), https://sipi.usc.edu/
2. Ken, C., Gent, P.: Image compression and the discrete cosine transform. College of the Redwoods, Tech. Rep (1998)
3. Stevo88: File:common chroma subsampling ratios.svg (Jul 2010), https://commons.wikimedia.org/wiki/File:Common_chroma_subsampling_ratios.svg