JRpeg

JPEG is a popular lossy image compression method based on the principle of quantising block sections of the discrete cosine transform of a YCbCr image. This allows for run length and Huffman encoding methods to be highly effective as most of the quantised block sections elements will get zeroed. Since the quantisation matrix applied to each section can be increased/decreased by a factor, user selectable compression levels can be easily implemented. The following section documents the implementation of a JPEG like compression called JRpeg.

Compression

All methods referenced for JRpeg compression can be found in JRpeg_compress.py, with the JRpeg_compress(input_filename, output_filename, cbcr_downsize_rate, QL_rate, QC_rate) method as a wrapper to complete a full JRpeg compression cycle given a set of input arguments.

RGB To YCbCr

Research suggests that luminance is a much more important factor in perceived image quality in comparison to colour (Meilikov, 2018). Therefore, the RGB image can be converted to a format where luminance is separate from chrominance, so that the chrominance component can be heavily down sampled without affecting the luminance (which leads to minimal loss in perceived image quality post compression). This can be done in the YCbCr colour space, for which the Y channel will contain the image luminance, and CbCr chrominance. The following formula **(fig. 1)** shows the conversion between RGB and YCbCr values [9]:

Conversion to $y'c_Bc_R$ from $r'g'b'$			
y' =	0.299 * r' +	0.587 * g' +	0.114 * b'
c_B =	-0.168736 * r' +	-0.331264 * g' +	0.500 * b'
c_R =	0.500 * r' +	-0.418688 * g' +	-0.081312 * b'

Figure 1

The implementation of this formula for JRpeg can be found in the rgb_to_ycbcr(img) method, which first takes in a cv2 style three channel RGB image array, then stored the R, G and B matrix components separately. The Y, CB and CR matrix arrays can then be calculated and stored in the corresponding cv2 image channel, which is subsequently returned.

Generally, the Cb and Cr components can downsized by a factor of two in both horizontal and vertical directions before any noticeable reduction in image quality is perceived. Although JPEG allows for down sampling at different rates for the horizontal and vertical components [10], JRpeg currently only allows for down sampling with the same factor in each component. The following method: down_sample_cbcr(YCbCr, sample_factor), will run a box averaging filter over the Cb and Cr components of size [sample_factor x sample_factor] to first smooth out each area that is being sampled, before creating a new image matrix from pixels spaced by the sample_factor. For example, if sample_factor is two, every two pixels will be used for new matrix. Each component matrix is then converted to a float before being returned. There is also validation around sample_factor so that any sample rate less than one will skip the above methods. Its important to note that down sampling is a lossy processes, although since only a small averaging mask is used, the information loss should be minimal.

Discrete Cosine Transform And Quantisation

The next stage in JPEG compression is to calculate the DCT of the image in 8x8 blocks, and apply a quantisation matrix in an attempt to zero out the pixel values that have little affect eventual perceived image quality. The basic theory behind using the DCT on 8x8 sections of the image is that each block can be represented using 64 cosine waves with the lower frequencies in the top left of the block and the higher frequencies nearer the bottom right (Computerphile, 2015). Since most of the image blocks 'information' will be in the low frequencies range, most of the high frequency values can be removed, resulting in a matrix of mostly zeros, which can be easily and heavily compressed with run length encoding. JPEG outlines two separate quantisation matrices for the luminance and chrominance components, as the latter can be quantised more heavily. These matrices are widely accepted as the default quantisation tables due to giving consistently satisfactory results across a range of applications [6]:

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

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

Figure 2

These quantisation tables are represented using NumPy arrays stored in JRpeg_util.py. The dtc_and_quantise_img(img, QL_rate, QC_rate) method first ensures the width and height of each image channel is divisible by 8, before using skimage.util.view_as_blocks to split the channels matrix array into 8x8 blocks. Each block is then converted to float and cantered to be within [-128 < x < 128] instead of [0 < x < 255] as the cosine wave ranges from [-1 < x < 1], before the cv2.dct function is applied. If the block belongs to the Y channel, the block is divided by the luminance quantisation matrix multiplied by the luminance quantisation rate (QL_rate), and then truncated resulting in all fractional values calculated being zeroed. A similar process is done for chrominance, but with applying the chroma quantisation matrix and rate. There is validation around the quantisation rates to ensure they are not 0 or below, as this would cause divide by zero errors and negative quantisation values. However, the quantisation rates can be fractional which will result in less information loss, but a lower compression ratio (as a result of the corresponding quantisation matrix being 'lighter').

Encoding and saving to disk

As briefly mentioned before, run length encoding (RLE) can be used to greatly minimize the size of each 8x8 block. The basic principle behind RLE is that consecutive elements of the same value are grouped together as [value, frequency], which greatly reduces the total length of the list whilst being lossless in its compression. The following example shows how RLE can be used to reduce a string from 9 to 6: [AAABBBBCC] -> [3A4B2C].

In JRpeg, the encoding is done in the following method: encode_and_save_quantised_dct_img(img_blocks, QL_rate, QC_rate, filename). First, each block has to be converted into single list array rather than a 2d matrix so that RLE can be most effective. This is done using a 'zig zag' style matrix traversal, shown in **fig. 3** below [11]:

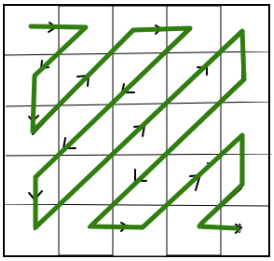

Figure 3

	0	1	2	3	4	5	6	7
0	-13.00000	-3.00000	6.00000	-0.00000	1.00000	-0.00000	0.00000	-0.00000
1	11.00000	-9.00000	1.00000	1.00000	-0.00000	0.00000	0.00000	-0.00000
2	2.00000	13.00000	-0.00000	0.00000	-0.00000	0.00000	-0.00000	0.00000
3	0.00000	2.00000	0.00000	-0.00000	-0.00000	0.00000	-0.00000	0.00000
4	1.00000	-0.00000	-0.00000	0.00000	-0.00000	0.00000	-0.00000	-0.00000
5	-0.00000	0.00000	0.00000	-0.00000	-0.00000	-0.00000	0.00000	0.00000
6	0.00000	0.00000	-0.00000	-0.00000	-0.00000	-0.00000	-0.00000	-0.00000
7	0.00000	-0.00000	0.00000	0.00000	0.00000	-0.00000	-0.00000	0.00000

Figure 4

The simplest and most efficient implementation for this traversal is to use a 8x8 matrix with the zig zag list index in each element, then loop through the block and place each element to its corresponding position in the list (defined by the index matrix). JRpeg implements this in JRpeg_util.py in the zigzag_block(block) method, that outputs a single 64 element list of zig zag format, which is then added to the corresponding channel of the output array, encoded_list. Each element of each channel in the output array is then converted from float to int (which greatly reduces storage size as the float values take up 64 bits each), and then grouped RLE style using itertools.groupby which creates tuples of [amount, value], where amount refers to the frequency of consecutive occurrences of an element, for example [0,0,0,0] -> (4, 0). A further optimisation that was subsequently added was to then unpack the tuples with only 1 occurrence and just store the raw value, as those tuples were actually twice as big as the raw value. This greatly increased the compression ratio, from approximately 55% to 80% (for image IC1.bmp).

An example output for JRpeg style RLE for the block **(fig.4)** (the first quantised block of IC1.bmp) would be: [-13, -3, 11, 2, -9, 6, 0, 1, 13, 0, 1, 2, 0, [2,1], [49,0]]. This one block was reduced from 64 elements to 15, which is a reduction of 188 bytes (as python reserves 32 bits for each int).

The final step in JRpeg encoding is to append the metadata, which is the block height and width values for each channel and the values for QL_rate and QC_rate as this data will be necessary to reconstruct the image during decompression. The encoded list is then saved as a .jrpg file using pickle.dump (which is essentially just a binary file)

Decompression

The referenced methods for the JRpeg decompression cycle can be found in JRpeg_decompress.py. JRpeg decompression takes in a binary file in .jrpg format, decompresses it and displays the output image. A BMP copy of the decompressed image is also stored to disk as well as calculating the MSE against the original image (before JRpeg compression).

Load and Decode .jrpg File

The load_and_decode_quantised_dct_img(filename) method first loads a .jrpg file into a list called compressed_img using pickle.load, followed by the quantisation rate meta data being extracted. Then for each channel, its corresponding block height and width metadata is extracted, and an empty 4D NumPy array called out_blocks containing 8x8 blocks for the given block [block_height x block_width] is initialised. Then each element in the current channel is checked to see if it's a tuple grouping, and if so unpack it and add consecutive elements for its corresponding frequency to the output list, decoded_list. If the element isn't a tuple the raw value is simply appended onto the output list. The current channel is then split into list chunks of 64 elements, which is then fed into the JRpeg_util.un_zigzag_block(block_list) method that rearranges the list into its original block format using the previously referenced zigzag_idx matrix array. The reconstructed block is then added to its corresponding position in the previously initialised out_blocks array, which is subsequently returned along with the quantisation rate metadata.

Inverse DCT

The previous function essentially returned the YCbCr image with the DCT and quantisation matrices applied, therefore the next logical step is to inverse this. The inverse_dct_blocks(decoded_list) function will first take in the decoded YCbCr image as well as the quantisation rates, and will multiply each block in the current channel by its corresponding quantisation matrix multiplied by its corresponding quantisation rate.

With quantisation now reversed, the YCbCr image is essentially back to the original DCT version from compression, with slight information loss of the higher frequencies. The inverse DCT is then calculated using cv2.dct with the DCT_INVERSE flag, followed by the values being readjusted to be back within the [0 < x < 255] range. The 4D matrix blocks array for the current channel is then combined back into a single matrix array using einops.rearrange, with the values being subsequently truncated. The image, now back in normal (possibly downsized) YCbCr format, is then returned.

Converting back into RGB

The final stage in the JRpeg decompression cycle is convert the YCbCr image values back into their corresponding RGB values. This is done in the YCbCr_to_rgb(YCbCr) method, which first resizes the Cb and Cr channels back to their original size, followed by the YCbCr variable being adjusted to be in cv2 format (with the channel index as the last element). The Y, Cb and Cr components are then extracted, for which the RGB values are calculated from using the formula in **fig.5** below:

Conversion to r'g'b' from $y'c_Bc_R$

$$
\begin{aligned}
r' &= 1.0 * y' + 0 * c_B + 1.402 * c_R \\
g' &= 1.0 * y' + -0.344136 * c_B + -0.714136 * c_R \\
b' &= 1.0 * y' + 1.772 * c_B + 0 * c_R
\end{aligned}
$$

Figure 5

The final RGB output is then adjusted so that all the pixels are within the [0 < x < 255] range, and then converted to np.unit8 (8 bit unsigned integers), before being returned.

Further optimisations

As briefly mentioned before, when down sampling the Cb and Cr components during compression, the horizontal and vertical directions are not adjustable independently, as only one parameter, sample_factor, is supplied to the method. Although this somewhat reduces complexity in the GUI, having both directions adjustable would allow for 4:2:2 sub sampling, which is commonly used in digital cameras and could potentially yield better results [12].

During the final step of encoding, it is mentioned that the values are stores and int data types. Python reserves 32 bits for each int in memory, and since no values really exceeded the [-70 < x <70] range, an 8 bit signed integer data type could be used (which would be within [-128 < x <128]). NumPy provides a datatype called np.int8, which supposedly is a 8 bit signed integer, however when tested, both the in memory and on disk compression rates were much higher than that of a regular python int, so more research could be done around the datatype used for storage which could potentially also yield better results.

JPEG uses a combination of RLE and Huffman encoding, whereas JRpeg only used RLE style encoding. The basic principle behind Huffman encoding is that a binary key is used in conjunction with the frequently occurring values, which is stored with a binary tree, resulting in lossless compression. This encoding method could have been applied after the RLE style encoding, however due to how RLE was implemented in JRpeg this would potentially not yield much better results.

Results Analysis

Appendix: Default Parameters(CbCr Downsize Rate = 2, QL_rate = 1, QC_rate = 1):

Using the default compression values with a CbCr down sampling rate of 2, and with standard quantisation matrices applied, images IC1 to IC5 have minimal to no noticeable perceived loss in image quality. Images with little change in intensity such as IC1 (78% compression) and IC3 (80% compression) are more effectively compressed by JRpeg than that of the more 'information heavy' images with frequent changes in intensity, such as IC4 (58% compression), which has frequent intensity changes where the goose and trees are. This is as expected since less of the DCT values will be zeroed during compression resulting in the RLE being less effective. The compression ratio when using the PNG equivalent images is always lower than that of the BMP image, which is also to be expected as PNG applies lossless compression to the image, whereas BMP has just the raw pixel values with no compression. This can be confirmed by looking at the disk and in memory size of the original BMP images, as its exactly the same for each image. Since the output MSE of both PNG and BMP images are exactly the same, this also confirms that there is no lossless compression within the PNG image format. The final observation to note, is that the in memory size for the encoded image is generally much than that of the original image. This is to be expected as not only is each value of the original image stored as an 8 bit unsigned integer (compared to 32 bit int of encoded image), there are also multiple embedded data structures within the encoded image that are each separately stored as objects in memory.

Appendix: Chrominance Parameters:

When the Cb and Cr components are not downsized there is only a small decrease in MSE from 61.2 to 59.8, with quite a large reduction in compression rate by almost 7%. When only applying a tenth (QC_rate = 0.1) of the chroma quantisation matrix, the MSE decrease and perceived image quality increase is minimal, at 58.5 MSE. However the compression rate is heavily decreased, at only 37% compared to the original 58%. Increasing the QC_rate to 4 yields slight improvement on the compression rate (with downsize still 0), but heavily increases MSE to 66 (although there is no noticeable perceived image quality loss). Increasing QC_rate to 6 with no downsizing gets the compression rate back up to 58%, the same as with a down sampling factor of 2 and standard chroma quantisation. However, MSE is increased to 69 and there is highly noticeable perceived image quality loss, with IC4 looking grey and washed out, and IC1 (with the same params applied) having slightly off colours, especially in the clouds. The downs sample rate of IC4 can be increased all the way to 16 until noticeable image quality reduction can be perceived, however, for IC1 at down sample of 4 noticeable colour spot blurring can be seen around the bright red areas near the house door. These results confirm that the Cb Cr down sampling does not reduce image quality (on images without condensed areas of bright colours) as much as other factors (as predicted by theory), whilst increasing compression rate significantly. This also shows chroma quantisation potentially has big impact on resultant image quality, however this could be due to the fact the JPEG standard chroma quantisation matrix is quite heavy (see JRpeg_util.py).

Appendix: Luminance Parameters:

By only increasing QL_rate to two, there is a huge increase in compression ratio by almost 10% for IC4, and even though the MSE increase is significant at 71.1, the perceived image quality loss isn't noticeable unless zoomed in. At QL_rate 8, the image quality loss starts to become noticeable at normal viewing distance (with MSE of 100), however, the compression rate is increased by almost 13%. At QL_rate 16, the image quality loss in very apparent, with significant blurring and 'blocky-ness', although the different parts of the picture such as the goose and trees are still distinguishable (with an increased MSE to only 105). The compression rate however goes up to almost 90%, which is a huge increase from the original 58%. The same QL_rate applied to IC1 gives similar results, with very noticeable sections within the clouds and grass. These results suggest the quantisation of the luminance has a large effect on resultant perceived image quality (as theorised), but also a large effect on the compression ratio. Although, as the luminance quantisation rate is increased exponentially, it begins to return diminishing increases to compression rate, with more significant reductions in perceived image quality.

Appendix: All Parameters:

Given the above results and testing, parameters of Downsize Rate = 4, QL_rate = 16 and QC_rate = 6 seem to be most effective values at getting the highest compression rate, without loosing too much perceived image quality in both luminance and chrominance. When these parameters are applied to IC4, the compression rate increases to 92% from 58% with the default parameters, with MSE increasing to 105.6 in comparison to its original value of 61.2. When these parameters are applied to a less 'information heavy' image such as IC1, the compression rate is even higher at 95%, although the perceived loss in image quality is much more noticeable (with vast sections of the image smoothed out). As mentioned earlier, further increases to luminance quantisation returns diminishing results, with IC1 returning a slight increase to 96.6% compression rate with a QL_rate of 25, with greatly increased image perception loss. The final figure in the appendix shows what happens when all parameters are set to an incredibly high number (30 for each). The compression rate goes to 99% which is incredibly high, however, the image quality is incredibly reduced, with no colour and almost no definition (although the building is still distinguishable in IC1)

Appendix:

Original Images:

IC1:

IC2:

IC3:

IC4:

IC5:

JRpeg

Default Parameters(CbCr Downsize Rate = 2, QL_rate = 1, QC_rate = 1):

IC1.bmp:

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	53101204 bytes	**Compressed On Disk Size**	7906032 bytes
Compression ratio	0.69	**Compression ratio**	4.63
Space saved	-45.171%	**Space saved**	78.386%
Mean Squared Error		31.282	

IC1.png:

Original In Mem Size	36578432 bytes	**Original On Disk Size**	12570903 bytes
Compressed In Mem Size	53101204 bytes	**Compressed On Disk Size**	7906032 bytes
Compression ratio	0.69	**Compression ratio**	1.59
Space saved	-45.171%	**Space saved**	37.108%
Mean Squared Error		31.282	

Decompressed Image:

IC2.bmp:

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	57520708 bytes	**Compressed On Disk Size**	8777556 bytes
Compression ratio	0.64	**Compression ratio**	4.17
Space saved	-57.253%	**Space saved**	76.003%
Mean Squared Error		30.628	

IC2.png:

Original In Mem Size	36578432 bytes	**Original On Disk Size**	14747737 bytes
Compressed In Mem Size	57520708 bytes	**Compressed On Disk Size**	8777556 bytes
Compression ratio	0.64	**Compression ratio**	1.68
Space saved	-57.253%	**Space saved**	40.482%
Mean Squared Error		30.628	

Decompressed Image:

IC3.bmp:

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	49812276 bytes	**Compressed On Disk Size**	7201999 bytes
Compression ratio	0.73	**Compression ratio**	5.08
Space saved	-36.179%	**Space saved**	80.311%
Mean Squared Error		25.265	

IC3.png:

Original In Mem Size	36578432 bytes	**Original On Disk Size**	10443637 bytes
Compressed In Mem Size	49812276 bytes	**Compressed On Disk Size**	7201999 bytes
Compression ratio	0.73	**Compression ratio**	1.45
Space saved	-36.179%	**Space saved**	31.039%
Mean Squared Error		25.265	

Decompressed Image:

IC4.bmp:

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	92206220 bytes	**Compressed On Disk Size**	15495421 bytes
Compression ratio	0.4	**Compression ratio**	2.36
Space saved	-152.078%	**Space saved**	57.638%
Mean Squared Error		61.192	

IC4.png:

Original In Mem Size	36578432 bytes	**Original On Disk Size**	25214292 bytes
Compressed In Mem Size	92206220 bytes	**Compressed On Disk Size**	15495421 bytes
Compression ratio	0.4	**Compression ratio**	1.63
Space saved	-152.078%	**Space saved**	38.545%
Mean Squared Error		61.192	

Decompressed Image:

IC5.bmp:

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	82114876 bytes	**Compressed On Disk Size**	13250119 bytes
Compression ratio	0.45	**Compression ratio**	2.76
Space saved	-124.49%	**Space saved**	63.776%
Mean Squared Error		52.67	

IC5.png:

Original In Mem Size	36578432 bytes	**Original On Disk Size**	21291035 bytes
Compressed In Mem Size	82114876 bytes	**Compressed On Disk Size**	13250119 bytes
Compression ratio	0.45	**Compression ratio**	1.61
Space saved	-124.49%	**Space saved**	37.767%
Mean Squared Error		52.67	

Decompressed Image:

Chrominance Parameters:

IC4 (CbCr Downsize Rate = 0, QL_rate = 1, QC_rate = 1):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	110278336 bytes	**Compressed On Disk Size**	17852066 bytes
Compression ratio	0.33	**Compression ratio**	2.05
Space saved	-201.485%	**Space saved**	51.195%
Mean Squared Error		59.823	

IC4 (CbCr Downsize Rate = 0, QL_rate = 1, QC_rate = 0.1):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	144015120 bytes	**Compressed On Disk Size**	23212994 bytes
Compression ratio	0.25	**Compression ratio**	1.58
Space saved	-293.716%	**Space saved**	36.539%
Mean Squared Error		58.492	

IC4 (CbCr Downsize Rate = 0, QL_rate = 1, QC_rate = 4):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	95994800 bytes	**Compressed On Disk Size**	15940988 bytes
Compression ratio	0.38	**Compression ratio**	2.29
Space saved	-162.436%	**Space saved**	56.42%
Mean Squared Error		65.833	

IC4 (CbCr Downsize Rate = 0, QL_rate = 1, QC_rate = 6):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	91365416 bytes	**Compressed On Disk Size**	15339389 bytes
Compression ratio	0.4	**Compression ratio**	2.38
Space saved	-149.779%	**Space saved**	58.064%
Mean Squared Error		69.03	

Decompressed image:

IC1 (CbCr Downsize Rate = 0, QL_rate = 1, QC_rate = 6):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	8442273 bytes
Compressed In Mem Size	56656392 bytes	**Compressed On Disk Size**	15339389 bytes
Compression ratio	0.65	**Compression ratio**	4.33
Space saved	-54.89%	**Space saved**	76.92%
Mean Squared Error		53.349	

Decompressed image:

IC4 (CbCr Downsize Rate = 4, QL_rate = 1, QC_rate = 1):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	87387680 bytes	**Compressed On Disk Size**	14842830 bytes
Compression ratio	0.42	**Compression ratio**	2.46
Space saved	-138.905%	**Space saved**	59.422%
Mean Squared Error		64.627	

IC4 (CbCr Downsize Rate = 16, QL_rate = 1, QC_rate = 1):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	85726996 bytes	**Compressed On Disk Size**	14616421 bytes
Compression ratio	0.43	**Compression ratio**	2.5
Space saved	-134.365%	**Space saved**	60.041%
Mean Squared Error		67.686	

Decompressed image:

IC1 (CbCr Downsize Rate = 4, QL_rate = 1, QC_rate = 1):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	47665324 bytes	**Compressed On Disk Size**	7230748 bytes
Compression ratio	0.77	**Compression ratio**	5.06
Space saved	-30.31%	**Space saved**	80.232%
Mean Squared Error		34.905	

Decompressed image:

Luminance Parameters:

IC4 (CbCr Downsize Rate = 2, QL_rate = 2, QC_rate = 1):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	74515880 bytes	**Compressed On Disk Size**	11922394 bytes
Compression ratio	0.49	**Compression ratio**	3.07
Space saved	-103.715%	**Space saved**	67.406%
Mean Squared Error		71.065	

IC4 (CbCr Downsize Rate = 2, QL_rate = 8, QC_rate = 1):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	47665324 bytes	**Compressed On Disk Size**	7230748 bytes
Compression ratio	0.77	**Compression ratio**	5.06
Space saved	-30.31%	**Space saved**	80.232%
Mean Squared Error		100.911	

Decompressed image:

IC4 (CbCr Downsize Rate = 2, QL_rate = 16**, QC_rate = 1):**

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	27839240 bytes	**Compressed On Disk Size**	3827430 bytes
Compression ratio	1.31	**Compression ratio**	9.56
Space saved	23.892%	**Space saved**	89.536%
Mean Squared Error		104.956	

Decompressed image:

IC1 (CbCr Downsize Rate = 2, QL_rate = 16, QC_rate = 1):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	20548500 bytes	**Compressed On Disk Size**	2547783 bytes
Compression ratio	1.78	**Compression ratio**	14.36
Space saved	43.823%	**Space saved**	93.035%
Mean Squared Error		98.5	

Decompressed image:

All Parameters:

IC4 (CbCr Downsize Rate = 4, QL_rate = 16, QC_rate = 6):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	20548500 bytes	**Compressed On Disk Size**	3020805 bytes
Compression ratio	1.67	**Compression ratio**	12.11
Space saved	40.045%	**Space saved**	91.742%
Mean Squared Error		105.633	

Decompressed image:

IC1 (CbCr Downsize Rate = 4, QL_rate = 16, QC_rate = 6):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	14401896 bytes	**Compressed On Disk Size**	1786273 bytes
Compression ratio	2.54	**Compression ratio**	20.48
Space saved	60.627%	**Space saved**	95.117%
Mean Squared Error		97.913	

Decompressed image:

IC1 (CbCr Downsize Rate = 4, QL_rate = 25, QC_rate = 6):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	10461712 bytes	**Compressed On Disk Size**	1262130 bytes
Compression ratio	3.5	**Compression ratio**	28.98
Space saved	71.399%	**Space saved**	96.55%
Mean Squared Error		103.982	

Decompressed image:

IC1 (CbCr Downsize Rate = 30, QL_rate = 30, QC_rate = 30):

Original In Mem Size	36578432 bytes	**Original On Disk Size**	36578442 bytes
Compressed In Mem Size	2594508 bytes	**Compressed On Disk Size**	303995 bytes
Compression ratio	14.1	**Compression ratio**	120.33
Space saved	92.907%	**Space saved**	99.169%
Mean Squared Error		103.982	

Decompressed image:

Bibliography

[1] Meilikov, Evgeny & Farzetdinova, Rimma. (2018). Color or Luminance Contrast – What Is More Important for Vision?. Studies in Computational Intelligence. 736. 147-156. 10.1007/978-3-319-66604-4_22.

[2] Computerphile. (2015). JPEG DCT, Discrete Cosine Transform (JPEG Pt2). [Online Video]. 22 May 2015. Available from: https://www.youtube.com/watch?v=Q2aEzeMDHMA&t=720s&ab_channel=Computerphile - accessed 05/12/2020

[3] https://stackoverflow.com/questions/2866380/how-can-i-time-a-code-segment-for-testing-performance-with-pythons-timeit - accessed 08/12/2020

[4] https://appliedmachinelearning.blog/2017/03/08/image-compression-using-k-means-clustering/ - accessed 05/12/2020

[5] https://towardsdatascience.com/image-compression-using-k-means-clustering-aa0c91bb0eeb - accessed 05/12/2020

[6] https://www.sciencedirect.com/topics/computer-science/quantization-matrix - accessed 08/12/2020

[7] http://devmag.org.za/2011/02/23/quadtrees-implementation/ - accessed 06/12/2020

[8] https://homepages.cae.wisc.edu/~ece533/project/f06/aguilera_rpt.pdf - Accessed 09/12/2020 - accessed 09/12/2020

[9] https://www.mir.com/DMG/ycbcr.html - accessed 08/12/2020

[10] https://www.impulseadventure.com/photo/chroma-subsampling.html - accessed 08/12/2020

[11] https://www.geeksforgeeks.org/print-matrix-zag-zag-fashion/ - accessed 08/12/2020

[12] https://www.impulseadventure.com/photo/chroma-subsampling.html.- accessed 08/12/2020