CMPT 365 – Assignment 2

Darien Flamont

Rastin Piri

JPEG Encoding

Image Resizing to fit 8x8 blocks: One of the first steps we take in our code is to make sure our image is a multiple of 8x8 blocks so we can later apply DCT and Quantization on blocks. The algorithm we used simply finds the floor after height and width are divided by the block size. Then we take that ratio and times it by our block size. This creates an image that has a height and width that is a multiple of 8 and we are trimming as few rows and columns of pixels as possible [1].

RGB-YUV: For the RGB to YUV compression we used a couple of tools built into python3 that helped us manipulate the images and get the proper format needed to apply the 3x3 matrix multiplication for RGB channels to YUV channels. Firstly we used numpy in order to create matrices, in python this can also be done by using nested listed, but because numpy already comes with tools such as colon notation, zero matrices, splitting, type conversion, matrix multiplication, and matrix division we decided to employ the built in numpy library for building matrices instead of using hardcoded nested lists. We also used the image library PIL that helped us convert and resize images to RGB format for use. Initially when opening an Image file that was of the type png it would be transferred in as a 4 channel matrix with RGBA and the alpha channel attached as a fourth channel so we were able to use the image library to convert the imported image as a pure RGB file without the alpha channel. From there the conversion from RGB to YUV was fairly straight forward, though can be computationally expensive for larger images. We just made a hardcoded YUV2RGB matrix based off the YUV conversion materials provided by Dr. Ze-Nian Li that was then matrix multiplied by every pixel RGB channel in our image to get our new YUV matrix with channels pertaining to Luminosity and Chrominance instead of RGB.

Chroma Subsampling: Chroma Subsampling by 4:2:0 consisted of splitting the image into 2x4 matrices for the whole image and each cell has 3 channels YUV. We take the U and V the channels and average them across the 4 pixels in segments of 2. The average that is calculated on the left side of the 2x4 matrix gets put into the left vertical dimension from the horizontal average and vice versa for the right hand side. The result is a matrix of 2x4 that averages the columns and puts them into the vertical dimension on the U and V channels. The Y channel is untouched.

Transform Coding: Transform coding was done with a hardcoded DCT matrix of size 8x8. We used the math library in order to use Cosine in our hardcoded matrix and to define pi and square root 2 in variable space. The DCT matrix transpose was calculated using the built in transpose function of numpy to reduce redundancy. For the blocks we used a double “for” loop that for each row and column we created a block by going from 0 to height and 0 to width of our image with a step size of 8. We calculated the starting and ending row index needed and likewise for the column indices. We iterated across the totality of our YUV image, after applying Chroma Subsampling, and for each iteration of the loop took the block specified above and did the DCT * Block * DCT^T matrix multiplication to convert to cosine frequency. This block was then inserted into the YUV matrix in its specific block index and then divided by the appropriate quantization matrix for that channel. We did this part separately for each channel of Y, U, and V because we found it was easier to manipulate the matrices by singular channels when using colon indexing and when dividing by our quantization matrices which differ between channels Y and U,V.

Quantization: This was done inside our double "for" loop mentioned in the DCT section of this report. For each DCT Transformed block in our YUV image we divided the transform frequency by our Scaled Luminance matrix for our Y channel and our Scaled Chrominance Matrix for our U and V channels after this division we rounded to the nearest integer. Rounding to the nearest integer plays a big role in loss of information and when dealing with numpy it kept everything in a float data type so we specifically had to cast the block to integer for rounding after division by our quantization matrices. For more information how the quantization matrices were scaled please see the next section. For our initial Luminance and Chrominance matrices we used the ones provided in the textbook that was shown by Dr. Ze-Nian Li, that are also used by the JPEG standard.

Quality Factor: Using the algorithm found in many forums and research papers for Quality Factor scaling of Quantization tables we scaled every cell of our 8x8 quantization tables (Chrominance and Luminance) by the pseudo-code if Quality Factor is less than 50 a Scale Factor = 5000/Quality Factor else Scale Factor = 200 - (Quality Factor*2). This scalar factor was then multiplied by each value in the matrix + 50 and divided by 100 then the floor was taken [2] [3]. This gave us our scaled quantization tables of integers and the rule of thumb found online was that for the Quality Factor of 50 the Quantization tables didn't change from the jpeg standard that was already hardcoded into our program which is true with this algorithm we implemented.

JPEG Decoding

Decoding was straight forward using our blocks that were divided by our quantization we just reversed the process and used the numpy multiplication function to get the DCT cosine frequencies back with loss of information due to the integer rounding. The block was then passed through our DCT matrices but in reverse order (i.e. DCT^T * Block * DCT) to get our original YUV image back but with compression and loss of information. After converting back to RGB for every pixel using the matrix given to us by Dr. Ze-Nian Li for YUV2RGB conversion we cast the image to the 8-bit unsigned integer data type and our image is displayed with its full lossy encoding based on Quality Factor.

Note: Unlike the DCT Transform, the multiplication by the scaled quantization matrices is not matrix multiplication it is something different. When trying to use matrix multiplication denoted by @ in python3 we had really skewed images in luminance and chrominance so we instead used the built in numpy multiplication function as we had previously used the built in numpy division function to encode the DCT quantization ratios. This got us the proper compression on the images in our YUV channels.

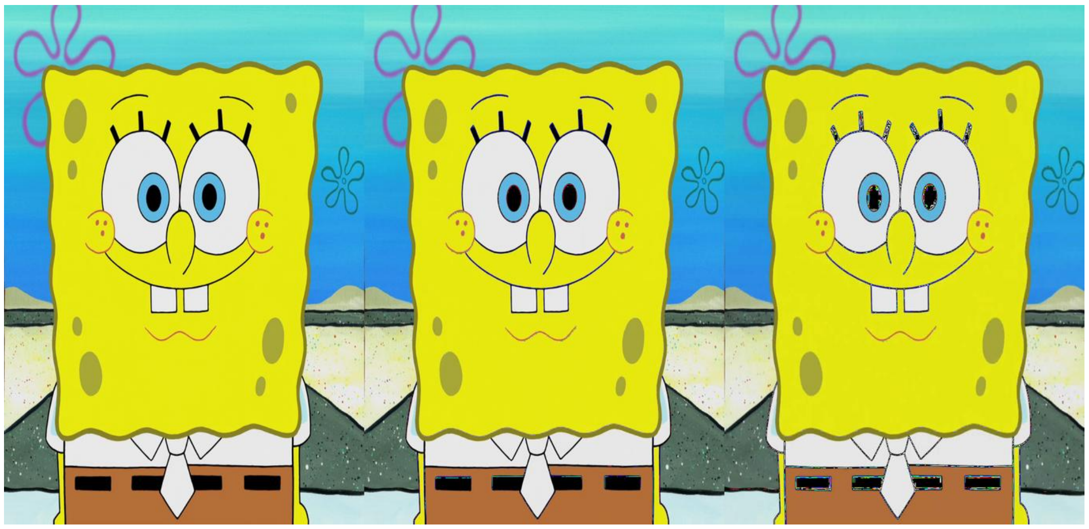
GUI

The GUI used arguments passed in with the python args parser with –qf flag being the quality factor integer and –F being the file path of the picture in the directory. We then used the Tkinter library to make a simple GUI interface that will open a file browser at your home root and you can then browse for a png or jpg file type, we only included those two for simplicity sake of the GUI and because they are the most common image file type we used. After you select your file and save a quality factor you then hit the ready and compress and you will receive an image output like the examples seen below. This is done by calling the JPEG encoder which does all the steps listed above and outputs the different images calculated in our JPEG encoder (Original, Chroma Subsample, JPEG compressed).

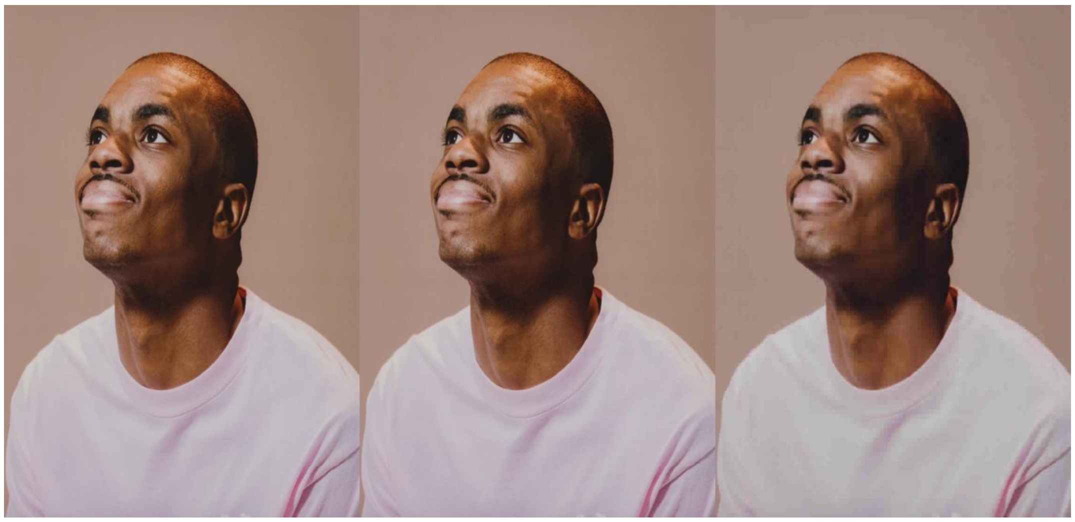
Examples of Chroma Subsampling and JPEG compression in action:

For all pictures, the original image is on the left, the chroma subsample image in the middle, and the compressed JPEG image on the right.

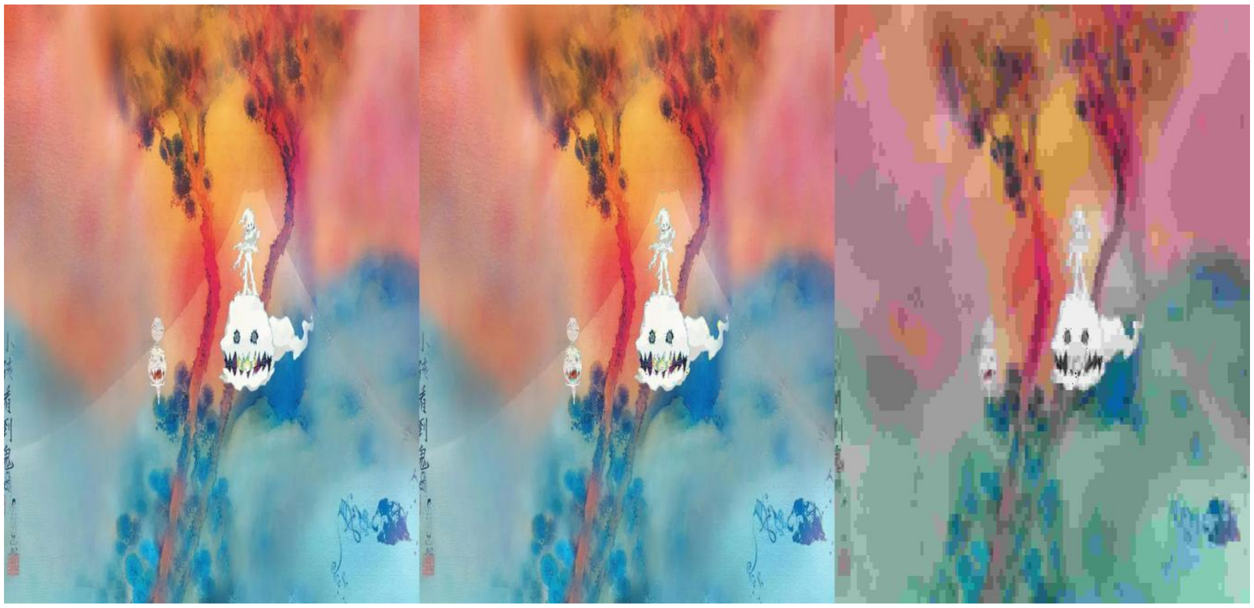
This image is using a Quality Factor of 50 so unchanged Quantization tables from the ones given we can see there is a slight loss of color in the compressed image and around the areas were we have really drastic changes in RGB values (eyes, belt) we have color conflictions and pixilation based on the average in the area. This is because of how JPEG compression and transform/quantization rounding works and is also due to chroma subsampling as we can already start to see the effects in the middle image.



This image is using a Quality Factor of 25 and we can see in the compressed image the lights are not as defined and there seems to be a slight blur to the photo but because the colors in RGB don’t jump drastically from 0-255 the changes even at a QF of 25 are not immediately noticeable like the previous example with Spongebob the image looks blurred and smoothed but no major details are lost on the subject.



The last example is uses a Quality Factor of 5 to really show the compression on the image and we can see immediately there are noticeable blocks in the image and averaged colours across large portions of the image because of the JPEG compression and transform/quantization rounding on the different block sections.



References

[1] Getsanjeev, “getsanjeev/compression-DCT,” GitHub. [Online]. Available: https://github.com/getsanjeev/compression-DCT/blob/master/image2RLE.py. [Accessed: 23-Feb-2019].

[2] “JPEG Compression Quality from Quantization Tables,” ImpulseAdventure - JPEG Compression and JPEG Quality. [Online]. Available: https://www.impulseadventure.com/photo/jpeg-quantization.html. [Accessed: 25-Feb-2019].

[3] Cogranne, Remi. “Determining JPEG Image Standard Quality Factor from the Quantization Tables.” Troyes University of Technology, Lab. for System Modelling and Dependability. [Online]. Available: https://arxiv.org/pdf/1802.00992.pdf. [Accessed: 25-Feb-2019].