# Lineariy 1: Final Report

Anna Griffin and Enmo Ren

May 2018

## 1 Introduction

JPEG is a widely used type of image compression that reduces the size of an image to save space when storing. It was first approved in the early 1990s and has remained a popular choice for photography, website designing, and publication. This type of compression is capable of a significant reduction of data storage making it extremely powerful. This process is crucial when working with large images due to the amount of storage they require, but it comes at a cost. It turns out a significant portion of the data an original image contains can be disregarded without the appearance of the final image deviating much from the original one. However, the image undergoes discrete cosine transform which is a type of lossy compression. This means some of the original image data is compromised and can not be retrieved in the decompression process. There is a trade off between storage space and image quality. When balanced correctly and efficiently, it is difficult for the human eye to detect changes between the image file and compressed version. This method works because the human eye is only sensitive to a portion of the information in the original image. By purposefully removing the less essential data, a smaller image file can be achieved without altering the appearance of the compressed image.

## 2 Color Spaces and Transformations

A digital image is comprised of tiny blocks of color called pixels. Each pixel is represented in the RGB colorspace with a red component, green component, and blue component. Each component has a different value within the range of 0 and 255 which determines the levels of red, blue, and green in each pixel. The RGB colorspace is used because combinations of varying amounts of just these three colors are able to produce every color in the spectrum. The downside however, is this colorspace requires a lot of space. Every pixel has three components, each of which takes up three bytes of storage. In the JPEG image compression process, images are converted to YCbCr colorspace. The YCbCr colorspace is comprised of three components as well, luminance (Y), blue chrominance (Cb) and red chromiance (Cr). Luminance describes the amount of light that gets emitted from the pixel, which is also thought of as the brightness. The chrominaces, on the other hand, hold information about the color. While there are three components to for every pixel in this colorspace as well, this representation of the information works out to be more advantageous due to biology.

The human eye is able to detect changes in the luminance more clearly than variation in the chrominance. Converting to this colorspace works to our advantage because it enables image compression to be able to keep most of the luminance while discarding a significant amount of information related to chrominance. We are able to fool our eyes in a sense because image compression can be done without us being able to tell a difference in the compressed version.

## 3 Image Splitting

After transforming from RGB colorspace to the YCbCr colorspace, the image gets split up into 8 x 8 pixel blocks. The image is divided in this way because the amount of variation between the 64 pixels in each block is small relative to the variation in the whole image and a 8x 8 block is more computational efficient than larger block. Every pixel in the block can be split up into its Y, Cb, and Cr components. There is an observable difference in the variation between the Y component and the Cb and Cr components which show that our our eyes are more sensitive to differences in luminance. This allows us to be able to compress notably more of the Cb and Cr components without having a large effect on the end image. Since the variance is small, the average value is meaningful because the amount by which each pixel varies is practically negligible.

The example matrix below displays the original pixel values of a 8-by-8 block in the input image after transforming it to YCbCr colorspace.

$$\begin{bmatrix} 182 & 188 & 198 & 206 & 214 & 218 & 215 & 210 \\ 189 & 198 & 209 & 215 & 216 & 215 & 213 & 210 \\ 202 & 209 & 216 & 218 & 218 & 214 & 210 & 207 \\ 212 & 213 & 213 & 215 & 215 & 213 & 204 & 196 \\ 213 & 214 & 213 & 212 & 208 & 202 & 191 & 182 \\ 208 & 212 & 213 & 206 & 195 & 184 & 177 & 171 \\ 205 & 207 & 204 & 192 & 179 & 173 & 169 & 171 \\ 205 & 200 & 187 & 175 & 168 & 167 & 171 & 173 \end{bmatrix}$$

## 4 Source Encoder

The main concept of image compression is to reduce the amount of data required to represent the digital image while still managing to preserve the appearance of the original image. JPEG compression achieves this by removing redundant information and building correlation between different color planes. During the process, the input signal is transformed in a way that sparse data and is compacted into a smaller number of coefficients. This results in the need for less storage for the compressed image. The specific linear transform that is applied to in the JPEG compression method is the Discrete Cosine Transform (DCT-II).

The essence of DCT is expressing a sequence of data points in terms of cosine functions that oscillate at different frequencies by taking the linear combination of frequencies. Each frequency's contribution to original pixel value in the matrix is calculated. In short, DCT transfers image in spatial domain to frequency domain.

For image compression, the reason that this specific linear transformation is applied is each pixel in image depends on its previous pixel and the periodicity of frequencies in DCT is applicable in this 8 x 8 block. The values of D(i,j) are called the DCT coefficients of corresponding pixel.

The DCT equation calculates the i,jth entry of the DCT of an image. p(x,y) is the (x,y) element of the block matrix and N is the size of the block ( 8 in an 8x8 matrix ).

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

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

For better computation efficiency, image compression applies DCT transform by using matrix multiplication. For a two-dimensional image matrix, a symmetric transformation matrix $T$ can be pre-computed based on the dct equation, since any mxm matrix A can be written as a sum of i,j functions of the equation (1).

$$\begin{aligned} T(i,j) &= \frac{1}{\sqrt{N}} && \text{if i = 0} \\ T(i,j) &= \sqrt{\frac{2}{N}} cos\left[\frac{(2x+1)i\pi}{2N}\right] && \text{if i > 0} \end{aligned} \tag{3}$$

For an mxm matrix A, $TA$ is an mxm matrix whose columns contain the one-dimensional DCT of the columns of A. The two-dimensional DCT of A can be computed as TA. Since T is a orthonormal matrix, its inverse is the same as its transpose. Therefore, the inverse two-dimensional DCT of D is given by $T'BT$.

$$\mathrm{T} = \begin{bmatrix} \sqrt{\frac{1}{8}} & \sqrt{\frac{1}{8}} & \sqrt{\frac{1}{8}} & \sqrt{\frac{1}{8}} & \sqrt{\frac{1}{8}} & \sqrt{\frac{1}{8}} & \sqrt{\frac{1}{8}} & \sqrt{\frac{1}{8}} \\ \sqrt{\frac{2}{8}}cos\left(\frac{\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{3\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{5\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{7\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{9\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{12\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{15\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{18\pi}{16}\right) \\ \sqrt{\frac{2}{8}}cos\left(\frac{2\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{6\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{10\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{14\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{18\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{22\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{26\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{30\pi}{16}\right) \\ \sqrt{\frac{2}{8}}cos\left(\frac{3\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{6\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{10\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{14\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{18\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{22\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{26\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{30\pi}{16}\right) \\ \sqrt{\frac{2}{8}}cos\left(\frac{4\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{12\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{20\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{28\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{36\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{44\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{52\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{60\pi}{16}\right) \\ \sqrt{\frac{2}{8}}cos\left(\frac{5\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{15\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{25\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{35\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{45\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{55\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{65\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{75\pi}{16}\right) \\ \sqrt{\frac{2}{8}}cos\left(\frac{6\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{18\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{30\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{42\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{54\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{66\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{78\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{90\pi}{16}\right) \\ \sqrt{\frac{2}{8}}cos\left(\frac{7\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{21\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{35\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{49\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{63\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{77\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{91\pi}{16}\right) & \sqrt{\frac{2}{8}}cos\left(\frac{105\pi}{16}\right) \end{bmatrix} \tag{4}$$

Before applying DCT to each 8x8 block, the coefficients have to be shifted. Since DCT uses cosine which is centred around zero and ranges from 1 to -1, the pixel values must be shifted to center around zero. Originally, they range from 0 to 255, but this transformation is done by subtracting the midpoint of range (128) from each entry in the block. The result is a shifted matrix with pixel values now in the range of [-128,127].

$$\begin{bmatrix} 54 & 60 & 70 & 78 & 86 & 90 & 87 & 82 \\ 61 & 70 & 81 & 87 & 88 & 87 & 85 & 82 \\ 74 & 81 & 88 & 90 & 90 & 86 & 82 & 79 \\ 84 & 85 & 85 & 87 & 87 & 85 & 76 & 68 \\ 85 & 86 & 85 & 85 & 80 & 74 & 63 & 54 \\ 80 & 84 & 85 & 78 & 67 & 56 & 49 & 43 \\ 77 & 79 & 76 & 64 & 51 & 45 & 41 & 43 \\ 77 & 72 & 59 & 47 & 40 & 39 & 43 & 45 \end{bmatrix}$$

Once we have our shifted matrix, we can use the DCT equation by multiplying this DCT matrix to find the coefficients of each frequency. These coefficients tell the weight of each frequency required to reconstruct this block. In DCT matrix, the block in the top left refers to the direct current (DC) coefficient because it is a constant value whereas the all of the other ones are referred to as alternating current (AC) coefficients corresponding to progressively increasing frequencies. The basis function exhibits a progressive increase in frequency both in vertical and horizontal direction. Generally, the coefficients in the top left are lower than the ones in the bottom right which means that the higher frequencies don't have as much importance. Due to this disparity between coefficients of high and low frequencies, we can afford to remove frequencies with lower coefficients, typically the higher frequencies, without there being a significant change from the

original image.

$$\begin{bmatrix} 578 & 38 & -22 & -3 & -5 & -11 & -0 & 0 \\ 71 & -71 & -26 & 5 & 0 & 2 & 0 & 0 \\ -45 & -20 & 16 & -5 & 0 & 1 & 0 & 1 \\ -5 & 5 & 12 & 0 & -1 & 1 & 1 & 0 \\ -4 & -7 & 7 & 11 & 0 & 0 & 1 & -1 \\ -10 & 2 & 1 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\ 1 & 0 & 0 & -1 & 0 & -1 & 1 & 0 \end{bmatrix}$$

## 5 Quantization

After applying DCT to an 8 x 8 pixel block, the transformed image DCT matrix is divided by a quantization matrix. More specifically, each coefficient in the matrix is divided by the quantizing factor that corresponds to its position in the matrix. Different JPEG compressors use different quantization matrices depending on the desired compression quality. A standard quality levels is 50 which balances the tradeoff between efficient compression and image quality. To adjust the quality, the standard quantization matrix gets scaled. For a decompressed image with a higher quality, although at the cost of less compression, the scaling factor is determined by subtracting the desired quality level from 100 and then divide it by 50. On the other hand, if compression is more of a priority, then the scaling factor can be found by dividing 50 by the new quality level.

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

Since the human eye is not particularly sensitive to rapid variation in the image, deemphasizing the higher frequency by dividing with a larger quantizing factor which reduce it to zero will not significantly affect human's visualization of image. This allows the quantization to reduce the number of bits needed to store by reducing the precision of those values.

The quantization matrices are designed in a way to optimize the space complexity. Generally, the factors in the top left of the matrix, which correspond to the lower frequencies, are small and the factors on the bottom right, which affect the higher frequencies, are larger. This pattern results in a matrix with many zeros in the bottom right half and leaves a few nonzero integers in the top left corner after element-wise division. Because a handful of these coefficients are now zero, the frequencies corresponding to them are left out and only the frequencies with nonzero coefficients are of any importance for our compressed image. After disregarding the frequencies with zero coefficients, the image block can be represented by just these frequencies and still appear (relatively, depending on the level of compression) identical to the original image.

To decrease the computational complexity, the last step in this part of the process is sequencing the coefficients. Most efficiently, this is done by serializing all the value following a zigzag path starting with the top left corner, moving one to the right, going all the way back, diagonally, to the second coefficient on the leftmost side, down one, and backup to the next coefficient in the top row. This pattern is continued until the bottom right hand corner is reached.

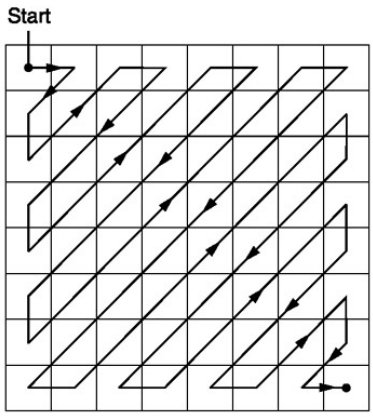


Figure 1: A zigzag path for matrix

The coefficients are lined up in this order which puts ones with nonzero values towards the front whereas the zeros is pushed to the end. Now they are strategically lined up to increase the efficiency of the encoding process.

## 6 Huffman Encoding

Next, Huffman Coding is applied to reduce storage space required for coefficients even further. Huffman coding is a type of compression that is classified as lossless which means it preserves all of the information. Once the long string of coefficients is obtained from the quantization process, they can be examined and ordered by their frequencies. After identifying the unique values and their corresponding frequencies, they can be ordered into a code tree. Starting at the bottom, the values with the two lowest frequency values can be joined by a two leaf node with the sum of their frequencies as the connector. Then, taking that frequency into consideration as well, the pair of the next lowest sum of frequencies is appended.

Once the tree is complete, the branches of the the tree get labeled with either a 1 or a 0. All of the branches on the left get a 0 and inversely, the ones on the right get a 1. From this, we are able to find the code for each value by following the branches of the tree accumulating the 1 and 0 branches of the path it takes. As we continue to build our tree in this way, it will be clear that the values with lower frequencies will be on the bottom and the higher ones will be on the top. This is useful because the code attached to the higher frequency values are able to be minimized with this type of compression. Then, we can reconstruct our sequence of values from the quantized matrix and replace the values with the codes obtained from the Huffman coding process. This new code that is significantly shrunken is then stored.

## 7 Decompression

The decompression phase is the reverse of compression process. The reconstruction of DCT coefficient matrix is done by adding the quantization matrix, which is stored in the file, back. After the DCT coefficient matrix is recomputed, the Inverse Discrete Cosine Transform will be applied in this coefficient matrix. The inverse DCT is doing DCT in opposite direction. Thus, the coefficient matrix is transformed back to spacial domain. After Inverse Discrete Cosine Transform is applied to our matrix, each element will be shifted back to its original range by adding its midpoint value

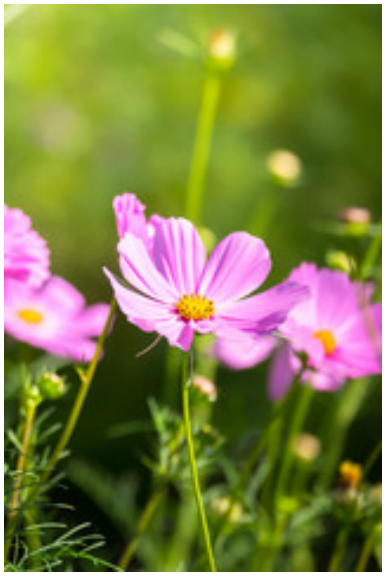

Figure 2: Original Image

Figure 3: Original Image Block

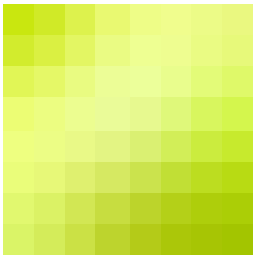

Figure 4: Decompressed Image Block-Quality 50

128 to each of them. Then, a decompressed 8x8 block of image is obtained. Once each decompressed block is put back to its original positions, a matrix, representing a decompressed JPEG digital image is generated and can be displayed by the JPEG file reader.

## Appendix

We wrote a DCT program to go along with our project. We programmed in python and used openCV2, PIL, numpy, and matplotlib. Our github repository https://github.com/annagriffin/JPEGcompression

```
def convert_RGB_to_YCbCr(filepath):
    """ Converts image from RGB colorspace to YCbCr colorspace

    filepath: string path to image file
    returns: image in YCC_Scale """

    img = cv2.imread(filepath)
    bgr_img = img[...,::-1]
    YCC_scale = cv2.cvtColor(bgr_img, cv2.COLOR_BGR2YCR_CB)
```

This function converts an image from the RGB colorspace to YCbCr colorspace

```
def partition(image):
    """ Separates image into 8x8 blocks

    image: image in YCbCr scale
    returns: list of 8x8 blocks """

    width = image.shape[1]
    height = image.shape[0]
    blocks = []
    image = np.array(image)
    delta_x = int(width/20)
    delta_y = int(height/30)
    index_x=0
    index_y=0
    for i in range(0,20):
        for j in range(0,30):
            temp_image = image[index_y:index_y+delta_y,index_x:index_x+delta_x]
            blocks.append(temp_image)
            index_y += delta_y
        index_y=0
        index_x += delta_x
    return blocks
```

This function splits the image up into 8x8 pixel blocks

```
def get_component_value(image, component):
""" Gets the Y component of an image
```

```
image: 8x8 block
returns: list of Y component values """

image = np.array(image,dtype = np.integer)
list_of_values = []
for row in range(len(image[0])):
    list_of_values1 = []
    for column in range(len(image[0])):
        pixel = image[row][column][component]
        list_of_values1.append(pixel)
    list_of_values.append(list_of_values1)
return list_of_values
```

This function isolates the components from a given pixel block.

```
def shift(matrix):
    """ Shifts one compoenent's values by 128

    component: matrix of just one component
    returns: matrix of shifted component """

    shifted = []
    for row in range(len(matrix)):
        shifted1 = []
        for column in range(len(matrix)):
            new = matrix[row][column] - 128
            shifted1.append(new)
        shifted.append(shifted1)

    return shifted
```

This function shifts the matrix.

```
def dct(matrix):

    transpose = get_transpose(dct_matrix)
    temp = np.matmul(dct_matrix, matrix)
    coefficient_matrix = np.matmul(temp, transpose)
    return coefficient_matrix
```

This function performs DCT and inverse DCT. It takes a matrix and goes through all of the steps inorder to compare how close the decompressed block is compared to the original one.

## References


[1] Frank Ong (presented by Jon Tamir): *JPEG DCT Demo*,

`https://inst.eecs.berkeley.edu/~ee123/sp16/Sections/JPEG_DCT_Demo.html`

`This source was helpful for the code portion of our project. We pulled information about ho`

[2] Ken Cabeen, Peter Gent *Image Compression and the Discrete Cosine Transform*,

`https://www.math.cuhk.edu.hk/~lmlui/dct.pdf`

This paper was useful for trying to understand what exactly happens to the pixels when they undergo DCT. We used the DCT equation and DCT matrix that the authors explained. Overall, it covered a pretty good overview of the process in a concise and easy to follow way.

[3] A.M.Raid, W.M.Khedr, M. A. El-dosuky and Wesam Ahmed *Jpeg Image Compression Using Discrete Cosine Transform - A Survey*,

`https://arxiv.org/pdf/1405.6147.pdf`

From this paper, we were able to get a better grasp on the basic principles of DCT. There was a brief introduction to spatial redundancy and variance, the biology of the human eye, and the fundamentals of various compression techniques.

[4] Gilbert Strang *The Discrete Cosine Transform*,

`http://www-math.mit.edu/~gs/papers/dct.pdf`

This paper helped us understand the differences between one dimensional DCT and two dimensional DCT.

[5] David Austin *Image Compression: Seeing What's Not There*,

`http://www.ams.org/publicoutreach/feature-column/fcarc-image-compression`

This essay gave a pretty clear overview of the image compression process and discussed the reasons for doing each step. There also was a lot of information about the color spaces and it described the reason why certain information can be disregarded by using side by side examples.

[6] *Discrete Cosine Transform and A Case Study on Image Compression*,

`http://bugra.github.io/work/notes/2014-07-12/`
`discre-fourier-cosine-transform-dft-dct-image-compression`

This case study provided a good starting place when writing our code. We didn't end up using it very much. They pointed us towards a library that does DCT, however did not find this to be very helpful since we had already started writing the code our own way. We ended up performing DCT ourselves, instead of relying on their library, but this paper was a good reference.

[7] Syed Ali Khayam *The Discrete COsine Transform(DCT): Theory and Application*,

`http://www.lokminglui.com/DCT_TR802.pdf`

This paper provided an overview of the process and why it is important. This just helped us generally when trying to get a better grasp on how the compression works.