Evaluating the Relevancy of JFIF Image Compression on the Internet

Uzen

Jul 22

`this is a DRAFT!!!!!!`

Contents

1 Introduction

Compression is the cornerstone of the internet. Behind the millions of bytes flowing between computers everyday around the world, compression algorithms are behind every single data packet. Compression algorithms are tools that can seemingly magically shrink a digital document, sometimes until they are a tenth of their size. Due the many uses of compression algorithms, they have been used in many different fields. From shrinking large backups for storage to speeding up the transfer of data across the web, compression algorithms are crucial to modern computing.

One specific field that I was interested in was website hosting. A personal website can be a good representation of themselves, similar to a more polished social media page. For a period of time, I was quite interested in starting my own website, renting virtual servers from Aliyun to start my project. However, I quickly ran into an issue. In the modern times, technology has progressed at an exponential rate, storage and bandwidth speeds are increasing at an extraordinary rate. It is common to see computers with hundreds of gigabytes of storage and gigabit connection to the internet. So it was quite a surprise for me when I found out that adding bandwidth and storage to my server costed money. A small, affordable server would only come with 10 to 50 gigabytes of storage a measly 1 megabit bandwidth connection (or purchase bandwidth by the gigabit used).

Due to this limitation, the website I made was slow. I couldn't upload many images or videos because the server would run out of storage, or make the website load at a snail's pace. This made me very careful about the files I upload, especially the sizes of image and videos. My curiosity eventually lead me to image compression. When an image is stored, there's different file types: `.png, .jpg, .gif`. These are actually different methods of storing images, using different methods image compression techniques. When creating my website, I had to choose what method of image compression to use. Each different method has its advantages and disadvantages.

One format of compression I am very interested in is JFIF compression (commonly found in JPEG files). JFIF uses a mixture of clever math and the characteristics of the human eye to shrink photos. It is commonly used in digital cameras. However, JFIF was originally invented over 30 years ago (August 1990 (Joint Photographic Experts Group)), and have not account for the many modern technological advances. For example, JFIF is bad at dealing with digital art and digital text due to limitation of the compression techniques. In the recent years, newer and more efficient image compression algorithms have been created, such as Webp. In fact, JPEG has released a newer version of JFIF, called JPEG-2000. Some has even used Artificial Intelligence to enhance compression rates (Johnston and Minnen).

Nevertheless, JFIF compression still remains as a dominant image compression technique. This paper aims to compare JFIF with modern image compression formats to evaluate whether it is still a competitive algorithm. The results of the paper would help website creators decide the best format to store images on their websites.

2 Background Information

2.1 Lossless and Lossy Compression

There are two major types of compression algorithms: lossy and lossless.

2.1.1 Lossless Compression

Lossless compression finds repetitive, similar patterns inside a file, condenses the data to the smallest possible size. One example of lossless compression is LZ77 (Brailsford). Commonly used on plain text documents[1], LZ77 finds repeated words, and stores it for reference. The next time the repeated word is used, LZ77 stores a pointer that points towards the last instance of word. In this paragraph, the word `compression` is used 5 times. LZ77 will only store the first `compression`, and save space from the rest.

Lossless compression is good for data with a lot of repetition. It also preserves the data it compresses, which lossy compression cannot. On the other hand, lossless compression is less efficient than lossy compression.

2.1.2 Lossy Compression

Lossy compression finds and remove redundant data. Lossy compression is used often in audio and video data. Due to its nature, lossy compression techniques varies greatly depending on the input data. While some lossless compression algorithms are generic, lossy compression algorithms cannot be interchange across different data types.

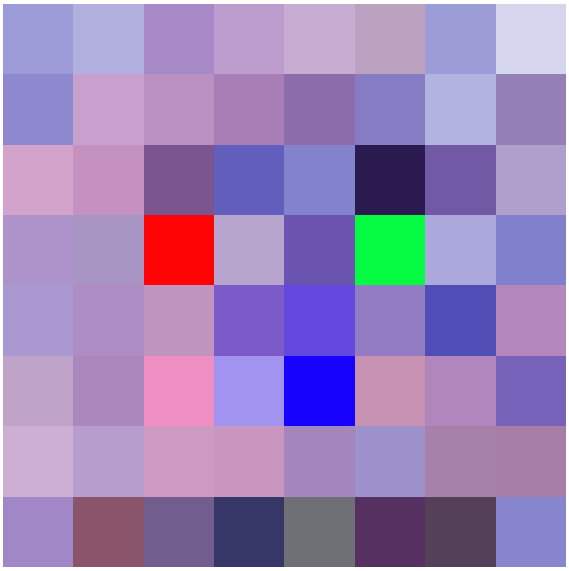
Lossy compression is more efficient than lossless compression. But it can degrade the quality of the data, as some data is thrown away. It is not suitable for all types of data. For example, we cannot use lossy compression on text, as every character matters in a sentence.

2.2 Storing Images

A pixel is a square block with a specific color. When a grid of pixels are placed together, it forms an image. An image is simply a 2D array of pixels. And each pixel can be stored as color. This method of storing images is called `bitmap`.

Color is generally represented in a colorspace[2]. One commonly used colorspace is `RGB`[3], which stores the mixture of red, green, and blueness of a color. Imagine that a color is composed by shining red, green, blue light on a black wall. Where the number associated with each value is the brightness of the light. The resulting image would be stored as a 2D array, each element containing a RGB value.

[1] `.txt` files
[2] method of storing color as data
[3] find rgb iso documents



(a) `art.png`, An 8 by 8 image made out of pixels

$$\begin{bmatrix} 156 & 177 & 167 & 188 & 200 & 188 & 156 & 214 \\ 142 & 200 & 186 & 169 & 139 & 133 & 179 & 149 \\ 212 & 196 & 122 & 97 & 131 & 42 & 113 & 174 \\ 174 & 169 & 253 & 183 & 106 & 4 & 171 & 129 \\ 170 & 174 & 191 & 124 & 101 & 148 & 80 & 180 \\ 191 & 170 & 238 & 161 & 23 & 199 & 176 & 119 \\ 205 & 183 & 206 & 200 & 164 & 157 & 166 & 167 \\ 162 & 138 & 113 & 54 & 111 & 86 & 84 & 134 \end{bmatrix}$$

(b) R Channel Data

$$\begin{bmatrix} 155 & 176 & 137 & 155 & 172 & 161 & 156 & 213 \\ 137 & 160 & 144 & 126 & 109 & 124 & 179 & 127 \\ 163 & 145 & 85 & 94 & 130 & 26 & 89 & 160 \\ 147 & 149 & 3 & 165 & 84 & 251 & 169 & 128 \\ 152 & 140 & 148 & 90 & 72 & 124 & 78 & 136 \\ 163 & 135 & 142 & 147 & 2 & 145 & 134 & 98 \\ 175 & 156 & 154 & 150 & 133 & 145 & 130 & 126 \\ 135 & 85 & 94 & 54 & 111 & 48 & 65 & 133 \end{bmatrix}$$

(c) G Channel Data

$$\begin{bmatrix} 215 & 223 & 199 & 205 & 209 & 192 & 215 & 238 \\ 207 & 206 & 195 & 183 & 172 & 195 & 224 & 183 \\ 203 & 192 & 144 & 188 & 205 & 79 & 166 & 203 \\ 202 & 196 & 3 & 205 & 175 & 65 & 219 & 204 \\ 209 & 196 & 191 & 200 & 221 & 195 & 182 & 189 \\ 201 & 189 & 195 & 239 & 254 & 177 & 188 & 185 \\ 214 & 206 & 196 & 191 & 190 & 203 & 171 & 166 \\ 199 & 107 & 143 & 103 & 118 & 96 & 89 & 206 \end{bmatrix}$$

(d) B Channel Data

Figure 1: RGB Colorspace

An alternative method of storing images is as mathematical equations. Images created this way are called `vector images`. Vector images have widely different use cases then bitmap images. However, this image type is severely limited at depicting complicated images, and is only suitable for digital art. Thus, this paper will work with bitmap images only.

2.2.1 Need for Compression

Storing images as arrays is fast and convenient, however it also results in large files. For example, a 1920 by 1080 image would take up 5.8MiB of data.

$$(1920 \cdot 1080)\ \texttt{pixels} \cdot 3\ \texttt{bytes} = 5.8\texttt{MiB} \tag{1}$$

Although this seems small, considering that most webpages have only kilobytes of data, images occupy a large amount of space. Videos are stored as a series of pictures. Using the same metrics as above, a 10 minute video would take up 104GiB of space. Uploading this to a websites would be extremely impractical.

For example, downloading the website `https://www.apple.com/iphone-12/` took 75`MiB` of data. Out of the 75`MiB` of data, 71.3`MiB` of data was from images. The rest of the website

(including fonts, html, javascript) only occupies 5.1MiB. Over 93% of the website is from images only.

2.3 Image Formats

Modern computer systems compresses images by default. Different images files have been developed with different algorithms and uses in mind. Below is a table of common image file types.

File Type	Full Name	Extension	C. Method	C. Strength
JPEG	Joint Photographic Expert Group image	.jpg	lossy	medium
PNG	Portable Network Graphics	.png	lossless	medium
GIF	Graphics Interchange Format	.gif	lossless	low
AVIF	AV1 Image File Format	.avif	both	high
WebP	Web Picture format	.webp	both	high
TIFF	Tagged Image File	.tiff	lossless	low
BMP	Bitmap File	.bmp	none	none

(a) Table of common image file types

File Type	Uses Case	Availability	Cons
JPEG	still images	high	can cause compression artifacts
PNG	high quality images	high	worse compression than lossless compression
GIF	animated graphics	high	low performance
AVIF	all	medium	basic support on some web browsers
WebP	all	high	slightly worse compression than AVIF
TIFF	high quality images	medium	support on some browsers only
BMP	uncompressed images	low	no compression, not suitable for web

(b) Table of image compression

2.4 Websites

Since this paper focuses on the use of compression on the internet, this subsection will briefly introduce the concepts of websites. Websites are hosted on rented cloud servers, with limitations on bandwidth, storage, processing power, and memory. Images are rendered client side by web browser, thus processing power and memory is irrelevant. Images require storage on the cloud server, large images may reduce the size of the website. When serving a website, bandwidth may limit the speed of transfer. Large images can increase loading times. Web developers have to balance image quality with image size when creating a website.

Due to the many available web browsers, the standards for web development varies. Web browsers support different image formats. The most common web browser is Google Chrome[4], whilst many web browsers are based on Chrome. When displaying an image, web browsers have to decompress, taking up processing power. Although processing power is abundant on modern

[4]requires citation

desktops, mobile devices and older computers may suffer. Web developers should be aware of the requirements of different image formats.

2.5 Code and Python

Image compression technology is old. Considering that it is an integral part of the internet, programmers have made it increasingly easy to perform image compression. Python will be used to write perform pseudo-JFIF compression used in this paper. This is because of the abundant amount of libraries available that simplify this process greatly. Although writing the program from scratch can give a greater understanding in the math that goes behind each step of compression, many of the math used are beyond the scope of this paper. Plus, using libraries help improve the speed and readability of the code. Any libraries used will be notated in the code, by the `import` statements at the start of the code.

3 JPEG Methodology

JFIF compression take advantages of two assumptions: (Townsend)

1. Humans are more sensitive to brightness than color.
2. Humans are not sensitive to high frequency contents.

The first step in the algorithm is to transform the colorspace from `RGB` to `YCbCr`. `Chroma Subsampling` (3.1) is then performed on the image. The image then undergoes `2D Discrete Cosine Transform` (3.2), and is `quantized` (3.3). Lastly, the resulting data matrix are compressed with `Huffman Encoding` (3.5).

This section will only detail the steps to compression. As the steps to decompressing is simply the reverse of the compression process.

3.1 Chroma Subsampling

3.1.1 YCbCr

Since the human eye is less sensitive to color than brightness, JFIF can downsample[5] the color element of an image. The commonly used `RGB` colorspace is convenient for projecting images, but does not separate the color from the luminance[6] of an image, `YCbCr` is used instead. YCbCr separates color into `Y` (luminance, brightness of a pixel), `Cb` (chroma blue, blueness), and `Cr` (chroma red, redness). Chroma green is not stored, as it can be calculated from Cb and Cr (learn media tech). The resulted image will be a matrix of pixels with color data stored in YCbCR.

The equation to convert `RGB` to `YCbCr` is:

$$Y' = 16 + (65.481 \cdot R' + 128.553 \cdot G' + 24.966 \cdot B') \tag{2}$$
$$C_B = 128 + (-37.797 \cdot R' + 74.203 \cdot G' + 112.0 \cdot B') \tag{3}$$
$$C_R = 128 + (112.0 \cdot R' - 93.786 \cdot G' - 18.214 \cdot B') \tag{4}$$

An code snippet in python for said conversion is listed below:

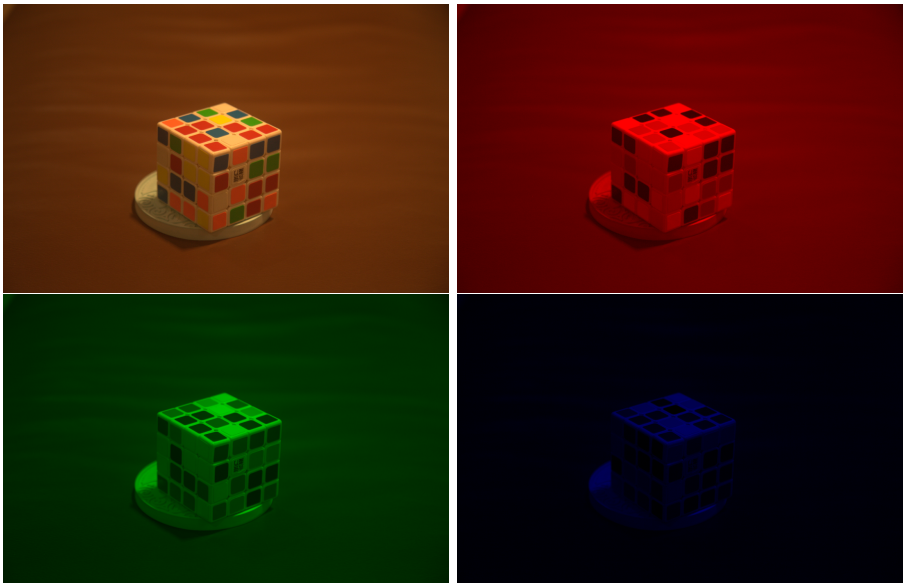
```
# Importing python packages
import numpy as np
import cv2

# Importing Image
img = cv2.imread('./Photos/cube.bmp')
```

[5] reduce the size of an image
[6] brightness

```
# Convert to YCbCr
ycbcr_img = cv2.cvtColor(img, cv2.COLOR_BGR2YCR_CB)
Y, Cr, Cb = cv2.split(ycbcr_img)
```

`numpy` is an math library that provides more flexible arrays used for python. `cv2` is the `opencv` library used for image processing. The code imports `cube.bmp` as three arrays into `img`, holding the `RGB` values. `cv2.cvtColor` is then used to convert the RGB matrix into corresponding YCbCr matrix. `cv2.split` splits the image into separate color channels, used for later processing.

(a) Original Picture (b) R colorspace



(c) G colorspace (d) B colorspace

Figure 3: `RGB` colorspace

Figure 3 shows the `RGB` colorspace. Figure 4 shows the same picture converted to `YCbCr` colorspace.

3.1.2 Subsampling

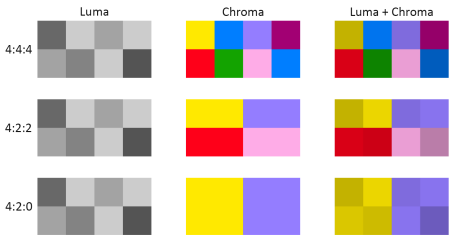
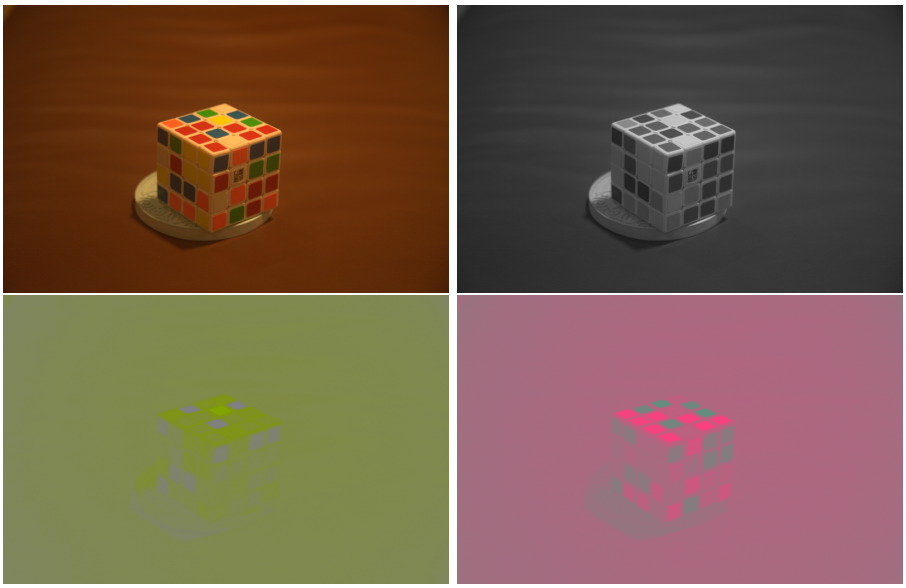


Figure 5: Chroma Subsampling illustrated

Humans are more sensitive to brightness that color. Since the brightness (luminance) is stored as Y in YCbCr, and color (chroma) stored as Cb and Cr, we process them separately. According to this statement, we can reduce the amount of details stored Cb and Cr, which would create minimal effects on the human eye.

We can perform this operation by only storing half

(a) Original Picture (b) Luminance



(c) Chroma Blue (d) Chroma Red

Figure 4: `YCbCr` colorspace

(or any other arbitrary number) the amount of pixels. When decompressing the image, the missing pixels would then be interpolated. This could be done by averaging out the nearest two available pixels, or simply referring to the pixel or the left.

There are several common schemes for discarding pixels: `4:4:4`, `4:2:2`, and `4:2:0` (refer to figure 5). Note, only chroma pixels are discarded, as discarding luminance pixels makes a noticeable difference. `4:4:4` discards no pixels, essentially no sub sampling. `4:2:2` discards every second column, discarding half the pixels. `4:2:0` discards every second column and every second row, discarding $\frac{3}{4}$ of the pixels. A less common `4:1:1` keeps only every fourth column, discarding $\frac{3}{4}$ of the pixels (but vertically).

The code used for chroma subsampling is listed below:

```
def comp422(inp):
    """4:2:2 chroma subsampling"""
    Y, Cr, Cb = cv2.split(inp)
    x, y = Y.shape
    comp_Cb = np.empty([x, y//2], dtype=np.uint8)
    comp_Cr = np.empty([x, y//2], dtype=np.uint8)

    for i in range(comp_Cb.shape[0]):
        for j in range(comp_Cb.shape[1]):
```

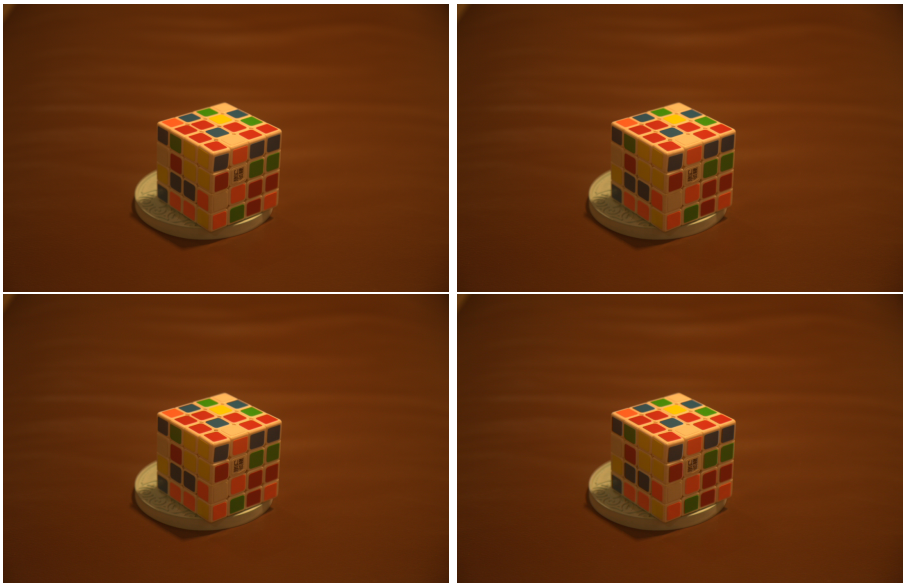
```
            comp_Cb[i][j] = Cb[i][j*2]

    for i in range(comp_Cr.shape[0]):
        for j in range(comp_Cr.shape[1]):
            comp_Cr[i][j] = Cr[i][j*2]

    return Y, comp_Cb, comp_Cr


def comp411(inp):
    """4:1:1 chroma subsampling"""
    Y, Cr, Cb = cv2.split(inp)
    x, y = Y.shape
    comp_Cb = np.empty([x, y//4], dtype=np.uint8)
    comp_Cr = np.empty([x, y//4], dtype=np.uint8)

    for i in range(comp_Cb.shape[0]):
        for j in range(comp_Cb.shape[1]):
            comp_Cb[i][j] = Cb[i][j*4]

    for i in range(comp_Cr.shape[0]):
        for j in range(comp_Cr.shape[1]):
            comp_Cr[i][j] = Cr[i][j*4]

    return Y, comp_Cb, comp_Cr


def comp420(inp):
    """4:2:0 chroma subsampling"""
    Y, Cr, Cb = cv2.split(inp)
    x, y = Y.shape
    comp_Cb = np.empty([x//2, y//2], dtype=np.uint8)
    comp_Cr = np.empty([x//2, y//2], dtype=np.uint8)

    for i in range(comp_Cb.shape[0]):
        for j in range(comp_Cb.shape[1]):
            comp_Cb[i][j] = Cb[i*2][j*2]

    for i in range(comp_Cr.shape[0]):
        for j in range(comp_Cr.shape[1]):
            comp_Cr[i][j] = Cr[i*2][j*2]

    return Y, comp_Cb, comp_Cr
```

`np.empty` creates two empty arrays `comp_Cb` and `comp_Cr`. The array length depends on the type of chroma subsampling (`[x, y//2]`). `dtype=np.uint8` defines the array to store an image. The `for` loop then iterates through the original chroma arrays, and selectively[7] copies the pixels

[7] `[i*2][j*2]` in this case skip 1 pixel horizontally and vertically

over.

(a) `4:4:4` (b) `4:2:2`



(c) `4:2:0` (d) `4:1:1`

Figure 6: Different types of chroma subsampling

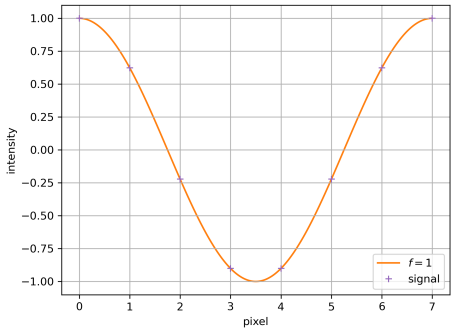
Figure 6 shows the end result of chroma subsampling. The even after discarding 75% of the color data (`4:2:0`), the human cannot easily perceive the difference in image quality. However, when zoomed in, the details are somewhat noticeable. Especially on sharp edges and quick changes in color.

3.2 Discrete Cosine Transform

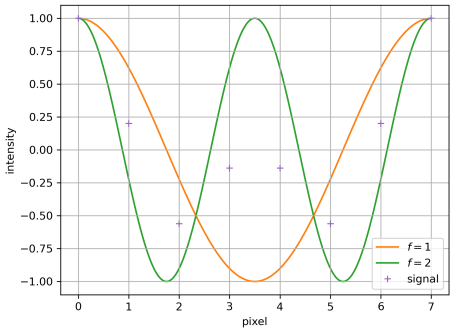
Discrete Cosine Transform (DCT) separates an image into frequency components. DCT is a subset of Fourier Transformation (Pound, "JPEG DCT, Discrete Cosine Transform (JPEG Pt2)- Computerphile"), which takes a time based function and transforms it into the frequency and amplitude components. This is performed by first splitting an image into small 8 by 8 pixel chunks, then performing the algorithm. Although the DCT used on imaged are 2 dimensions, we can explain the concept from a 1 dimension perspective first.

Lets take a single row of 8 pixels, and plot them on a graph with pixels on the x axis and intensity on the y axis. The resulted data points forms a signal. If we were to recreate the curve mathematically, we can plot a curve through the data points. DCT reconstructs the signal by stacking differently weighted cosine curves of different frequency and amplitude.

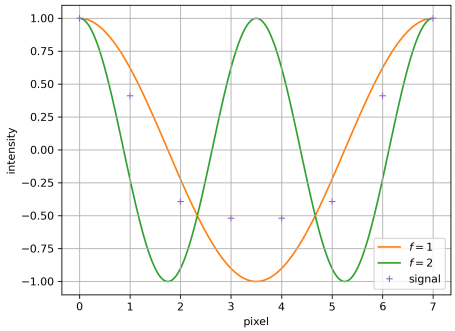
For example, figure 7a shows a cosine curve with a period of 8 pixels. Lets say this curve has a frequency of 1 unit. The purple coordinates represents the resultant signal.



(a) $f = 1$



(b) $f = 1 + f = 2$



(c) Weighted cosine curves

Figure 7: Cosine curves with varying frequency

If we add an additional cosine curve with a frequency of 2 units, the signal will be changed. In this case, the signal is the average of both curves.

If we change to weighting of the average, we can change the signal. In this case, the cosine curve with a frequency of 1 unit is weighed 3 times as much as the other cosine curve.

DCT[8] propose that we can represent any signal by adding 8 weighted cosine curves of increasing frequency. The resulted signal is modeled by this mathematical equation. The exact mathematics of DCT is too advance for the scope of this paper, implementation of DCT will be done through code.

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)\right] \qquad k = 0, \ldots, N-1 \tag{5}$$

Lets use the first row of `art.png` as an example. Below is the luminance data from the first row from `art.png`.

$$\begin{bmatrix} 162 & 182 & 153 & 171 & 185 & 173 & 163 & 216 \end{bmatrix}$$

[8]More specifically DCT-II, used by JIFF compression

Since a cosine curve moves from 1 to -1, we need to normalize the data around the midpoint, 128[9]. We minus 128 for every value.

$$\begin{bmatrix}34 & 54 & 25 & 43 & 57 & 45 & 35 & 88\end{bmatrix}$$

Perform 1D DCT.

$$\begin{bmatrix}2.658 & 2.534 & 2.236 & 2.204 & 1.892 & 2.050 & 2.176 & 2.432\end{bmatrix}$$

The example showed 1 dimension DCT, 2 dimension DCT is used on photos. Similar to decomposing a signal into many stacking cosine waves, 2D DCT decompose an 8 by 8 image into a stack of 8 by 8 grids. These grids are formed by 2D cosine waves, each with increasing frequency. 2D DCT proposes that an 8 by 8 image can be taken apart into 64 different cosine grids. The results generated by this operation is called the DCT coefficients.

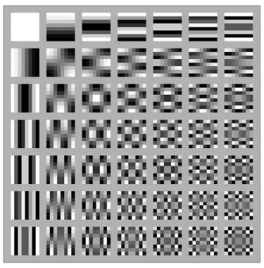


Figure 8: DCT matrix

Continuing to use the previous example, this time we will incorporate the full picture. Below is the luminance data from `art.png`.

$$\begin{bmatrix}
162 & 182 & 153 & 171 & 185 & 173 & 163 & 216 \\
146 & 177 & 162 & 145 & 125 & 135 & 184 & 140 \\
182 & 166 & 103 & 106 & 139 & 37 & 105 & 169 \\
161 & 160 & 78 & 175 & 101 & 156 & 175 & 137 \\
164 & 157 & 166 & 113 & 98 & 139 & 90 & 155 \\
176 & 152 & 177 & 162 & 37 & 165 & 153 & 114 \\
188 & 170 & 174 & 170 & 149 & 155 & 145 & 143 \\
150 & 103 & 105 & 60 & 112 & 65 & 73 & 142
\end{bmatrix}$$

Normalize around 128.

$$\begin{bmatrix}
34 & 54 & 25 & 43 & 57 & 45 & 35 & 88 \\
18 & 49 & 34 & 17 & 253 & 7 & 56 & 12 \\
54 & 38 & 231 & 234 & 11 & 165 & 233 & 41 \\
33 & 32 & 206 & 47 & 229 & 28 & 47 & 9 \\
36 & 29 & 38 & 241 & 226 & 11 & 218 & 27 \\
48 & 24 & 49 & 34 & 165 & 37 & 25 & 242 \\
60 & 42 & 46 & 42 & 21 & 27 & 17 & 15 \\
22 & 231 & 233 & 188 & 240 & 193 & 201 & 14
\end{bmatrix}$$

Perform 2D DCT.

$$\begin{bmatrix}
762.00 & -102.01 & 55.19 & -42.42 & 89.10 & -128.36 & -18.27 & -47.98 \\
892.00 & -61.95 & -394.48 & 221.98 & 217.79 & -361.52 & -301.94 & 517.95 \\
2014.00 & -138.43 & -372.83 & -279.54 & -462.45 & 793.54 & 116.16 & -105.93 \\
1262.00 & 148.90 & -551.01 & -101.17 & 7.07 & -177.11 & 107.31 & 679.04 \\
1652.00 & -260.79 & -594.95 & 19.08 & 330.93 & 416.22 & -675.06 & 228.99 \\
1248.00 & -419.99 & 139.83 & -200.20 & 500.63 & -426.76 & 138.02 & 202.34 \\
540.00 & 159.15 & 11.46 & 4.47 & 8.49 & 43.30 & 35.05 & -19.82 \\
2644.00 & 89.74 & -719.73 & -19.09 & -557.20 & -120.82 & -311.11 & 138.31
\end{bmatrix}$$

[9] $255 \div 2$, 255 is the maximum possible intensity

The code used for 2D dct is listed below:

```
# Importing python packages
import numpy as np
from scipy import fftpack

# Import from new image
Y = data['Y']
Cb = data['Cb']
Cr = data['Cr']

# split into 8x8 blocks
Yy, Yx = Y.shape
Yblocks = np.zeros([Yx * Yy // 64, 8, 8], dtype=int)

for i in range(Yy):
    for j in range(Yx):
        Yblocks[i//8*Yx//8 + j//8][(i) % 8][(j) % 8] = Y[i][j]

Cy, Cx = Cr.shape
Crblocks = np.zeros([Cx * Cy // 64, 8, 8], dtype=int)
Cbblocks = np.zeros([Cx * Cy // 64, 8, 8], dtype=int)

for i in range(Cy):
    for j in range(Cx):
        Crblocks[i//8*Cx//8 + j//8][(i) % 8][(j) % 8] = Cr[i][j]
        Cbblocks[i//8*Cx//8 + j//8][(i) % 8][(j) % 8] = Cb[i][j]

# normalize to zero (-128)
Yblocks -= 128
Crblocks -= 128
Cbblocks -= 128

# DCT
for i in range(Yx*Yy//64):
    dctBlocks = fftpack.dct(Yblocks[i], axis=1)

for i in range(Cx*Cy//64):
    CrBlocks = fftpack.dct(Crblocks[i], axis=1)
    CbBlocks = fftpack.dct(Cbblocks[i], axis=1)
```

The code takes first imports the data from the previous chroma subsampling. It then splits the picture into 8 by 8 chunks. Then, it normalizes the picture around 128. Lastly, it performs DCT on each block.

3.3 Quantization

The beauty of DCT lies in that most of the image can be recreated by the lower frequency cosine blocks ("JPEG 'files' & Colour (JPEG Pt1)- Computerphile"). This is exploiting the second flaw in the human eye, reduced ability to see high frequency content. Through this logical, we can theoretically create an importance value for each cosine grid (from DCT). Using this importance value, we can throw away the less important data, while keeping the more important data. This process is called quantization.

$$\begin{bmatrix} 8 & 6 & 6 & 8 & 12 & 14 & 16 & 17 \\ 6 & 6 & 6 & 8 & 10 & 13 & 12 & 15 \\ 6 & 6 & 7 & 8 & 13 & 14 & 18 & 24 \\ 8 & 8 & 8 & 14 & 13 & 19 & 24 & 35 \\ 12 & 10 & 13 & 13 & 20 & 26 & 34 & 39 \\ 14 & 13 & 14 & 19 & 26 & 34 & 39 & 39 \\ 16 & 12 & 18 & 24 & 34 & 39 & 39 & 39 \\ 17 & 15 & 24 & 35 & 39 & 39 & 39 & 39 \end{bmatrix} \tag{6}$$

Figure 9: Luminance quantization table from Photoshop Save To Web 050 (Hass)

This is done mathematically by dividing the DCT coefficients by a quantization table. Each element in the DCT coefficients is divided by their responding quantization table, the resulted value rounded and stored as the quantized data matrix. The process of rounding quantized data is removing information depending on the importance of each frequency. Notice how in figure **??**, the element in the quantization table increases as we move to the bottom right corner. The higher the quantization factor, the lower the importance of that specific frequency block.

We can imagine the rounding process as a process of finding factors. By dividing the DCT coefficients with the quantization coefficients, we are forcefully factoring the DCT coefficients by the quantization coefficients. The smaller the quantization coefficient, the more accurate we can factor the DCT coefficients. By tuning the quantization coefficients, we can control how much data about a specific frequency to throw out.

The code used to perform this operation is listed below:

```
# Importing Python libraries
import numpy as np

# Importing Quantization Tables
ql = np.genfromtxt("./quantization table/ps010l.csv", delimiter=",", dtype=int)
qc = np.genfromtxt("./quantization table/ps010c.csv", delimiter=",", dtype=int)

# Quantization
qYblocks = np.rint(Yblocks/ql)
qCrblocks = np.rint(Crblocks/qc)
qCbblocks = np.rint(Cbblocks/qc)
```

`ql` and `qc` are the quantization tables imported (`np.genfromtxt`) for luminance and chroma.

Empty arrays (`qYblocks`, `qCrblocks`, `qCbblocks`) are created. Then, we quantize the value (`Yblocks/ql`), and store it in the empty array. The values are rounded (`np.rint`, round to nearest integer) instead of truncating, as it is more accurate (without increasing data size).

3.4 Run Length Encoding

After quantization, the resulted matrix is compressed into a string. Due to the increased amount of quantization the further south east down the matrix, the quantized matrix often have a lot of similar values towards the south east side. In order to take advantage of this repetition, a pointer will traverse the matrix in a zig zag manner, compacting it into a string.

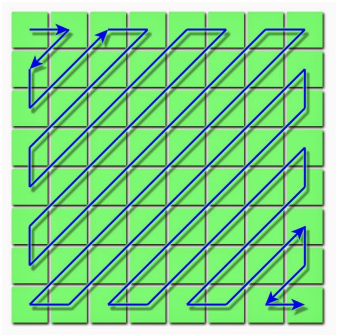


Figure 10: Path of the pointer (Programming Techniques)

The code used to perform zig zag manner is shown below. Due to the uniqueness of the operation, no libraries were used.

```
# Importing python packages
import numpy as np

qYblocks = data['qYblocks']
qCbblocks = data['qCbblocks']
qCrblocks = data['qCrblocks']

# Entropy Encoder
# This just works, very messy but works
def entropyData(array):

    # create empty array for output, move pointer to top left corner
    output = np.zeros(64)
    y = x = 0
    i = 0

    # Copy value of 0 0
    output[i] = array[y][x]
    i += 1

    # Move right by one, copy value of 1 0
    x += 1
    output[i] = array[y][x]
    i += 1

    # Repeat 3 times
    for j in range(3):

        # Move diagonally left downwards until hitting left wall of array
        while x != 0:
            x -= 1
            y += 1
```

```
            output[i] = array[y][x]
            i += 1

                # Move down by one
        y += 1
        output[i] = array[y][x]
        i += 1

                # Move diagonally right upwards until hitting top wall of array
        while y != 0:
            x += 1
            y -= 1
            output[i] = array[y][x]
            i += 1

                # Move right by one
        x += 1
        output[i] = array[y][x]
        i += 1

        # Complete to halfway point
    while x != 0:
        x -= 1
        y += 1
        output[i] = array[y][x]
        i += 1

    # Midpoint
    x += 1
    output[i] = array[y][x]
    i += 1

        # Complete other half
    for j in range(3):

                # Move diagonally right upwards until hitting top wall of array
        while x < 7:
            x += 1
            y -= 1
            output[i] = array[y][x]
            i += 1

                # Move down by one
        y += 1
        output[i] = array[y][x]
        i += 1

                # Move diagonally left downwards until hitting bottom wall of array
        while y < 7:
```

```
            x -= 1
            y += 1
            output[i] = array[y][x]
            i += 1

                # Move right by one
        x += 1
        output[i] = array[y][x]
        i += 1

        # Return 1D array output
    return output
```

The result of this transformation is a 64 integer long string. Most of the integers with be repeated. JFIF performs run length encoding to losslessly compress the string. It does this by representing each integer as two values, the `integer` and the `amount of time it is repeated`. For a string with many repetitions, run length encoding can reduce the space required to the data drastically.

3.5 Huffman Coding

To convert the string of data into binary, huffman coding is performed. Huffman coding furthers losslessly compresses the data by effectively assigning symbols to each element in the string.

4 Evaluation!!!!!!

4.1 Effectiveness/Benchmark

4.2 Speed

4.3 Artifacts

4.4 Modern Features

5 Conclusion

JFIF compression is an integral part of the internet in both its history and current time. Although old, JFIF compression still holds up when used on the correct types of images. Its compression is high, without losing out majorly on image quality. However, development of technology has also bring forwards more types of images, some unsuitable for JFIF compression. Image types like digital art and raw photos are better left for newer image compression methods.

Works Cited

Brailsford, David. "Elegant Compression in Text (The LZ 77 Method)". 1 Sept. 2013. <https://www.youtube.com/watch?v=goOa3DGezUA>.

Hass, Calvin. "JPEG Compression Quality from Quantization Tables". 2018. <https://www.impulseadventure.com/photo/jpeg-quantization.html>.

Johnston, Nick and David Minnen. "Image Compression with Neural Networks". *Google AI Blog* (29 Sept. 2016). <https://ai.googleblog.com/2016/09/image-compression-with-neural-networks.html>.

Joint Photographic Experts Group. "Overview of JPEG 1". 27 July 2021. <https://jpeg.org/jpeg/>.

learn media tech. "Chroma Subsampling – 4:4:4 vs 4:2:2 vs 4:2:0". *Learn Media Tech* (2020). <https://learnmediatech.com/chroma-subsampling-444-vs-422-vs-420/>.

Pound, Mike. "JPEG 'files' & Colour (JPEG Pt1)- Computerphile". 22 Apr. 2015. <https://www.youtube.com/watch?v=n_uNPbdenRs>.

——."JPEG DCT, Discrete Cosine Transform (JPEG Pt2)- Computerphile". 22 May 2015. <https://www.youtube.com/watch?v=Q2aEzeMDHMA>.

Programming Techniques. "Discrete Cosine Transform and JPEG compression : Image Processing". *Following Tutorials* (7 Feb. 2014). <https://followtutorials.com/2014/02/discrete-cosine-transform-and-jpeg-compression-image-processing.html>.

Townsend, Alex. "JPEG: Image compression algorithm". Mar. 2017. <http://pi.math.cornell.edu/~web6140/TopTenAlgorithms/JPEG.html>.