Assignment:
JPEG Compression Algorithm

اسـم الطالب:	حازم محمد أحمد الشتيحي
الفصل:	2
الرقم الأكاديمي:	1700396
القسـم/الشعبة:	هندسة و علوم الحاسب
الفرقة/المستوي:	الرابعة
اسـم المقـرر:	Digital Multimedia
اشـــراف:	د/ محمد عبده بربار م/ أحمد القوصي

➢ JPEG Compression Algorithm Steps:

- Original Image(.tiff format image)

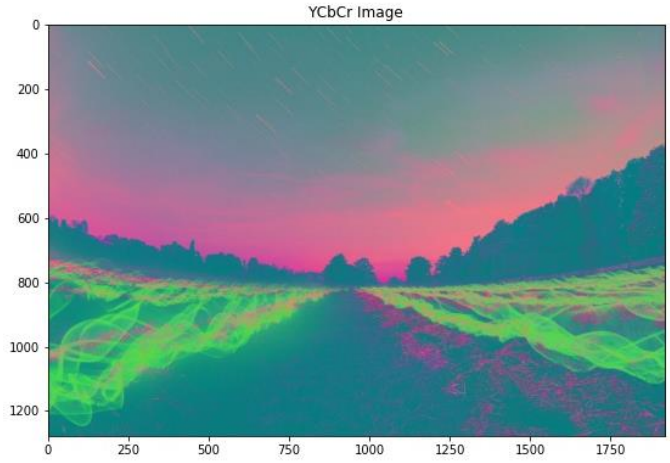
- Result Image

1) Color Space Conversion

- Reading and displaying the image.
- Notice the image dimensions is 1280 x 1920 pixels

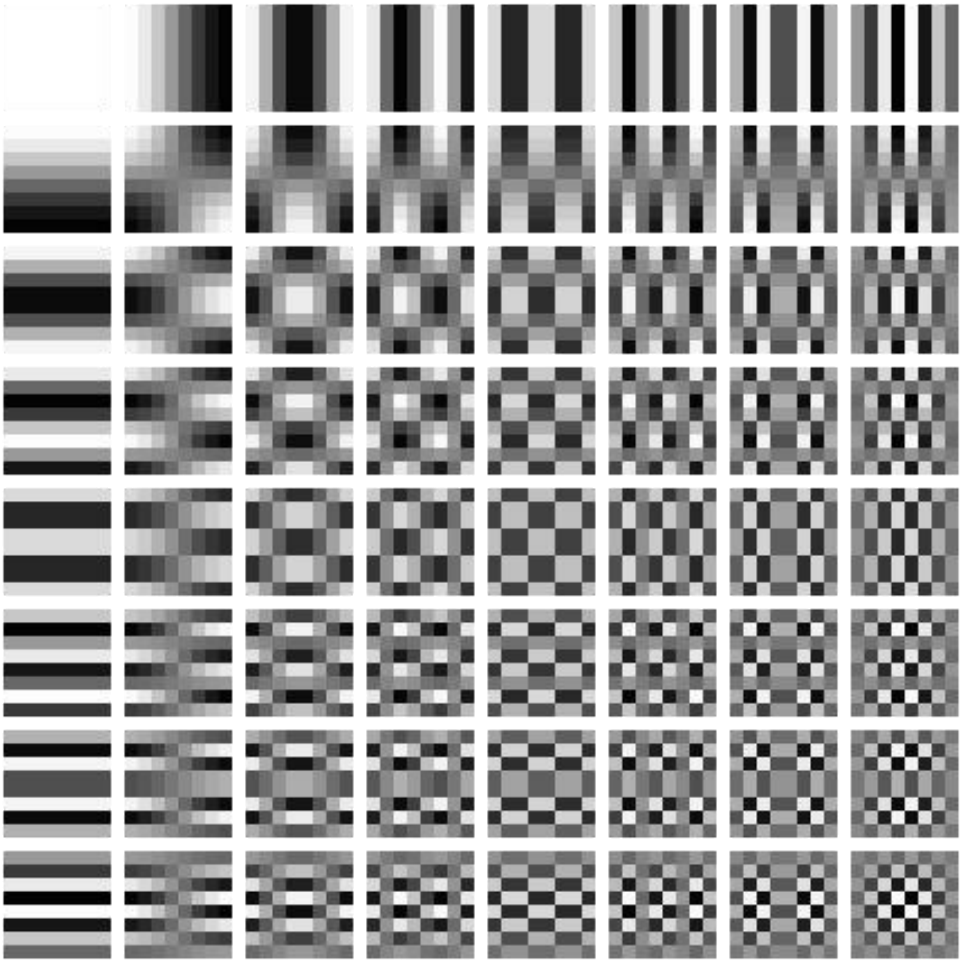
- This is the result after converting the color space of the image from RGB to YCbCr using the function 'color_conversion' that is built from scratch using the equations(delta=128):
- $Y(Luminance) = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$
- $Cr(Red\ Chrominance) = (R - Y) \cdot 0.713 + delta$
- $Cb(Blue\ Chrominance) = (B - Y) \cdot 0.564 + delta$

- $R(Red) = Y + 1.403 \cdot (Cr - delta)$
- $G(Green) = Y - 0.714 \cdot (Cr - delta) - 0.344 \cdot (Cb - delta)$
- $B(Blue) = Y + 1.773 \cdot (Cb - delta)$



2) Discrete Cosine Transform (DCT)

- JPEG converts an image into chunks of 8x8 blocks of pixels, then applies Discrete Cosine Transformation to each block and then uses quantization to compress the resulting block.
- DCT is a method for converting discrete data points into a combination of cosine waves. DCT will take an 8x8 image block and reproduces it using an 8x8 matrix of cosine functions. The 8x8 matrix of cosine functions look like this:



- The top left corner represents the lowest frequency cosine function and the bottom right represents the highest frequency cosine function. Turning the bottom right components to 0, the resulting image would appear the same but missing the high frequency components that are less receptive to the human eye.

3) Quantization

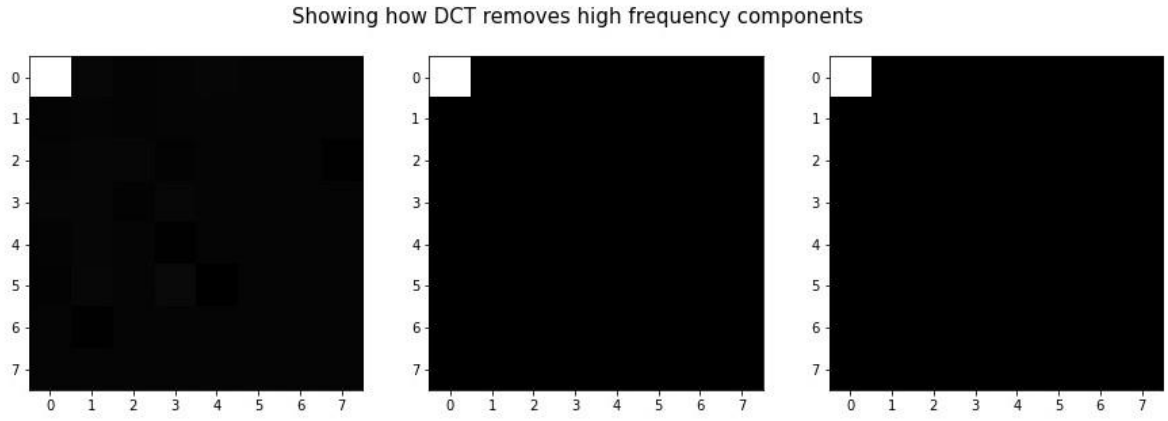
- The JPEG algorithm is a lossy algorithm, and the loss of data happens in the quantization process. The quantization process defines how much you want to compress the image by offering different quantization tables.
- Every 8x8 image block is divided by the quantization table element wise to remove high frequency data.
- Luminance quantization table:

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

- Chrominance quantization table:

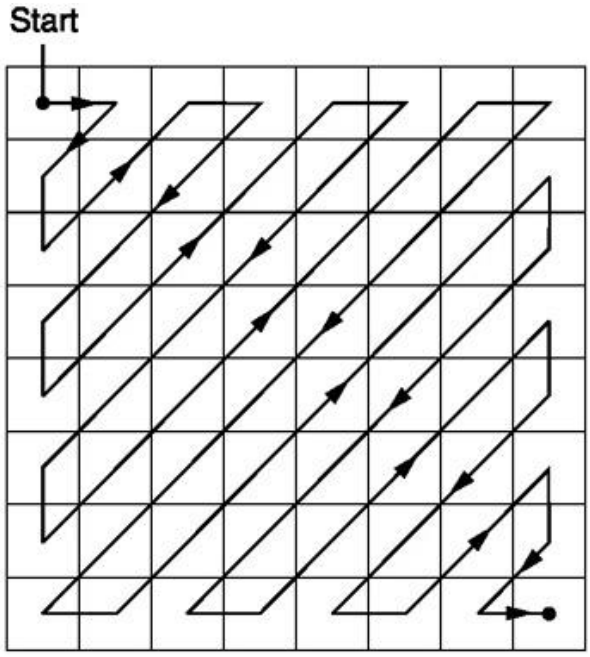
17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

- A result of three 8x8 blocks after removing high frequency components.



4) Run-Length Encoding

- **Zig-Zag Process:** to convert the image into 1-Dimensional array to encode it.




```
def ZigZag(block):
  result = [[] for i in range(side+side-1)]
  for y in range(side):
    for x in range(side):
      i = y + x
      if(i%2 == 0):
          result[i].insert(0,block[y][x])
      else:
          result[i].append(block[y][x])

  return array([coefficient for line in result for coefficient in line])

example = Q_DCT_blocks[0][0][:,:,0]
print('Example of an 8x8 block before ZigZag:')
print(example, '\n')

print('Example of an 8x8 block after ZigZag:')
print(ZigZag(example))
```

```
Example of an 8x8 block before ZigZag:
[[72  1  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]]

Example of an 8x8 block after ZigZag:
[72  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
```

- **Run-length encoding**: is used to compress repeated data. At the end of the zig-zag process, we saw that most of the 1D array had so many 0s at the end. Run-length encoding allows us to save more space.
- The following figure illustrates the compression after applying the zigzag encoding and run-length encoding to the previous block example. Clearly the difference in size is massive.

```
Size Comparisson before and after zig-zag & run length:

Block Example:
 1111001000000110000100001000010011000000

Original size of this block: 192 KB
After Compresion size of this block: 5.125 KB
```

5) Huffman Encoding

- Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression.
- The process below illustrates Huffman encoding by saving the letters in fewer bits.

```
a: 01100001
b: 01100010
c: 01100011
d: 01100100
e: 01100101
```

```
# Mapping

000: 01100001
001: 01100010
010: 01100011
100: 01100100
011: 01100101
```

```
a: 000
b: 001
c: 010
d: 100
e: 011
```

➢ Finally, Save the image as jpg

- Obviously, the resolution of the image didn't change after compression, and hardly any important information is lost.
- Last, we save the result image as jpg format.