Faculty of Engineering

Computer Engineering

# Signals Report

**Importing needed libraries then reading the image**

```
import numpy as np
import matplotlib.pyplot as plt
import cv2

original_image = cv2.imread('Resources/original.png')
```

**Extracting BGR channels and showing them**

```
blue_, green_, red_ = cv2.split(original_image)
cv2.imshow("Blue channel Gray Scaled", cv2.resize(blue_, (960, 540)))
cv2.imshow("Green channel Gray Scaled", cv2.resize(green_, (960, 540)))
cv2.imshow("Red channel Gray Scaled", cv2.resize(red_, (960, 540)))
```

Blue Channel in gray scale

Green Channel in gray scale

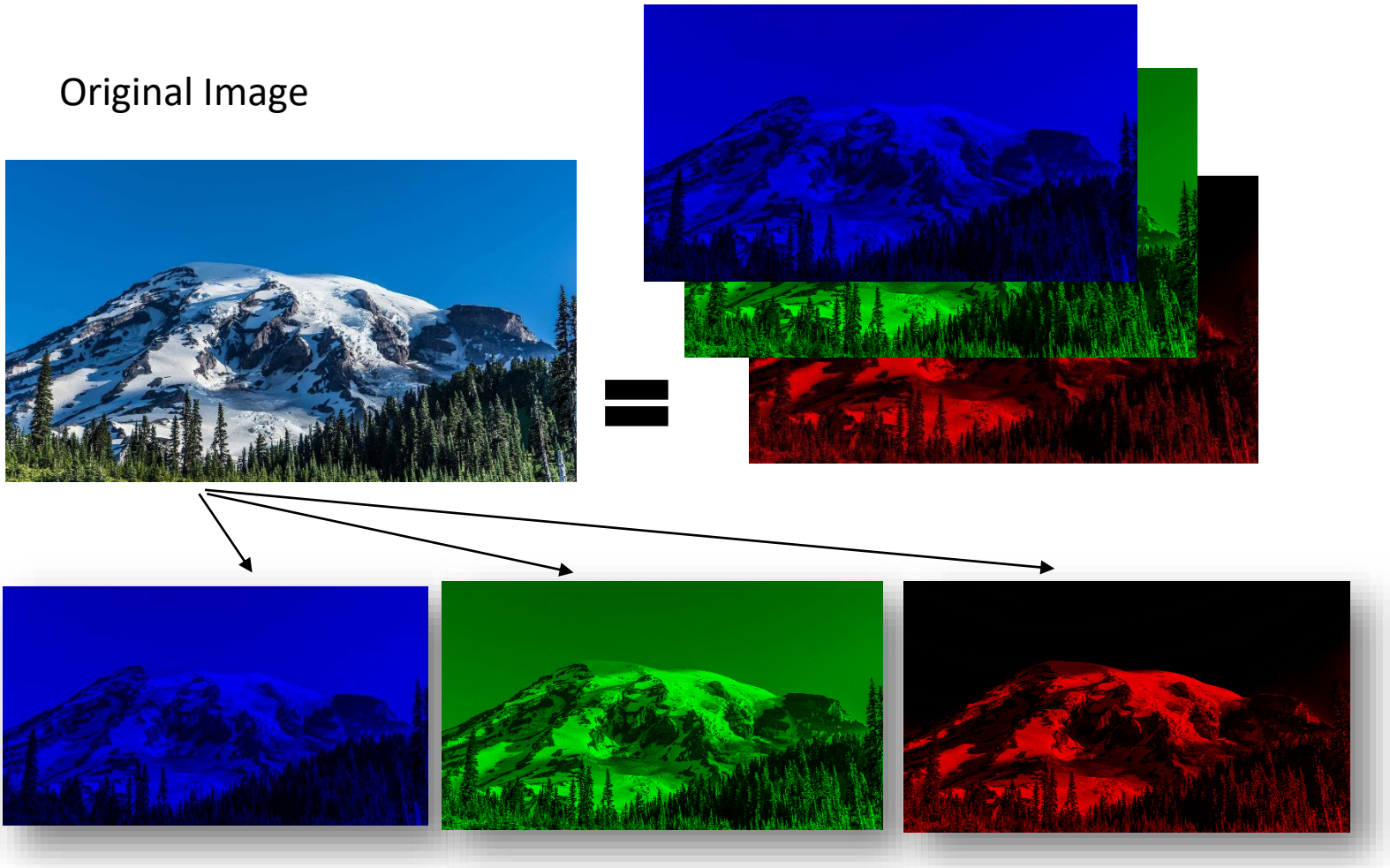
Red Channel in gray scale

We can see that blue channel has more brightness than another channels as the most common color in original image is the blue color.

**Showing BGR channels as colored image**

```
blue, green, red = original_image.copy(), original_image.copy(), original_image.copy()
blue[:, :, (1, 2)] = 0
green[:, :, (0, 2)] = 0
red[:, :, (0, 1)] = 0

cv2.imshow("Blue channel", cv2.resize(blue, (960, 540)))
cv2.imshow("Green channel", cv2.resize(green, (960, 540)))
cv2.imshow("Red channel", cv2.resize(red, (960, 540)))
```




Any colored image can be constructed from three channels Blue, Green and Red as those colors are the basis colors, our job now is to perform the algorithm of image compression to each channel then combine them again to construct the needed colored image.

The performed algorithm well be discussed along with the next code.

**Used Function to compress and decompress the image with comments**

```
def compressImage(m, show_imgs=False):
    """ compress and decompress image """

    decompressed_image = np.zeros_like(original_image)
    m = m
    compressed_image_width = m * (HEIGHT // 8)
    compressed_image = np.empty((m, compressed_image_width, 3))

    for i in range(0, WIDTH, 8):
        compressed_horizonal_slice = np.empty((m, m, 3))
        for j in range(0, HEIGHT, 8):
            block = original_image[i: i + 8, j:j + 8]
            block = block.astype(np.float32) / 255.0
            # split the three channels
            blue, green, red = np.float32(cv2.split(block))

            # apply 2d DCT to each channel to compress the image
            blue_dct = cv2.dct(blue)
            green_dct = cv2.dct(green)
            red_dct = cv2.dct(red)

            # keep the top-left block and ignore the rest
            blue_dct[m:, :] = 0
            blue_dct[:, m:] = 0
            green_dct[m:, :] = 0
            green_dct[:, m:] = 0
            red_dct[m:, :] = 0
            red_dct[:, m:] = 0

            # apply inverse DCT to each channel to decompress the image
            blue_idct = cv2.idct(blue_dct)
            green_idct = cv2.idct(green_dct)
            red_idct = cv2.idct(red_dct)

            # merge the BGR channels to construct our compressed and decompressed images
            decompressed_block = cv2.merge((blue_idct, green_idct, red_idct))
            decompressed_block = (decompressed_block * 255).clip(0, 255).astype(np.uint8)
            decompressed_image[i: i + 8, j:j + 8] = decompressed_block

            compressed_block = cv2.merge((blue_dct[:m, :m], green_dct[:m, :m], red_dct[:m, :m]))
            compressed_block = (compressed_block * 255).clip(0, 255).astype(np.uint8)

            if j == 0:
                compressed_horizonal_slice = compressed_block
            else:
                compressed_horizonal_slice = np.hstack((compressed_horizonal_slice, compressed_block))
```

```
    if i == 0:
        compressed_image = compressed_horizonal_slice
    else:
        compressed_image = np.vstack((compressed_image, compressed_horizonal_slice))

# show the output images
if show_imgs:
    cv2.imshow("Original image", cv2.resize(original_image, (960, 540)))
    cv2.imshow(f"Compressed image at m = {m}", compressed_image)
    cv2.imshow(f"Decompressed image at m = {m}", cv2.resize(decompressed_image, (960, 540)))
    cv2.waitKey(0)
    cv2.destroyAllWindows()

return compressed_image, decompressed_image
```

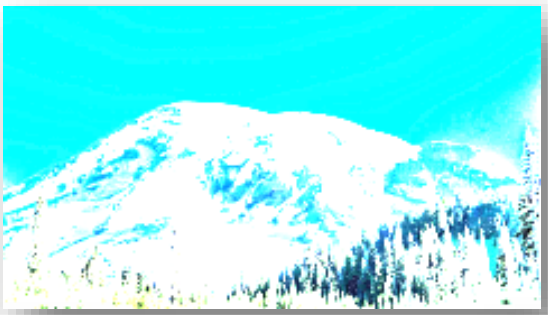
**The above function perform the compression algorithm as following:**

- Extract an 8x8 block from original image at a time.
- Extract the three channels from this block.
- Apply 2D Discrete Cosine Transform (DCT) to each channel to get the coefficients for basis images.
- Retain the mxm top left block of DCT matrix and set the rest to zeros.
- Construct the compressed image from those mxm blocks.
- Apply inverse DCT to obtain the decompressed image.

**To Show the output images for passed m**

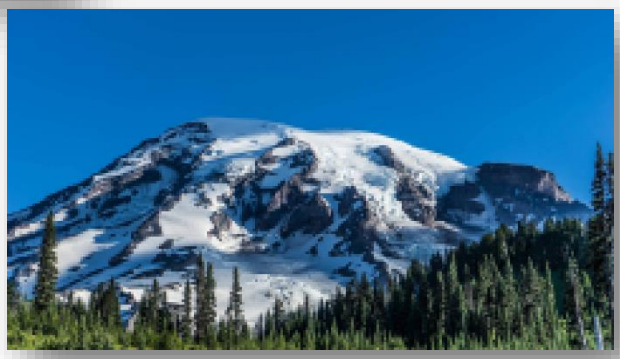
```
compressed_image, decompressed_image = compressImage(m= 1, show_imgs= True)
```

Compressed image at m = 1



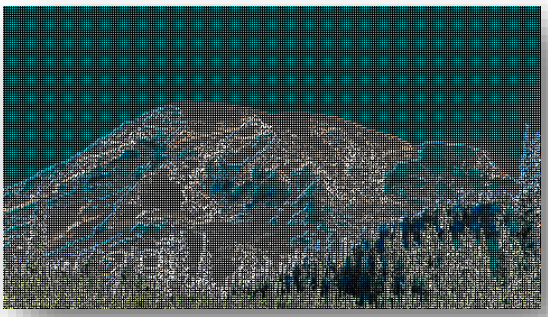
Original Image

Decompressed image at m = 1



```
compressed_image, decompressed_image = compressImage(m= 2, show_imgs= True)
```

Compressed image at m = 2

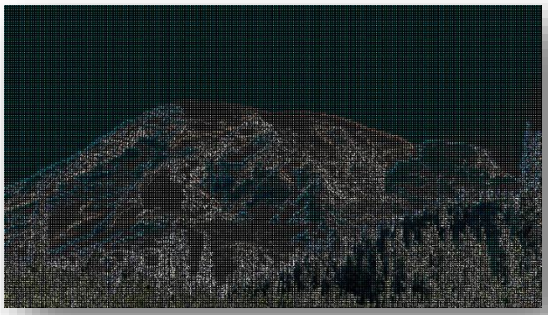


Original Image

Decompressed image at m = 2

```
compressed_image, decompressed_image = compressImage(m= 3, show_imgs= True)
```

Compressed image at m = 3

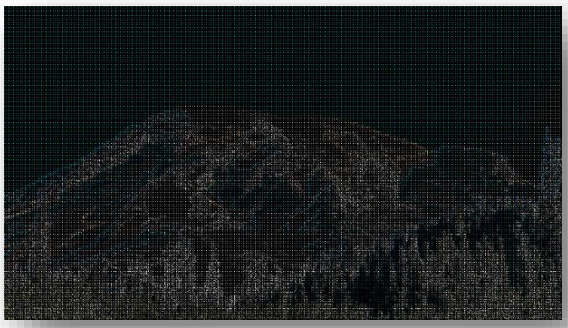


Original Image

Decompressed image at m = 3

```
compressed_image, decompressed_image = compressImage(m= 4, show_imgs= True)
```

Compressed image at m = 4



Original Image

Decompressed image at m = 4

We can see that as we choose higher value for m we get a higher quality image close to the original one, the quality of the decompressed image is measured using the Peak Signal-to-Noise Ratio (PSNR), which is defined by $PSNR = 10\log_{10}(\frac{peak^2}{MSE}), peak = 255$

MSE stands for mean square error $MSE = \frac{(original - compressed)^2}{number\ of\ pixels}$

**Function to calculate PSNR**

```
def calculate_psnr(original_image, decompressed_image):
  """ calculate psnr value between two images """

MSE = np.mean((original_image - decompressed_image) ** 2)
psnr = 10 * np.log10(255 ** 2 / MSE)
return psnr
```

**To get compressed images sizes and calculate PSNR for each value of m**

```
img_sizes = []
PSNR_array = []
m_array = [1, 2, 3, 4]
for m_value in m_array:
  compressed_image, decompressed_image = compressImage(m=m_value)
  psnr_value = calculate_psnr(original_image, decompressed_image)
  PSNR_array.append(psnr_value), img_sizes.append(compressed_image.size)
```

| m | Original shape | Compressed shape | Original size | Compressed size |
|---|---|---|---|---|
| **1** | (1080, 1920, 3) | (135, 240, 3) | 6220800 | 97200 |
| **2** | | (270, 480, 3) | | 388800 |
| **3** | | (405, 720, 3) | | 874800 |
| **4** | | (540, 960, 3) | | 1555200 |

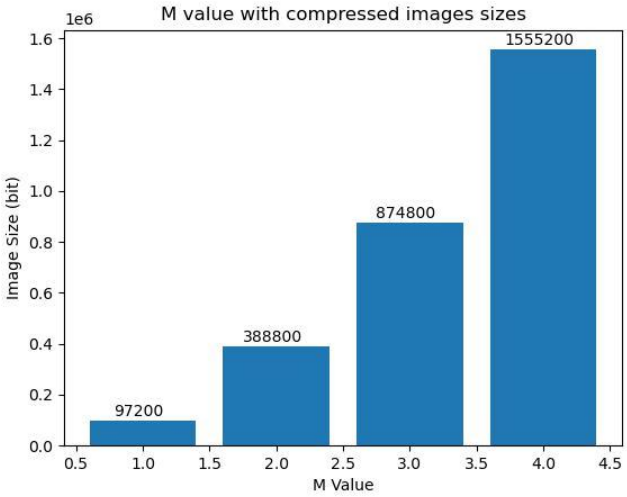
We can find the the size of compressed image decrease for lower value of m but that leads to poor quilaty of the decompressed image as we mentioned before so it is a trade off.
Also note that the size and shape of the decompressed image are the same as for original one.

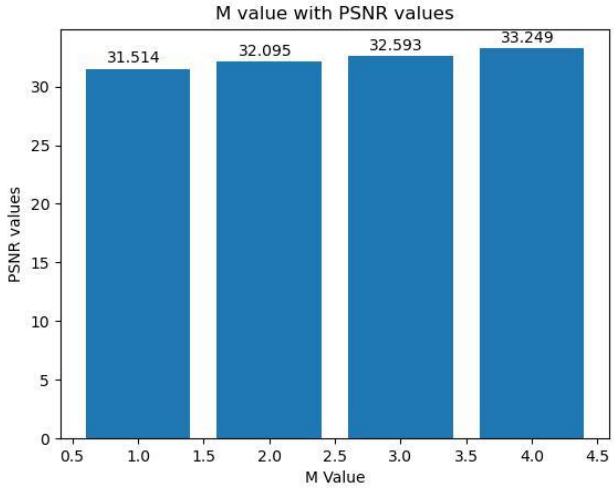
**plot m values with compressed images sizes**

```
plt.bar(m_array, img_sizes)
for x, y in zip(m_array, img_sizes):
    plt.text(x - 0.03, y + 5000, y, ha='center', va='bottom')
plt.xlabel("M Value")
plt.ylabel("Image Size (bit)")
plt.title("M value with compressed images sizes")
```

**plot m values with PSNR values**

```
plt.bar(m_array, PSNR_array)
for x, y in zip(m_array, PSNR_array):
    plt.text(x - 0.03, y + 0.2, round(y, 3), ha='center', va='bottom')
plt.xlabel("M Value")
plt.ylabel("PSNR values")
plt.title("M value with PSNR values")
```




The higher the psnr value the closer the image to the original one, so it is clear that for higher values of m we get a higher value of psnr and higher quality image as discussed before.

# Appendix

Repo: [Elkhiat15/Simple-image-compression-using-DCT]

Note that all the codes included in this report can be copied and pasted, but for entire code and output images check out the above GitHub repo.

# Used tools

**Python** as main programing language.

- **Numby** for pixels matrix manipulation.
- **Opencv** for image processing operations.
- **Matplotlib** for plotting graphs.