Assignment - 5 Report

Github Link -

1. Preprocessing:

1.Preprocessing

```
from skimage import data
import matplotlib.pyplot as plt
import numpy as np

img = data.camera()

h, w = img.shape
img = img[:h - h % 8, :w - w % 8]
print("Image shape:", img.shape)
plt.imshow(img, cmap='gray')
plt.title("Built-in Grayscale Image")
plt.axis('off')
plt.show()
```

```
Image shape: (512, 512)
```

This loads a sample grayscale image of a man using a camera from the skimage library. It's a commonly used built-in image for testing image processing techniques.

Since the SVD compression will be done on 8×8 blocks, the code ensures that both the height and width of the image are divisible by 8. This is done by cropping a few pixels from the edges if necessary.

After that, it displays the image using matplotlib, along with the image shape (which is 512×512 pixels). The image is already in grayscale, so no additional conversion is needed.

In short, this step prepares the image for the next steps in the compression pipeline by resizing and displaying it properly.

2.Block-wise SVD Function:

```
Image shape: (512, 512)
Extracting first few 8x8 blocks...

Block at (0, 0):
 [[200 200 200 200 199 200 199 198]
 [200 199 199 200 199 200 199 198]
 [199 199 199 200 200 200 200 200]
 [200 200 199 199 199 199 199 199]
 [200 200 200 200 199 199 199 200]
 [200 199 199 200 199 199 199 199]
 [200 201 200 200 199 200 198 199]
 [201 200 200 200 200 199 199 200]]

Block at (0, 8):
 [[199 198 198 198 198 198 198 198]
 [198 199 199 199 199 199 198 198]
 [200 200 200 199 200 199 198 198]
 [198 199 198 198 199 198 198 199]
 [199 199 198 198 199 199 200 199]
 [199 199 198 198 199 199 199 198]
 [199 199 199 199 200 200 199 198]
 [200 199 200 199 199 200 199 199]]

Block at (8, 0):
 [[200 200 200 199 200 200 200 199]
 [200 200 199 200 200 200 201 200]
 [200 200 201 200 200 201 200 200]
 [201 199 200 200 199 200 200 200]
 [200 199 198 200 200 200 200 202]
 [200 199 199 199 200 201 201 201]
 [201 199 200 200 199 201 201 200]
 [201 201 201 199 201 200 200 201]]

Block at (8, 8):
 [[199 199 199 198 199 200 199 199]
 [199 199 199 199 200 200 200 199]
 [199 199 200 200 200 200 199 199]
 [201 199 200 199 200 200 200 199]
 [200 200 199 200 199 200 200 199]
 [199 200 199 201 199 200 199 201]
 [201 201 200 199 200 200 201 200]
 [202 200 200 200 200 201 200 200]]
```

The image is 512×512 pixels, and since 512 is divisible by 8, the image is perfectly split into (512 / 8) = 64 blocks per row and column, or 4096 total blocks.

This code is extracting and displaying the pixel values (from 0 to 255) of the first four 8×8 blocks, located at:

- (0, 0) - top-left corner
- (0, 8) - one block to the right
- (8, 0) - one block down
- (8, 8) - diagonally down-right

Implement compress_block(block, k):

Implement compress_block(block, k)

```
[10] def compress_block(block, k):
        U, S, Vt = np.linalg.svd(block, full_matrices=False)
        S[k:] = 0
        compressed = np.dot(U, np.dot(np.diag(S), Vt))
        return compressed
```

```
sample_block = img[0:8, 0:8]
compressed_block = compress_block(sample_block, k=4)
print("Original Block:\n", sample_block)
print("\nCompressed Block (k=4):\n", compressed_block.astype(np.uint8))
```

```
Original Block:
 [[200 200 200 200 199 200 199 198]
 [200 199 199 200 199 200 199 198]
 [199 199 199 200 200 200 200 200]
 [200 200 199 199 199 199 199 199]
 [200 200 200 200 199 199 199 200]
 [200 199 199 200 199 199 199 199]
 [200 201 200 200 199 200 198 199]
 [201 200 200 200 200 199 199 200]]

Compressed Block (k=4):
 [[200 200 199 200 198 200 198 198]
 [199 199 199 199 199 199 199 197]
 [199 199 198 200 199 199 200 200]
 [199 199 199 199 198 198 198 199]
 [200 200 199 199 199 199 198 199]
 [200 199 199 199 199 199 199 198]
 [199 201 200 199 198 199 198 198]
 [201 200 199 200 199 198 199 200]]
```

Testing how Singular Value Decomposition (SVD) works on a single 8×8 block of the image.

- First, you extract the top-left 8×8 region from the image this small grid of pixel values is called the original block.
- Then, you apply SVD compression to this block by keeping only the top 4 singular values (`k=4`). This reduces the amount of data used to represent the block.
- After compression, the block is reconstructed using only those top 4 values

Recombine blocks into a full image:

Recombine blocks into a full image

```
[12] def compress_image_blockwise(img, k):
        h, w = img.shape
        compressed_img = np.zeros_like(img, dtype=np.float32)

        for i in range(0, h, 8):
            for j in range(0, w, 8):
                block = img[i:i+8, j:j+8]
                compressed_block = compress_block(block, k)
                compressed_img[i:i+8, j:j+8] = compressed_block

        return np.clip(compressed_img, 0, 255).astype(np.uint8)
```

This function compress_image_blockwise(img, k) compresses an entire image by applying SVD to every non-overlapping 8×8 block. It loops through the image block by block, compresses each block using the compress_block function, and stores the result in a new image. Finally, it clips the pixel values to the valid range (0–255) and returns the reconstructed image.

Final Image:

Final Image

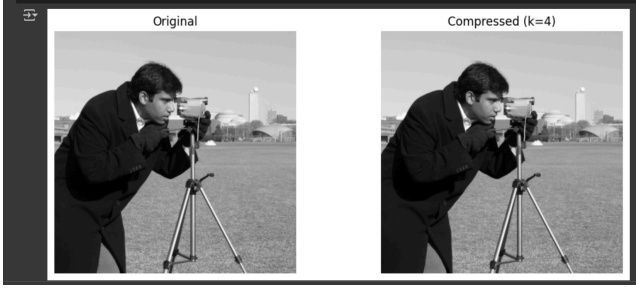
```
compressed_img = compress_image_blockwise(img, k=4)

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.imshow(img, cmap='gray')
plt.title("Original")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(compressed_img, cmap='gray')
plt.title("Compressed (k=4)")
plt.axis('off')

plt.tight_layout()
plt.show()
```



This step shows a side-by-side comparison of the original image and the compressed image using block-wise SVD with `k=4`. Each 8×8 block in the image is compressed by keeping only the top 4 singular values. Despite this reduction, the compressed image looks almost identical to the original, with only slight blurring. This demonstrates that block-wise SVD can effectively reduce data while preserving visual quality.

3. Compression Analysis:

Apply Block-wise SVD for each k ∈ {1,…,8} and Compute Compression Ratio for each k

3.Compression Analysis

Apply Block-wise SVD for each k ∈ {1,...,8}

```
compressed_images = {}
k_values = list(range(1, 9))

for k in k_values:
    compressed_images[k] = compress_image_blockwise(img, k)
```

Compute Compression Ratio for each k

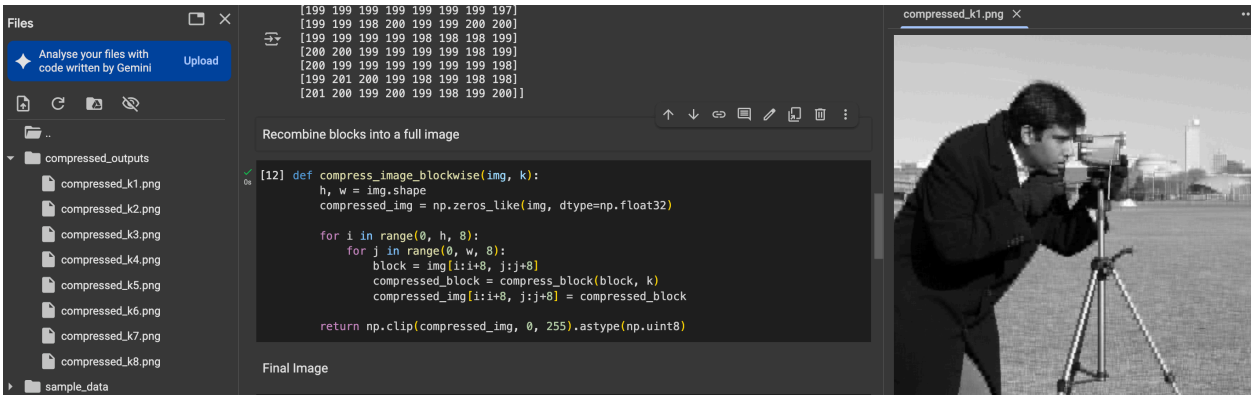
```
[15] compression_ratios = []

    for k in k_values:
        original_values = 64
        retained_values = k * (8 + 8 + 1)
        ratio = original_values / retained_values
        compression_ratios.append(ratio)

    print("Compression Ratios for k = 1 to 8:\n", compression_ratios)
```

```
Compression Ratios for k = 1 to 8:
 [3.764705882352941, 1.8823529411764706, 1.2549019607843137, 0.9411764705882353, 0.7529411764705882, 0.6274509803921569, 0.5378151260504201, 0.47058823529411764]
```

This step performs compression analysis by applying block-wise SVD to the image for different values of `k` ranging from 1 to 8. For each `k`, the image is compressed using only the top `k` singular values in every 8×8 block, and the results are stored. Then, the compression ratio for each `k` is calculated by comparing the original data size (64 values per block) to the number of values retained after compression, which is `k×(8+8+1)`. The output shows that smaller `k` values result in higher compression ratios for example, `k=1` achieves a ratio of about 3.76, meaning the data is reduced by nearly four times. As `k` increases, the compression ratio decreases, reflecting a trade-off between compression efficiency and image quality.



saved compressed images for each k = 1 to 8 in a folder named compressed_outputs. Each file (like compressed_k1.png, compressed_k2.png, etc.) represents the full image reconstructed using only the top k singular values per block.

The image compressed_k1.png and so on….is displayed this is the version of the original image reconstructed using only 1 singular value per 8×8 block.

- It's still recognizable, but visibly blurrier compared to the original.
- So there are total 8 images if you keep going the image looks good in last image
- This demonstrates that even with very aggressive compression (keeping only 1 value out of 64), some visual structure is retained.

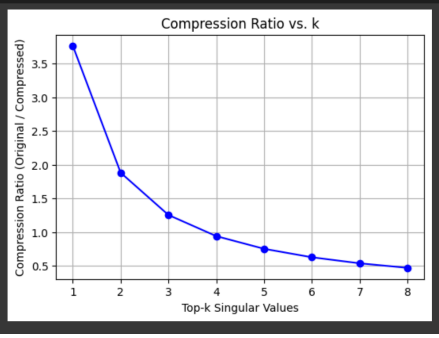
4. Visualization:

Compression Ratio vs. k

4.Visualization

Compression Ratio vs. k

```
plt.figure(figsize=(6, 4))
plt.plot(k_values, compression_ratios, marker='o', linestyle='-', color='blue')
plt.title("Compression Ratio vs. k")
plt.xlabel("Top-k Singular Values")
plt.ylabel("Compression Ratio (Original / Compressed)")
plt.grid(True)
plt.xticks(k_values)
plt.show()
```

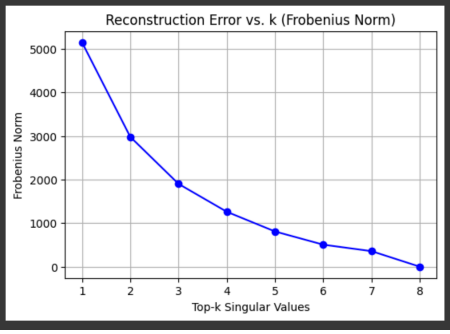


This step visualizes how the compression ratio changes as the number of singular values `k` increases. The plot titled "Compression Ratio vs. k" shows that when fewer singular values are retained (low `k`), the compression ratio is high indicating stronger compression. As `k` increases, more data is kept in each block, so the compression ratio decreases. The graph follows a downward curve, meaning that each additional singular value retained reduces the compression benefit. This visualization clearly shows the trade-off between compression and data retention: using a small `k` gives high compression but may lose quality, while a higher `k` keeps more image detail but compresses less efficiently.

Reconstruction Error (Frobenius norm) vs. k

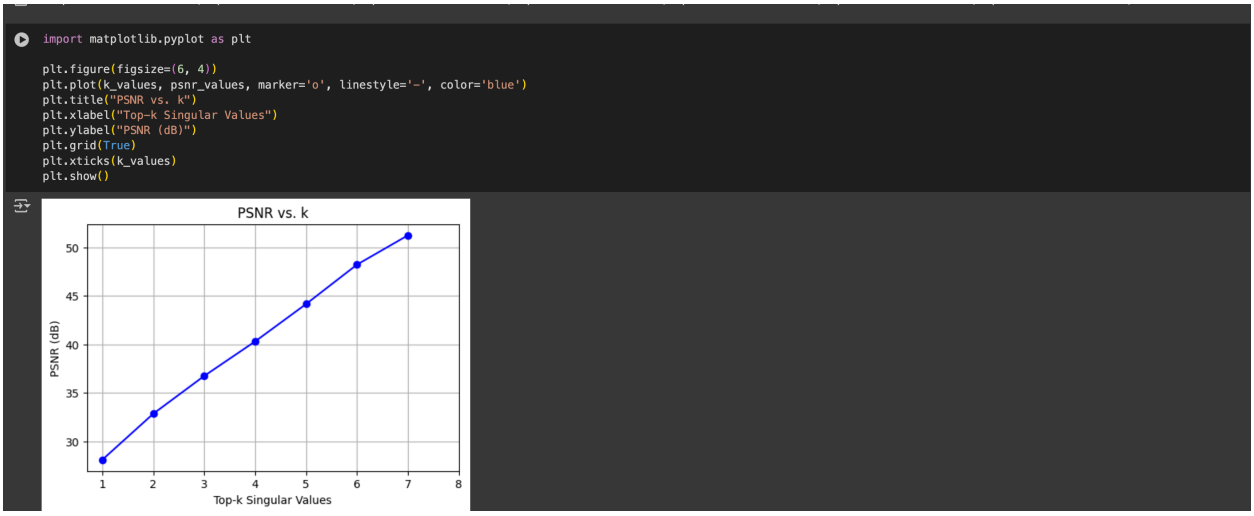
```
import matplotlib.pyplot as plt

plt.figure(figsize=(6, 4))
plt.plot(k_values, frobenius_errors, marker='o', linestyle='-', color='blue')
plt.title("Reconstruction Error vs. k (Frobenius Norm)")
plt.xlabel("Top-k Singular Values")
plt.ylabel("Frobenius Norm")
plt.grid(True)
plt.xticks(k_values)
plt.show()
```



This step visualizes how the reconstruction error changes with different values of `k`, where `k` represents the number of top singular values used during block wise SVD compression. The plot titled "Reconstruction Error vs. k (Frobenius Norm)" shows that as `k` increases, the reconstruction error (measured using Frobenius norm) decreases significantly. This makes sense because retaining more singular values helps better preserve the original image details during reconstruction. At `k=1`, the error is the highest, meaning the image loses the most information, while at `k=8`, the error drops to near zero, indicating a near-perfect reconstruction. The curve shows a smooth and steady decline, clearly illustrating the trade-off between compression level and image accuracy higher compression introduces more error, while better image quality requires more data.

PSNR (Peak Signal-to-Noise Ratio) vs K



This step visualizes the relationship between the number of singular values (`K`) retained during compression and the Peak Signal-to-Noise Ratio (PSNR), a common measure of image quality. The plot titled "PSNR vs. K" shows that as `K` increases, PSNR also increases, indicating that the compressed image becomes more similar to the original. At low `K` values, PSNR is relatively low, meaning more distortion and lower image quality. As `K` approaches 8, PSNR rises significantly, crossing 50 dB, which suggests near perfect reconstruction. This plot clearly illustrates that while lower `k` gives higher compression, retaining more singular values leads to better visual quality. PSNR serves as a helpful metric to balance between compression and image fidelity.