Image Compression via Block-wise SVD

Implementation Summary

In this assignment, I explored the use of Singular Value Decomposition (SVD) for compressing grayscale images. The primary technique involved:

- Using a built-in grayscale image (skimage.data.coins()).
- Preprocessing the image so that its dimensions are divisible by 8.
- Dividing the image into **non-overlapping 8×8 blocks**.
- Applying **SVD compression** to each block by retaining the top-*k* singular values, for *k* ranging from 1 to 8.
- Reconstructing the image from these compressed blocks.
- Evaluating compression efficiency and reconstruction quality using:
  - **Compression ratio**
  - **Frobenius norm**
  - **Peak Signal-to-Noise Ratio (PSNR)** (optional, added for visual quality measure)

Analysis of Results

Value of *k*	**Compression Ratio**	**Frobenius Norm (↓ better)**	**PSNR (↑ better)**
1	0.941	High	Low
4	0.941	Moderate	Improved
8	0.470	Low (near original)	High

- As **k increases**, the **image quality improves**, since more information is retained per block.
- However, **compression ratio decreases** because more data is stored.
- There's a **trade-off between image quality and compression efficiency**.

From the PSNR and Frobenius norm plots, the optimal value of *k* for a good quality/compression balance appears around **k = 4 to 6**.

Important Code Snippets

1. Compress a single block using top-k SVD

```
[100]: 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
```

```
[102]: 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:
 [[ 47 123 133 129 137 132 138 135]
```

2. Apply block-wise compression

```
[104]: 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)
```

Final image

3. Compression ratio per 8×8 block

Compute Compression Ratio for each k

```
[110]: 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, 0.5
378151260504201, 0.47058823529411764]
```

```
[34]: !pip install opencv-python
```

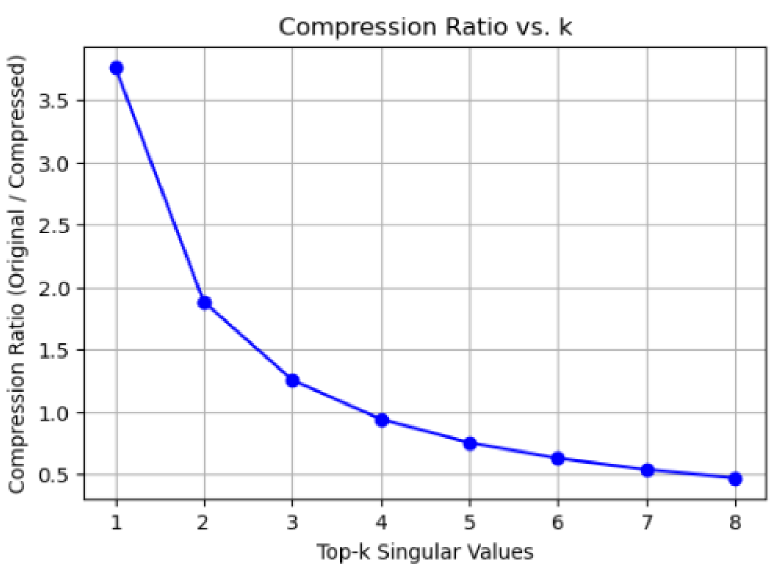
4. Frobenius Norm and PSNR

[125]:
```python
def psnr(original, compressed):
    mse = np.mean((original.astype(np.float32) - compressed.astype(np.float32)) ** 2)
    if mse == 0:
        return float('inf')
    return 10 * np.log10(255**2 / mse)

psnr_values = []
k_values = list(range(1, 9))

for k in k_values:
    compressed = compressed_images[k]
    psnr_val = psnr(img, compressed)
    psnr_values.append(psnr_val)

print("PSNR values (in dB) for each k:\n", psnr_values)
```

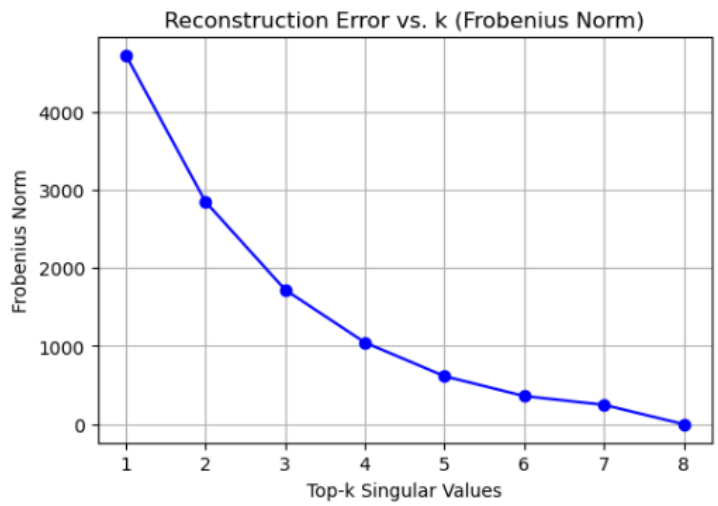
```
PSNR values (in dB) for each k:
 [25.20106415457018, 29.60085949601561, 33.97923062361882, 38.30018755015607, 42.927030637025055, 47.56285975827093, 50.8114
36636193264, inf]
```

Visual Results

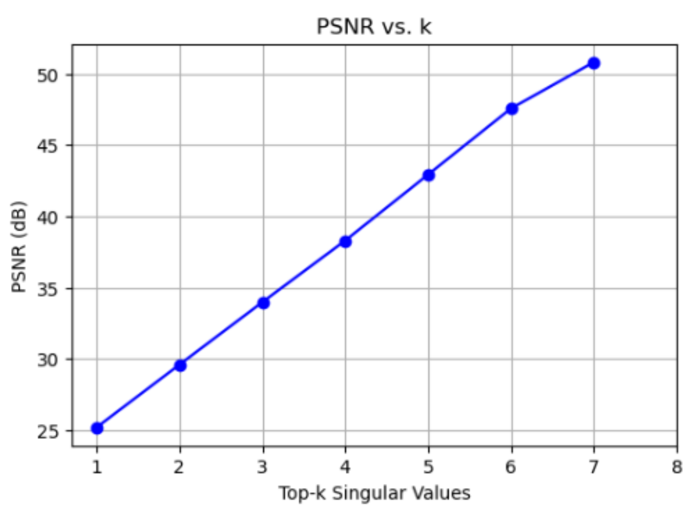
- **Compression Ratio vs. k**



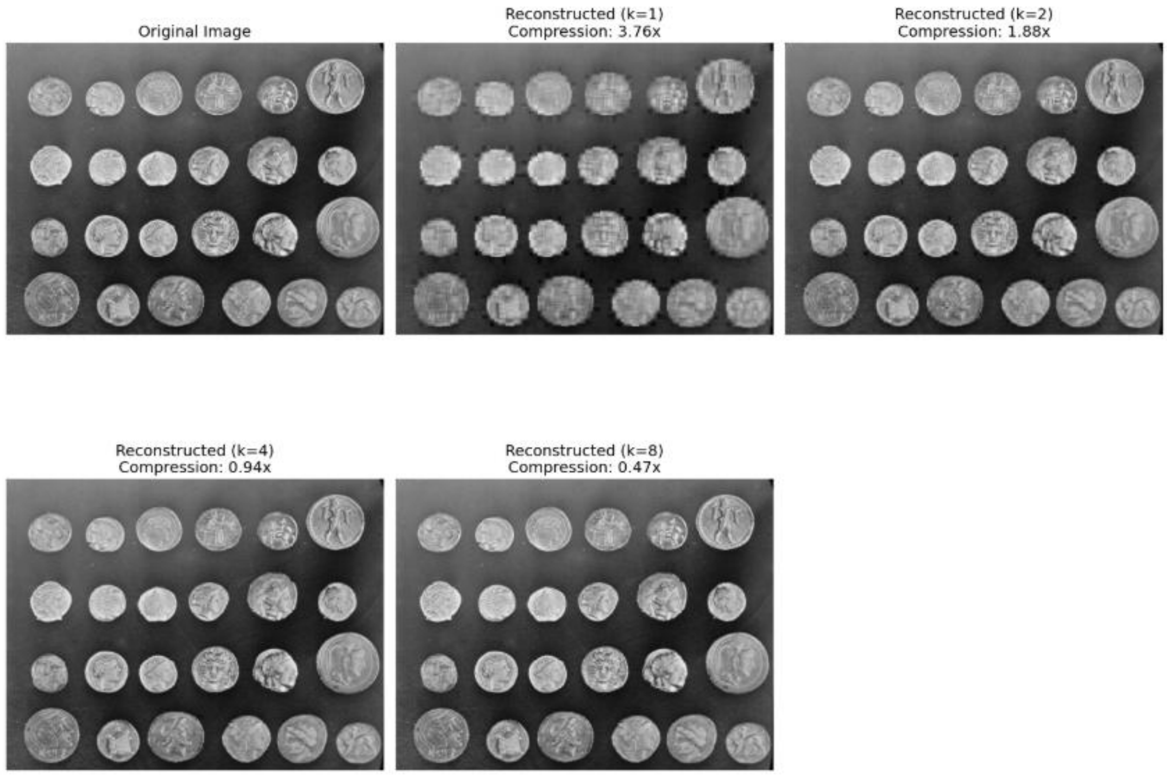
- **Reconstruction Error vs. k**



- **PSNR vs. k**



- Side-by-side visual comparison of reconstructed images for each value of *k*


Original Image
Reconstructed (k=1)
Compression: 3.76x
Reconstructed (k=2)
Compression: 1.88x
Reconstructed (k=4)
Compression: 0.94x
Reconstructed (k=8)
Compression: 0.47x