---

title: SVD workbook

format: html

jupyter: python3

---

**# Introduction**

Image processing has become integral to numerous fields, from medical imaging to digital forensics, where large volumes of visual data demand efficient storage, transmission, and quality retention techniques. Among the many mathematical transformations applied to images, Singular Value Decomposition (SVD) has emerged as a particularly valuable tool. SVD is a matrix factorization technique that represents a given matrix as a product of three matrices: $U$, $\Sigma$, and $V^T$. This decomposition is significant in image processing because it maximizes the energy contained in the largest singular values, enabling the creation of compact, high-quality approximations of the original data. Unlike other transformations, SVD does not require a specific image size or type, making it highly adaptable and robust for various image processing tasks.

The primary strength of SVD lies in its capacity to separate image data into meaningful components. For instance, in an image represented by SVD, the larger singular values and their corresponding vectors encode most of the structural content, while smaller singular values can often represent noise. This property is beneficial for applications requiring data reduction, such as image compression and denoising, where maintaining the primary structure while reducing extraneous information is essential. Additionally, SVD’s stable mathematical foundation and adaptability have made it increasingly popular in other specialized applications, including watermarking for digital forensics and security.

In image compression, SVD enables reduced data storage by approximating the image using fewer singular values, providing a balance between quality and compression ratio. This application is critical in fields where storage and bandwidth are constrained. Similarly, in denoising, SVD can isolate noise by exploiting the decomposition’s ability to differentiate between dominant and subdominant subspaces, allowing effective noise suppression without significantly affecting the image’s core structure. Furthermore, SVD is also used in watermarking, where slight modifications to specific singular values embed unique patterns within images, enhancing security and ensuring authenticity.

Despite these advantages, SVD in image processing remains an area with unexplored potential. This paper explores these established applications while addressing underutilized SVD properties to uncover new applications. By investigating SVD's adaptive properties in compressing and filtering images, as well as its potential for encoding data securely, this work contributes to a growing body of research on SVD-based image processing and presents promising directions for further study.

**# SVD Application in Image Processing**

Singular Value Decomposition (SVD) has several important applications in image processing. The SVD can be used to reduce the noise or compress matrix data by eliminating small singular values or higher ranks @Chen2018SingularVD. This allows for the size of stored images to be reduced @cao2006singular. Additionally, the SVD has properties that make it useful for various image processing tasks, such as enhancing image quality and filtering out noise. The main theorem of SVD is reviewed in the search results, and numerical experiments have been conducted to illustrate its applications in image processing.

**## Image Compression**

Image compression represents a vital technique to reduce the data needed to represent an image. This is crucial for achieving efficient storage and transmission across various applications, including digital photography, video streaming, and web graphics. Compression methods are primarily categorized into two distinct types: lossy and lossless.

Lossy compression diminishes file size by irreversibly eliminating certain image data, which can result in a degradation of image quality, as observed in JPEG formats. This method is frequently employed when the reduction of file size is of paramount importance, and any resultant loss in quality is considered acceptable.

Conversely, lossless compression techniques allow for the compression of images without any loss of data, facilitating the exact reconstruction of the original image, as exemplified by PNG formats. This approach is beneficial when preserving image quality is essential and minimizing file size is of lesser importance.

The decision to use either lossy or lossless compression hinges on the specific needs of the application, balancing the trade-offs between file size and image quality.

SVD-based image compression functions by decomposing the image matrix into three components and subsequently approximating the original matrix with only the most significant singular values and vectors. This process results in a compact image representation while preserving the essential information.

Mathematically, given an image represented as a matrix $A$ with dimensions $m \times n$, the Singular Value Decomposition (SVD) decomposes $A$ into three matrices: $U$, $\Sigma$, and $V^T$. Here, $U$ is an $m \times m$ orthogonal matrix containing the left singular vectors, $\Sigma$ is an $m \times n$ diagonal matrix containing singular values, and $V^T$ is the transpose of an $n \times n$ orthogonal matrix containing the right singular vectors. To compress the image, we keep only the top $k$ singular values (where $k$ is significantly smaller than both $m$ and $n$). The compressed image can be reconstructed as

$$

A\_k = U\_k \Sigma\_k V\_k^T,

$$

where $U\_k$ contains the first $k$ columns of $U$, $\Sigma\_k$ is a $k \times k$ diagonal matrix of the top $k$ singular values, and $V\_k^T$ consists of the first $k$ rows of $V^T$.

```{python}

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

# Read and convert the image to grayscale

img = Image.open('amrita\_campus.jpg')  # Specify your image file

gray\_img = img.convert('L')  # Convert to grayscale

A = np.array(gray\_img, dtype=np.float64)  # Convert to float64 for SVD computation

original=A

# Apply Singular Value Decomposition (SVD)

U, S, Vt = np.linalg.svd(A, full\_matrices=False)

# Choose the number of singular values to keep for compression

k = 50  # You can adjust this value to see different compression levels

# Create a compressed version of the image using the first k singular values

S\_k = np.zeros\_like(A)  # Initialize a zero matrix for S\_k

S\_k[:k, :k] = np.diag(S[:k])  # Keep only the top k singular values

# Reconstruct the compressed image

A\_k = np.dot(U[:, :k], np.dot(S\_k[:k, :k], Vt[:k, :]))  # Reconstruct the image from the reduced SVD

# Display the original and compressed images

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.imshow(A, cmap='gray', vmin=0, vmax=255)  # Display original image

plt.title('Original Image')

plt.axis('off')

plt.subplot(1, 2, 2)

plt.imshow(A\_k, cmap='gray', vmin=0, vmax=255)  # Display compressed image

plt.title(f'Compressed Image (k = {k})')

plt.axis('off')

plt.show()

```

To assess the quality of the original and compressed images, various metrics can be employed. Commonly used measures are discussion in this section.

**### Image Quality Assessment Metrics**

To evaluate the quality of compressed images relative to their original versions, several standardized metrics are commonly employed. These metrics provide quantitative comparisons across aspects such as pixel-level error, signal fidelity, structural similarity, and compression efficiency. The following are the key metrics used in image quality assessment:

**####  Mean Squared Error (MSE)**

The Mean Squared Error quantifies the average squared difference between corresponding pixel values of the original and compressed images. Lower values indicate higher fidelity to the original. Mathematically, MSE is defined as:

$$

\text{MSE} = \frac{1}{m \cdot n} \sum\_{i=1}^{m} \sum\_{j=1}^{n} (A(i,j) - A\_k(i,j))^2

$$

where $A(i,j)$ and $A\_k(i,j)$ denote the pixel values of the original and compressed images, respectively, and $m \times n$ represents the image dimensions.

**#### Peak Signal-to-Noise Ratio (PSNR)**

PSNR is a widely used metric that compares the maximum possible signal value to the noise level introduced by compression. It is computed as:

$$

\text{PSNR} = 10 \cdot \log\_{10} \left( \frac{\text{MAX}^2}{\text{MSE}} \right)

$$

where $\text{MAX}$ represents the maximum pixel value (e.g., 255 for 8-bit images). Higher PSNR values indicate better image quality, as they correspond to lower MSE values.

**#### Structural Similarity Index (SSIM)**

The Structural Similarity Index assesses perceptual similarity by analyzing luminance, contrast, and structural information between the original and compressed images. The SSIM index, ranging from -1 to 1, is calculated as:

$$

\text{SSIM}(A, A\_k) = \frac{(2 \mu\_A \mu\_{A\_k} + C\_1)(2 \sigma\_{AA\_k} + C\_2)}{(\mu\_A^2 + \mu\_{A\_k}^2 + C\_1)(\sigma\_A^2 + \sigma\_{A\_k}^2 + C\_2)}

$$

where $\mu$, $\sigma$, and $\sigma\_{AA\_k}$ denote means, variances, and covariances of $A$ and $A\_k$, with constants $C\_1$ and $C\_2$ to prevent division by zero. Higher SSIM values suggest higher structural fidelity.

**#### Compression Ratio (CR)**

Compression Ratio quantifies the efficiency of compression, calculated as the ratio of the original image size to the compressed size:

$$

\text{Compression Ratio} = \frac{\text{Size of Original Image}}{\text{Size of Compressed Image}}

$$

A higher compression ratio indicates a greater reduction in file size, which is desirable in applications requiring efficient storage or transmission.

**#### Normalized Cross-Correlation (NCC)**

Normalized Cross-Correlation measures the similarity in pixel intensity patterns between the original and compressed images. NCC is calculated as:

$$

\text{NCC} = \frac{\sum (A \cdot A\_k)}{\sqrt{\sum A^2 \cdot \sum A\_k^2}}

$$

Values closer to 1 indicate a stronger correlation, signifying greater retention of the original image characteristics in the compressed version.

These metrics collectively provide a comprehensive assessment of image quality by addressing both objective and perceptual aspects of compression, making them suitable for a wide range of applications in image processing and computer vision.

**:::{#tbl-quality-metrics}**

| Metric                               | Value          |

|:-------------------------------------|----------------|

| Mean Squared Error (MSE)             | 110.2853       |

| Peak Signal-to-Noise Ratio (PSNR)    | 27.7056 dB     |

| Structural Similarity Index (SSIM)   | 0.8116         |

| Compression Ratio (CR)               | 10.78          |

| Normalized Cross-Correlation (NCC)   | 0.9976         |

| Original Image Size                  | 9709.38 KB     |

| Compressed Image Size                | 900.78 KB      |

| Size Reduction                       | 8808.59 KB     |

: Quality assessment metrics for original and compressed images, detailing standard measures of image compression and fidelity.

**:::**

The quality assessment metrics indicate effective compression with minimal loss of fidelity in the image. A Mean Squared Error (MSE) of 110.29 suggests that the average pixel intensity differences between the original and compressed images are small. The Peak Signal-to-Noise Ratio (PSNR) of 27.71 dB, typically above the 30 dB threshold for high-quality compression, indicates moderate quality but acceptable for many applications.

The Structural Similarity Index (SSIM) of 0.8116, close to 1, suggests that the perceptual similarity between the images remains high. The Compression Ratio (CR) of 10.78 shows significant size reduction, and the Normalized Cross-Correlation (NCC) of 0.9976 demonstrates a high correlation between the original and compressed images, supporting strong structural consistency.

The compressed image achieves substantial size reduction (from 9709.38 KB to 900.78 KB) with reasonable preservation of visual quality, making it suitable for applications prioritizing storage efficiency without heavily compromising visual fidelity.

The table below presents a comparison of compression quality metrics for three different image compression methods: Singular Value Decomposition (SVD), Discrete Cosine Transform (DCT), and Wavelet Transform. The metrics included are Mean Squared Error (MSE), Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index (SSIM), Compression Ratio (CR), Normalized Cross-Correlation (NCC), Compressed Size, and Size Reduction. Each metric provides insight into the effectiveness of the compression techniques in terms of image quality and storage efficiency.

**:::{#tbl-quality-assessment}**

| Method   |      MSE      |     PSNR (dB)     |      SSIM      |    CR    |     NCC     | Compressed Size (KB) |

|----------|---------------|-------------------|----------------|----------|-------------|----------------------|

| SVD      | 282.9933     | 23.6130           | 0.7122         | 6.88     | 0.9938      | 176.33               |

| DCT      | 1172.0801    | 17.4412           | 0.5914         | 2.28     | 0.9741      | 531.82               |

| Wavelet  | 0.0197       | 65.1859           | 0.9999         | 0.12     | 1.0000      | 9715.31              |

: Quality assessment metrics for original and compressed images in comparison with popular image compression algorithms.

**:::**

The results demonstrate that Singular Value Decomposition (SVD) offers a superior balance between image quality and compression efficiency compared to Discrete Cosine Transform (DCT) and Wavelet Transform. With a significantly lower Mean Squared Error (MSE) and a Peak Signal-to-Noise Ratio (PSNR) of 23.6130 dB, SVD preserves the original image quality more effectively than DCT (17.4412 dB) and offers practical structural similarity (SSIM) of 0.7122. In contrast, while the Wavelet method achieves excellent PSNR (65.1859 dB) and SSIM (0.9999), its large compressed size (9715.31 KB) renders it impractical for many applications.

In terms of compression efficiency, SVD yields a Compression Ratio (CR) of 6.88 with a manageable compressed size of 176.33 KB, resulting in a significant size reduction of 1037.34 KB. This contrasts sharply with DCT’s lower CR of 2.28 and Wavelet’s CR of 0.12, which implies an increase in size for the latter. Overall, SVD stands out as a robust image compression method, effectively maintaining quality while achieving substantial reductions in storage requirements, making it particularly advantageous for applications prioritizing both quality and efficiency.

**## SVD Architecture and Denoising**

The Singular Value Decomposition (SVD) architecture provides a powerful framework for analyzing and compressing images. In the context of image decomposition, the singular values (SVs) represent the luminance levels of various layers within the image, while the corresponding singular vectors (SCs) define the geometric characteristics of these layers.

When applied to a high-resolution image, SVD enables the extraction of significant image content through the left singular matrix, capturing the primary structures and features. Conversely, the right singular matrix isolates the noise components, which are typically linked to the smaller singular values found in the diagonal matrix, $\Sigma$.

Thus, the largest singular values correspond to the most prominent image features, often referred to as eigenimages, while the noise components are associated with the smaller singular values. This decomposition allows for a clear distinction between meaningful image information and noise, facilitating effective compression and analysis. By leveraging SVD, one can efficiently manage and manipulate image data, ensuring that essential visual content is retained while minimizing the impact of noise.

**:::{#fig-2 layout-ncol=2}**

![Original Image](original\_image.pdf){#fig-2a}

![Extracted Noise](extracted\_noise.pdf){#fig-2b}

![Reconstructed Signal](reconstructed\_signal\_k\_40.pdf){#fig-2c}

An example with sub-figure illustrating the effectiveness of SVD in separating significant image content from noise.

**:::**

**## A starting example**

An example demonstrating the image compression using SVD is given below.

**:::{.panel-tabset}**

**## Code**

```{.matlab}

% Read and convert the image to grayscale

img = imread('amrita\_campus.jpg'); % Specify your image file

gray\_img = rgb2gray(img); % Convert to grayscale

A = double(gray\_img); % Convert to double for SVD computation

% Apply Singular Value Decomposition (SVD)

[U, S, V] = svd(A)

% Choose the number of singular values to keep for compression

k = 50; % You can adjust this value to see different compression levels

% Create a compressed version of the image using the first k singular values

S\_k = zeros(size(A)); % Initialize a zero matrix for S\_k

S\_k(1:k, 1:k) = S(1:k, 1:k); % Keep only the top k singular values

% Reconstruct the compressed image

A\_k = U\*S\_k\*V'; % Reconstruct the image from the reduced SVD

% Display the original and compressed images

figure;

subplot(1, 2, 1);

imshow(uint8(A)); % Display original image

title('Original Image');

subplot(1, 2, 2);

imshow(uint8(A\_k)); % Display compressed image

title(['Compressed Image (k = ', num2str(k), ')']);

```

**## Output**

```{python}

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

# Read and convert the image to grayscale

img = Image.open('amrita\_campus.jpg')  # Specify your image file

gray\_img = img.convert('L')  # Convert to grayscale

A = np.array(gray\_img, dtype=np.float64)  # Convert to float64 for SVD computation

original=A

# Apply Singular Value Decomposition (SVD)

U, S, Vt = np.linalg.svd(A, full\_matrices=False)

# Choose the number of singular values to keep for compression

k = 50  # You can adjust this value to see different compression levels

# Create a compressed version of the image using the first k singular values

S\_k = np.zeros\_like(A)  # Initialize a zero matrix for S\_k

S\_k[:k, :k] = np.diag(S[:k])  # Keep only the top k singular values

# Reconstruct the compressed image

A\_k = np.dot(U[:, :k], np.dot(S\_k[:k, :k], Vt[:k, :]))  # Reconstruct the image from the reduced SVD

# Display the original and compressed images

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.imshow(A, cmap='gray', vmin=0, vmax=255)  # Display original image

plt.title('Original Image')

plt.axis('off')

plt.subplot(1, 2, 2)

plt.imshow(A\_k, cmap='gray', vmin=0, vmax=255)  # Display compressed image

plt.title(f'Compressed Image (k = {k})')

plt.axis('off')

plt.show()

```

**:::**

**## Assessing quality of compression**

**:::{.panel-tabset}**

**### Code**

```{.matlab}

% Calculate Mean Squared Error (MSE)

mse = mean((A(:) - A\_k(:)).^2);

% Calculate Peak Signal-to-Noise Ratio (PSNR)

max\_pixel\_value = 255; % Maximum pixel value for 8-bit images

psnr = 10 \* log10((max\_pixel\_value^2) / mse);

% Calculate sizes

original\_size = numel(A) \* 8; % Size of the original image in bytes (double data type)

compressed\_size = (k \* (size(A, 1) + size(A, 2))) \* 8; % Size of compressed representation (U, S\_k, V)

% Display results

fprintf('Mean Squared Error (MSE): %.4f\n', mse);

fprintf('Peak Signal-to-Noise Ratio (PSNR): %.4f dB\n', psnr);

fprintf('Original Image Size: %.2f KB\n', original\_size / 1024); % Convert to KB

fprintf('Compressed Image Size: %.2f KB\n', compressed\_size / 1024); % Convert to KB

fprintf('Size Reduction: %.2f KB\n', (original\_size - compressed\_size) / 1024); % Convert to KB

```

**### Output**

```{.python}

import numpy as np

from skimage.metrics import structural\_similarity as ssim

import math

# Mean Squared Error (MSE)

mse = np.mean((A - A\_k) \*\* 2)

# Peak Signal-to-Noise Ratio (PSNR)

max\_pixel\_value = 255.0  # For an 8-bit image

psnr = 10 \* np.log10((max\_pixel\_value \*\* 2) / mse)

# Structural Similarity Index (SSIM)

ssim\_index = ssim(A, A\_k, data\_range=max\_pixel\_value)

# Compression Ratio (CR)

original\_size = A.nbytes

compressed\_size = (U[:, :k].nbytes + S\_k[:k, :k].nbytes + Vt[:k, :].nbytes)

compression\_ratio = original\_size / compressed\_size

# Normalized Cross-Correlation (NCC)

ncc = np.sum(A \* A\_k) / np.sqrt(np.sum(A \*\* 2) \* np.sum(A\_k \*\* 2))

# Display results

print(f"Mean Squared Error (MSE): {mse:.4f}")

print(f"Peak Signal-to-Noise Ratio (PSNR): {psnr:.4f} dB")

print(f"Structural Similarity Index (SSIM): {ssim\_index:.4f}")

print(f"Compression Ratio (CR): {compression\_ratio:.2f}")

print(f"Normalized Cross-Correlation (NCC): {ncc:.4f}")

print(f'Original Image Size: {original\_size / 1024:.2f} KB')  # Convert to KB

print(f'Compressed Image Size: {compressed\_size / 1024:.2f} KB')  # Convert to KB

print(f'Size Reduction: {(original\_size - compressed\_size) / 1024:.2f} KB')

```

**:::**

```{.python}

#pip install PyWavelets

import cv2

import numpy as np

import pywt

from skimage.metrics import structural\_similarity as ssim

import matplotlib.pyplot as plt

# Load and convert image to grayscale

img = cv2.imread('amrita\_campus.jpg', cv2.IMREAD\_GRAYSCALE)

# Function to display images side-by-side

def display\_images(original, compressed, title):

    plt.figure(figsize=(10,5))

    plt.subplot(1, 2, 1)

    plt.imshow(original, cmap='gray')

    plt.title("Original Image")

    plt.subplot(1, 2, 2)

    plt.imshow(compressed, cmap='gray')

    plt.title(title)

    plt.show()

# 1. Discrete Cosine Transform (DCT) Compression

def dct\_compression(img, k=50):

    img = img.astype(np.float32)

    dct\_img = cv2.dct(img)  # Apply DCT

    dct\_img[np.abs(dct\_img) < k] = 0  # Thresholding

    compressed\_img = cv2.idct(dct\_img)  # Apply inverse DCT

    display\_images(img, compressed\_img, "DCT Compressed Image")

    return compressed\_img

# 2. Wavelet Transform Compression (JPEG 2000 equivalent)

def wavelet\_compression(img, wavelet='haar', level=1, threshold=10):

    coeffs = pywt.wavedec2(img, wavelet, level=level)

    coeffs\_thresholded = []

    for c in coeffs:

        if isinstance(c, tuple):  # For detail coefficients

            coeffs\_thresholded.append(tuple(pywt.threshold(arr, threshold, mode='soft') for arr in c))

        else:  # For approximation coefficients

            coeffs\_thresholded.append(pywt.threshold(c, threshold, mode='soft'))

    compressed\_img = pywt.waverec2(coeffs\_thresholded, wavelet)

    display\_images(img, compressed\_img, "Wavelet Compressed Image")

    return compressed\_img

# 3. Fractal Compression (simplified example with downsampling)

def fractal\_compression(img, scale\_factor=0.5):

    small\_img = cv2.resize(img, (0, 0), fx=scale\_factor, fy=scale\_factor)

    compressed\_img = cv2.resize(small\_img, (img.shape[1], img.shape[0]))  # Upscale back

    display\_images(img, compressed\_img, "Fractal Compressed Image (downsampled)")

    return compressed\_img

# 4. Run-Length Encoding (RLE) Compression

def rle\_compression(img):

    pixels = img.flatten()

    rle = []

    i = 0

    while i < len(pixels):

        count = 1

        while i + 1 < len(pixels) and pixels[i] == pixels[i + 1]:

            i += 1

            count += 1

        rle.append((pixels[i], count))

        i += 1

    # Decoding RLE for display (just a simple reconstruction)

    decompressed = np.concatenate([np.full(count, val) for val, count in rle])

    decompressed\_img = decompressed.reshape(img.shape)

    display\_images(img, decompressed\_img, "RLE Compressed Image")

    return decompressed\_img

# 5. Predictive Coding Compression

def predictive\_coding\_compression(img):

    img = img.astype(np.int16)  # To handle negative differences

    prediction\_error = img.copy()

    for i in range(1, img.shape[0]):

        for j in range(1, img.shape[1]):

            prediction = (img[i-1, j] + img[i, j-1]) // 2

            prediction\_error[i, j] = img[i, j] - prediction

    compressed\_img = np.clip(prediction\_error + img.mean(), 0, 255).astype(np.uint8)

    display\_images(img, compressed\_img, "Predictive Coded Image")

    return compressed\_img

# Run all compression methods

dct\_compressed = dct\_compression(img)

wavelet\_compressed = wavelet\_compression(img)

fractal\_compressed = fractal\_compression(img)

rle\_compressed = rle\_compression(img)

predictive\_coded = predictive\_coding\_compression(img)

```

```{.python}

import numpy as np

import cv2

import pywt

import matplotlib.pyplot as plt

def load\_image(file\_path):

    img = cv2.imread(file\_path, cv2.IMREAD\_GRAYSCALE)  # Load image in grayscale

    return img

def calculate\_metrics(original, compressed):

    mse = np.mean((original - compressed) \*\* 2)

    psnr = 10 \* np.log10(255\*\*2 / mse) if mse != 0 else float('inf')

    # Update SSIM to include data\_range

    from skimage.metrics import structural\_similarity as ssim

    ssim\_value = ssim(original, compressed, data\_range=original.max() - original.min())

    return mse, psnr, ssim\_value

def svd\_compression(img, k=50):

    A = img.astype(np.float32)

    U, S, Vt = np.linalg.svd(A, full\_matrices=False)

    S\_k = np.zeros\_like(S)

    S\_k[:k] = S[:k]

    A\_k = np.dot(U, np.dot(np.diag(S\_k), Vt))

    compressed = {

        'U': U[:, :k],

        'S': S\_k[:k],

        'Vt': Vt[:k, :]

    }

    # Debug: Check sizes

    compressed\_size = compressed['U'].nbytes + compressed['S'].nbytes + compressed['Vt'].nbytes

    print(f"SVD Compressed Size: {compressed\_size} bytes")

    return A\_k

def dct\_compression(img, threshold=10):

    dct = cv2.dct(np.float32(img))

    dct[dct < threshold] = 0  # Zero out small coefficients

    idct = cv2.idct(dct)

    # Debug: Check sizes

    compressed\_size = dct.nbytes + idct.nbytes

    print(f"DCT Compressed Size: {compressed\_size} bytes")

    return idct

def wavelet\_compression(img, threshold=0.1):

    coeffs = pywt.wavedec2(img, 'haar', level=2)

    coeffs\_thresholded = [coeffs[0]] + [tuple(pywt.threshold(c, threshold, mode='soft') for c in detail) for detail in coeffs[1:]]

    # Reconstruct the image from the thresholded coefficients

    img\_reconstructed = pywt.waverec2(coeffs\_thresholded, 'haar')

    # Calculate the compressed size correctly

    compressed\_size = sum(c.nbytes for c in coeffs\_thresholded[1]) + coeffs\_thresholded[0].nbytes + img\_reconstructed.nbytes

    print(f"Wavelet Compressed Size: {compressed\_size} bytes")

    return img\_reconstructed

def display\_images(original, compressed, title1, title2):

    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)

    plt.title(title1)

    plt.imshow(original, cmap='gray')

    plt.axis('off')

    plt.subplot(1, 2, 2)

    plt.title(title2)

    plt.imshow(compressed, cmap='gray')

    plt.axis('off')

    plt.show()

def main():

    file\_path = r'D:\SVD\_project\amrita\_campus.jpg'  # Use a raw string for Windows paths

    original\_image = load\_image(file\_path)

    # Get original image size in KB

    original\_size = original\_image.nbytes / 1024  # Convert bytes to KB

    # Perform compression

    svd\_compressed = svd\_compression(original\_image)

    dct\_compressed = dct\_compression(original\_image)

    wavelet\_compressed = wavelet\_compression(original\_image)

    # Calculate metrics

    svd\_mse, svd\_psnr, svd\_ssim = calculate\_metrics(original\_image, svd\_compressed)

    dct\_mse, dct\_psnr, dct\_ssim = calculate\_metrics(original\_image, dct\_compressed)

    wavelet\_mse, wavelet\_psnr, wavelet\_ssim = calculate\_metrics(original\_image, wavelet\_compressed)

    # Calculate Compressed Sizes and additional metrics

    svd\_compressed\_size = svd\_compressed.nbytes / 1024  # Size in KB

    dct\_compressed\_size = dct\_compressed.nbytes / 1024  # Size in KB

    wavelet\_compressed\_size = wavelet\_compressed.nbytes / 1024  # Size in KB

    svd\_cr = original\_size / svd\_compressed\_size

    dct\_cr = original\_size / dct\_compressed\_size

    wavelet\_cr = original\_size / wavelet\_compressed\_size

    # NCC calculation

    def normalized\_cross\_correlation(original, compressed):

        return np.sum(original \* compressed) / (np.linalg.norm(original) \* np.linalg.norm(compressed))

    svd\_ncc = normalized\_cross\_correlation(original\_image, svd\_compressed)

    dct\_ncc = normalized\_cross\_correlation(original\_image, dct\_compressed)

    wavelet\_ncc = normalized\_cross\_correlation(original\_image, wavelet\_compressed)

    # Size Reduction

    svd\_size\_reduction = original\_size - svd\_compressed\_size

    dct\_size\_reduction = original\_size - dct\_compressed\_size

    wavelet\_size\_reduction = original\_size - wavelet\_compressed\_size

    # Print Comparison Table

    print(f"{'Method':<10} {'MSE':<20} {'PSNR (dB)':<15} {'SSIM':<15} {'CR':<10} {'NCC':<10} {'Compressed Size (KB)':<25} {'Size Reduction (KB)':<20}")

    print(f"{'SVD':<10} {svd\_mse:<20.4f} {svd\_psnr:<15.4f} {svd\_ssim:<15.4f} {svd\_cr:<10.2f} {svd\_ncc:<10.4f} {svd\_compressed\_size:<25.2f} {svd\_size\_reduction:<20.2f}")

    print(f"{'DCT':<10} {dct\_mse:<20.4f} {dct\_psnr:<15.4f} {dct\_ssim:<15.4f} {dct\_cr:<10.2f} {dct\_ncc:<10.4f} {dct\_compressed\_size:<25.2f} {dct\_size\_reduction:<20.2f}")

    print(f"{'Wavelet':<10} {wavelet\_mse:<20.4f} {wavelet\_psnr:<15.4f} {wavelet\_ssim:<15.4f} {wavelet\_cr:<10.2f} {wavelet\_ncc:<10.4f} {wavelet\_compressed\_size:<25.2f} {wavelet\_size\_reduction:<20.2f}")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

```

```{.python}

import numpy as np

import cv2

import pywt

import matplotlib.pyplot as plt

from skimage.metrics import structural\_similarity as ssim

def load\_image(file\_path):

    img = cv2.imread(file\_path, cv2.IMREAD\_GRAYSCALE)  # Load image in grayscale

    return img

def calculate\_metrics(original, compressed):

    mse = np.mean((original - compressed) \*\* 2)

    psnr = 10 \* np.log10(255\*\*2 / mse) if mse != 0 else float('inf')

    ssim\_value = ssim(original, compressed, data\_range=original.max() - original.min())

    return mse, psnr, ssim\_value

def svd\_compression(img, k=20):

    A = img.astype(np.float32)

    U, S, Vt = np.linalg.svd(A, full\_matrices=False)

    S\_k = np.zeros\_like(S)

    S\_k[:k] = S[:k]

    A\_k = np.dot(U[:, :k], np.dot(np.diag(S\_k[:k]), Vt[:k, :]))

    compressed = {

        'U': U[:, :k],

        'S': S\_k[:k],

        'Vt': Vt[:k, :]

    }

    # Calculate the size of compressed data

    compressed\_size = compressed['U'].nbytes + compressed['S'].nbytes + compressed['Vt'].nbytes

    print(f"SVD Compressed Size: {compressed\_size} bytes")

    return A\_k, compressed\_size

def dct\_compression(img, threshold=10):

    dct = cv2.dct(np.float32(img))

    dct[dct < threshold] = 0  # Zero out small coefficients

    # Count non-zero coefficients

    non\_zero\_coeffs = np.count\_nonzero(dct)

    compressed\_size = dct.nbytes - (dct.size - non\_zero\_coeffs) \* dct.dtype.itemsize  # Size excluding zeros

    idct = cv2.idct(dct)  # Reconstruct the image (not needed for size calculation)

    print(f"DCT Compressed Size: {compressed\_size} bytes")

    return idct, compressed\_size

def wavelet\_compression(image, wavelet='haar', threshold=0.2):

    # Perform 2D wavelet decomposition

    coeffs = pywt.wavedec2(image, wavelet)

    # Threshold the detail coefficients

    coeffs\_thresholded = list(coeffs)

    for i in range(1, len(coeffs\_thresholded)):

        coeffs\_thresholded[i] = tuple(pywt.threshold(c, threshold, mode='soft') for c in coeffs\_thresholded[i])

    # Calculate compressed size

    compressed\_size = sum(np.prod(c.shape) \* c.dtype.itemsize for detail in coeffs\_thresholded[1:] for c in detail) + coeffs\_thresholded[0].nbytes

    # Reconstruct the image

    img\_reconstructed = pywt.waverec2(coeffs\_thresholded, wavelet)

    return img\_reconstructed, compressed\_size

def display\_images(original, compressed, title1, title2):

    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)

    plt.title(title1)

    plt.imshow(original, cmap='gray')

    plt.axis('off')

    plt.subplot(1, 2, 2)

    plt.title(title2)

    plt.imshow(compressed, cmap='gray')

    plt.axis('off')

    plt.show()

def main():

    file\_path = r'D:\SVD\_project\amrita\_campus.jpg'  # Use a raw string for Windows paths

    original\_image = load\_image(file\_path)

    # Get original image size in bytes

    original\_size = original\_image.nbytes  # Size in bytes

    # Perform compression

    svd\_compressed, svd\_compressed\_size = svd\_compression(original\_image)

    dct\_compressed, dct\_compressed\_size = dct\_compression(original\_image)

    wavelet\_compressed, wavelet\_compressed\_size = wavelet\_compression(original\_image)

    # Calculate metrics

    svd\_mse, svd\_psnr, svd\_ssim = calculate\_metrics(original\_image, svd\_compressed)

    dct\_mse, dct\_psnr, dct\_ssim = calculate\_metrics(original\_image, dct\_compressed)

    wavelet\_mse, wavelet\_psnr, wavelet\_ssim = calculate\_metrics(original\_image, wavelet\_compressed)

    # Calculate Compressed Sizes and additional metrics

    svd\_cr = original\_size / svd\_compressed\_size

    dct\_cr = original\_size / dct\_compressed\_size

    wavelet\_cr = original\_size / wavelet\_compressed\_size

    # NCC calculation

    def normalized\_cross\_correlation(original, compressed):

        return np.sum(original \* compressed) / (np.linalg.norm(original) \* np.linalg.norm(compressed))

    svd\_ncc = normalized\_cross\_correlation(original\_image, svd\_compressed)

    dct\_ncc = normalized\_cross\_correlation(original\_image, dct\_compressed)

    wavelet\_ncc = normalized\_cross\_correlation(original\_image, wavelet\_compressed)

    # Size Reduction

    svd\_size\_reduction = original\_size - svd\_compressed\_size

    dct\_size\_reduction = original\_size - dct\_compressed\_size

    wavelet\_size\_reduction = original\_size - wavelet\_compressed\_size

    # Print Comparison Table

    print(f"{'Method':<10} {'MSE':<20} {'PSNR (dB)':<15} {'SSIM':<15} {'CR':<10} {'NCC':<10} {'Compressed Size (KB)':<25} {'Size Reduction (KB)':<20}")

    print(f"{'SVD':<10} {svd\_mse:<20.4f} {svd\_psnr:<15.4f} {svd\_ssim:<15.4f} {svd\_cr:<10.2f} {svd\_ncc:<10.4f} {svd\_compressed\_size/1024:<25.2f} {svd\_size\_reduction/1024:<20.2f}")

    print(f"{'DCT':<10} {dct\_mse:<20.4f} {dct\_psnr:<15.4f} {dct\_ssim:<15.4f} {dct\_cr:<10.2f} {dct\_ncc:<10.4f} {dct\_compressed\_size/1024:<25.2f} {dct\_size\_reduction/1024:<20.2f}")

    print(f"{'Wavelet':<10} {wavelet\_mse:<20.4f} {wavelet\_psnr:<15.4f} {wavelet\_ssim:<15.4f} {wavelet\_cr:<10.2f} {wavelet\_ncc:<10.4f} {wavelet\_compressed\_size/1024:<25.2f} {wavelet\_size\_reduction/1024:<20.2f}")

    display\_images(original\_image, svd\_compressed,"original","compressed")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

```

```{.python}

import numpy as np

from skimage import io, color

import matplotlib.pyplot as plt

# Load the image and convert it to grayscale

image = io.imread('TestImage.jpg')

if image.ndim == 3:

    image = color.rgb2gray(image)

image = image.astype(float)

# Perform SVD

U, S, Vt = np.linalg.svd(image, full\_matrices=False)

# Set number of components to visualize

num\_components = 2

# Function to normalize and visualize singular vectors

def visualize\_singular\_vectors(vectors, title, n\_components, shape):

    fig, axs = plt.subplots(1, n\_components, figsize=(15, 5))

    fig.suptitle(title, fontsize=16)

    for i in range(n\_components):

        vector = vectors[:, i] if title == 'Column Space (U)' else vectors[i, :]

        # Normalize vector for better visibility

        normalized\_vector = (vector - np.min(vector)) / (np.max(vector) - np.min(vector))

        axs[i].imshow(normalized\_vector.reshape(shape), cmap='gray', aspect='auto')

        axs[i].axis('off')

        axs[i].set\_title(f'Component {i+1}')

    plt.show()

# Visualize Column Space (U matrix columns)

visualize\_singular\_vectors(U, "Column Space (U)", num\_components, (image.shape[0], 1))

# Visualize Row Space (V^T matrix rows)

visualize\_singular\_vectors(Vt, "Row Space (V^T)", num\_components, (1, image.shape[1]))

# Plot Singular Values

plt.figure(figsize=(8, 6))

plt.plot(np.log(1+S), 'o-', label="Singular Values")

plt.xlabel("Index")

plt.ylabel("Singular Value")

plt.title("Singular Values Plot")

plt.legend()

plt.grid()

plt.show()

```

**### Ploting the signal and noise part of an image**

```{.python}

import numpy as np

import matplotlib.pyplot as plt

from skimage import io, color

# Load the image and convert it to grayscale

image = io.imread('.TestImage.jpg')

if image.ndim == 3:

    image = color.rgb2gray(image)

# Perform SVD

U, S, VT = np.linalg.svd(image, full\_matrices=False)

# Number of components to keep

k = 40  # Adjust this for more or fewer components

# Reconstruct the signal part

S\_k = np.zeros\_like(S)  # Create a zero array for singular values

S\_k[:k] = S[:k]  # Keep the largest k singular values

# Reconstruct the signal image

reconstructed\_signal = U @ np.diag(S\_k) @ VT

# Extract noise

noise = image - reconstructed\_signal

# Convert images to uint8 for saving

image\_uint8 = (image \* 255).astype(np.uint8)

reconstructed\_signal\_uint8 = (reconstructed\_signal \* 255).astype(np.uint8)

noise\_uint8 = (noise \* 255).astype(np.uint8)

# Save each image as a PDF

io.imsave('original\_image.pdf', image\_uint8)

io.imsave('reconstructed\_signal\_k\_{}.pdf'.format(k), reconstructed\_signal\_uint8)

io.imsave('extracted\_noise.pdf', noise\_uint8)

# Plotting

plt.figure(figsize=(15, 10))

plt.subplot(1, 3, 1)

plt.title('Original Image')

plt.imshow(image, cmap='gray')

plt.axis('off')

plt.subplot(1, 3, 2)

plt.title('Reconstructed Signal (k={})'.format(k))

plt.imshow(reconstructed\_signal, cmap='gray')

plt.axis('off')

plt.subplot(1, 3, 3)

plt.title('Extracted Noise')

plt.imshow(noise, cmap='gray')

plt.axis('off')

plt.tight\_layout()

# Save the entire figure as a PDF

plt.savefig('comparison\_plot.pdf', bbox\_inches='tight')

plt.show()

```

**### Compression quality with different values of k**

```{.python}

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from skimage import io, color

from skimage.metrics import peak\_signal\_noise\_ratio as psnr

from skimage.metrics import structural\_similarity as ssim

# Load the image and convert it to grayscale

image = io.imread('TestImage.jpg')

if image.ndim == 3:

    image = color.rgb2gray(image)

# Initialize a list to store results

results = []

# Define a range of k values

k\_values = [1, 5, 10, 20, 50, 100, 200, 400, 600, 800, 1000]

for k in k\_values:

    # Perform SVD

    U, S, VT = np.linalg.svd(image, full\_matrices=False)

    # Reconstruct the signal part with k components

    S\_k = np.zeros\_like(S)

    S\_k[:k] = S[:k]

    reconstructed\_signal = U @ np.diag(S\_k) @ VT

    # Compute PSNR and SSIM

    current\_psnr = psnr(image, reconstructed\_signal)

    # Set data\_range for SSIM

    data\_range = 1  # Use 255 if your image is in the range [0, 255]

    current\_ssim = ssim(image, reconstructed\_signal, data\_range=data\_range)

    # Append results

    results.append({'k': k, 'PSNR': current\_psnr, 'SSIM': current\_ssim})

# Create a DataFrame from the results

results\_df = pd.DataFrame(results)

# Display the DataFrame as a table

print(results\_df)

# Optionally, save the results to a CSV file

results\_df.to\_csv('psnr\_ssim\_variation.csv', index=False)

# Plot the results

plt.figure(figsize=(12, 6))

plt.plot(results\_df['k'], results\_df['PSNR'], marker='o', label='PSNR')

plt.plot(results\_df['k'], results\_df['SSIM'], marker='o', label='SSIM')

plt.xscale('log')  # Log scale for better visualization

plt.xlabel('Number of Components (k)')

plt.ylabel('Value')

plt.title('Variation of PSNR and SSIM with Different k Values')

plt.legend()

plt.grid()

plt.savefig('psnr\_ssim\_variation\_plot.pdf')

plt.show()

```

**### Creating images from the left singular and right singular matrices**

```{python}

import numpy as np

from skimage import io, color

import matplotlib.pyplot as plt

# Load the image and convert it to grayscale

image = io.imread('TestImage.jpg')

if image.ndim == 3:

    image = color.rgb2gray(image)

image = image.astype(float)

# Perform SVD

U, S, Vt = np.linalg.svd(image, full\_matrices=False)

# Normalize function for better visualization

def normalize\_matrix(matrix):

    return (matrix - np.min(matrix)) / (np.max(matrix) - np.min(matrix))

# Save the original image

plt.imshow(image, cmap='gray')

plt.axis('off')

plt.title('Original Image')

plt.savefig('original\_image.pdf', format='pdf', bbox\_inches='tight')

plt.close()

# Save the left singular matrix (U)

plt.imshow(normalize\_matrix(U), cmap='gray', aspect='auto')

plt.axis('off')

plt.title('Left Singular Matrix (U)')

plt.savefig('left\_singular\_matrix\_U.pdf', format='pdf', bbox\_inches='tight')

plt.close()

# Save the right singular matrix (V^T)

plt.imshow(normalize\_matrix(Vt), cmap='gray', aspect='auto')

plt.axis('off')

plt.title('Right Singular Matrix (V^T)')

plt.savefig('right\_singular\_matrix\_Vt.pdf', format='pdf', bbox\_inches='tight')

plt.close()

```

**### Ploting the Dominant and sub dominant components**

```{python}

import numpy as np

from skimage import io, color

import matplotlib.pyplot as plt

# Load the image and convert it to grayscale

image = io.imread('TestImage.jpg')

if image.ndim == 3:

    image = color.rgb2gray(image)

image = image.astype(float)

# Perform SVD

U, S, Vt = np.linalg.svd(image, full\_matrices=False)

# Number of components for partial reconstruction

k = 50  # Choose k to retain enough detail without full reconstruction

# Reconstructing left singular matrix U and right singular matrix V^T with k components

U\_reconstructed = U[:, :k] @ np.diag(S[:k])

Vt\_reconstructed = np.diag(S[:k]) @ Vt[:k, :]

# Set figure size based on the original image dimensions

figsize = (image.shape[1] / 100, image.shape[0] / 100)

# Function to save reconstructed singular matrix images with the same dimensions as the original

def save\_reconstructed\_matrix(matrix, title, filename):

    fig, ax = plt.subplots(figsize=figsize)

    # Normalize for better visualization

    #normalized\_matrix = (matrix - np.min(matrix)) / (np.max(matrix) - np.min(matrix))

    ax.imshow(matrix, cmap='gray', aspect='auto')

    ax.axis('off')

    plt.title(title)

    plt.savefig(filename, bbox\_inches='tight', pad\_inches=0)

    plt.close(fig)

# Save the original image in grayscale for reference

plt.imsave("original\_image.pdf", image, cmap='gray')

# Save reconstructed left singular matrix U

save\_reconstructed\_matrix(U\_reconstructed, "Reconstructed Left Singular Matrix (U)", "left\_singular\_matrix\_U\_traces.pdf")

# Save reconstructed right singular matrix V^T

save\_reconstructed\_matrix(Vt\_reconstructed, "Reconstructed Right Singular Matrix (V^T)", "right\_singular\_matrix\_Vt\_traces.pdf")

```

**### Code for Denoising with SVD**

```{python}

import numpy as np

from skimage import io, color

import matplotlib.pyplot as plt

# Load the original image and convert it to grayscale

image = io.imread('TestImage.jpg')

if image.ndim == 3:

    image = color.rgb2gray(image)

image = image.astype(float)

# Check the original image statistics

print("Original Image - Min:", np.min(image), "Max:", np.max(image))

# Step 1: Add Gaussian noise to the image

noise\_variance = 0.1  # Set this value based on your requirements

# Generate Gaussian noise

noise = np.random.normal(0, np.sqrt(noise\_variance), image.shape)

# Add noise to the original image

noisy\_image = image + noise

# Ensure the noisy image values are clipped to [0, 1]

noisy\_image = np.clip(noisy\_image, 0, 1)

# Check the noisy image statistics

print("Noisy Image - Min:", np.min(noisy\_image), "Max:", np.max(noisy\_image))

# Step 2: Perform SVD on the noisy image

U, S, Vt = np.linalg.svd(noisy\_image, full\_matrices=False)

# Step 3: Filter Noise by Truncating Smaller Singular Values

# Adjust `k` to retain more singular values, which should improve quality

k = 50  # Adjust this value as needed

S\_denoised = np.zeros\_like(S)

S\_denoised[:k] = S[:k]  # Keep the top `k` singular values

# Step 4: Reconstruct the denoised image

denoised\_image = U @ np.diag(S\_denoised) @ Vt

# Ensure the denoised image values are clipped to [0, 1]

denoised\_image = np.clip(denoised\_image, 0, 1)

# Check the denoised image statistics

print("Denoised Image - Min:", np.min(denoised\_image), "Max:", np.max(denoised\_image))

# Plotting and saving the results

fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Original Image

axs[0].imshow(image, cmap='gray')

axs[0].set\_title("Original Image")

axs[0].axis('off')

# Noisy Image

axs[1].imshow(noisy\_image, cmap='gray')

axs[1].set\_title("Noisy Image")

axs[1].axis('off')

# Denoised Image (SVD)

axs[2].imshow(denoised\_image, cmap='gray')

axs[2].set\_title("Denoised Image (SVD)")

axs[2].axis('off')

plt.tight\_layout()

plt.show()

# Saving each image as a standalone PDF

plt.imsave("original\_image.pdf", image, cmap='gray')

plt.imsave("noisy\_image.pdf", noisy\_image, cmap='gray')

plt.imsave("denoised\_image.pdf", denoised\_image, cmap='gray')

```

**## Denoising BSD400 Dataet using SVD**

```{python}

import numpy as np

from skimage import io, color, metrics

import matplotlib.pyplot as plt

# Load the original image from the BSD400 dataset and convert it to grayscale

image = io.imread('DenoiseImage.png')

if image.ndim == 3:

    image = color.rgb2gray(image)

image = image.astype(float)

# Normalize the image to the range [0, 1]

image -= np.min(image)

image /= np.max(image)

# Step 1: Add Gaussian noise to the image

noise\_variance = 0.001  # Use a smaller noise variance relative to the normalized image

# Generate Gaussian noise scaled by the maximum value of the image

noise = np.random.normal(0, np.sqrt(noise\_variance \* np.max(image)), image.shape)

# Add noise to the original image

noisy\_image = image + noise

# Ensure the noisy image values are clipped to [0, 1]

noisy\_image = np.clip(noisy\_image, 0, 1)

# Step 2: Perform SVD on the noisy image

U, S, Vt = np.linalg.svd(noisy\_image, full\_matrices=False)

# Step 3: Determine a threshold for singular values

threshold =0.618\*np.mean(S)  # Example threshold: mean of singular values

S\_denoised = np.where(S > threshold, S, 0)  # Set small singular values to zero

# Step 4: Reconstruct the denoised image

denoised\_image = U @ np.diag(S\_denoised) @ Vt

# Ensure the denoised image values are clipped to [0, 1]

denoised\_image = np.clip(denoised\_image, 0, 1)

data\_range = 1.0  # If your images are in the range [0, 1]. Use 255 if they are in [0, 255].

# Step 5: Calculate PSNR and SSIM to assess denoising performance

psnr\_noisy = metrics.peak\_signal\_noise\_ratio(image, noisy\_image, data\_range=data\_range)

ssim\_noisy = metrics.structural\_similarity(image, noisy\_image, data\_range=data\_range)

psnr\_denoised = metrics.peak\_signal\_noise\_ratio(image, denoised\_image, data\_range=data\_range)

ssim\_denoised = metrics.structural\_similarity(image, denoised\_image, data\_range=data\_range)

print(f"PSNR (Noisy Image): {psnr\_noisy:.2f}")

print(f"SSIM (Noisy Image): {ssim\_noisy:.4f}")

print(f"PSNR (Denoised Image): {psnr\_denoised:.2f}")

print(f"SSIM (Denoised Image): {ssim\_denoised:.4f}")

# Plotting and saving the results

fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Original Image

axs[0].imshow(image, cmap='gray')

axs[0].set\_title("Original Image")

axs[0].axis('off')

# Noisy Image

axs[1].imshow(noisy\_image, cmap='gray')

axs[1].set\_title(f"Noisy Image\nPSNR: {psnr\_noisy:.2f}, SSIM: {ssim\_noisy:.4f}")

axs[1].axis('off')

# Denoised Image (SVD)

axs[2].imshow(denoised\_image, cmap='gray')

axs[2].set\_title(f"Denoised Image (SVD)\nPSNR: {psnr\_denoised:.2f}, SSIM: {ssim\_denoised:.4f}")

axs[2].axis('off')

plt.tight\_layout()

plt.show()

# Saving each image as a standalone PDF

plt.imsave("BSDoriginal\_image.pdf", image, cmap='gray')

plt.imsave("BSDnoisy\_image.pdf", noisy\_image, cmap='gray')

plt.imsave("BSDdenoised\_image.pdf", denoised\_image, cmap='gray')

```

**### Assesing quality of SVD denoiser**

```{python}

import numpy as np

from skimage import io, color, metrics

import matplotlib.pyplot as plt

# Load the original image and convert it to grayscale

image = io.imread('TestImage.jpg')

if image.ndim == 3:

    image = color.rgb2gray(image)

image = image.astype(float)

# Check the original image statistics

print("Original Image - Min:", np.min(image), "Max:", np.max(image))

# Step 1: Add Gaussian noise to the image

noise\_variance = 0.1  # Set this value based on your requirements

# Generate Gaussian noise

noise = np.random.normal(0, np.sqrt(noise\_variance), image.shape)

# Add noise to the original image

noisy\_image = image + noise

# Ensure the noisy image values are clipped to [0, 1]

noisy\_image = np.clip(noisy\_image, 0, 1)

# Check the noisy image statistics

print("Noisy Image - Min:", np.min(noisy\_image), "Max:", np.max(noisy\_image))

# Step 2: Perform SVD on the noisy image

U, S, Vt = np.linalg.svd(noisy\_image, full\_matrices=False)

# Step 3: Filter Noise by Truncating Smaller Singular Values

# Adjust `k` to retain more singular values, which should improve quality

k = 50  # Adjust this value as needed

S\_denoised = np.zeros\_like(S)

S\_denoised[:k] = S[:k]  # Keep the top `k` singular values

# Step 4: Reconstruct the denoised image

denoised\_image = U @ np.diag(S\_denoised) @ Vt

# Ensure the denoised image values are clipped to [0, 1]

denoised\_image = np.clip(denoised\_image, 0, 1)

# Check the denoised image statistics

print("Denoised Image - Min:", np.min(denoised\_image), "Max:", np.max(denoised\_image))

data\_range = 1.0  # If your images are in the range [0, 1]. Use 255 if they are in [0, 255].

# Step 5: Calculate PSNR and SSIM to assess denoising performance

psnr\_noisy = metrics.peak\_signal\_noise\_ratio(image, noisy\_image, data\_range=data\_range)

ssim\_noisy = metrics.structural\_similarity(image, noisy\_image, data\_range=data\_range)

psnr\_denoised = metrics.peak\_signal\_noise\_ratio(image, denoised\_image, data\_range=data\_range)

ssim\_denoised = metrics.structural\_similarity(image, denoised\_image, data\_range=data\_range)

print(f"PSNR (Noisy Image): {psnr\_noisy:.2f}")

print(f"SSIM (Noisy Image): {ssim\_noisy:.4f}")

print(f"PSNR (Denoised Image): {psnr\_denoised:.2f}")

print(f"SSIM (Denoised Image): {ssim\_denoised:.4f}")

# Plotting and saving the results

fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Original Image

axs[0].imshow(image, cmap='gray')

axs[0].set\_title("Original Image")

axs[0].axis('off')

# Noisy Image

axs[1].imshow(noisy\_image, cmap='gray')

axs[1].set\_title(f"Noisy Image\nPSNR: {psnr\_noisy:.2f}, SSIM: {ssim\_noisy:.4f}")

axs[1].axis('off')

# Denoised Image (SVD)

axs[2].imshow(denoised\_image, cmap='gray')

axs[2].set\_title(f"Denoised Image (SVD)\nPSNR: {psnr\_denoised:.2f}, SSIM: {ssim\_denoised:.4f}")

axs[2].axis('off')

plt.tight\_layout()

plt.show()

# Optionally save each image as a standalone PDF

plt.imsave("original\_image.pdf", image, cmap='gray')

plt.imsave("noisy\_image.pdf", noisy\_image, cmap='gray')

plt.imsave("denoised\_image.pdf", denoised\_image, cmap='gray')

```

**### Ploting the correlation between regenerated images over k**

```{python}

import numpy as np

from skimage import io, color

import matplotlib.pyplot as plt

# Load the original grayscale image

image = io.imread('TestImage.jpg')

if image.ndim == 3:

    image = color.rgb2gray(image)

image = image.astype(float)

# Perform SVD on the original image

U, S, Vt = np.linalg.svd(image, full\_matrices=False)

# List of k-values to experiment with

k\_values = [10, 20, 50, 100, 200, 400, 600]

# Flatten the original image to compare correlations

original\_flattened = image.flatten()

# Array to store correlations with the original image for each k

correlations = []

# Reconstruct images using different numbers of singular values and compute correlations

for k in k\_values:

    S\_k = np.zeros\_like(S)

    S\_k[:k] = S[:k]  # Retain only the top k singular values

    reconstructed\_image = U @ np.diag(S\_k) @ Vt

    reconstructed\_flattened = reconstructed\_image.flatten()

    # Calculate correlation with the original image

    corr = np.corrcoef(original\_flattened, reconstructed\_flattened)[0, 1]

    correlations.append(corr)

# Plotting k-values against their corresponding correlations

plt.figure(figsize=(8, 5))

plt.plot(k\_values, correlations, marker='o', linestyle='-')

plt.xlabel("Number of Singular Values (k)")

plt.ylabel("Correlation with Original Image")

plt.title("Correlation between Reconstructed Images and Original Image")

plt.grid()

plt.show()

```

**### Image Forensic with SVD**

```{python}

import numpy as np

import matplotlib.pyplot as plt

from skimage import io, color

def preprocess\_image(image):

    if image.ndim == 3:

        if image.shape[2] == 4:  # Check for RGBA

            image = image[:, :, :3]  # Take RGB only

        gray\_image = color.rgb2gray(image)

    else:

        gray\_image = image  # Already grayscale

    return gray\_image.astype(float)

def embed\_watermark(image, watermark, alpha=0.1):

    U, S, Vt = np.linalg.svd(image, full\_matrices=False)

    watermark\_resized = np.resize(watermark, S.shape)

    S\_watermarked = S + alpha \* watermark\_resized

    watermarked\_image = np.dot(U, np.dot(np.diag(S\_watermarked), Vt))

    return np.clip(watermarked\_image, 0, 1)

def extract\_watermark(original\_image, watermarked\_image, alpha=0.1):

    U1, S1, Vt1 = np.linalg.svd(original\_image, full\_matrices=False)

    U2, S2, Vt2 = np.linalg.svd(watermarked\_image, full\_matrices=False)

    extracted\_watermark = (S2 - S1) / alpha

    return extracted\_watermark

# Load images

host\_image = io.imread('TESTIMAGE.png')

host\_image = preprocess\_image(host\_image)

watermark\_image = io.imread('amritha\_TL.png')

watermark\_image = preprocess\_image(watermark\_image)

# Ensure images are in [0, 1] range

host\_image = np.clip(host\_image, 0, 1)

watermark\_image = np.clip(watermark\_image, 0, 1)

# Embed watermark

watermarked\_image = embed\_watermark(host\_image, watermark\_image, alpha=0.1)

# Extract watermark

extracted\_watermark = extract\_watermark(host\_image, watermarked\_image, alpha=0.1)

# Reshape the extracted watermark to the size of the original watermark image

extracted\_watermark = np.clip(extracted\_watermark, 0, 1)  # Ensure valid pixel range

extracted\_watermark = np.resize(extracted\_watermark, watermark\_image.shape)  # Resize to match original watermark

# Plotting results

fig, axs = plt.subplots(1, 3, figsize=(15, 5))

axs[0].imshow(host\_image, cmap='gray')

axs[0].set\_title("Original Image")

axs[0].axis('off')

axs[1].imshow(watermarked\_image, cmap='gray')

axs[1].set\_title("Watermarked Image")

axs[1].axis('off')

axs[2].imshow(extracted\_watermark, cmap='gray')

axs[2].set\_title("Extracted Watermark")

axs[2].axis('off')

plt.tight\_layout()

plt.show()

```

```{python}

import numpy as np

import cv2

from scipy.linalg import svd

import matplotlib.pyplot as plt

def read\_image(filename):

    # Read the image using OpenCV

    img = cv2.imread(filename, cv2.IMREAD\_UNCHANGED)  # Read with unchanged channels

    if img is None:

        raise ValueError("Image not found or unable to read.")

    # Check if the image has an alpha channel

    if img.shape[2] == 4:  # If there are 4 channels (RGBA)

        img = cv2.cvtColor(img, cv2.COLOR\_BGRA2BGR)  # Convert to BGR without alpha

    # Convert to grayscale if it's a 3-channel image

    if len(img.shape) == 3 and img.shape[2] == 3:

        img = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

    return img

# Load host image

host\_image = read\_image('TESTIMAGE.png')  # Replace with your actual path

host\_image = cv2.resize(host\_image, (256, 256))  # Resize to 256x256

host\_image = host\_image.astype(np.float64)  # Convert to double

# Perform SVD

Uimg, Simg, Vimg = svd(host\_image)

# Read watermark

watermark\_image = read\_image('copyright1.png')  # Replace with your actual path

watermark\_image = cv2.resize(watermark\_image, (256, 256))  # Resize to 256x256

alfa = 0.1  # Alpha value for watermark embedding

watermark\_image = watermark\_image.astype(np.float64)  # Convert to double

# Ensure watermark is in the same shape as Simg

if watermark\_image.shape[0] > len(Simg):

    watermark\_image = watermark\_image[:len(Simg), :]

# Modify the singular values by adding the scaled watermark

for i in range(len(Simg)):

    Simg[i] += alfa \* watermark\_image[i, 0]  # Update singular values

# Reconstruct the watermarked image

Simg\_matrix = np.diag(Simg)  # Create a diagonal matrix from singular values

watermarked\_image = np.dot(Uimg, np.dot(Simg\_matrix, Vimg))

# Normalize the watermarked image to the range [0, 255]

watermarked\_image = np.clip(watermarked\_image, 0, 255).astype(np.uint8)

# Display the original host image

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

plt.imshow(host\_image, cmap='gray')

plt.title('The Original Image')

plt.axis('off')

# Display the watermarked image

plt.subplot(1, 2, 2)

plt.imshow(watermarked\_image, cmap='gray')

plt.title('Watermarked Image')

plt.axis('off')

plt.tight\_layout()

plt.show()

```

**## Extraction part**

```{python}

import numpy as np

import cv2

from scipy.linalg import svd

import matplotlib.pyplot as plt

def read\_image(filename):

    img = cv2.imread(filename, cv2.IMREAD\_UNCHANGED)

    if img is None:

        raise ValueError(f"Image not found or unable to read: {filename}")

    if len(img.shape) == 3 and img.shape[2] == 3:

        img = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

    return img

def embed\_watermark(host\_image, watermark\_image, alfa=0.1):

    Uimg, Simg, Vimg = svd(host\_image)

    if watermark\_image.shape[0] > len(Simg):

        raise ValueError("Watermark size exceeds singular values length.")

    for i in range(len(Simg)):

        Simg[i] += alfa \* watermark\_image[i, 0]

    Simg\_matrix = np.diag(Simg)

    watermarked\_image = np.dot(Uimg, np.dot(Simg\_matrix, Vimg))

    return watermarked\_image, Simg, Uimg, Vimg

def extract\_watermark(watermarked\_image, original\_singular\_values, alfa=0.1, watermark\_shape=None):

    U\_Wimg, S\_Wimg, V\_Wimg = svd(watermarked\_image)

    # Calculate the extracted watermark

    extracted\_watermark = (S\_Wimg - original\_singular\_values) / alfa

    if watermark\_shape is not None:

        extracted\_watermark = extracted\_watermark.reshape(watermark\_shape)

    # Normalize to uint8 range

    extracted\_watermark = np.clip(extracted\_watermark, 0, 255).astype(np.uint8)

    return extracted\_watermark

def calculate\_psnr(original\_image, watermarked\_image):

    mse = np.mean((original\_image.astype(np.float64) - watermarked\_image.astype(np.float64)) \*\* 2)

    if mse == 0:

        return 100

    max\_pixel\_value = 255.0

    psnr = 10 \* np.log10((max\_pixel\_value \*\* 2) / mse)

    return psnr

# Load host image

host\_image = read\_image('TESTIMAGE.png')  # Replace with your actual path

host\_image = cv2.resize(host\_image, (256, 256)).astype(np.float64)

# Load watermark image

watermark\_image = read\_image('copyright1.png')  # Replace with your actual path

watermark\_image = cv2.resize(watermark\_image, (256, 1)).astype(np.float64)

# Embed watermark

alfa = 0.1

watermarked\_image, Simg, Uimg, Vimg = embed\_watermark(host\_image, watermark\_image, alfa)

# Extract watermark

extracted\_watermark = extract\_watermark(watermarked\_image, Simg, alfa, watermark\_shape=(256, 1))

# Calculate PSNR

psnr\_value = calculate\_psnr(host\_image, watermarked\_image)

# Display the images

plt.figure(figsize=(18, 6))

plt.subplot(1, 3, 1)

plt.imshow(host\_image, cmap='gray')

plt.title('The Original Image')

plt.axis('off')

plt.subplot(1, 3, 2)

plt.imshow(watermarked\_image, cmap='gray')

plt.title('Watermarked Image')

plt.axis('off')

plt.subplot(1, 3, 3)

plt.imshow(extracted\_watermark, cmap='gray')

plt.title('Extracted Watermark')

plt.axis('off')

plt.tight\_layout()

plt.show()

print(f"PSNR between the original and watermarked image: {psnr\_value:.2f} dB")

```