---

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, Σ, 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, Σ, and

V^T. Here, U is an $m \times m$ orthogonal matrix containing the left singular vectors, Σ 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, Σ_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 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 μ, σ, and σ_{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.

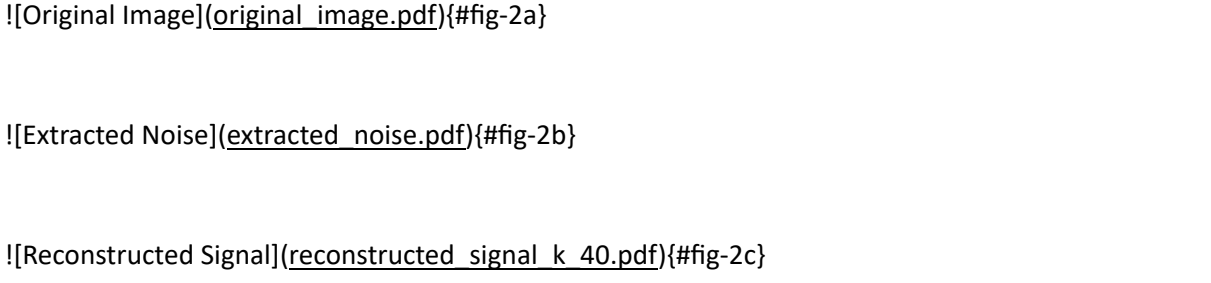
SVD Architecture and Denoising

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, Σ.

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}



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")
```