2025

Project 3: SVD Image Compression

STUDENT ID: 800777831

GREGORY E. SCHWARTZ

Abstract

This project documents the manual implementation of a Singular Value Decomposition (SVD) algorithm for compressing a full-color image, using only custom-built numerical routines. The core objectives were to (1) apply the theoretical framework outlined in Section 12.4.3 of Sauer's Numerical Analysis, (2) process an actual photograph of the student (as the sole team member), and (3) evaluate the effectiveness of the compression for multiple values of p, the number of retained singular values.

To meet the assignment requirements, all computations were performed without calling any built-in SVD routines. Instead, a custom power iteration method was implemented to compute the eigenvalues and eigenvectors of the transposed matrix product. This was used to generate the right singular vectors and singular values directly. Left singular vectors were computed via matrix multiplication and normalization.

The input image was processed in its red, green, and blue channels separately, with compression applied to each before recombination. For each value of p, the report includes both the percentage of storage used relative to the original image and the reconstruction error based on the Frobenius norm, satisfying all points in Section 3(a) and 3(b) of the project outline. The results demonstrate that high compression ratios can be achieved with modest loss in image quality, validating the utility of low-rank approximations in image processing.

Introduction

Image compression via SVD allows a matrix to be approximated using only a small number of its singular values and associated vectors. For an image matrix A, the compression is performed by computing the factorization:

$$A \approx U \times \Sigma \times V^t$$

where U is a matrix of left singular vectors, Σ is a diagonal matrix of singular values, and V^t is the transpose of the matrix of right singular vectors.

By truncating the decomposition to retain only the top p singular values, we form a low-rank approximation of the image that requires significantly less storage.

This project was completed individually. I took a photograph of myself and processed it as the subject image. The assignment required computing the decomposition without using any built-in SVD functionality. All compression and reconstruction were done using manually implemented code. In addition to image reconstruction, quantitative metrics—storage percentage and reconstruction error—are evaluated in accordance with the project outline.

Methodological Decisions

1. Overview of the Problem

The goal was to perform image compression using a custom SVD implementation. This required avoiding library SVD functions, supporting color images, and computing compression quality for multiple values of p.

2. Structure of the Code

The code was divided into functions for modularity and clarity. Each function was responsible for a distinct step in the pipeline. The notebook follows a logical order: image loading → SVD computation → image reconstruction → error and compression analysis.

3. Manual SVD Construction

3.1 Power Iteration

A function was written to perform power iteration on the matrix $A^t \times A$, which returns the dominant eigenvalue and its corresponding eigenvector.

3.2 Orthogonalization and Deflation

After each eigenpair was found, Gram-Schmidt orthogonalization was used to ensure orthogonality with previously computed vectors. The matrix $A^t \times A$ was deflated by subtracting the projection onto the found eigenvector. This enabled the successive computation of the next dominant singular vector.

3.3 Singular Value and Left Vector Computation

For each eigenvalue λ_i, the singular value σ_i was computed as the square root of λ_i. The corresponding left singular vector u_i was computed as $\mathbf{u_i} = (\mathbf{1}/\sigma_i) \times \mathbf{A}\mathbf{v}_i$. All results were stored in matrices U, Σ, and V^t.

This approach follows the structure outlined in Section 12.4.2 of *Numerical Analysis*, where low-rank approximations retain the dominant energy of the matrix. By focusing on the top *p* singular values, the reconstruction emphasizes the most significant directions of variance in the data — a principle that underpins many SVD-based applications, including compression and principal component analysis. The observed drop in reconstruction error as *p* increases reflects the rapid decay of singular values typical in natural image matrices.

4. Color Image Handling

The sample notebook provided by the professor only supported grayscale images. I modified the approach to handle color by separating the image into red, green, and blue channels, applying the SVD to each channel independently, and recombining the results.

5. Reconstruction

Each channel was reconstructed using the expression $U \times \Sigma \times V^t$ with the top p singular values. The compressed RGB image was formed by stacking the three channels.

6. Runtime and Image Size Management

The initial image was a high-resolution .jpeg file (2448 × 3264 pixels), which caused the notebook to freeze during SVD processing. I switched to a .jpg version with smaller resolution, and then further reduced the pixel density using:

```
img = img[::3, ::3, :]
```

This allowed reliable processing across multiple values of p without stalling and made the experiments repeatable.

7. Parameterization of p

A loop was written to compute compression and error metrics for multiple p values. The following outputs were collected:

- Compression ratio
- Frobenius norm error
- Compressed image preview

This directly supports Sections 3(a) and 3(b) of the project assignment, which require reporting both storage reduction and reconstruction error.

8. Code Originality

Every numerical method was implemented from scratch. The display and image-handling structure followed the professor's examples, but all computational components — including the SVD algorithm, matrix deflation, and image downsampling — were designed for this project per the instructions.

Computer Experiments/Simulations and Results

The following section contains results that quantitatively fulfill the requirements of Sections 3(a) and 3(b) in the project outline: percentage of storage used, and error between original and compressed images.

Image Dimensions (after resizing): Shape: 481 × 222 × 3

Compression Summary:

p	Compression Ratio	Frobenius Error
10	0.0659	47.1837
25	0.1648	30.2752
50	0.3296	19.7116
100	0.6593	9.7216

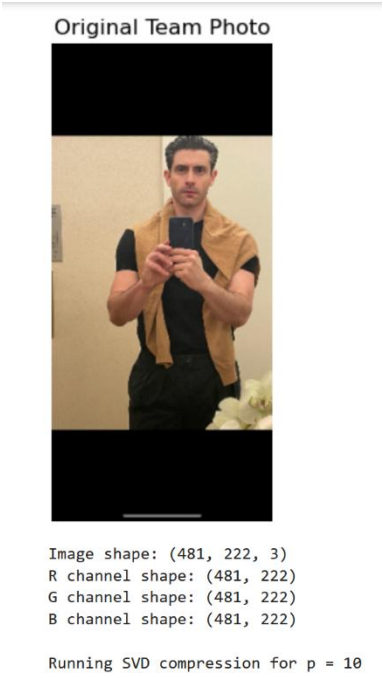
* Compression ratio was computed using the formula:

$$\textbf{Compression Ratio} = 3p(m + n + 1)/(3mn)$$

*Frobenius norm error was computed as the sum of the individual Frobenius errors from each channel.

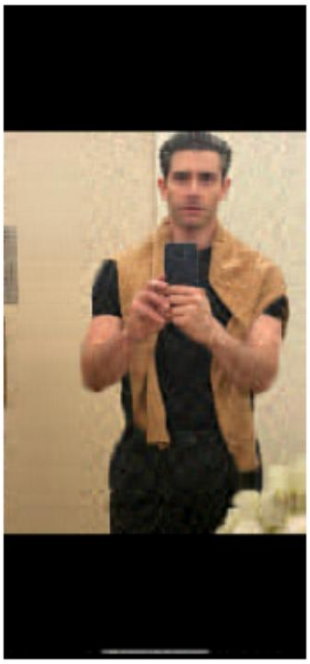
$$\textbf{Frobenius Error} = \|\mathbf{A} - \mathbf{A}_p\|^f$$

Visual Results:



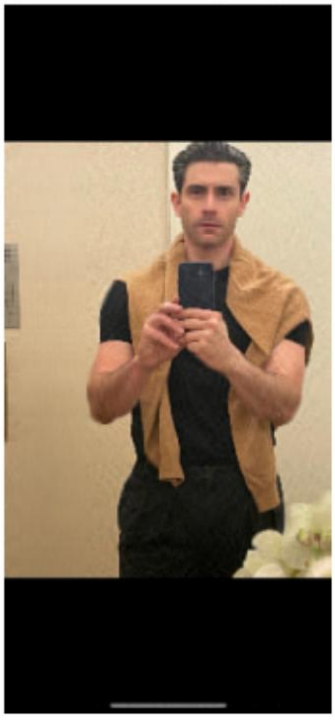
Compressed Image (p = 10)

Compressed Image (p = 25)

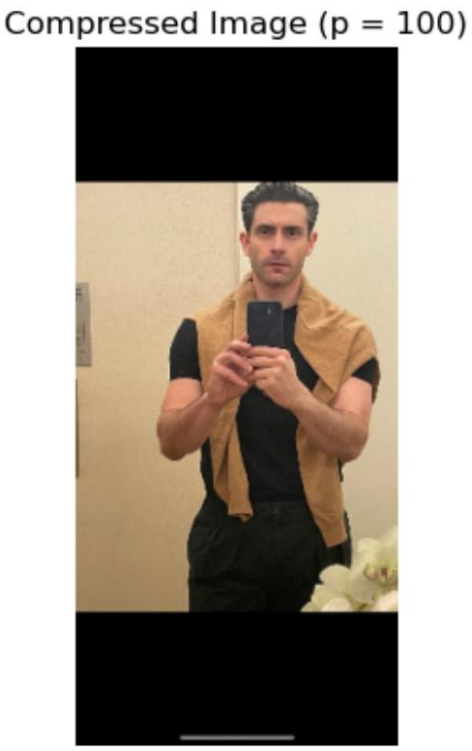


Running SVD compression for p = 50

Compressed Image (p = 50)



Running SVD compression for p = 100




```
Summary of Results:
p = 10  | Compression Ratio: 0.0659 | Frobenius Error: 47.1837
p = 25  | Compression Ratio: 0.1648 | Frobenius Error: 30.2752
p = 50  | Compression Ratio: 0.3296 | Frobenius Error: 19.7116
p = 100 | Compression Ratio: 0.6593 | Frobenius Error: 9.7216
```

Conclusions

Singular Value Decomposition proved to be an effective method for compressing color images using a low-rank approximation framework. Across all tested values of *p*—10, 25, 50, and 100—the compressed images retained coherent visual structure, with increasing values of *p* yielding progressively lower reconstruction error. At *p* = 25, the compression ratio was 0.1648 with a Frobenius error of 30.2752; at *p* = 100, the approximation approached the original with an error of 9.72.

The balance between compression and visual fidelity became particularly apparent in the lower-*p* range. Notably, *p* = 50 appeared to offer a favorable trade-off between memory savings and image quality, with a compression ratio of 0.3296 and a Frobenius error of 19.71. This suggests a practical sweet spot where enough structure is preserved to maintain perceptual integrity while still achieving significant data reduction.

Although perceptual error metrics were not used, the Frobenius norm served as a reliable proxy for reconstruction quality. Computational constraints, such as processing large matrix sizes with manually implemented iterative methods, were addressed by downsampling the image, allowing for stable and repeatable analysis.

The behavior of the singular values reflected the theoretical predictions described in Section 12.4.2 of *Numerical Analysis* by Sauer. A sharp drop in error as *p* increased confirmed that most of the matrix's energy is captured by the leading singular components. This aligns with the core premise of SVD-based approximation: that dominant singular values encode the most significant structure of the original matrix.

The results reinforce the relevance of SVD as both a theoretical and practical tool for data compression. Beyond image storage, the approach supports a broader class of applications in scientific computing where dimensionality reduction and matrix efficiency are required.

References

Blakie, D. (n.d.). *Advanced Python Coding*. NYU, Class Notes.

Gidea, M. (2023). *Lecture 9–11: Eigenvalues and Singular Values I–III* [Class lecture slides]. Yeshiva University.

Google Developers. (n.d.). *Google Python Style Guide: Section 3.8 — Comments and Docstrings*. Retrieved from https://google.github.io/styleguide/pyguide.html#38-comments-and-docstrings

Matplotlib Developers. (n.d.). *Matplotlib Documentation*. Retrieved from https://matplotlib.org/

MAT 5003. (n.d.). *Class Notes and Codes*. Course Materials, Yeshiva University.

NumPy Developers. (n.d.). *NumPy Documentation (Linear Algebra API)*. Retrieved from https://numpy.org/doc/stable/reference/routines.linalg.html

OpenCV Developers. (n.d.). *OpenCV Image Tutorials (Python)*. Retrieved from https://docs.opencv.org/4.x/dc/d2e/tutorial_py_image_display.html

Project 3: SVD Image Compression. (2023). *Assignment Outline*. Yeshiva University.

PythonExamples.org. (n.d.). *Read and Display Color Images*. Retrieved from https://pythonexamples.org/python-read-image-using-matplotlib-imread/

PythonReferences.com. (n.d.). *Python Coding Resources*. Retrieved from https://pythonreferences.com

Sauer, T. (2012). *Numerical Analysis* (3rd ed.). Pearson Education.

StackOverflow. (2013). Image compression using SVD in Python. *Stack Overflow*. Retrieved from https://stackoverflow.com/questions/18625085/image-compression-using-svd-in-python

University of Utah. (n.d.). *Guidelines on Commenting Code*. Retrieved from https://www.cs.utah.edu/~germain/PPS/Topics/commenting.html

---

Appendix

Final.Project3SVDimage

April 23, 2025

[1]:
```
# ============================================================================
# Project 3 - SVD Image Compression
# Student: Gregory Schwartz
# Course: Numerical Methods
# Description: This script compresses a color image using a manually
# Implemented Singular Value Decomposition (SVD) algorithm.
# ============================================================================
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.image import imread


# ============================================================================
# Load and Normalize Image
# ============================================================================
# Load image
img = imread('team_photo.JPG')

# Aggressively downsample
img = img[::3, ::3, :]  # SUPER small for fast compression

# Normalize if needed
if img.max() > 1.0:
    img = img / 255.0


# Display original image
plt.imshow(img)
plt.title("Original Team Photo")
plt.axis('off')
plt.show()

# Check image dimensions
print("Image shape:", img.shape)


# ============================================================================
```

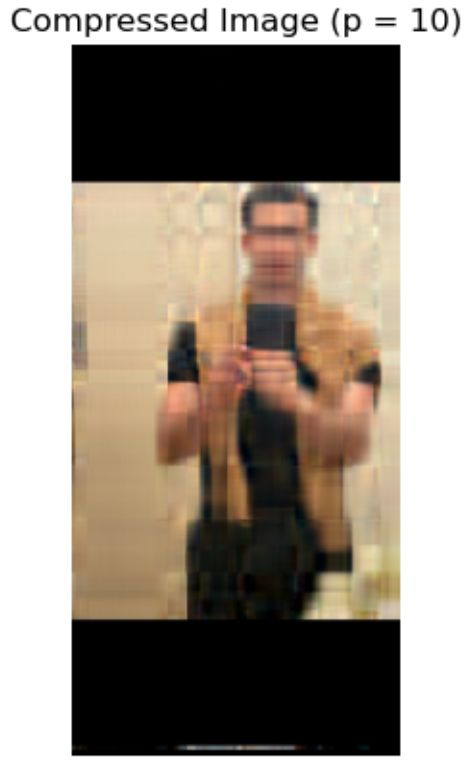
```python
# Function: power_iteration
# Purpose:  Compute the dominant eigenvalue and eigenvector of a symmetric matrix
# ============================================================================
def power_iteration(B, num_iters=1000, tol=1e-10):
    """
    Perform power iteration to find the dominant eigenvalue and eigenvector.

    Parameters:
    B : ndarray
        Symmetric matrix (e.g., A^T A)
    num_iters : int
        Maximum number of iterations
    tol : float
        Tolerance for convergence

    Returns:
    lambda_1 : float
        Dominant eigenvalue
    v : ndarray
        Corresponding normalized eigenvector
    """
    n = B.shape[1]
    v = np.random.rand(n)
    v = v / np.linalg.norm(v)

    for _ in range(num_iters):
        Bv = B @ v
        v_new = Bv / np.linalg.norm(Bv)
        if np.linalg.norm(v_new - v) < tol:
            break
        v = v_new

    lambda_1 = v.T @ B @ v
    return lambda_1, v


# ============================================================================
# Function: compute_manual_svd
# Purpose:  Compute top p singular values/vectors manually using power iteration
# ============================================================================
def compute_manual_svd(A, p, num_iters=1000, tol=1e-10):
    """
    Compute the top p singular values and vectors of matrix A.

    Parameters:
    A : ndarray
```

```
        Input matrix (e.g., color channel of image)
    p : int
        Number of components to keep
    num_iters : int
        Maximum iterations for power iteration
    tol : float
        Tolerance for convergence

    Returns:
    U : ndarray
        Left singular vectors (m x p)
    S : ndarray
        Diagonal matrix of singular values (p x p)
    Vt : ndarray
        Transposed right singular vectors (p x n)
    """
    m, n = A.shape
    ATA = A.T @ A
    V_list, U_list, S_list = [], [], []

    for _ in range(p):
        lambda_i, v_i = power_iteration(ATA, num_iters, tol)

        for v_prev in V_list:
            v_i -= np.dot(v_i, v_prev) * v_prev
        v_i /= np.linalg.norm(v_i)

        sigma_i = np.sqrt(lambda_i)
        u_i = (A @ v_i) / sigma_i

        V_list.append(v_i)
        U_list.append(u_i)
        S_list.append(sigma_i)

        ATA -= sigma_i**2 * np.outer(v_i, v_i)

    U = np.column_stack(U_list)
    V = np.column_stack(V_list)
    S = np.diag(S_list)
    return U, S, V.T


# ================================================================================
# Function: reconstruct_image
# Purpose:  Rebuild matrix from truncated SVD
# ================================================================================
def reconstruct_image(U, S, Vt):
```

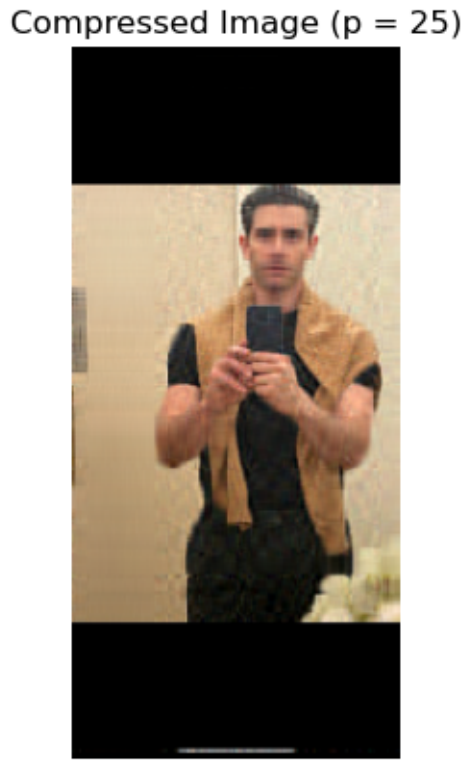
```
    """
    Reconstruct matrix using U, S, Vt.

    Returns:
    A_p : ndarray
        Approximated matrix
    """
    return U @ S @ Vt


# ================================================================================
# Split the image into R, G, B channels
# ================================================================================
R = img[:, :, 0]
G = img[:, :, 1]
B = img[:, :, 2]

print("R channel shape:", R.shape)
print("G channel shape:", G.shape)
print("B channel shape:", B.shape)


# ================================================================================
# Compress and Reconstruct Image for Different p Values
# ================================================================================
p_values = [10, 25, 50, 100]
results = []

for p in p_values:
    print(f"\nRunning SVD compression for p = {p}")

    U_R, S_R, Vt_R = compute_manual_svd(R, p)
    U_G, S_G, Vt_G = compute_manual_svd(G, p)
    U_B, S_B, Vt_B = compute_manual_svd(B, p)

    R_p = reconstruct_image(U_R, S_R, Vt_R)
    G_p = reconstruct_image(U_G, S_G, Vt_G)
    B_p = reconstruct_image(U_B, S_B, Vt_B)

    img_p = np.stack([R_p, G_p, B_p], axis=2)

    # Display the compressed image
    plt.imshow(img_p)
    plt.title(f"Compressed Image (p = {p})")
    plt.axis('off')
    plt.show()
```

```
    # Compute compression ratio
    m, n = R.shape
    original_size = m * n * 3
    compressed_size = 3 * p * (m + n + 1)
    compression_ratio = compressed_size / original_size

    # Compute Frobenius norm error
    error = (
        np.linalg.norm(R - R_p, 'fro') +
        np.linalg.norm(G - G_p, 'fro') +
        np.linalg.norm(B - B_p, 'fro')
    )

    results.append((p, compression_ratio, error))


# ================================================================================
# Summary Output: Compression Ratios and Errors
# ================================================================================
print("\nSummary of Results:")
for p, ratio, err in results:
    print(f"p = {p:<3} | Compression Ratio: {ratio:.4f} | Frobenius Error: {err:
 ↪.4f}")
```

```
Image shape: (481, 222, 3)
R channel shape: (481, 222)
G channel shape: (481, 222)
B channel shape: (481, 222)

Running SVD compression for p = 10

Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers). Got range
[-0.12658579722192093..1.058398244870288].
```




```
Running SVD compression for p = 25

Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers). Got range
[-0.1401202883411203..1.0562795630393662].
```




```
Running SVD compression for p = 50

Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers). Got range
[-0.06087821680068069..1.0565041627480773].
```


```
Running SVD compression for p = 100

Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers). Got range
[-0.04056289921785705..1.0355100940334347].
```


```
Summary of Results:
p = 10  | Compression Ratio: 0.0659 | Frobenius Error: 47.1837
p = 25  | Compression Ratio: 0.1648 | Frobenius Error: 30.2752
p = 50  | Compression Ratio: 0.3296 | Frobenius Error: 19.7116
p = 100 | Compression Ratio: 0.6593 | Frobenius Error: 9.7216
```

[ ]: