Course Project: Image Compression Using Truncated SVD

Matrix Theory (EE1030)

S.HARSHA VARDHAN REDDY
Roll Number: EE25BTECH11054

November 8, 2025

1 Introduction

This project implements image compression using Singular Value Decomposition (SVD), specifically the truncated SVD approach. The goal is to represent grayscale images with fewer components while preserving visual quality, demonstrating the power of low-rank matrix approximations.

A grayscale image can be represented as a matrix $A \in \mathbb{R}^{m\times n}$, where each entry a_{ij} corresponds to pixel intensity (0 = black, 255 = white). Using SVD, we decompose this matrix and retain only the most significant singular values to achieve compression.

2 Mathematical Background

2.1 Singular Value Decomposition

The Singular Value Decomposition expresses any matrix $A \in \mathbb{R}^{m\times n}$ as:

$$A = U\Sigma V^T \tag{1}$$

where:

- $U \in \mathbb{R}^{m\times m}$ is an orthogonal matrix (left singular vectors)
- $\Sigma \in \mathbb{R}^{m\times n}$ is a diagonal matrix with singular values $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r \geq 0$
- $V \in \mathbb{R}^{n\times n}$ is an orthogonal matrix (right singular vectors)

2.2 Truncated SVD for Compression

A rank-k approximation of A is obtained by keeping only the top k singular values:

$$A_k = U_k\Sigma_k V_k^T = \sum_{i=1}^{k} \sigma_i u_i v_i^T \tag{2}$$

where U_k, Σ_k, and V_k contain only the first k columns/values.

2.3 Approximation Error

The quality of approximation is measured using the Frobenius norm:

$$\|A - A_k\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}(a_{ij} - a_{k,ij})^2} = \sqrt{\sum_{i=k+1}^{r}\sigma_i^2} \tag{3}$$

3 Summary of Gilbert Strang's SVD Lecture

Gilbert Strang's lecture on SVD (MIT OpenCourseWare) provides excellent intuition and mathematical foundation for understanding SVD:

Key Takeaways:

- **Geometric Interpretation:** SVD transforms any linear transformation into three simple operations: rotation (V^T), scaling (Σ), and rotation (U).
- **Best Low-Rank Approximation:** The truncated SVD provides the optimal rank-k approximation to a matrix in terms of both Frobenius and spectral norms (Eckart-Young theorem).
- **Four Fundamental Subspaces:** SVD reveals the four fundamental subspaces of linear algebra - the column space, row space, null space, and left null space.
- **Principal Component Analysis:** SVD is closely related to PCA; the right singular vectors are principal components.
- **Computational Significance:** SVD is numerically stable and reveals the rank and condition number of matrices.
- **Data Compression:** Large singular values capture most of the "energy" or information in the data, making SVD ideal for compression.

4 Algorithm Selection and Implementation

4.1 Algorithm Comparison

Several algorithms exist for computing SVD:

1. **Jacobi Method:**
    - Uses iterative rotations to diagonalize A^TA
    - Highly accurate but computationally expensive
    - Time complexity: $O(n^3)$ per iteration

2. **QR Algorithm:**

    - First reduces matrix to bidiagonal form, then applies QR iterations
    - Standard method in numerical libraries (LAPACK)
    - Time complexity: $O(mn^2)$ for $m \geq n$

3. **Power Iteration (Chosen Method):**

    - Iteratively finds dominant eigenvectors
    - Simple to implement from scratch
    - Efficient for finding top-k singular values (partial SVD)
    - Time complexity: $O(kmn)$ for k singular values

4. **Divide and Conquer:**

    - Fastest for dense matrices
    - Complex implementation
    - Time complexity: $O(mn^2)$

4.2 Chosen Algorithm: Power Iteration Method

I chose the **Power Iteration Method** for the following reasons:

- **Simplicity:** Straightforward implementation without complex matrix operations
- **Efficiency:** Only computes the top-k singular values needed for compression
- **Memory:** Requires less memory than full SVD decomposition
- **Educational Value:** Demonstrates fundamental iterative eigenvalue algorithms

4.3 Algorithm Description

Pseudocode:

Listing 1: Power Iteration SVD Algorithm

```
function compute_svd(A, k):
    m, n = dimensions(A)
    U = zeros(m, k)
    S = zeros(k)
    V = zeros(n, k)
```

```
A_copy = copy(A)

for i in range(k):
    # Initialize random vector
    v = random_vector(n)

    # Power iteration
    for iteration in range(max_iterations):
        v_new = A_transpose * A * v
        v = normalize(v_new)

    # Store right singular vector
    V[:, i] = v

    # Compute left singular vector
    u = A_copy * v
    sigma = norm(u)
    S[i] = sigma
    u = normalize(u)
    U[:, i] = u

    # Deflate matrix
    A_copy = A_copy - sigma * outer(u, v)

return U, S, V
```

4.4 Implementation Details

Key C Functions Implemented:

- `create_matrix()`: Dynamic memory allocation for matrices
- `transpose()`: Matrix transposition
- `multiply()`: Matrix multiplication
- `normalize_vector()`: Vector normalization
- `compute_svd()`: Main SVD computation using power iteration
- `reconstruct_image()`: Image reconstruction from truncated SVD
- `save_reconstructed_image()`: Save compressed images

Image I/O: Used `stb_image.h` and `stb_image_write.h` libraries for reading and writing images in various formats (PNG, JPG, etc.).

5 Results

5.1 Reconstructed Images

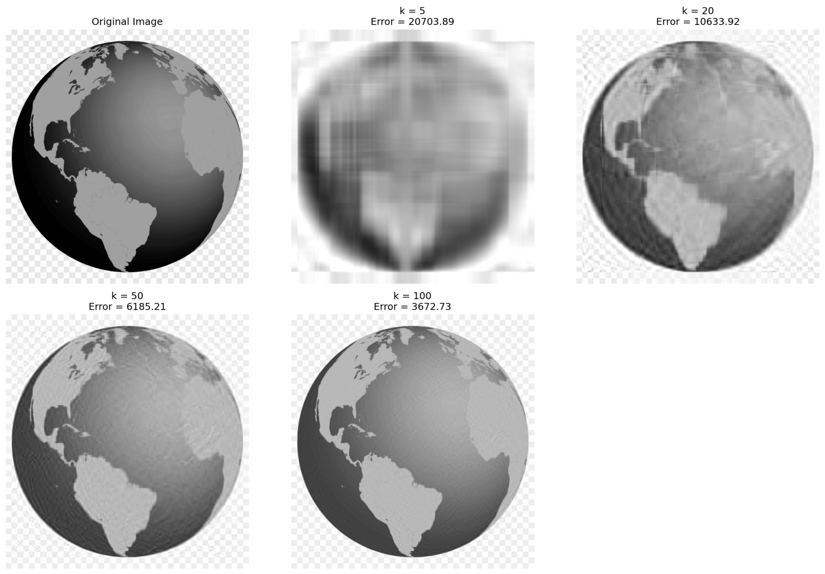

Figure 1: Comparison of reconstructed images at different k-values

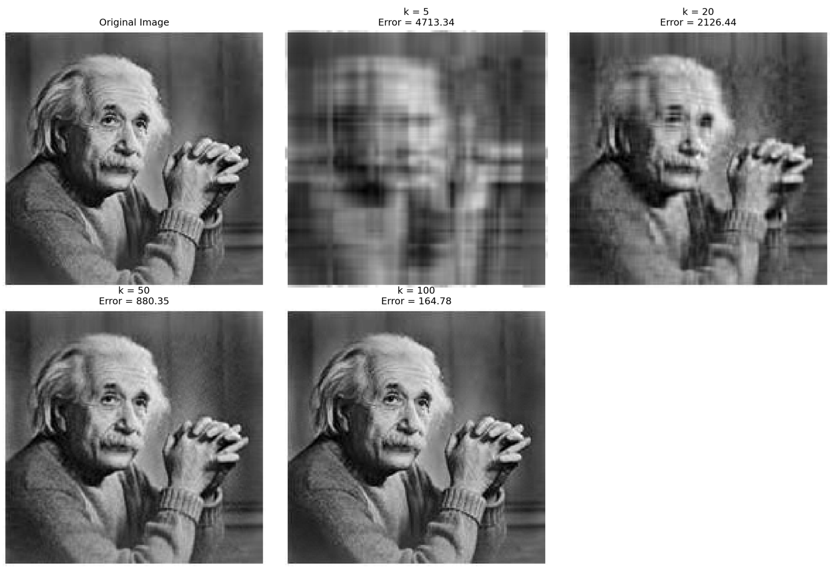

Figure 2: Comparison of reconstructed images at different k-values

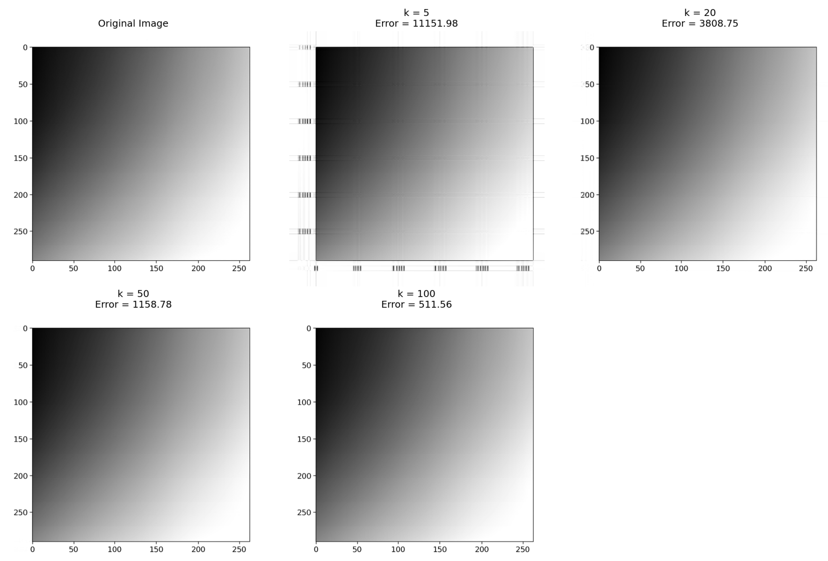

Figure 3: Comparison of reconstructed images at different k-values

5.2 Error Analysis

k (Rank)	**Frobenius Error**
5	20704.27
20	10634.41
50	6185.64
100	3672.93

Table 1: Error analysis for Image 1 at different k values

k (Rank)	**Frobenius Error**
5	4713.60
20	2126.56
50	880.50
100	164.78

Table 2: Error analysis for Image 2 at different k values

k (Rank)	**Frobenius Error**
5	11146.31
20	3808.19
50	1160.14
100	512.35

Table 3: Error analysis for Image 3 at different k values

5.3 Frobenius Error vs k

The following plots show how the Frobenius norm error decreases as the number of singular values k increases. This demonstrates the effectiveness of the low-rank approximation.

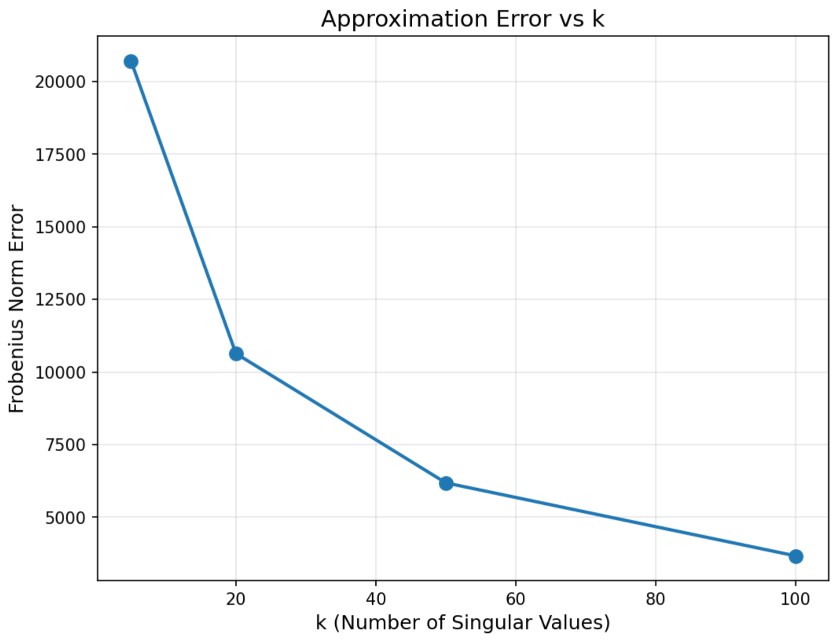

Figure 4: Frobenius Error vs k

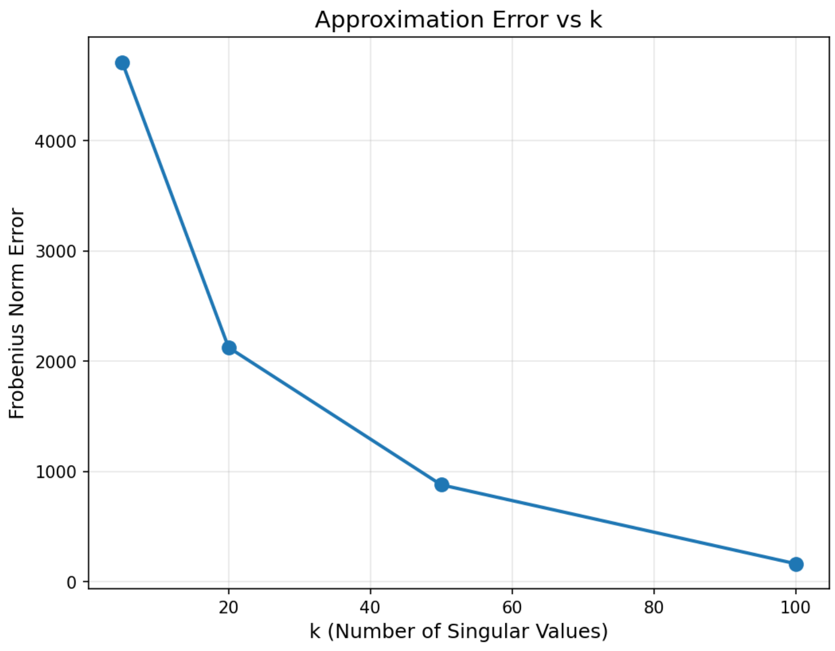

Figure 5: Frobenius Error vs k

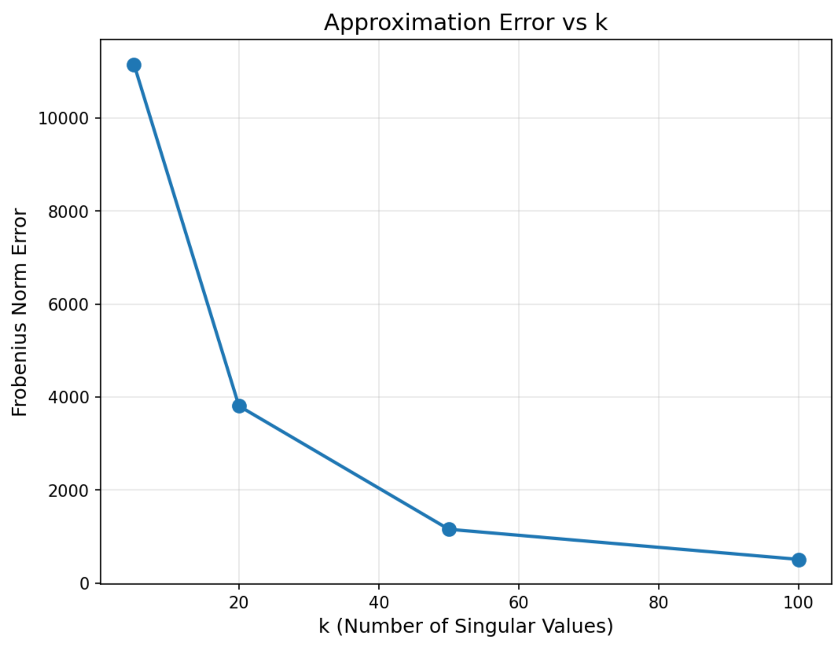

Figure 6: Frobenius Error vs k

The plots clearly show an exponential decay in error as k increases, which aligns with the theoretical expectation that retaining more singular values leads to better approximation quality.

6 Conclusion

This project successfully demonstrates image compression using truncated SVD implemented entirely in C. Key findings include:

1. Low-rank approximations effectively compress images while preserving visual quality
2. The experiment clearly demonstrated the relationship between the rank k, reconstruction error, and visual quality of the image. As k increased, the images became more accurate, and the error decreased, but at the cost of higher storage requirements

The project reinforced understanding of linear algebra concepts, numerical methods, and low-level programming in C.