SOFTWARE ASSIGNMENT - IMAGE COMPRESSION USING TRUNCATED SVD

KAVIN B

ROLL NO: EE25BTECH11033

1 INTRODUCTION

This project implements image compression using Singular Value Decomposition (SVD).In this context, a grayscale image is represented as a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, where each entry a_{ij} corresponds to a pixel's intensity (ranging from 0 for black to 255 for white).

The primary objectives of this project are to implement this truncated SVD algorithm, reconstruct images using varying values of k (e.g., 5, 20, 50, 100), and quantitatively analyze the quality of compression by computing the approximation error using the Frobenius norm ($\|\mathbf{A} - \mathbf{A}_k\|_F$).

2 THEORY

The fundamental principle relied upon is that SVD can factorize this image matrix into three components: $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, where

- $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices
- $\mathbf{\Sigma}$ is a diagonal matrix containing singular values in decreasing order

By retaining only the top k singular values, we can generate a low-rank approximation of the original image, defined as $\mathbf{A}_k = \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^T$. This technique forms the basis of image compression, as it often allows for the preservation of the majority of the image's visual content while significantly reducing the amount of data required to store it.

3 SUMMARY OF STRANG'S VIDEO

3.1 Concept

SVD is described as the "final and best" factorization for *any* matrix $\mathbf{A}$ (whether rectangular or square). It factorizes matrix $\mathbf{A}$ into three specific components:

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

where:

- $\mathbf{U}$ is an orthogonal matrix whose columns (u_i) form an orthonormal basis for the **column space**.
- $\mathbf{\Sigma}$ is a diagonal matrix containing non-negative **singular values** ($\sigma_i \geq 0$).
- $\mathbf{V}^T$ is the transpose of an orthogonal matrix $\mathbf{V}$, whose columns (v_i) form an orthonormal basis for the **row space**.

Unlike eigenvalue decomposition for symmetric matrices ($\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$), SVD applies to all matrices by utilizing *two different* orthogonal bases, $\mathbf{U}$ and $\mathbf{V}$.

3.2 Geometric Interpretation

The geometric goal of SVD is to find specific orthonormal bases for the row and column spaces such that the matrix **A** acts purely as a "stretcher" (scaling) without any rotation between these bases. The fundamental relationship is:

$$\mathbf{A}v_i = \sigma_i u_i \tag{2}$$

Matrix **A** maps a basis vector from the row space (v_i) to a scaled basis vector in the column space ($\sigma_i u_i$).

3.3 Computation Method

The components **U**, **Σ**, and **V** are found using symmetric positive (semi) definite matrices:

1. **Finding V and Σ**: Compute $\mathbf{A}^T\mathbf{A}$.

$$\mathbf{A}^T\mathbf{A} = (\mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T)(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T) = \mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T \tag{3}$$

   - The eigenvectors of $\mathbf{A}^T\mathbf{A}$ are the columns of **V**.
   - The square roots of the eigenvalues of $\mathbf{A}^T\mathbf{A}$ are the **singular values** (σ_i) in **Σ**.

2. **Finding U**: Compute $\mathbf{A}\mathbf{A}^T$.

$$\mathbf{A}\mathbf{A}^T = (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)(\mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T) = \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T \tag{4}$$

   - The eigenvectors of $\mathbf{A}\mathbf{A}^T$ are the columns of **U**.

3.4 The Four Fundamental Subspaces

SVD perfectly aligns with the four fundamental subspaces of a matrix:

- The first r columns of **V** (where r is the rank) form the basis for the **row space**.
- The remaining columns of **V** form the basis for the **null space** (corresponding to $\sigma_i = 0$).
- The first r columns of **U** form the basis for the **column space**.
- The remaining columns of **U** form the basis for the **left null space**.

4 IMPLEMENTED ALGORITHM

JACOBI METHOD

The Jacobi method is an iterative algorithm used to find the eigenvalues and eigenvectors of a real symmetric matrix **A**. The core idea is to apply a sequence of orthogonal similarity transformations (Givens rotations) to systematically annihilate the off-diagonal elements.

4.1 Transformation to a Symmetric Problem

Standard SVD works on any rectangular matrix $\mathbf{A}$ (height $\times$ width). However, the Jacobi algorithm specifically requires a **square, symmetric matrix**.

- **Action:** We compute $\mathbf{B} = \mathbf{A}^T\mathbf{A}$.
- **Reason:** The resulting matrix $\mathbf{B}$ is guaranteed to be square (width $\times$ width) and symmetric. Crucially, it shares the same "right" singular vectors ($\mathbf{V}$) as the original image $\mathbf{A}$, and its eigenvalues are the squared singular values (σ^2) of $\mathbf{A}$.

4.2 The Jacobi "Annihilation" Strategy

This is the core engine designed to turn the dense matrix $\mathbf{B}$ into a diagonal matrix.

1. **Find the largest off-diagonal element:** In every iteration, the algorithm searches the entire matrix to find the largest off-diagonal element B_{pq}.
2. **Rotate it away:** A **Givens Rotation** is applied, affecting only rows/columns p and q. This rotation is mathematically designed to be exactly the angle required to make that specific B_{pq} zero.
3. **Repeat:** By repeatedly attacking the largest remaining element, the matrix inevitably converges to a diagonal state.

The values finally left on the diagonal are the eigenvalues (importance scores), and the accumulated rotations form the eigenvectors ($\mathbf{V}$).

4.3 Extracting SVD Components

Once Jacobi converges, we possess $\mathbf{V}$ and the eigenvalues (λ_i). The remaining SVD components are derived as follows:

- **Singular Values ($\mathbf{\Sigma}$):** Calculated as the square root of the eigenvalues found by Jacobi: $\sigma_i = \sqrt{\lambda_i}$.
- **Left Singular Vectors (U):** These represent the "vertical" patterns in the image, computed using the relationship:
    1. Start with $\mathbf{A} = \mathbf{U\Sigma V}^T$.
    2. Multiply both sides by $\mathbf{V}$ on the right:

$$\mathbf{AV} = \mathbf{U\Sigma V}^T\mathbf{V} \tag{5}$$

    3. Since $\mathbf{V}$ is orthogonal ($\mathbf{V}^T\mathbf{V} = \mathbf{I}$):

$$\mathbf{AV} = \mathbf{U\Sigma} \tag{6}$$

    4. To isolate $\mathbf{U}$, multiply by the inverse of $\mathbf{\Sigma}$ (which is just dividing by the singular values because $\mathbf{\Sigma}$ is diagonal):

$$\mathbf{U} = \mathbf{AV\Sigma}^{-1} \tag{7}$$

For an individual column vector $\mathbf{u}_j$ (the j-th column of $\mathbf{U}$), this formula simplifies to:

$$\mathbf{u}_j = \frac{1}{\sigma_j}\mathbf{A}\mathbf{v}_j \tag{8}$$

Where $\mathbf{v}_j$ is the j-th column of $\mathbf{V}$ and σ_j is the j-th singular value.

4.4 Image Compression (Truncation)

Perfectly reconstructing the image requires all N singular values. However, the first few larger singular values typically contain the main structure (shapes, shadows), while smaller ones contain noise or fine texture.

- **Action:** We retain only the top k (e.g., 50) singular values and their corresponding vectors.
- **Reconstruction:** The approximate compressed image is built using only these top k components:

$$\mathbf{A}_{approx} = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T \tag{9}$$

5 COMPARISON OF DIFFERENT ALGORITHMS

5.1 Jacobi Algorithm

This is the method implemented in the this project. It iteratively applies Givens rotations to zero out off-diagonal elements.

- **Pros:** Extremely accurate, especially for small singular values. Inherently parallelizable.
- **Cons:** Very slow for large general matrices compared to modern methods, typically requiring many sweeps through the entire matrix.

5.2 Power Iteration

A simple iterative method primarily used to find the dominant eigenvalue (the one with the largest absolute value) and its corresponding eigenvector.

- **Mechanism:** Starts with a random vector v_0 and iteratively computes $v_{k+1} = Av_k$ (normalizing at each step) until it converges to the dominant eigenvector.
- **Pros:** Very simple to implement. Useful when only the single largest eigenvalue is needed (e.g., Google's original PageRank algorithm). Memory-efficient for sparse matrices as it only requires matrix-vector multiplication.
- **Cons:** Only finds one eigenvalue at a time. Convergence can be very slow if the top two eigenvalues are close in magnitude. Does not work well if the dominant eigenvalue is not unique (e.g., complex conjugate pairs).

5.3 Golub-Kahan-Reinsch (Bidiagonalization + QR)

The historical "gold standard" for dense matrix SVD (often found in standard libraries like LAPACK's `dgesvd`).

- **Mechanism:** Reduces the dense matrix to **bidiagonal form** using Householder reflections, then applies the QR algorithm to find singular values.
- **Pros:** Very stable, robust, and reliable.
- **Cons:** Slower than Divide and Conquer for very large matrices.

5.4 Divide and Conquer

The modern default for many high-performance dense linear algebra libraries (e.g., LAPACK's `dgesdd`).

- **Mechanism:** Splits the matrix into smaller subproblems, solves them recursively, and "glues" the answers back together.
- **Pros:** Significantly faster (often 2x-4x) than Golub-Kahan for large matrices by exploiting highly optimized BLAS Level 3 operations.
- **Cons:** Requires more workspace memory than QR.

5.5 Randomized SVD (rSVD)

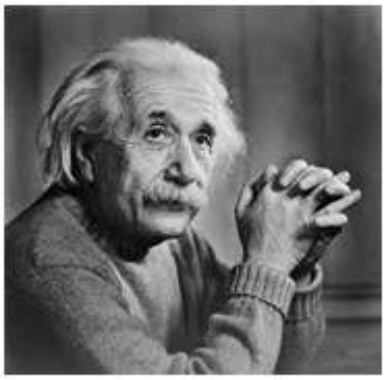
A modern class of algorithms ideal for huge data (e.g., machine learning) where only the top k singular values are needed (truncated SVD).

- **Mechanism:** Multiplies the huge matrix $\mathbf{A}$ by a random matrix $\mathbf{\Omega}$ to get a smaller representative matrix $\mathbf{Y}$. SVD is then computed on this smaller matrix.
- **Pros:** Extremely fast for huge matrices; requires fewer passes over data.
- **Cons:** It is an **approximation**. It might miss very small singular values, but is usually highly accurate for dominant ones.

6 RECONSTRUCTED IMAGES

6.1 EINSTEIN

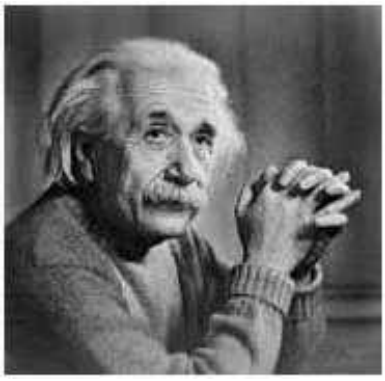
Original Image



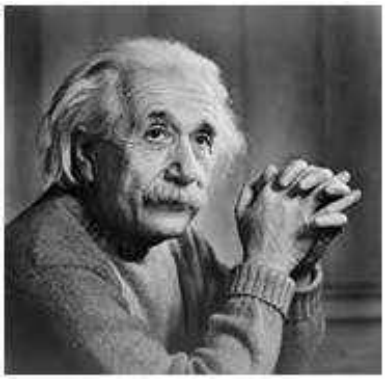
Reconstructed Images:

k=5

k=20



k=50



k=100

6.2 GLOBE

Original Image

Reconstructed Images:

k=5

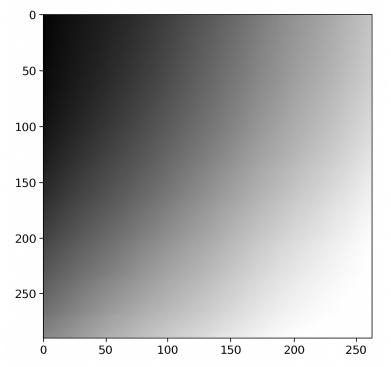
k=20

k=50

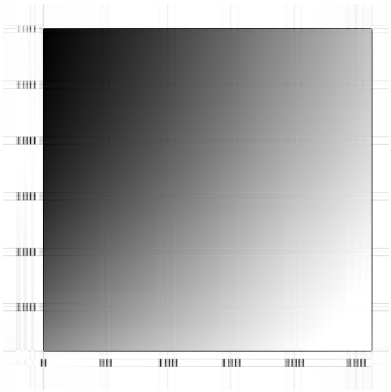
k=100

6.3 GREYSCALE

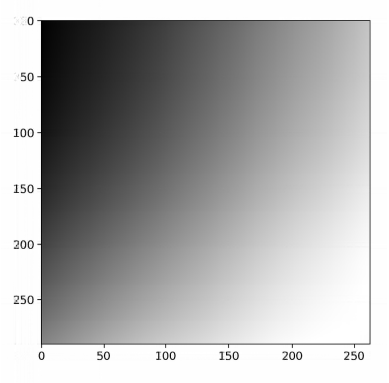
Original Image



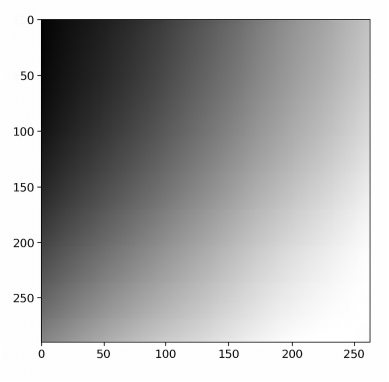
Reconstructed Images:



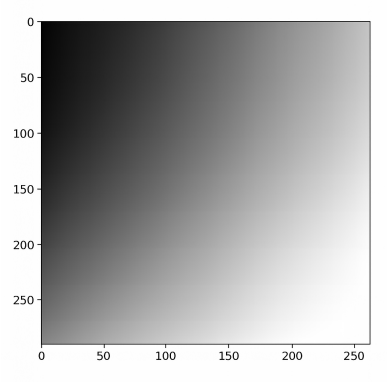
k=5


0
50
100
150
200
250
0
50
100
150
200
250

k=20


0
50
100
150
200
250
0
50
100
150
200
250

k=50


0
50
100
150
200
250
0
50
100
150
200
250

k=100

7 ERROR ANALYSIS

EINSTEIN

Table 4: Einstein Reconstruction Error Analysis

k	Frobenius Norm	Percentage Error (%)
5	4628.676439	21.2234
20	2124.663914	9.7420
50	880.015709	4.0350
100	185.238866	0.8494

GLOBE

Table 4: Globe Reconstruction Error Analysis

k	Frobenius Norm	Percentage Error (%)
5	20500.294336	12.9492
20	10530.550051	6.6518
50	6123.277645	3.8678
100	3676.958940	2.3226

GREYSCALE

Table 4: Greyscale Reconstruction Error Analysis

k	Frobenius Norm	Percentage Error (%)
5	10992.299278	5.6790
20	3525.721393	1.8215
50	1040.807945	0.5377
100	523.337708	0.2704

8 TRADE OFFS AND REFLECTIONS ON IMPLEMENTATION CHOICE

8.1 Algorithm Choice: Classical Jacobi

We implemented the **Classical Jacobi algorithm** from scratch. The primary trade-off here was accepting slower computational performance in exchange for high numerical accuracy and a clear, iterative mechanism that is easier to understand and debug than more complex modern methods.

- **Trade-off:** The choice of the Classical Jacobi algorithm prioritized **simplicity and accuracy**

over **speed**. By iteratively zeroing out off-diagonal elements with straightforward rotations, the method is easy to implement and highly precise for symmetric matrices like $\mathbf{A}^T\mathbf{A}$, ensuring numerical stability even for small eigenvalues. However, this comes at the cost of performance, as the algorithm's $O(N^3)$ complexity makes it significantly slower for large images compared to more complex, modern alternatives like QR or Divide and Conquer.

- **Reflection:** While highly accurate for symmetric matrices like $\mathbf{A}^T\mathbf{A}$, its $O(N^3)$ time complexity makes it prohibitively slow for very high-resolution images compared to industry-standard algorithms (e.g., Divide and Conquer or Golub-Kahan).

8.2 Memory Management in C

Our implementation uses manual dynamic memory allocation in C, eventually standardizing on **flattened 1D arrays** to represent 2D matrices.

- **Trade-off:** By manually managing memory, we gain precise control over data layout, which improves cache locality and performance. However, this increases complexity, as it requires manual index arithmetic (e.g., `A[i*width + j]`) and rigorous error checking to prevent memory leaks or segmentation faults, issues that are automatically handled in higher-level languages.

- **Reflection:** Provides fine-grained control over memory layout, improving cache locality. However, it introduces significant risks of segmentation faults and memory leaks, requiring rigorous testing and careful indexing logic compared to using high-level languages with automatic garbage collection.

8.3 Full vs. Approximate SVD

We computed the **full SVD** before truncating to rank k.

- **Trade-off:** This prioritizes **exactness** over **efficiency**. While this approach guarantees precision by calculating all singular values, it is computationally expensive, as many of these values are ultimately discarded. In a production environment for large-scale images, **Randomized SVD** would be a preferable alternative, offering significant speedups by directly computing only the top-k components at the cost of a negligible approximation error.

- **Reflection:** This approach is computationally expensive as it calculates all singular values, even those that are eventually discarded. A production system for large images might prefer **Randomized SVD**, which directly computes only the top-k components, offering massive speedups at the cost of a small approximation error.

9 CONCLUSION

This project successfully demonstrated the application of Singular Value Decomposition (SVD) for image compression. By decomposing an image's matrix into its constituent singular values and vectors, we were able to reconstruct approximations of the image using only a truncated subset of this data.

Our key finding was the direct and quantifiable trade-off between the level of compression and the resulting image quality. This relationship is controlled by the number of singular values, k, used for reconstruction:

1. **High Compression**(Low k): Using a very small k (e.g., $k = 10$) resulted in a significant compression ratio, as only a fraction of the original data was needed. However, this came at the cost of severe image degradation, with only the most basic structural elements of the image being preserved.

2. **High Quality**(High k): As k was increased, the reconstructed image's fidelity to the original improved dramatically. The largest singular values clearly correspond to the most significant visual information. Smaller, later singular values add finer details, textures, and noise.