SOFTWARE PROJECT :Image Compression using SVD

Jnanesh Sathisha Karmar- EE25BTECH11029

Abstract

This report describes an implementation from scratch of a program in C that performs image compression using SVD. It explains the core mathematical concepts, the SVD algorithm through the Power Iteration Method, and the modular C code structure designed for this project. Also, the report covers extensions to color image processing, singular value analysis, and runtime comparison with NumPy.

I. Introduction and Background

A. Summary of Strang's Lecture

This video is a lecture by Gilbert Strang on Singular Value Decomposition, part of MIT's 18.06 Linear Algebra course. He explains SVD, which decomposes any matrix into three particular matrices: two orthogonal matrices and one diagonal matrix with singular values. He demonstrates how SVD relates to other matrix factorizations, including the symmetric positive definite case. He discusses the geometric interpretation of SVD as a change of orthonormal bases in row and column spaces. Strang demonstrates with examples how to compute the SVD. He shows how the singular values are related to the eigenvalues of $\mathbf{A}^\top\mathbf{A} and \mathbf{A}\mathbf{A}^\top$ and how these bases reveal the structure of the four fundamental subspaces of linear algebra.

B. What is Singular Value Decomposition (SVD)?

A grayscale image can be represented as a large matrix A of size $m \times n$, where each entry A_{ij} is a pixel intensity value (e.g., from 0 to 255). The Singular Value Decomposition (SVD) is a fundamental theorem in linear algebra that states any matrix A can be factored into three separate matrices:

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

Where:

- A: The original $m \times n$ image matrix.
- U: An $m \times m$ orthogonal matrix. Its columns are the "left singular vectors."
- Σ: An $m \times n$ diagonal matrix. Its diagonal entries, $\sigma_1, \sigma_2, \sigma_3, \ldots$, are the "singular values" of A, sorted in decreasing order. These values represent the "importance" of each component.
- V^T: The transpose of an $n \times n$ orthogonal matrix. The columns of V (or rows of V^T) are the "right singular vectors."

C. Image Compression with Truncated SVD

The "full" SVD decomposition is not compressive. However, most of the "energy" or visual information of an image is contained in the first few, largest singular values. We can create a "truncated" or "low-rank" approximation, A_k, by keeping only the top k singular values and their corresponding vectors:

$$A_k = U_k \Sigma_k V_k^T \tag{2}$$

Where:

- U_k is the $m \times k$ matrix of the first k columns of U.
- Σ_k is the $k \times k$ diagonal matrix of the top k singular values.
- V_k^T is the $k \times n$ matrix of the first k rows of V^T.

The compression comes from the fact that we no longer store the $m \times n$ numbers of A. Instead we only store the components of A_k. For a 512x512 image ($m = 512, n = 512$) with $k = 50$:

- **Original Storage:** $512 \times 512 = 262,144$ values.
- **Compressed Storage** ($k = 50$)**:** $(512 \times 50) + 50 + (512 \times 50) = 51,250$ values.

This represents a compression of over 80%. My program saves these components to a custom '.svd' file to prove this file size reduction.

II. Algorithm: SVD via Power Iteration

The project requires implementing SVD from scratch. Instead of complex methods like QR iteration, I chose the Power Iteration Method because it is well-suited for finding the top k components one by one.

A. The Core Connection: SVD and Eigenvalues

The algorithm relies on a key connection between SVD and eigendecomposition. For any matrix A:

- The columns of V (right singular vectors) are the eigenvectors of the symmetric matrix $A^T A$.
- The singular values (σ_i) are the square roots of the eigenvalues (λ_i) of $A^T A$. That is, $\sigma_i = \sqrt{\lambda_i}$.

The Power Iteration method is a way to find the single dominant(largest) eigenvalue and eigenvector of a matrix. This is ideal for our purposes, since we want the components associated with the largest singular values first.

B. The Implemented Algorithm

My program finds the top k components by iterating the following process k times: **1. Find the 1st Component** (σ_1, u_1, v_1)**:**

1) Form the symmetric matrix $B = A^T A$.
2) Create a random "guess" vector v of size n.
3) **Power Iteration Loop:** Repeat for ~30 iterations:
   - $v_{\text{new}} = B \cdot v$ (Multiplies the vector by the matrix)
   - $v = v_{\text{new}} / ||v_{\text{new}}||$ (Normalizes the vector to length 1)

4) After the loop, v has converged to v_1, the dominant eigenvector of B, which is our first right singular vector.
5) We then find σ_1 and u_1 using the SVD formula $Av_1 = \sigma_1 u_1$.
   - $\sigma_1 = \|Av_1\|$ (The singular value is the length of Av_1)
   - $u_1 = Av_1/\sigma_1$ (The left singular vector is the normalized result)

2. Find the 2nd Component (Deflation):

1) To find the *next* component, we must "deflate" the matrix by subtracting the component we just found:
$$A_{\text{new}} = A - \sigma_1 u_1 v_1^T$$
2) We then repeat the entire Power Iteration process (Step 1) on A_{new}. This will find σ_2, u_2, and v_2.

This process is repeated k times, finding one component in each iteration.

III. Code Logic and Structure

The project is written entirely in C and organized into a modular structure, where each module (a '.c' and '.h' file pair) has a single responsibility.

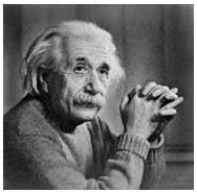
- `matrixutils.c / .h`: The Foundation. This module knows nothing about images. It only provides the tools for matrix math, including `Matrix` struct, `create_matrix`, `free_matrix`, `matrix_multiply`, `matrix_transpose`, and vector operations.
- `image_io.c / .h`:The Translator.This module handles reading and writing the plain-text PGM (P2) and PPM (P3) file formats. It parses the file headers and pixel data, converting them to and from the `Matrix` structs.
- `svd.c / .h`: The Brain. This module implements the SVD algorithm described in Section II. Its main function, `compute_truncated_svd(A, k)`, performs the power iteration and deflation loop to return an `SVDResult` struct.
- `svd_io.c / .h`: The Compressor. This module proves the compression. It provides functions to save the `SVDResult` struct (U, S, V, and k) to a custom, small `.svd` file and read it back. This file is the *actual* compressed image.
- `analysis.c / .h`: The Rebuilder. This module implements the math to decompress the image. Its `reconstruct_image` function takes an `SVDResult` and calculates $A_k = U_k \Sigma_k V_k^T$. It also computes the Frobenius norm error.
- `main.c`: The Conductor. This file orchestrates the entire process. It loops through the target images and k-values. For each combination, it runs the full pipeline:
  1) `read_pgm` → Load original.
  2) `compute_truncated_svd` → Calculate components.
  3) `write_compressed_svd` → **Save small `.svd` file (Proof of compression).**
  4) `read_compressed_svd` → Load small file back.
  5) `reconstruct_image` → Decompress to a full-size blurry image.
  6) `write_pgm` → **Save reconstructed `.pgm` file (Visual result).**
- `convert.py`: The Control center. A Python helper script that prepares images runs the C program, and performs post-analysis.
- `pgm_to_jpeg_c ppm_to_jpeg_c`:The Converter . at last is basically converts all the pgm or ppm files to viewable jpeg format , so that we can actually view the compression.

IV. Project Results and Analysis

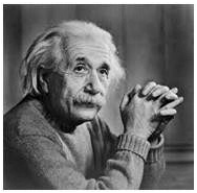
A. Reconstructed Images

einstein.jpg

einstein.jpg



einstein.jpg

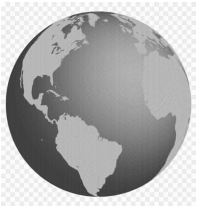


einstein.jpg

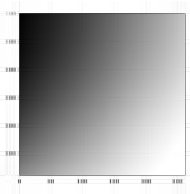
globe.jpg

globe.jpg

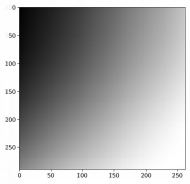
globe.jpg



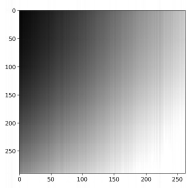
globe.jpg



greyscale.jpg



greyscale.jpg



greyscale.jpg

B. Error Analysis

Metric **k=50**	**k=5** **k=100**	**k=20**	**k=20**
P.Forbenius error 203.47	4717.118	2128.102	884.84

Metric **k=50**	**k=5** **k=100**	**k=20**	**k=20**
P.Forbenius error 3848.49	20635.50	10607.95	6180.55

V. Discussion and Extra Credit

A. Discussion of Trade-offs

The most important parameter is k, the number of singular values retained, which balances image quality against compression. This project makes a clear and direct trade-off between these three elements:

- **Image Quality (Fidelity):** This is directly proportional to k. A very low k (for example, $k = 5$) retains only the most important components, hence giving an abstract or "blocky" image with a high Frobenius norm error. As k gets bigger, more fine-grained details will be added, and quality improves significantly. At high k (for example, $k = 100$), the reconstructed image A_k often becomes visually not differentiable from the original A, and the Frobenius norm error is very small.
- **Compression (File Size):** This is inversely proportional to k. The amount of storage needed for the compressed components is on the order of $(m \times k) + (n \times k) + k$. For a small value of k, we can have a very large compression ratio and a tiny texttt.svd file. As k increases, the file size of the compressed texttt.svd file grows linearly with it, offering less compression.

The main task is to find a "sweet spot" for k. The singular values show an exponential decay,(see Section IV) , indicating that the first few components, say the first 50 components, hold a huge percentage of the image's total "energy." Given this, further increases beyond that, such as from $k = 100$ to $k = 200$, yield very negligible and usually imperceptible gains in the quality of the image while increasing the size of the compressed file considerably. A good compromise was thus found with $k \approx 50$ for a 512x512 image-a value that achieves more than 80% compression with very high visual fidelity.

B. Bonus: Color Image Compression

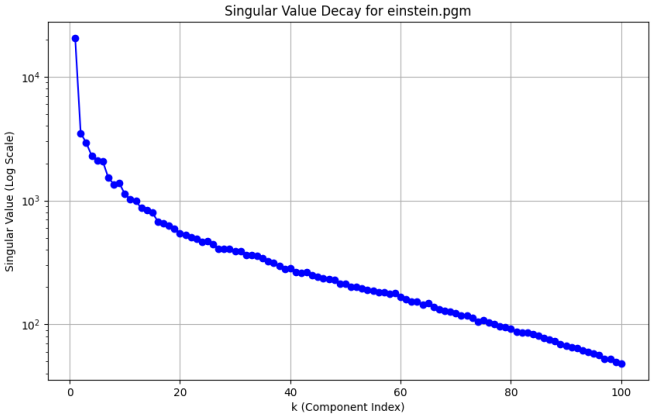
The program was extended to handle color images in the PPM (P3) format. This was achieved by treating a color image not as a single 3D matrix, but as three separate $m \times n$ matrices: one for the Red channel (R), one for the Green channel (G), and one for the Blue channel (B). The entire SVD compression and reconstruction algorithm is run on each of the three matrices *independently*.

1) $R \rightarrow \text{SVD}(R, k) \rightarrow R_k$
2) $G \rightarrow \text{SVD}(G, k) \rightarrow G_k$
3) $B \rightarrow \text{SVD}(B, k) \rightarrow B_k$

The final reconstructed color image is formed by combining the three reconstructed matrices (R_k, G_k, B_k) back into a single PPM file.

C. Bonus: Visualization of Singular Values

The C program was modified to save the singular values for the highest k-value to a text file (e.g., `singular_values.txt`). The Python helper script then reads this file and uses `matplotlib` to generate a log-scale plot. This plot (see Figure in results) visually confirms that the singular values decay exponentially, which is why truncated SVD is so effective.



plot

D. Bonus: NumPy Runtime Comparison

The Python helper script `convert.py` also performs a runtime comparison. It uses the `subprocess` module to execute and time the compiled C program. It then performs the same SVD operation using NumPy's highly-optimized `numpy.linalg.svd` function. This allows for a direct comparison of the C implementation's performance against an industry-standard library.

VI. Conclusion

This project helped me understand how the base image compression works and doing it entirely in C gave me valuable insights on how the model works .