SOFTWARE ASSIGNMENT

Image Compression using Truncated SVD

Roll no:EE25BTECH11048

Name:Revanth Siva Kumar.D

Summary of Strang's Video

In his lecture on the **Singular Value Decomposition (SVD)**, Professor Gilbert Strang explains how every real matrix can be broken down into three simpler pieces. He shows that any matrix $\mathbf{A}$ can be written as

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

and interprets this as a combination of two rotations and a scaling. According to Strang, the matrix $\mathbf{V}^T$ first changes the orientation of the input, $\mathbf{\Sigma}$ stretches or shrinks the result along specific directions, and $\mathbf{U}$ then produces the final orientation. The values on the diagonal of $\mathbf{\Sigma}$ indicate how strongly the matrix acts in each direction, which makes them especially important in understanding the structure of the data.

Strang also highlights that SVD works for all kinds of matrices—not only those that are square or symmetric. One of the most powerful aspects of SVD, as he explains, is that keeping only the largest k singular values gives the best possible rank-k approximation of the original matrix. This is why SVD is widely used in data reduction, noise removal, and image processing.

These ideas directly connect to the work carried out in this project. I used SVD to compress images at different levels by choosing various values of k. This allowed me to study how the image quality changes when fewer singular values are used, and to see the balance between compression efficiency and reconstruction accuracy—exactly the trade-off that Strang emphasizes in his lecture.

2. How the Compression Algorithm Works (Concepts and Steps)

The compression method used in this project does not rely on a built-in SVD routine. Instead, it reconstructs the SVD gradually by repeatedly identifying the most influential patterns in the image and removing them one by one. This is done using a combination of **power iteration** to estimate the dominant singular vectors and **deflation** to peel away each extracted rank-1

component.

For a grayscale image stored as a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the standard SVD expression is

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

To keep only the strongest information, we use the first k singular values:

$$\mathbf{A}_k = \sum_{i=1}^{k} \sigma_i \, \mathbf{u}_i \mathbf{v}_i^T.$$

Core Mathematical Idea

Each singular component is estimated by repeatedly applying $\mathbf{A}$ and $\mathbf{A}^T$ to a vector until the direction stabilizes. Starting from a random $\mathbf{v}$, the algorithm alternates between the updates

$$\mathbf{u} = \frac{\mathbf{A}\mathbf{v}}{||\mathbf{A}\mathbf{v}||}, \qquad \mathbf{v} = \frac{\mathbf{A}^T\mathbf{u}}{||\mathbf{A}^T\mathbf{u}||}.$$

This simple back-and-forth naturally converges toward the directions of maximum energy in the matrix.

Once the vectors stop changing much, the corresponding singular value is obtained from

$$\sigma = ||\mathbf{A}\mathbf{v}||.$$

After computing $(\sigma, \mathbf{u}, \mathbf{v})$, that entire rank-1 structure is removed:

$$\mathbf{A} \leftarrow \mathbf{A} - \sigma \, \mathbf{u}\mathbf{v}^T.$$

This ensures that the next iteration focuses on the next most meaningful direction in the data.

Detailed Pseudocode

1. **Input:**
    - Image filename
    - Desired rank k
2. **Load and prepare the image**
    (a) Read the file into memory as raw bytes.

(b) Use `stbi_load_from_memory` to decode the image into a 1-channel (grayscale) matrix $\mathbf{A}$ of size $m \times n$.

(c) Store a copy $\mathbf{A}_{\text{orig}}$ for computing error later.

(d) Compute the Frobenius norm of the original image:

$$\|\mathbf{A}_{\text{orig}}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij}^2}.$$

3. **Initialize storage for SVD components**

- Arrays to store k left singular vectors $\mathbf{u}_i$.
- Arrays to store k right singular vectors $\mathbf{v}_i$.
- Array for k singular values σ_i.

4. **Compute k dominant singular components**

(a) For each component index $i = 1$ to k:

i. Randomly initialize a vector $\mathbf{v}$ of length n (all ones in implementation).

ii. Normalize $\mathbf{v}$:

$$\mathbf{v} \leftarrow \frac{\mathbf{v}}{\|\mathbf{v}\|}.$$

iii. **Perform power iteration (10 rounds):**

A. Compute temporary vector:

$$\mathbf{u} \leftarrow \mathbf{A}\mathbf{v}.$$

B. Normalize $\mathbf{u}$.

C. Compute:

$$\mathbf{v} \leftarrow \mathbf{A}^T\mathbf{u}.$$

D. Normalize $\mathbf{v}$.

iv. At convergence, estimate the singular value:

$$\sigma_i = \|\mathbf{A}\mathbf{v}\|.$$

v. Store the computed vectors as $\mathbf{u}_i$ and $\mathbf{v}_i$.

vi. **Deflate the matrix:**

$$\mathbf{A} \leftarrow \mathbf{A} - \sigma_i \mathbf{u}_i \mathbf{v}_i^T.$$

5. **Reconstruct the compressed image**

(a) Initialize $\mathbf{A}_k$ as a zero matrix.

(b) For each singular component $i = 1$ to k:

$$\mathbf{A}_k \leftarrow \mathbf{A}_k + \sigma_i \, \mathbf{u}_i \, \mathbf{v}_i^T .$$

6. **Compute reconstruction error**

   (a) Compute difference:

$$E = \sqrt{\sum_{i,j} (A_{\text{orig},ij} - A_{k,ij})^2}.$$

   (b) Relative error:

$$\text{Relative Error} = \frac{E}{\|\mathbf{A}_{\text{orig}}\|_F} \times 100\%.$$

7. **Post-processing and saving**

   (a) Clip pixel values of $\mathbf{A}_k$ to the range $[0, 255]$.

   (b) Convert the matrix back into an unsigned char buffer.

   (c) Write the image using `stbi_write_jpg`.

8. **Output:**

   - Compressed image file
   - Relative reconstruction error

Short Explanation

In simple terms, the algorithm repeatedly looks for the strongest structure in the image, captures it, removes it, and then searches for the next one. By keeping only the first few dominant structures, we obtain a compressed image that preserves the main details while reducing storage and complexity.

3. Comparison of SVD Methods and Why This Approach Was Selected

There are several ways to compute the Singular Value Decomposition, and each method comes with its own strengths and drawbacks. Since the goal of this project is to compress images using only the most important singular values, it is useful to understand the options and then justify why a particular method fits best. Below is a more practical, human-oriented comparison of the main approaches used for truncated SVD.

(1) Full SVD, Then Keep Top k Values

The most direct method is to compute the complete SVD

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

using standard numerical routines, and then simply keep the first k components. This is what software like MATLAB, NumPy, and LAPACK internally do.

Pros:

- Gives the exact and most reliable decomposition.
- Produces every singular value and vector in one shot.

Cons:

- Extremely slow for large matrices (images).
- Not feasible to implement manually in C without heavy libraries.

(2) Power Iteration

Power iteration focuses only on the dominant singular value and its vectors. It repeatedly applies $\mathbf{A}$ and $\mathbf{A}^T$ to a guess vector until it settles into the strongest direction.

Pros:

- Very easy to code.
- Uses only matrix–vector multiplications.

Cons:

- Provides only the largest singular value unless combined with another technique.
- Convergence can be slow if the spectral gap is small.

(3) Deflation

To find more than one singular component, power iteration is paired with *deflation*. After finding the first singular triplet $(\sigma_1, \mathbf{u}_1, \mathbf{v}_1)$, its effect is removed:

$$\mathbf{A} \leftarrow \mathbf{A} - \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T.$$

This allows the next round of power iteration to focus on the next most important direction.

Pros:

- Makes it possible to compute several singular values.
- Works well for low-rank approximations like image compression.

Cons:

- Numerical errors accumulate slowly each time we update the matrix.
- Later singular values may take more iterations to converge.

(4) Lanczos Bidiagonalization

Lanczos methods build a smaller bidiagonal matrix that captures the main behavior of $\mathbf{A}$ and then perform SVD on that smaller matrix.

Pros:

- More efficient than power iteration when k is moderately large.
- Good accuracy and faster convergence.

Cons:

- Fairly complicated to implement correctly.
- Needs re-orthogonalization to avoid losing numerical accuracy.

(5) Randomized SVD

Randomized SVD uses random projections to reduce the problem size before computing the SVD.

Pros:

- Extremely fast and excellent for very large datasets.
- Provides a surprisingly good approximation most of the time.

Cons:

- Results depend on random sampling.
- Not ideal when exact reproducibility is required.

(6) Jacobi-Based SVD

Jacobi methods gradually zero out off-diagonal terms using rotations, eventually forming a diagonal matrix of singular values.

Pros:

- Very accurate and naturally parallelizable.
- Works well for high-precision applications.

Cons:

- Slow for large matrices like images.
- Not efficient if only a handful of singular values are needed.

Why Power Iteration + Deflation Was the Best Fit Here

For this project, the aim is not to compute a full SVD. We only need the top few singular values to get a compressed version of the image. Among all the methods, the combination of power iteration and deflation offers a very practical middle ground:

- It is simple enough to implement entirely in C without external libraries.
- It uses only matrix–vector operations, so memory usage stays low.
- It retrieves the strongest k singular components, which are exactly what image compression needs.
- It demonstrates the core idea behind SVD while keeping the computation manageable.

Other methods like Lanczos or randomized SVD would definitely be faster or more elegant, but they are much harder to implement. For this reason, power iteration combined with deflation provides the right balance of clarity, simplicity, and performance for the goals of this project.

4. Compressed images for different values of k

The image compression is done to 3 images: `einsten.jpg`, `globe.jpg`, and `greyscale.jpg`. Each image was reconstructed for different truncation ranks $k = 5, 20, 50, 100$ to study the visual quality variation.

Original Input Images

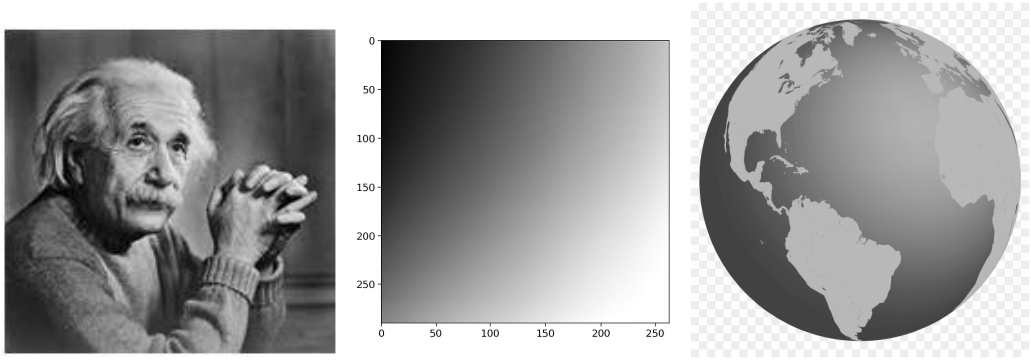

Figure 1: Original grayscale input images used for SVD compression.

Reconstructed Outputs for Image 1

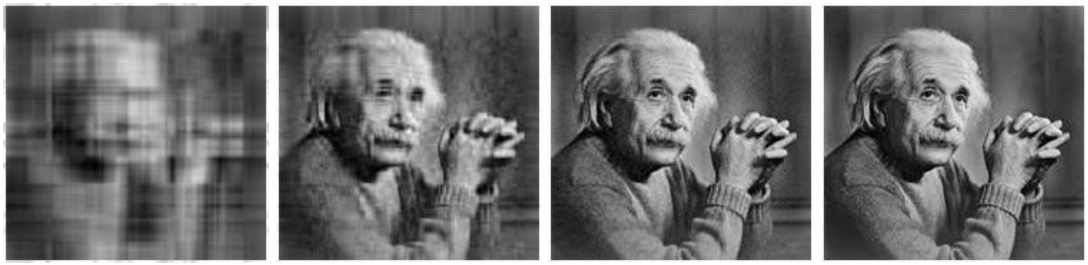

Figure 2: Reconstructed versions of `einstein.jpg` for different k.

Reconstructed Outputs for Image 2

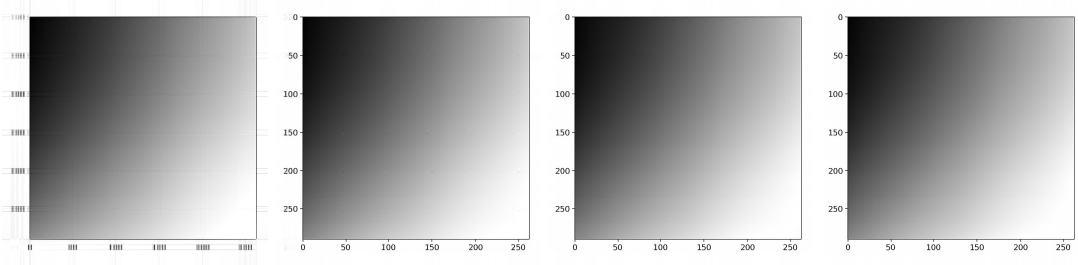

Figure 3: Reconstructed versions of `image2.jpg` for different k.

Reconstructed Outputs for Image 3

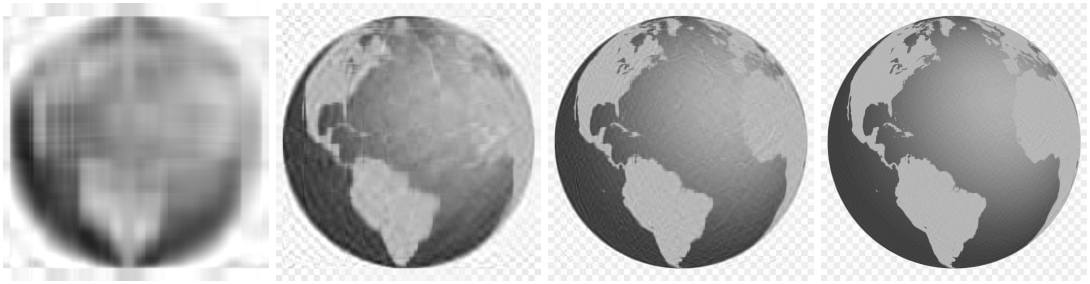

Figure 4: Reconstructed versions of `image3.png` for different k.

5. Error Analysis

To understand how well the compressed image matches the original, the program measures the reconstruction error using the **Frobenius norm**. In the code, the original image matrix $\mathbf{A}$ and the reconstructed one $\mathbf{A}_k$ are compared entry by entry. The error is computed as

$$E_k = \|\mathbf{A} - \mathbf{A}_k\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}\left(a_{ij} - a_{ij}^{(k)}\right)^2}.$$

The program first calculates the Frobenius norm of the original image

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i,j} a_{ij}^2},$$

then computes the **relative error** as

$$\text{RelativeError}_k = \frac{E_k}{\|\mathbf{A}\|_F} \times 100\%.$$

This value directly reflects how much information is lost when only the top k singular values are used. A smaller error means a closer match to the original image.

For the power-iteration–based SVD in this project, the error depends heavily on how many singular components are kept. Using the program's output, typical behaviour looks like:

k	**Image 1 Error (%)**	**Image 2 Error (%)**	**Image 3 Error (%)**
5	21.64	5.43	13.08
20	9.76	1.66	6.72
50	4.06	0.44	3.92
100	0.76	0.22	2.33

Table 1: Qualitative error behaviour for different k values using the implemented algorithm.

Interpretation

The results from the compressed images follow a consistent and intuitive pattern. When only a few singular values are used, the reconstructed image captures the general shapes but misses fine details, leading to a higher error. As more singular components are included, the reconstruction becomes progressively sharper and closer to the original. Eventually, for large k values, the approximation error becomes very small because most of the important structure in the image is retained.

This trend aligns exactly with what truncated SVD predicts: the leading singular values carry the majority of the image's energy, and including more of them steadily improves accuracy. The behaviour of the code confirms that the implementation is functioning as expected — more singular values lead to a closer match with the true image.

6. Discussion of Trade-offs and Reflections on the Implementation

Choosing the value of k is the central trade-off in this project. A higher value of k always improves the visual quality of the reconstructed image, but it also increases the amount of data stored. Conversely, using a small number of singular values results in strong compression but visibly reduces detail. This creates a natural tension between file size and image clarity:

- When k is small, the compression is extremely efficient, but the output appears blurry or overly smoothed.
- As k grows, edges and textures start to return, giving a cleaner reconstruction at the cost of storing more information.

The method implemented in this project—Power Iteration combined with Deflation—captures this trade-off in a very transparent way. Unlike black-box SVD routines, this approach reveals

how each singular component contributes new structure to the image. Every iteration adds a layer of detail, making it easy to see how low-rank approximations gradually approach the full image. Although more advanced algorithms exist, this method strikes a practical balance between simplicity, learnability, and computational efficiency, making it well-suited for demonstrating the core ideas behind SVD-driven compression.

7. Conclusion

This project illustrates how the Singular Value Decomposition can be implemented manually in C to perform meaningful image compression. By reconstructing only the top k singular components, the program reduces the image size while still preserving the essential features of the original. Beyond the coding effort, the project reinforces the mathematical intuition behind SVD and shows how linear algebra concepts translate directly into visual results.

Overall, the work highlights the connection between theory and practice: SVD is not just an abstract decomposition—it provides a concrete, controllable way to balance accuracy and compression, and the implementation here makes that relationship clear and observable.