Software Project Report

Image Compression (SVD)

ee25btech11056 - Suraj.N

Date: November 8, 2025

Summary of Strang's video

Every real (or complex) matrix $\mathbf{A}$ can be factorized as

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

where

$$\mathbf{U}^\top\mathbf{U} = \mathbf{I}, \quad \mathbf{V}^\top\mathbf{V} = \mathbf{I}, \quad \mathbf{\Sigma} = \begin{pmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_r \end{pmatrix} \tag{2}$$

The σ_i are called the **singular values** of $\mathbf{A}$ and are always non-negative:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0 \tag{3}$$

Conceptual Meaning:

1) $\mathbf{V}$ contains an orthonormal basis $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ for the row space.
2) $\mathbf{U}$ contains an orthonormal basis $\{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_m\}$ for the column space.
3) $\mathbf{A}$ maps $\mathbf{v}_i$ to $\sigma_i\mathbf{u}_i$, that is:

$$\mathbf{A}\mathbf{v}_i = \sigma_i\mathbf{u}_i \tag{4}$$

Finding the Components:

Compute

$$\mathbf{A}^\top\mathbf{A} = \mathbf{V}\mathbf{\Sigma}^\top\mathbf{\Sigma}\mathbf{V}^\top \tag{5}$$

Thus, $\mathbf{A}^\top\mathbf{A}$ is symmetric and positive semi-definite. The eigenvectors of $\mathbf{A}^\top\mathbf{A}$ form the columns of $\mathbf{V}$, and the eigenvalues are σ_i^2.

Similarly,

$$\mathbf{A}\mathbf{A}^\top = \mathbf{U}\mathbf{\Sigma}\mathbf{\Sigma}^\top\mathbf{U}^\top \tag{6}$$

The eigenvectors of $\mathbf{A}\mathbf{A}^\top$ form the columns of $\mathbf{U}$, and the eigenvalues are again σ_i^2.

Summary of Relations:

$$\mathbf{A}\mathbf{v}_i = \sigma_i\mathbf{u}_i, \qquad \mathbf{A}^\top\mathbf{u}_i = \sigma_i\mathbf{v}_i \tag{7}$$

Special Case: Symmetric Positive Definite Matrix

If $\mathbf{A}$ is symmetric and positive definite:

$$\mathbf{A} = \mathbf{A}^\top, \quad \lambda_i > 0 \tag{8}$$

then $\mathbf{U} = \mathbf{V}$ and $\mathbf{\Sigma}$ contains the positive eigenvalues λ_i:

$$\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top \tag{9}$$

The program performs image compression by approximating the image matrix $\mathbf{A}$ using a truncated Singular Value Decomposition (SVD).

A grayscale image can be represented as a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ where each element A_{ij} represents the intensity of a pixel ranging from 0 to 255.

Goal: Approximate $\mathbf{A}$ using a lower-rank matrix $\mathbf{A}_k$ such that

$$\mathbf{A} \approx \mathbf{A}_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top \tag{10}$$

where σ_i are the singular values, and $\mathbf{u}_i$, $\mathbf{v}_i$ are the left and right singular vectors respectively.

This reduces the storage size while retaining most of the visual information.

—

Mathematical Concept :

The Singular Value Decomposition (SVD) of a matrix $\mathbf{A}$ is given by:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top \tag{11}$$

where

- $\mathbf{U} \in \mathbb{R}^{m \times m}$ is an orthogonal matrix containing left singular vectors.
- $\mathbf{V} \in \mathbb{R}^{n \times n}$ is an orthogonal matrix containing right singular vectors.
- $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$ is a diagonal matrix containing singular values $\sigma_1, \sigma_2, \ldots, \sigma_r$ where $r = \text{rank}(\mathbf{A})$.

The best rank-k approximation of $\mathbf{A}$ is:

$$\mathbf{A}_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top \tag{12}$$

—

Implementation Logic :

The program approximates the leading singular vectors using the **Power Iteration method**. The steps are as follows:

1) Initialize a random vector $\mathbf{v} \in \mathbb{R}^n$.

2) Iteratively compute

$$\mathbf{v} \leftarrow \frac{\mathbf{A}^\top \mathbf{A}\mathbf{v}}{||\mathbf{A}^\top \mathbf{A}\mathbf{v}||} \tag{13}$$

to find the dominant right singular vector.

3) Compute

$$\mathbf{u} = \frac{\mathbf{A}\mathbf{v}}{||\mathbf{A}\mathbf{v}||} \tag{14}$$

as the corresponding left singular vector.

4) Compute the dominant singular value as

$$\sigma = ||\mathbf{A}\mathbf{v}|| \tag{15}$$

5) Construct the rank-1 approximation:

$$\mathbf{A}_1 = \sigma \mathbf{u}\mathbf{v}^\top \tag{16}$$

6) Deflate the matrix:

$$\mathbf{A} \leftarrow \mathbf{A} - \mathbf{A}_1 \tag{17}$$

7) Repeat steps 1 - 6 for k iterations to obtain a rank-k approximation.

—

Pseudo Code :

1) **Input:** Image file $\rightarrow$ grayscale matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$

2) **Parameters:** target ranks $\{k\}$, iterations per singular vector N (e.g. $N = 50$)

3) **Output:** low-rank reconstruction $\mathbf{A}_k$ and error E_k

4) **Algorithm:**

a) Set $\mathbf{A}_{\text{temp}} \leftarrow \mathbf{A}$ and $\mathbf{A}_k \leftarrow \mathbf{0}_{m \times n}$

b) **for** $t = 1 \ldots k$ **do**

i) Initialize random vector $\mathbf{v} \in \mathbb{R}^n$ and normalize:

$$\mathbf{v} \leftarrow \frac{\mathbf{v}}{||\mathbf{v}||} \tag{18}$$

ii) **Power iteration:** repeat $i = 1 \ldots N$

$$\mathbf{w} \leftarrow \mathbf{A}_{\text{temp}}\, \mathbf{v} \tag{19}$$

$$\mathbf{v} \leftarrow \mathbf{A}_{\text{temp}}^{\top} \mathbf{w} \tag{20}$$

$$\mathbf{v} \leftarrow \frac{\mathbf{v}}{||\mathbf{v}||} \tag{21}$$

iii) Compute left singular vector and singular value:

$$\mathbf{u} \leftarrow \frac{\mathbf{A}_{\text{temp}}\, \mathbf{v}}{||\mathbf{A}_{\text{temp}}\, \mathbf{v}||} \tag{22}$$

$$\sigma \leftarrow ||\mathbf{A}_{\text{temp}}\, \mathbf{v}|| \tag{23}$$

iv) Rank-1 update and deflation:

$$\mathbf{A}_k \leftarrow \mathbf{A}_k + \sigma\, \mathbf{u}\mathbf{v}^{\top} \tag{24}$$

$$\mathbf{A}_{\text{temp}} \leftarrow \mathbf{A}_{\text{temp}} - \sigma\, \mathbf{u}\mathbf{v}^{\top} \tag{25}$$

v) If σ is numerically zero (below tolerance), **break**

c) **end for**

d) Compute Frobenius error:

$$E_k = ||\mathbf{A} - \mathbf{A}_k||_F \tag{26}$$

(Mathematical concept for Power Iteration) :

Let $\mathbf{A}$ be a matrix. We want to find the eigenvector corresponding to the largest eigenvalue of $\mathbf{A}$. Let $\mathbf{v}_0$ be an arbitrary initial vector that has a nonzero component in the direction of the dominant eigenvector.

The eigen-decomposition of $\mathbf{A}$ is given by:

$$\mathbf{A} = \mathbf{X}\mathbf{\Lambda}\mathbf{X}^{-1} \tag{27}$$

where $\mathbf{X}$ contains the eigenvectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$, and $\mathbf{\Lambda}$ is a diagonal matrix with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ such that

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n| \tag{28}$$

The initial vector $\mathbf{v}_0$ can be expressed as a linear combination of the eigenvectors:

$$\mathbf{v}_0 = c_1\mathbf{x}_1 + c_2\mathbf{x}_2 + \cdots + c_n\mathbf{x}_n \tag{29}$$

Applying $\mathbf{A}$ repeatedly gives:

$$\mathbf{v}_k = \mathbf{A}^k\mathbf{v}_0 \tag{30}$$

Substitute the eigen-decomposition of $\mathbf{A}$:

$$\mathbf{v}_k = \mathbf{A}^k\mathbf{v}_0 = \mathbf{X}\mathbf{\Lambda}^k\mathbf{X}^{-1}\mathbf{v}_0 \tag{31}$$

Expanding using the eigenvector basis:

$$\mathbf{v}_k = c_1\lambda_1^k\mathbf{x}_1 + c_2\lambda_2^k\mathbf{x}_2 + \cdots + c_n\lambda_n^k\mathbf{x}_n \tag{32}$$

Divide both sides by λ_1^k:

$$\frac{\mathbf{v}_k}{\lambda_1^k} = c_1\mathbf{x}_1 + c_2\left(\frac{\lambda_2}{\lambda_1}\right)^k\mathbf{x}_2 + \cdots + c_n\left(\frac{\lambda_n}{\lambda_1}\right)^k\mathbf{x}_n \tag{33}$$

As k = large value, since $\left|\frac{\lambda_i}{\lambda_1}\right| < 1$ for $i > 1$, the higher-order terms vanish:

$$\frac{\mathbf{v}_k}{\lambda_1^k} \approx c_1\mathbf{x}_1 \tag{34}$$

Hence, after normalization, $\mathbf{v}_k$ converges to the dominant eigenvector $\mathbf{x}_1$ corresponding to the largest eigenvalue λ_1.

So,using this power iteration technique , we can obtain the dominant eigen vector of $\mathbf{A}^\top\mathbf{A}$ and dominant singular value corresponding to it.

—

Error Calculation :

The reconstruction error between the original and compressed images is given by the Frobenius norm:

$$E = \|\mathbf{A} - \mathbf{A}_k\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}(A_{ij} - A_{k,ij})^2} \tag{35}$$

A smaller E indicates better approximation, but higher compression requires a smaller k.

—

Reason for choosing Power Iteration :

1) **Simple to Implement :** The Power Iteration algorithm involves only basic matrix-vector multiplications and normalization steps. It does not require complex matrix factorizations or transformations.
2) **Memory Efficient :** It stores only a few vectors and intermediate results, making it well-suited for large matrices and image data where memory efficiency is crucial.
3) **Efficient for Top-k Singular Values :** The method directly computes the dominant singular vectors corresponding to the largest singular values without computing the full decomposition. Hence, it is computationally optimal for truncated SVD and image compression.

Reasons for not choosing other algorithms :

1) **Jacobi Algorithm :**
   a) Slow convergence for large matrices.
   b) Requires multiple sweeps to achieve numerical stability.
   c) Higher computational complexity $O(n^3)$.
   d) Inefficient for sparse matrices.
2) **Golub Kahan Bidiagonalization :**
   a) Complex multi-step procedure involving orthogonal transformations.
   b) Implementation complexity is high for general-purpose image data.
   c) Not memory efficient as it requires storing orthogonal matrices **U** and **V** explicitly.
3) **Divide and Conquer SVD :**
   a) Requires recursive decomposition and back-substitution steps.
   b) Implementation is mathematically and programmatically complex.
   c) Memory intensive for large matrices due to intermediate subproblems.

Power Iteration

The power iteration algorithm used in the code performs 50 iterations for each singular vector computation. This number of iterations is chosen to ensure a good trade-off between computational efficiency and convergence accuracy.

$$\mathbf{v}_{k+1} = \frac{\mathbf{A}^\top \mathbf{A}\mathbf{v}_k}{\|\mathbf{A}^\top \mathbf{A}\mathbf{v}_k\|} \tag{36}$$

Each iteration brings the vector $\mathbf{v}_k$ closer to the dominant eigenvector of $\mathbf{A}^\top\mathbf{A}$ corresponding to the largest singular value. For most image matrices, convergence to within machine precision occurs in fewer than 50 iterations, but fixing it to 50 iterations guarantees stable results even for ill-conditioned matrices.

Numerical Stability and Normalization

At each iteration, normalization of the vectors $\mathbf{u}$ and $\mathbf{v}$ is crucial for preventing numerical overflow and underflow. The normalization ensures that the magnitude of the iterates remains bounded, i.e.,

$$\mathbf{v}_{k+1} = \frac{\mathbf{v}_{k+1}}{\|\mathbf{v}_{k+1}\|} \tag{37}$$

Without this step, repeated multiplication by $\mathbf{A}$ could cause exponential growth or decay depending on the magnitude of the eigenvalues, leading to instability or loss of precision. Hence, normalization preserves numerical stability throughout the iterations.

Significance of Singular Value Decay in Images

In image matrices, the singular values $\sigma_1, \sigma_2, \ldots, \sigma_n$ typically exhibit rapid decay, meaning that the first few singular values capture most of the image's energy and structural information.

$$\text{Energy captured by top-}k\text{ singular values} = \frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{n} \sigma_i^2} \tag{38}$$

Larger singular values correspond to dominant image features such as edges and intensity gradients, while smaller singular values represent finer details or noise. Thus, retaining only the largest k singular values leads to significant compression with minimal perceptual loss in image quality.

Frobenius Norm Error and Image Quality

The reconstruction error between the original and the approximated image is computed using the Frobenius norm:

$$E_k = \|\mathbf{A} - \mathbf{A}_k\|_F \tag{39}$$

As k increases, the approximation $\mathbf{A}_k$ includes more singular components, reducing the Frobenius error monotonically:

$$E_{k+1} < E_k \tag{40}$$

The Frobenius norm error correlates directly with the visual quality of the reconstructed image:

1) For small k (e.g., $k = 5$), E_k is large, and the image appears blurry due to the loss of high-frequency information.
2) As k increases (e.g., $k = 50$ or $k = 100$), E_k decreases, and the image quality improves, retaining sharper details and textures.

Observations and Limitations

1) The algorithm effectively captures the main image features even for small k values due to the rapid decay of singular values.
2) Beyond a certain k, visual improvement becomes negligible while computation time and memory usage increase.
3) Power iteration assumes distinct dominant singular values; convergence can slow down if multiple singular values are close in magnitude.
4) The normalization step ensures numerical stability, but round-off errors may still accumulate slightly for very large matrices.

Trade-off Between Rank k, Image Quality, and Compression

The parameter k in the truncated Singular Value Decomposition (SVD) directly controls the balance between **image quality** and **compression ratio**. Choosing an appropriate k is crucial for achieving efficient compression without significant visual degradation.

Mathematical Background

The original image matrix $\mathbf{A}$ of size $m \times n$ can be represented as:

$$\mathbf{A} = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top \tag{41}$$

where $r = \text{rank}(\mathbf{A})$, σ_i are singular values with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r$, and $\mathbf{u}_i$, $\mathbf{v}_i$ are the left and right singular vectors, respectively.

In the low-rank approximation, only the top k singular values and vectors are used:

$$\mathbf{A}_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top \tag{42}$$

Compression Ratio

The compression ratio measures how efficiently the image is stored. The number of values required to store $\mathbf{A}_k$ is:

$$N_k = k(m + n + 1) \tag{43}$$

where m and n are the image dimensions and 1 accounts for each singular value σ_i. For the original image:

$$N_{\text{original}} = m \times n \tag{44}$$

Hence, the compression ratio is:

$$\text{Compression Ratio} = \frac{N_{\text{original}}}{N_k} = \frac{mn}{k(m + n + 1)} \tag{45}$$

As k decreases, the compression ratio increases (higher compression), but image quality deteriorates.

Image Quality and Energy Preservation

The fraction of image energy preserved in $\mathbf{A}_k$ is given by:

$$E_k = \frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{r} \sigma_i^2} \tag{46}$$

A larger k retains more energy and thus higher image quality, while a smaller k retains less energy, leading to visible loss of details.

Trade-off Discussion

1) For small k, only a few singular components are retained. The image is highly compressed, but edges and fine textures become blurred.
2) For large k, more singular values are included, resulting in better reconstruction but larger storage requirements.
3) Beyond a certain k, the increase in image quality is minimal, while storage and computation costs grow significantly.

Optimal Choice of k

The optimal k depends on the desired balance between visual quality and compression. A practical approach is to select k such that E_k exceeds a chosen threshold (e.g., 90% or 95% of total energy).

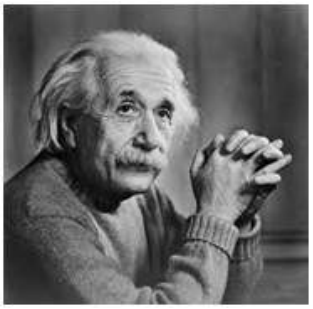
$$E_k \geq 0.95 \tag{47}$$

Summary of the Trade-off

1) Increasing $k \Rightarrow$ higher image quality, lower compression efficiency.
2) Decreasing $k \Rightarrow$ higher compression, lower image quality.
3) The ideal k achieves sufficient perceptual quality while minimizing storage and computation.

Images

(a) Original einstein Image

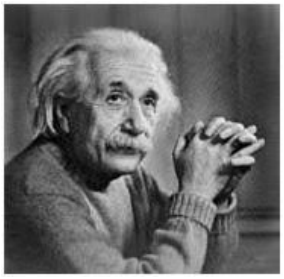


Original Image

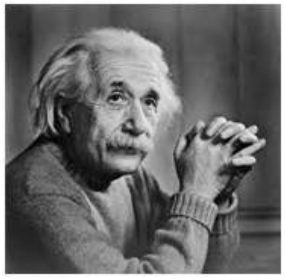
Compressed Images

k=5

k=20



k=50



k=100

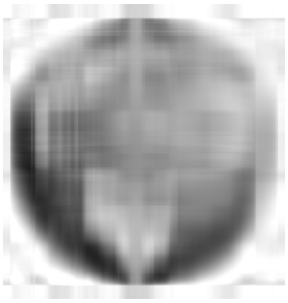
k	**Runtime (s)**	**Frobenius Error**
5	0.08	4714.53
20	0.26	2126.56
50	0.65	880.51
100	1.30	164.78

Table : einstein.jpg

(b) Original globe Image

Original Image

Compressed Images



k=5

k=20

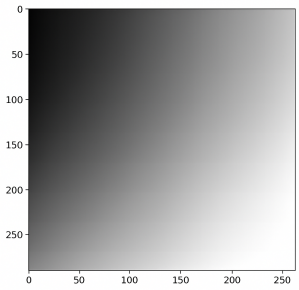
k=50

k=100

k	**Runtime (s)**	**Frobenius Error**
5	1.64	20704.27
20	6.37	10634.88
50	16.08	6185.65
100	31.52	3673.31

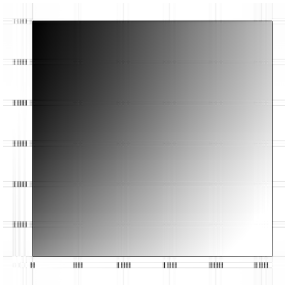
Table : globe.jpg

(c) Original greyscale Image

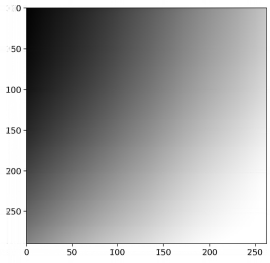


Original Image

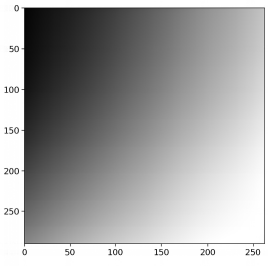
Compressed Images



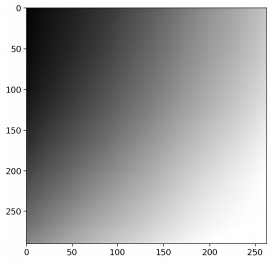
k=5



k=20



k=50



k=100

k	**Runtime (s)**	**Frobenius Error**
5	2.48	11088.89
20	10.66	3782.21
50	27.65	1120.90
100	55.74	393.47

Table : greyscale.png

Extension to Colored Images

The previous implementation performed image compression for grayscale images represented by a single intensity matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$. In the extended version, the algorithm is generalized to handle **colored (RGB) images** by independently processing each color channel â Red (**R**), Green (**G**), and Blue (**B**) â using the same truncated SVD-based power iteration method.

—

Algorithm

1) **Image Loading:** The program reads an image file (JPEG or PNG) and determines the number of channels using the `stb_image` library.
   - If the image has 1 channel, it is treated as a grayscale image.
   - If it has 3 channels, it is treated as a color image (RGB).
2) **Channel Separation:** For a color image, the pixel values are separated into three matrices:
$$\mathbf{R},\ \mathbf{G},\ \mathbf{B} \in \mathbb{R}^{m \times n} \tag{48}$$
   where each element represents the intensity of that color component.
3) **Independent Channel Compression:** The `low_rank()` function (based on power iteration and deflation) is applied separately to each channel:
$$\mathbf{R}_k = \text{low\_rank}(\mathbf{R}, k), \quad \mathbf{G}_k = \text{low\_rank}(\mathbf{G}, k), \quad \mathbf{B}_k = \text{low\_rank}(\mathbf{B}, k) \tag{49}$$
   Each $\mathbf{C}_k$ (for $C \in \{R, G, B\}$) represents the rank-k approximation of the corresponding color matrix.
4) **Channel Merging:** After compression, the three reconstructed matrices are merged to form the final RGB image:
$$\mathbf{A}_k(i, j) = \begin{bmatrix} R_k(i, j) & G_k(i, j) & B_k(i, j) \end{bmatrix} \tag{50}$$
5) **Error Calculation:** The Frobenius norm error is computed separately for each color channel:
$$E_R = \|\mathbf{R} - \mathbf{R}_k\|_F, \quad E_G = \|\mathbf{G} - \mathbf{G}_k\|_F, \quad E_B = \|\mathbf{B} - \mathbf{B}_k\|_F \tag{51}$$
   These values provide quantitative measures of reconstruction quality for each channel.
6) **Output:** The compressed image is saved as `base_name_k.jpg` or `base_name_k.png`, depending on the input format.

—

Key Advantages of the Extended Version

1) **Generalized to RGB Images:** The method now supports colored images without any change in the core SVD logic.
2) **Independent Channel Processing:** Each color channel is processed separately, maintaining the natural color structure of the original image.
3) **Parallelizable Implementation:** Since **R**, **G**, and **B** are independent, the compression for each channel can be parallelized in future implementations.
4) **Consistent Error Metrics:** The Frobenius norm error per channel helps evaluate compression quality precisely for each color component.
5) **Scalable to Any Resolution:** The algorithm works for arbitrary image sizes since memory allocation and computation are performed dynamically.

—

Mathematical Representation

For a color image, let the three component matrices be:

$$\mathbf{A}_{RGB} = \{\mathbf{R}, \mathbf{G}, \mathbf{B}\} \tag{52}$$

Each component is approximated as:

$$\mathbf{C}_k = \sum_{i=1}^{k} \sigma_i^{(C)} \mathbf{u}_i^{(C)} \mathbf{v}_i^{(C)^\top}, \quad C \in \{R, G, B\} \tag{53}$$

The reconstructed color image is then:

$$\mathbf{A}_k = \begin{bmatrix} \mathbf{R}_k & \mathbf{G}_k & \mathbf{B}_k \end{bmatrix} \tag{54}$$

Thus, the overall compression preserves the structural features and color distribution while reducing storage requirements by storing only k singular components per channel.

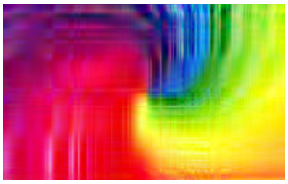
—

Images

Original color image

Original Image

Compressed Images



k=5

k=20

k=50

k=100

k	**Runtime (s)**	**R Channel Error**	**G Channel Error**	**B Channel Error**
5	22.33	40072.61	42626.46	55874.72
20	88.56	22923.95	28886.00	36443.59
50	251.53	12365.71	16384.83	20067.50
100	448.12	6995.57	8617.86	9864.13

Table : color.jpg

Runtime Comparison with Python Numpy

To validate and compare the performance of the C implementation, a similar image compression algorithm is implemented in Python using the built-in `NumPy` library. The `np.linalg.svd()` function directly computes the full Singular Value Decomposition (SVD) of a given matrix.

—

Algorithm

1) **Image Reading:** The image is read using the `PIL.Image` module and converted into a NumPy array $\mathbf{A} \in \mathbb{R}^{m \times n \times c}$, where channels = 1 for grayscale and channels= 3 for RGB images.
2) **Channel Handling:**
   - If the image is grayscale, the SVD is directly applied to $\mathbf{A}$.
   - If the image is colored, the image is divided into its Red, Green, and Blue channels:

$$\mathbf{A}_{RGB} = \{\mathbf{R}, \mathbf{G}, \mathbf{B}\} \tag{55}$$

   Each channel is processed independently.
3) **Singular Value Decomposition:** For each channel matrix $\mathbf{C} \in \{\mathbf{R}, \mathbf{G}, \mathbf{B}\}$, the full SVD is computed as:

$$\mathbf{C} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top \tag{56}$$

4) **Low-Rank Approximation:** The top-k singular values and corresponding vectors are retained:

$$\mathbf{C}_k = \mathbf{U}_{:,1:k}\mathbf{\Sigma}_{1:k,1:k}\mathbf{V}^\top_{:,1:k} \tag{57}$$

   This provides a rank-k approximation of the channel.
5) **Image Reconstruction:** The compressed channels are merged back together to form the reconstructed image:

$$\mathbf{A}_k = \begin{bmatrix} \mathbf{R}_k & \mathbf{G}_k & \mathbf{B}_k \end{bmatrix} \tag{58}$$

   The pixel values are clipped to $[0, 255]$ and saved using `PIL.Image`.

—

Error and Runtime Analysis

The reconstruction error for each channel is computed using the Frobenius norm:

$$E_C = \|\mathbf{C} - \mathbf{C}_k\|_F, \quad C \in \{R, G, B\} \tag{59}$$

The total runtime for each k value is measured using the Python `time` module.

The python code was executed for all the four images and the following resutls were obtained

k	**Runtime (s)**	**Frobenius Error**
5	0.08	4713.34
20	0.07	2126.44
50	0.07	880.35
100	0.08	164.78

Table : einstein.jpg

k	**Runtime (s)**	**Frobenius Error**
5	6.01	20703.89
20	6.28	10633.92
50	6.31	6185.21
100	6.10	3672.61

Table : globe.jpg

k	**Runtime (s)**	**Frobenius Error**
5	10.59	11087.74
20	10.72	3782.74
50	10.71	1122.44
100	11.08	396.82

Table : greyscale.png

k	**Runtime (s)**	**R Channel Error**	**G Channel Error**	**B Channel Error**
5	28.57	40071.30	42632.25	55872.70
20	27.41	22924.77	28887.07	36442.29
50	27.59	12367.51	16386.31	20066.60
100	27.82	6997.62	8617.40	9861.32

Table : color.jpg

—

Performance Comparison and Observations

1) The `np.linalg.svd()` function computes the **complete SVD** of the matrix, regardless of the chosen k. Hence, the runtime remains approximately constant for $k = 5, 20, 50$, and 100.
2) Since NumPy internally performs the full decomposition,it is relatively fast but still computationally more expensive for small k, because unnecessary singular vectors are computed.
3) In contrast, the C implementation using the **Power Iteration Method** computes only the top-k singular values and vectors. This makes it much faster for small k values, as no full decomposition is performed.
4) For larger k values, however, the gap between both implementations narrows. The full SVD in NumPy becomes competitive, sometimes outperforming the C power iteration for very high k values.
5) In summary:
   - For small k (e.g., $k = 5$ or $k = 20$): Power iteration (C) is significantly faster.
   - For large k (e.g., $k = 50$ or $k = 100$): NumPy's complete SVD may approach or exceed the C implementation in speed due to highly optimized matrix operations.

—

Discussion

The comparison highlights the trade-off between **algorithmic specificity** and **library optimization**:

- Power Iteration (C) is efficient for computing only a few dominant singular values and vectors, suitable for real-time compression.
- Full SVD (NumPy) is more general and numerically robust, but performs redundant computation when only a truncated approximation is required.

Therefore, for image compression applications where only the top-k components are needed, the iterative C implementation remains more efficient and memory-conscious.