IMAGE COMPRESSION USING TRUNCATED SVD WITH QR-BASED EIGENVALUE DECOMPOSITION - SOFTWARE PROJECT

Bhargav K - EE25BTECH11013

CONTENTS

I. INTRODUCTION

Any digital image consists of a huge number of pixels, where each pixel stores colour information. For grayscale images, these pixel values can be represented in a matrix form $\mathbf{A} \in \mathbf{R}^{m \times n}$, where m and n are the number of rows and columns respectively. The pixel values usually range from 0(black) to 255(white). As the resolution of the image increases, the size of these matrices grow tremendously, thereby occupying more storage. Thus, efficient ways of image compression are required in signal processing and machine learning and many other fields.

II. SINGULAR VALUE DECOMPOSITION

To solve this problem, the method of Singular Value Decomposition(SVD) can be used. In this method, any matrix can be represented as the product of 3 other simpler matrices. These 3 matrices contribute to the geometric feature of the original matrix.

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

where $\mathbf{U}$ and $\mathbf{V}^{\mathbf{T}}$ are the orthogonal matrices containing the left and right singular vectors respectively. $\mathbf{\Sigma}$ is a diagonal matrix consisting of the singular values.
These singular values capture the amount of information about the image along each principal direction.

III. TRUNCATED SVD

A remarkable property of SVD is that the image can be accurately approximated by retaining only the largest k singular values by discarding the smaller ones. This produces a matrix

$$\mathbf{A}_k = \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}^{\mathbf{T}}{}_k \tag{2}$$

which is a k-rank matrix(Since all the columns are linearly independent). This matrix captures the most significant features of the image while using much less data.

IV. SUMMARY OF DR GILBERT STRANG'S VIDEO

Any matrix can be written as

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

where $\mathbf{U}$ and $\mathbf{V}^{\mathbf{T}}$ are the orthogonal matrices, $\mathbf{\Sigma}$ is a diagonal matrix consisting of the singular values.

Eigenvalue decomposition can be used only for square matrices, while SVD can be used for any rectangular or non-symmetric matrices.

A matrix acts as a linear map from one vector space to another. SVD decomposes the matrix into 3 simpler operations:
(1) reflection/rotation $\mathbf{V}^{\mathbf{T}}$
(2) stretching Σ
(3) Second rotation/reflection $\mathbf{U}$

The matrix $\mathbf{V^T}$ rotates the input space, $\mathbf{\Sigma}$ scales the result along the new axis, $\mathbf{U}$ rotates the final output. Singular Values indicate the importance of each component of the transformation.
Some singular values may have value = 0, and these contribute to the null space of the matrix $\mathbf{A}$. Larger singular values correspond to larger stretches of the transformation.

(1) Finding $\mathbf{V}$ and $\mathbf{\Sigma}$:

Multiply $\mathbf{A}^T$ and $\mathbf{A}$

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

$$\mathbf{A}^T\mathbf{A} = \mathbf{V}\mathbf{\Sigma}^2 = \mathbf{V}\mathbf{\Sigma}^T\left(\mathbf{U}^T\mathbf{U}\right)\mathbf{\Sigma}\mathbf{V}^T \tag{5}$$

Since $\mathbf{U}$ is an orthogonal matrix, $\mathbf{U}^T\mathbf{U} = \mathbf{I}$. Also, $\mathbf{\Sigma}$ is a diagonal matrix. So $\mathbf{\Sigma}^T\mathbf{\Sigma} = \mathbf{\Sigma}^2$

$$\mathbf{A}^T\mathbf{A} = \mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T \tag{6}$$

Here $\mathbf{V}$ contain the eigenvectors of $\mathbf{A}^T\mathbf{A}$.
So, after finding the expression of $\mathbf{A}^T\mathbf{A}$, we find the square root of the diagonal elements of the $\mathbf{\Sigma}^2$, which gives the singular values of the matrix $\mathbf{A}$.

(2) Finding $\mathbf{U}$:

Multiply $\mathbf{A}$ and $\mathbf{A}^T$

$$\mathbf{A}\mathbf{A}^T = \left(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\right)\left(\mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T\right) \tag{7}$$

Since $\mathbf{V}$ is an orthogonal matrix, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$. Also, $\mathbf{\Sigma}$ is a diagonal matrix. So $\mathbf{\Sigma}^T\mathbf{\Sigma} = \mathbf{\Sigma}^2$

$$\mathbf{A}\mathbf{A}^T = \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T \tag{8}$$

Here $\mathbf{U}$ contain the eigenvectors of $\mathbf{A}\mathbf{A}^T$.
Here $\mathbf{\Sigma}$ and $\mathbf{U}$ and $\mathbf{V}$ can be found out by finding the eigenvalues and eigenvectors appropriately.

V. ALGORITHM AND MATHEMATICAL EXPLANATION - QR ITERATION-BASED EIGENVALUE DECOMPOSITION

A. Overview

The objective is to compute the truncated Singular Value Decomposition (SVD) of a matrix. Given a matrix $\mathbf{A}$ of size $m \times n$, SVD decomposes it as:

$$\mathbf{A} = \mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^T \tag{9}$$

where $\mathbf{U}$ is $m \times m$ (left singular vectors), $\mathbf{\Sigma}$ is $m \times n$ rectangular diagonal (singular values), and $\mathbf{V}$ is $n \times n$ (right singular vectors).

For truncated SVD, we retain only the top k singular values:

$$\mathbf{A_k} = \mathbf{U_k}\,\mathbf{\Sigma_k}\,\mathbf{V_k}^T \tag{10}$$

B. MATHEMATICAL BACKGROUND

1) Relating SVD to Eigen Decomposition: We observe that

$$\mathbf{A}^T\mathbf{A} = \mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T \tag{11}$$

Hence, the columns of $\mathbf{V}$ are eigenvectors of $\mathbf{A}^T\mathbf{A}$, and the eigenvalues are σ_i^2.
Similarly,

$$\mathbf{A}\mathbf{A}^T = \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T \tag{12}$$

Thus, we compute eigenvectors of $\mathbf{A}^T\mathbf{A}$ to get $\mathbf{V}$. Then, each column $\mathbf{U_i}$ is computed using:

$$\mathbf{U_i} = \frac{\mathbf{A}\mathbf{V_i}}{\sigma_i} \tag{13}$$

2) QR Iteration for Eigenvalues: To find eigenvalues of a symmetric matrix, we repeatedly perform QR decomposition.

3) Gram-Schmidt Orthogonalization: Given vectors $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$, we construct orthonormal vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n$ using

$$\mathbf{q}_1 = \frac{\mathbf{a}_1}{\|\mathbf{a}_1\|} \tag{14}$$

$$\mathbf{q}_k = \frac{\mathbf{a}_k - \sum_{i=1}^{k-1} \left\|\mathbf{a}_k^T\mathbf{q}_i\right\| \mathbf{q}_i}{\left\|\mathbf{a}_k - \sum_{i=1}^{k-1} \left(\mathbf{a}_k^T\mathbf{q}_i\right)\mathbf{q}_i\right\|}, \qquad k = 2, \ldots, n \tag{15}$$

Each new vector is formed by subtracting projections onto previously computed orthonormal vectors and normalizing. This leads to numerical stability and ensures $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$.

In practice, Modified Gram-Schmidt is preferred because it reduces accumulation of floating-point errors.

For a given symmetric matrix $\mathbf{M}$:

$$\mathbf{M} = \mathbf{Q}\mathbf{R} \tag{16}$$

Then,

$$\mathbf{M}_{\text{next}} = \mathbf{R}\mathbf{Q} \tag{17}$$

Repeated iteration converges to an upper triangular matrix whose diagonal entries are eigenvalues. Accumulating the $\mathbf{Q}$ matrices yields eigenvectors.

4) FINDING EIGENVECTORS BY QR ITERATION: In QR iteration, the eigenvectors of a symmetric matrix are obtained by repeatedly multiplying the orthogonal matrices $\mathbf{Q}$ from each QR decomposition.
We start with

$$\mathbf{A} = \mathbf{Q_1}\mathbf{R_1} \tag{18}$$

Then get

$$\mathbf{A_1} = \mathbf{R_1}\mathbf{Q_1} \tag{19}$$

In the next iteration,

$$\mathbf{A_1} = \mathbf{Q_2}\mathbf{R_2} \tag{20}$$

$$\mathbf{A_2} = \mathbf{R_2}\mathbf{Q_2} \tag{21}$$

Repeating this process for k iterations yields

$$\mathbf{A_k} = \mathbf{R_k}\mathbf{Q_k} \tag{22}$$

Define

$$\mathbf{Q}_{\text{tot}} = \mathbf{Q_1}\mathbf{Q_2}\cdots\mathbf{Q_k} \tag{23}$$

$$\mathbf{A}_k = \mathbf{Q}_{k-1}^T\mathbf{A}_{k-1}\mathbf{Q}_{k-1} \tag{24}$$

Further expanding $\mathbf{A}_{k-1}$, we get:

$$\mathbf{A}_k = (\mathbf{Q}_1\mathbf{Q}_2\cdots\mathbf{Q}_{k-1})^T\,\mathbf{A}\,(\mathbf{Q}_1\mathbf{Q}_2\cdots\mathbf{Q}_{k-1}) \tag{25}$$

$$\implies \mathbf{A_k} = {\mathbf{Q}_{\text{tot}}}^T\mathbf{A}\,\mathbf{Q}_{\text{tot}} \tag{26}$$

As k increases, $\mathbf{A_k}$ converges to a diagonal matrix

$$\mathbf{A_k} \to \mathbf{\Lambda} \tag{27}$$

Therefore,

$$\mathbf{A} \approx \mathbf{Q}_{\text{tot}}\,\mathbf{\Lambda}\,{\mathbf{Q}_{\text{tot}}}^T \tag{28}$$

Hence, the columns of $\mathbf{Q}_{\text{tot}}$ are the eigenvectors of $\mathbf{A}$.
In the implementation, this is achieved by initializing

$$\mathbf{V} = \mathbf{I} \tag{29}$$

where $\mathbf{I}$ is the identity matrix

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \tag{30}$$

Then, during each iteration, we update

$$\mathbf{V} \leftarrow \mathbf{V}\mathbf{Q} \tag{31}$$

After k iterations, we obtain

$$\mathbf{V} = \mathbf{Q_1}\mathbf{Q_2}\cdots\mathbf{Q_k} \tag{32}$$

Thus, $\mathbf{V}$ converges to the eigenvector matrix.

C. Truncated SVD Algorithm

1) Step 1: Compute $\mathbf{A}^T\mathbf{A}$*:*

$$\mathbf{M} = \mathbf{A}^T\mathbf{A} \tag{33}$$

2) Step 2: QR Iteration: Initialize $\mathbf{V}$ as identity

$$\mathbf{V} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \tag{34}$$

Repeatedly:

$$\mathbf{M} = \mathbf{QR} \tag{35}$$

$$\mathbf{M} = \mathbf{RQ} \tag{36}$$

$$\mathbf{V} = \mathbf{VQ} \tag{37}$$

After iterations, diagonal entries of $\mathbf{M}$ approximate eigenvalues λ_i.

3) Step 3: Compute Singular Values:

$$\sigma_i = \sqrt{\lambda_i} \tag{38}$$

4) Step 4: Compute $\mathbf{U}$*:*

$$\mathbf{U_i} = \frac{\mathbf{AV_i}}{\sigma_i} \tag{39}$$

Normalize $\mathbf{U_i}$.

5) Step 5: Truncate: Select top k singular values and vectors:

$$\mathbf{U_k} = (\mathbf{U_1}\ \mathbf{U_2}\ \cdots\ \mathbf{U_k}) \tag{40}$$

$$\mathbf{\Sigma_k} = \begin{pmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_k \end{pmatrix} \tag{41}$$

$$\mathbf{V_k^T} = (\mathbf{V_1}\ \mathbf{V_2}\ \cdots\ \mathbf{V_k})^T \tag{42}$$

6) Step 6: Reconstruct Approximation:

$$\mathbf{A_k} = \mathbf{U_k}\,\mathbf{\Sigma_k}\,\mathbf{V_k}^T \tag{43}$$

D. Python Code

(1)Python code uses ctypes to interface Python with your C implementation of Truncated SVD.
(2)Read the image using Matplotlib and converts NumPy arrays into double** pointers so they can be passed to C.
(3)Stores the result from C into ../figs/
(4)Shows the compressed image using matplotlib.
(5)Calculates the Frobenius error between the original and reconstructed image and reports the normalized error.
(6)Calculates the time taken to run the Truncated SVD function.

VI. PSEUDO-CODE

$$\text{Input: } \mathbf{A},\ m,\ n,\ k \tag{44}$$

$$\mathbf{A}^T \leftarrow \mathbf{A} \tag{45}$$

$$\mathbf{A}^T\mathbf{A} \leftarrow \mathbf{A}^T \cdot \mathbf{A} \tag{46}$$

A. Modified Gram–Schmidt Orthogonalisation

Input: $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$
Output: $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n$

$$\mathbf{u}_k = \mathbf{a}_k \tag{47}$$

For $k = 1$ to n:

$$\mathbf{q}_k = \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|} \tag{48}$$

For $j = k + 1$ to n:

$$r_{kj} = \mathbf{q}_k^T \mathbf{u}_j \tag{49}$$

$$\mathbf{u}_j \leftarrow \mathbf{u}_j - r_{kj}\mathbf{q}_k \tag{50}$$

End
Perform QR iteration on $\mathbf{A}^T\mathbf{A}$ to obtain eigenvalues λ_i and eigenvectors $\mathbf{V}$

Algorithm 1 Truncated SVD via QR Iteration

Input: $\mathbf{A}$
$\mathbf{M} \leftarrow \mathbf{A}^\top\mathbf{A}$
$\mathbf{V} \leftarrow I$
while not converged **do**
  $[Q, R] \leftarrow QR(\mathbf{M})$
  $\mathbf{M} \leftarrow RQ$
  $\mathbf{V} \leftarrow \mathbf{V}Q$
end while
Extract diagonal entries λ_i from $\mathbf{M}$
$\sigma_i \leftarrow \sqrt{\lambda_i}$
$\mathbf{u}_i \leftarrow \frac{1}{\sigma_i}\mathbf{A}\mathbf{v}_i$
=0

$$\sigma_i = \sqrt{\lambda_i} \tag{51}$$

Sort singular values by using qsort (most efficient method for sorting)
For each i:

$$\mathbf{U}_i = \frac{\mathbf{A}\mathbf{V}_i}{\sigma_i} \tag{52}$$

Normalize $\mathbf{U_i}$.
Form:

$$\mathbf{A_k} = \mathbf{U_k}\,\mathbf{\Sigma_k}\,\mathbf{V_k}^T \tag{53}$$

Return $\mathbf{A_k}$.

VII. CHOICE OF ALGORITHM: SVD USING QR ITERATION

In this project, I implemented truncated Singular Value Decomposition (SVD) using an iterative QR-based eigenvalue computation. Several possible algorithms were considered, including the Jacobi method, power iteration, and the full Golub–Kahan bidiagonalization approach. After evaluating implementation effort, numerical behavior, and suitability for image compression, the QR iteration method was selected. The main reasons are summarized below.

A. *Comparison With Other Algorithms*

- **Jacobi SVD:** Although conceptually simple and highly accurate, the Jacobi algorithm converges slowly for medium–sized matrices. Its $O(n^3)$ complexity together with repeated plane rotations makes it practically inefficient for image sizes such as 256×256 or 512×512 and so on. Runtime is significantly longer. Jacobi relies on element-wise rotations, which leads to irregular memory access. Even if only a few singular values are required, this algorithm tends to operate on the entire matrix. Also precision loss occurs when scaled to a large matrix because repeated transformations may cause rounding errors.
- **Power Iteration / Lanczos:** Power iteration can compute only the largest singular value and corresponding singular vector. To extract multiple singular vectors, repeated deflation is required, which is numerically unstable and may not preserve orthogonality. Lanczos improves convergence but is significantly more complicated to implement reliably. Convergence slows dramatically when singular values are similar, making it unsuitable for similar image matrices.
- **Golub–Kahan Bidiagonalization + Implicit QR:** This method is used in industrial libraries (e.g: LAPACK) and yields excellent numerical accuracy. However, it requires Householder transformations, implicit QR steps, and careful implementation to avoid numerical breakdown. This complexity is beyond the scope of a low-level implementation without numerical libraries. This method is more complex to implement from scratch; standard libraries use it and can efficiently compute truncated SVD. It will require extra memory to store householder vectors, intermediate bidiagonal form, perform QR iterations.

B. Advantages of QR Iteration SVD

The QR iteration method offers an excellent trade–off between implementation complexity and numerical performance. Its advantages include:

1) **Conceptually Straightforward:** The algorithm requires only matrix transpose, computing $\mathbf{A}^T\mathbf{A}$, QR factorization, and iterative refinement. These operations are simple to implement using basic linear algebra concepts.
2) **Numerical Stability:** QR iteration is applied on $\mathbf{A}^T\mathbf{A}$, which is symmetric and positive semidefinite. For such matrices, QR iteration is known to be stable and to converge reliably to eigenvalues and eigenvectors.
3) **Finding V:** The eigenvectors of $\mathbf{A}^T\mathbf{A}$ give the right singular vectors $\mathbf{V}$. The left singular vectors can be recovered using:

$$\mathbf{U} = \frac{\mathbf{AV}}{\mathbf{\Sigma}}. \tag{54}$$

   There is no necessity to find the eigenvectors of $\mathbf{A}\mathbf{A}^T$ again, as it is computationally double the efforts.
4) **Support for Truncated SVD:** Since image compression requires only the top k singular components, the iteration can be terminated early after extracting $\mathbf{V}_k$ and $\mathbf{\Sigma}_k$, which improves efficiency.
5) **No External Libraries Needed:** All the necessary operations can be implemented in C from scratch.

Hence, QR iteration was selected as it provides the best balance between feasibility, robustness, and educational clarity, while still giving accurate truncated SVD results suited for image compression.

Method	Complexity	Remarks
Jacobi	$O(n^3)$	Slow convergence
Power Iteration	$O(kn^2)$	Not orthogonal for multiple vectors
QR Iteration	$O(n^3)$	Good balance, stable
Bidiagonalization + QR	$O(n^3)$	Hard to implement

C. Conclusion

We chose the QR-iteration SVD approach because it's much simpler to implement from scratch compared to bidiagonalization methods, while still being more stable than repeatedly using power iteration and generally faster than Jacobi-based techniques. It can efficiently compute truncated SVD, which is exactly what we need for image compression. The fact that it also works properly on colour images shows that the method is both flexible and reliable.

VIII. EXTENSION TO COLOUR IMAGES

Although the project initially focuses on grayscale images, the implementation naturally extends to colour images. An RGB image can be expressed as three matrices:

$$\mathbf{A}_R,\ \mathbf{A}_G,\ \mathbf{A}_B \in \mathbb{R}^{m\times n}. \tag{55}$$

The same SVD procedure is applied independently to each channel:

$$\mathbf{A}_R \approx \mathbf{U}_R\mathbf{\Sigma}_R\mathbf{V}_R^T, \qquad \mathbf{A}_G \approx \mathbf{U}_G\mathbf{\Sigma}_G\mathbf{V}_G^T, \qquad \mathbf{A}_B \approx \mathbf{U}_B\mathbf{\Sigma}_B\mathbf{V}_B^T. \tag{56}$$

The reconstructed image is obtained by combining the three components:

$$\mathbf{A} \approx (\mathbf{A}_R,\ \mathbf{A}_G,\ \mathbf{A}_B). \tag{57}$$

This method worked on colour images without needing to change the main SVD code, showing that the approach is flexible and not limited to grayscale inputs. In our experiments, the reconstructed RGB images still looked visually similar to the originals even when using a moderate rank k. This suggests that most of the important visual details in each colour channel are captured by only a small number of singular values.

IX. RECONSTRUCTED IMAGES FOR DIFFERENT K

(a) $k = 5$ (b) $k = 10$ (c) $k = 50$ (d) $k = 100$ (e) Original Image

(f) $k = 5$ (g) $k = 10$ (h) $k = 50$ (i) $k = 100$ (j) Original Image

(k) $k = 5$ (l) $k = 10$ (m) $k = 50$ (n) $k = 100$ (o) Original Image

(p) $k = 5$ (q) $k = 10$ (r) $k = 50$ (s) $k = 100$

(t) Original Image

Fig. 1: Reconstruction of four images across four values of k.

X. ERROR ANALYSIS

Frobenius Error = $\|\mathbf{A} - \mathbf{A}_k\|_F$

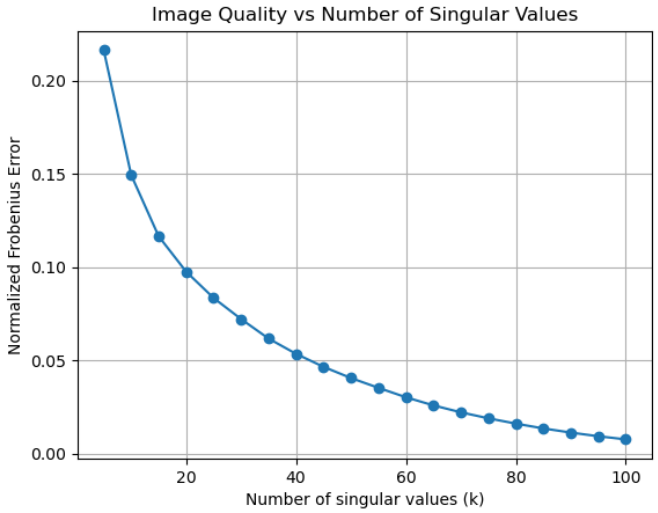
Normalised Frobenius Error = $\frac{\|\mathbf{A}-\mathbf{A}_k\|_F}{\|\mathbf{A}\|_F}$

More is the Frobenius error, then less is the image quality and vice-versa.

Because singular values are sorted in decreasing order, retaining more (k larger) captures more information of the image.

Normalised Frobenius Error closer to 1, indicates the poorest image quality and closer to 0, indicates highest image quality.

The following graph and table are generated for the image (einstein.jpg) in /figs/einstein.jpg



The table is for einstein.jpg

Singular Values	**Frob Error**	**Normalised Error**
5	8170.60168	0.216
20	3683.87018	0.097
50	1528.38576	0.040
100	286.27236	0.007

The table is for greyscale.png

Singular Values	**Frob Error**	**Normalised Error**
5	19336.15705	0.0575
20	6634.51578	0.0197
50	2094.1928	0.00624
100	1049.03611	0.0031

The table is for globe.jpg

Singular Values	**Frob Error**	**Normalised Error**
5	35860.9564	0.13078
20	18430.2799	0.067
50	10725.6972	0.039
100	6377.8504	0.02325

The table is for scenery.jpg

Singular Values	**Frob Error**	**Normalised Error**
5	29254.7311	0.283
20	21792.9555	0.2108
50	16040.9439	0.1552
100	10879.6896	0.1052

From these generated images and the graph, it can be observed that as k(number of singular values) increases, the frobenius error decreases. So image quality increases.

XI. DISCUSSION OF TRADE-OFFS AND REFLECTIONS ON IMPLEMENTATION CHOICE

The objective of this project was to implement image compression using the Singular Value Decomposition (SVD). Several factors influence the balance between computational efficiency, accuracy, storage savings, and implementation complexity.

A. Choice of SVD with QR Iteration

In this work, the QR iteration method was chosen because it is conceptually simpler and can be implemented from scratch without depending on external numerical libraries such as LAPACK. The method computes eigenvalues of $\mathbf{A}^T\mathbf{A}$, from which singular values are obtained.

Advantages:

- Conceptually easy to implement.
- Does not depend on external libraries.
- Produces valid singular values and vectors.
- Naturally accommodates truncated SVD.

Limitations:

- Computational complexity is high ($O(mn^2)$).
- Not very suitable for large matrices.
- Convergence is slower than bidiagonalization approaches.

This results in significantly higher execution time for large images (e.g: 1024×1024).

B. Truncation Parameter k: Accuracy vs. Compression

Truncated SVD reconstructs

$$\mathbf{A_k} = \mathbf{U_k}\,\mathbf{\Sigma_k}\,\mathbf{V_k}^T, \tag{58}$$

where k controls the rank of approximation. The singular values decay rapidly, meaning most of the energy is captured within the first few components.

Observations:

- Small k: High compression, but poor quality (loss of detail).
- Large k: Lower compression, but higher quality.

This introduces a trade-off between storage efficiency and accuracy.

C. Computational Trade-offs

Method	Speed	Implementation Difficulty
QR-iteration SVD	Slow	Simple
NumPy/LAPACK SVD	Fast	Requires library support
Randomized SVD	Very Fast	Algorithmically complex

Thus, although pedagogically valuable, the implementation is not computationally optimal.

D. Per-channel RGB Processing

The SVD is applied independently to each of the three color channels. This simplifies implementation but has some disadvantages.

Pros:

- Straightforward to implement.
- No color-space conversion needed.

Cons:

- Computation time is tripled (three channels - R,G,B).

E. Memory vs. Accuracy

Storing $\mathbf{U_k}$, $\mathbf{\Sigma_k}$, and $\mathbf{V_k}$ requires

$$k(m+n+1) \tag{59}$$

This value should be less than the initial memory of the image (mn).

$$k(m+n+1) < mn \tag{60}$$

$$\implies k < \frac{mn}{1+m+n} \tag{61}$$

So if the value of k is more than $\frac{mn}{1+m+n}$, then the image won't be compressed. Rather the memory of the final image will be more.

F. Reflections

The implementation demonstrates:

- SVD effectively preserves most image information in the first few singular values.
- Image quality increases monotonically with k.
- There is an inherent trade-off between compression and accuracy.

However, it also highlights that practical systems rely on optimized numerical backends (e.g., LAPACK), and more efficient strategies such as randomized SVD can dramatically improve performance.

G. Conclusion

The QR-based truncated SVD implementation offers flexible compression and accurately demonstrates theoretical behavior. However, computational cost limits scalability. In practice, optimized SVD libraries or randomized approaches are recommended for real-time, large-scale applications. This work nevertheless provided meaningful insight into the numerical structure of image data and the working of SVD-based compression.