Image Compression Using Truncated SVD
Matrix Theory (EE1030)

Kartik Lahoti
EE25BTECH11032

Abstract

This report details the implementation of an image compression algorithm using truncated Singular Value Decomposition (SVD) from scratch in C.The core algorithm leverages a hybrid approach: a Randomized SVD algorithm to reduce the dimensionality of the large image matrix, followed by a one-sided Jacobi SVD method to compute the singular values and vectors of the reduced matrix.This report presents how the algorithm is structured and provides the pseudo code to implement in C. The report also include why this algorithm is preferred over other and a caparison with implementing svd only with One-Sided-Jacobi. The report also provides error in the compressed image for different stages and provides an overview about the quality of image with compressions.

1 Summary of Gilbert Strang's video

Singular Value Decomposition is a powerful decomposition that can be applied on any matrix , irrespective of the fact that its square, fat or tall. To make this possible we have to get two set of singular vectors, $\mathbf{u}$ and $\mathbf{v}$. The $\mathbf{u}$'s are the eigen-vectors of $\mathbf{AA}^\top$ and the $\mathbf{v}$'s are the eigen-vectors of $\mathbf{A}^\top\mathbf{A}$.

Looking at the column and row space of a matrix $\mathbf{A}$ of order $m \times n$. Let this matrix have rank r. Thus both these spaces will have r linearly independent vectors.

Now we can say the A is diagonalized ,

$$\mathbf{Av_1} = \sigma_1 \mathbf{u_1} \tag{1}$$

$$\mathbf{Av_2} = \sigma_2 \mathbf{u_2} \tag{2}$$

$$\vdots \tag{3}$$

$$\mathbf{Av_r} = \sigma_r \mathbf{u_r} \tag{4}$$

This can be rewritten as

$$\mathbf{A}\begin{pmatrix}\mathbf{v_1} & \mathbf{v_2} & \dots \mathbf{v_r}\end{pmatrix} = \begin{pmatrix}\mathbf{u_1} & \mathbf{u_2} & \dots \mathbf{u_r}\end{pmatrix}\begin{pmatrix}\sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_r\end{pmatrix} \tag{5}$$

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

Since $\mathbf{V}$ and $\mathbf{U}$ are orthogonal i.e. $\mathbf{V}^\top\mathbf{V} = \mathbf{I}$ and $\mathbf{U}^\top\mathbf{U} = \mathbf{I}$, we can write

$$\mathbf{A} = \mathbf{U\Sigma V}^\top \tag{7}$$

Here, $\mathbf{U}$ and $\mathbf{V}^\top$ have left and right singular vectors respectively. The $\mathbf{\Sigma}$ has the singular values of matrix $\mathbf{A}$.

Look at Matrix $\mathbf{A}^\top\mathbf{A}$

$$\mathbf{A}^\top\mathbf{A} = \mathbf{V\Sigma}^\top\mathbf{U}^\top\mathbf{U\Sigma V}^\top \tag{8}$$

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

where $\mathbf{\Sigma}^2 = \mathbf{\Lambda}$

This matrix $\mathbf{A}^\top\mathbf{A}$ have special properties like positive semi-definite and symmetric. Again we look at

$$\mathbf{AA}^\top = \mathbf{U\Sigma V}^\top\mathbf{V\Sigma}^\top\mathbf{U}^\top \tag{10}$$

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

Since we can perform eigen value decomposition of these two matrices that are $\mathbf{A}^\top\mathbf{A}$ and $\mathbf{AA}^\top$, we can obtain the matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{\Lambda}$.

This way we can obtain all the matrices $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$.

2 Introduction

A grayscale image can be represented as a real-valued matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, where each entry (i, j) corresponds to a pixel's intensity. The Singular Value Decomposition (SVD) is a fundamental matrix factorization that decomposes $\mathbf{A}$ into three other matrices:

$$\mathbf{A} = \mathbf{U\Sigma V}^\top \tag{12}$$

where $\mathbf{U} \in \mathbb{R}^{m\times m}$ and $\mathbf{V} \in \mathbb{R}^{n\times n}$ are orthogonal matrices, and $\mathbf{\Sigma} \in \mathbb{R}^{m\times n}$ is a diagonal matrix containing the singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$.

The SVD provides the best low-rank approximation of a matrix. By keeping only the top k singular values and their corresponding vectors, we can construct a compressed approximation A_k:

$$\mathbf{A_k} = \mathbf{U_k\Sigma_k V_k}^\top \tag{13}$$

where U_k is $m \times k$, Σ_k is $k \times k$, and V_k is $n \times k$. This approximation requires storing only $(m \times k) + k + (n \times k)$ values, which is far less than $m \times n$ for small k, forming the basis of SVD-based image compression.

3 Algorithm Implementation

Computing the full SVD of a large image matrix is computationally expensive. To optimize this, we implemented a two-stage algorithm. First, we use a Randomized SVD (rSVD) technique to find a low-dimensional subspace that captures the ”action” of the matrix. Second, we apply the accurate but more intensive Jacobi SVD algorithm to this much smaller matrix.

3.1 Randomized SVD (rSVD)

Since the matries that we have to operate on are becoming of higher and higher dimension (ex. video being shot presently might be in 4k but a year ago there was HD) and applying SVD on this huge data becomes difficult. rSVD is one such method evolved to faster our calculations. Even if a matrix has a very huge dimension , it may still have a low intrinsic rank (i.e. lesser numbr of very features).

The core idea of rSVD is , we have a matrix $\mathbf{A}$, we will randomly sample the column space of this matrix (project the n-dimensional column space of $\mathbf{A}$ onto a much smaller $(k + p)$-dimensional subspace, where k is the target rank and p is a small oversampling parameter (e.g., $p = 10$, this p value is generally suggested).

3.1.1 Mathematical Formulation:

1) **Step 1:** We generate a random Gaussian matrix $\mathbf{P} \in \mathbb{R}^{n\times r}$, where $r = k + p$. We then form a ”sketch”(project) matrix $Z \in \mathbb{R}^{m\times r}$ by computing:

$$\mathbf{Z} = \mathbf{AP} \tag{14}$$

The matrix $\mathbf{Z}$ and $\mathbf{A}$ will have same dominant column space if $\mathbf{P}$ is a proper random matrix.

2) **Step 2 Orthogonalization :** We find an orthogonal basis for the column space of $\mathbf{Z}$. This is done by computing a **QR** decomposition of $\mathbf{Z}$:

$$\mathbf{Z} = \mathbf{QR} \tag{15}$$

where $\mathbf{Q} \in \mathbb{R}^{m\times r}$ has orthogonal columns.
To perform this QR decompositon we used Householder QR.

3) **Working of Householder QR :**

We carefully find householder reflections one by one , i.e.

$$\mathbf{H_i} = \mathbf{I} - \frac{2}{\mathbf{v}^\top \mathbf{v}}\mathbf{v}\mathbf{v}^t \tag{16}$$

where $\mathbf{H_i}$ is both Symmetric and Orthogonal.

$$\mathbf{Hx} = \alpha\mathbf{e_1} \tag{17}$$

where $\alpha = \pm \|\mathbf{x}\|$, and sign is chosen such that first terms doesn't cancel out. Substituing $\mathbf{H}$ and rearranging $\mathbf{v}$ we get,

$$\mathbf{v} = \mathbf{x} - \alpha\mathbf{e_1} \tag{18}$$

Now,for a Matrix $\mathbf{A}$ with n columns we start with multiplying $\mathbf{H_i}$ from $i = 0$ to n.
First Multiplication

$$\mathbf{A'} = \mathbf{H_1 A} \tag{19}$$

where , $\mathbf{A'}$ has all element below a_{11} as zero.

$$\mathbf{A''} = \mathbf{H_2 A'} \tag{20}$$

where , $\mathbf{A''}$ has all element below a_{22} as zero and so on till.

$$\mathbf{A^{n_{th}}} = \mathbf{H_n} \dots \mathbf{H_2 H_1 A} \tag{21}$$

where $\mathbf{A^{n_{th}}} = \mathbf{R}(say)$ becomes upper triangular matrix.

$$\mathbf{A} = \mathbf{H_1 H_2} \dots \mathbf{H_n R} \tag{22}$$

We achived the QR Decompostion where $\mathbf{Q} = \mathbf{H_1 H_2} \dots \mathbf{H_n}$

4) **Reason For Choosing This QR Algo:**
Householder QR is much more stable than Classical Gram-Schmidt (CGS).
It avoids catastrophic cancellation because it never directly subtracts nearly collinear vectors.
It achieves stability close to machine precision - nearly as good as performing a full orthogonalization.

5) **Projection:** We project the original matrix $\mathbf{A}$ onto this new, smaller basis $\mathbf{Q}$:

$$\mathbf{B} = \mathbf{Q}^\top \mathbf{A} \tag{23}$$

The resulting matrix $\mathbf{B} \in \mathbb{R}^{r \times n}$ is much smaller than $\mathbf{A}$ (since $r \ll m$). The key insight is that the SVD of $\mathbf{B}$ is closely related to the SVD of $\mathbf{A}$.

3.2 Pseudocode for rSVD

function RANDOMIZED_SVD($A, m, n, k, U_k, vec_k, V_k$)
    Matrix A order $m \times n$, target rank k
    $A_k = U_k \operatorname{diag}(vec_k)\ (V_k)^\top$
    p ▷ oversampling parameter
    $r \leftarrow k + p$ ▷ r is effective rank
    $P \leftarrow \text{RandomMatrix}(n, r)$ ▷ Generate $n \times r$ random matrix
    $Z \leftarrow \text{MatrixMultiply}(A, P, m, n, r)$
    $Q \leftarrow I_{m \times m}$ ▷ Identity matrix
    $Q \leftarrow \text{HouseholderQR}(Z, m, r)$
    while temp **do**
        if $i < r$ **and** $i < m$ **then**
            $len \leftarrow m - i$
            $v \leftarrow$ i-th column of Z
            $norm \leftarrow \|v\|$
            $Q \leftarrow \text{HouseholderReflector}(v, Z, Q, r, i, norm)$

$R \leftarrow \text{HouseholderReflector}(v, Z, Q, r, i, norm)$
end if
$temp \leftarrow 0$
end while
$A^\top \leftarrow \text{Transpose}(A, m, n)$
$Q^\top \leftarrow \text{Transpose}(Q, m, r)$
$B \leftarrow \text{MatrixMultiply}(Q^\top, A, r, m, n)$ ▷ Code uses $B = Q^A$
$V \leftarrow I_{n \times n}$ ▷ Identity matrix
▷ Now compute SVD of the smaller matrix B
$(U_B, \Sigma, V) \leftarrow \text{JacobiSVD}(B, r, n)$
▷ Reconstruct final U
$U \leftarrow \text{MatrixMultiply}(Q, U_B, m, r, r)$
$U_k \leftarrow U$ ▷ Gets first k columns from U
$vec_k \leftarrow \Sigma$
$V_k \leftarrow V$
return (U_k, vec_k, V_k) ▷ Full SVD components
end function

3.3 One-Sided Jacobi SVD

After reducing the problem to the smaller matrix $\mathbf{B}$, we compute its SVD using a one-sided Jacobi algorithm. This method iteratively applies Jacobi (Givens) rotations to the columns of $\mathbf{B}$ to make them orthogonal. It implicitly diagonalizes $\mathbf{B}^\top\mathbf{B}$ without ever forming it, which improves numerical stability.

3.3.1 Mathematical Formulation:

1) **Goal:** We want to find an orthogonal $\mathbf{V}$ such that the columns of $\mathbf{B}' = \mathbf{BV}$ are orthogonal. If $\mathbf{B}' = \mathbf{U\Sigma}$, then $\mathbf{B} = \mathbf{U\Sigma V}^\top$ is the SVD.
2) **Iteration:** The algorithm proceeds in "sweeps." In each sweep, we iterate over all pairs of columns (p, q) of B . We find the pairs that are not Orthogonal or close to Orthogonal and apply Jacobi Rotation to those column pairs.
3) **Jacobi Rotation:** For a pair (p, q), we want to find a rotation angle θ to make columns p and q orthogonal. We compute the inner products of the columns:

$$n_{pp} = \mathbf{B_p}^\top \mathbf{B_p} \tag{24}$$
$$n_{qq} = \mathbf{B_q}^\top \mathbf{B_q} \tag{25}$$
$$n_{pq} = \mathbf{B_p}^\top \mathbf{B_q} \tag{26}$$

If n_{pq} is not close to zero, we compute the cosine (c) and sine (s) of a rotation angle θ that will zero out the (p, q) entry of the $B^T B$ matrix. These c and s form the Jacobian Rotation Matrix.

$$\mathbf{J}(\theta) = \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \tag{27}$$

θ is found from the relation

$$\frac{1}{\tan 2\theta} = \frac{n_{qq} - n_{pp}}{2n_{pq}} \tag{28}$$

4) **Making Columns Orthogonal:** The rotation is applied to the columns p and q of $\mathbf{B}$:

$$\begin{pmatrix} \mathbf{B'_p} & \mathbf{B'_q} \end{pmatrix} = \begin{pmatrix} \mathbf{B_p} & \mathbf{B_q} \end{pmatrix} \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \tag{29}$$

We also accumulate this rotation in our $\mathbf{V}$ matrix, which is initialized to $\mathbf{I}$:

$$\begin{pmatrix} \mathbf{V'_p} & \mathbf{V'_q} \end{pmatrix} = \begin{pmatrix} \mathbf{V_p} & \mathbf{V_q} \end{pmatrix} \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \tag{30}$$

5) **Convergence:** We repeat these sweeps until all n_{pq} are below a small tolerance, meaning all columns of B are orthogonal.
6) **Final SVD:** Once converged, the singular values σ_j are the Euclidean norms of the columns of the final matrix $\mathbf{B'}$. The left singular vectors $\mathbf{U_B}$ are the normalized columns of $\mathbf{B'}$. The right singular vectors $\mathbf{V}$ are the accumulated rotations.

3.3.2 Pseudocode for Jacobi SVD:

```
function JacobiSVD(B, r, n)
    V ← IdentityMatrix(n, n)
    U ← Matrix(r, n)
    Σ ← Vector(n)
    converged ← false
    while not converged and sweeps < MAX_SWEEPS do
        converged ← true
        for p ← 0 to n − 2 do
            for q ← p + 1 to n − 1 do
                n_pp ← Dot(B_p, B_p)
                n_qq ← Dot(B_q, B_q)
                n_pq ← Dot(B_p, B_q)
                if |n_pq| > TOLERANCE then
                    converged ← false
                    c, s ← CalculateRotation(n_pp, n_qq, n_pq)
                    ApplyRotation(B, c, s, p, q)
                    ApplyRotation(V, c, s, p, q)
                end if
            end for
        end for
    end while
                                                    ▷ Extract U and Sigma
    for j ← 0 to n − 1 do
        σ_j ← Norm(B_j)
        U_j ← B_j/σ_j
    end for
```

27: SortSVD(U, Σ, V) ▷ Sorting Singular values
28: **return** U, Σ, V
29: **end function**

4 Results and Analysis

The following are the Images Generated by the program in comparision to original.

4.1 Einstein Image

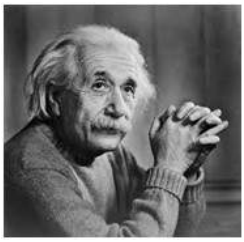

Fig. 1: Original Image - 6kB

The following figures show the reconstructed images for different values of k.

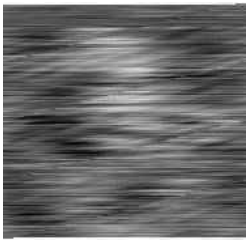

Fig. 2: $k = 5$

Fig. 3: $k = 20$

Fig. 4: $k = 50$

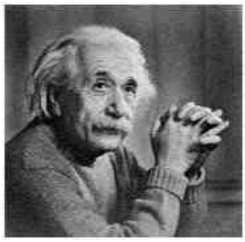

Fig. 5: $k = 100$

K Value	Size (kb)	Frobenius Norm	Time
5	5.3	8196.592526	1.614
20	5.4	5644.658531	1.905
50	5.3	3199.401578	2.684
100	4.6	1222.847285	2.724

TABLE 6: Einstein Image Comparison

4.2 Globe Image

Fig. 6: Original Image - 146.6kB

The following figures show the reconstructed images for different values of k.

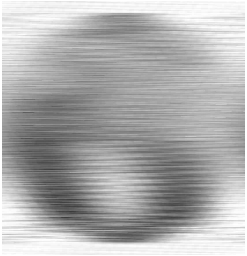

Fig. 7: $k = 5$

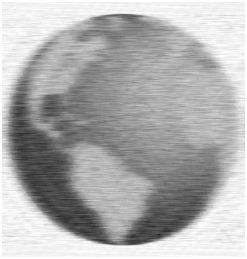

Fig. 8: $k = 20$

Fig. 9: $k = 50$

Fig. 10: $k = 100$

K Value	Size (kb)	Frobenius Norm	Time
5	71.2	33218.41908	$3min13s$
20	65.6	20200.8566	$3min24sec$
50	56.8	13645.73036	$3min44s$
100	47.3	8878.954987	$4min27s$

TABLE 6: Globe Image

4.3 Grey Scale Image

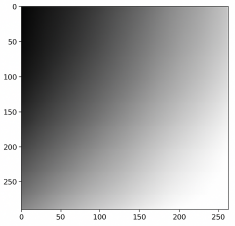

Fig. 11: Original Image - 926kB

The following figures show the reconstructed images for different values of k.

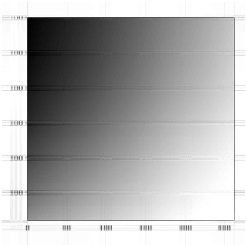

Fig. 12: $k = 5$

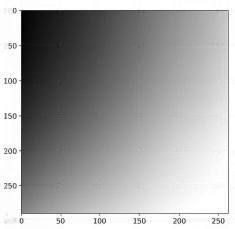

Fig. 13: $k = 20$

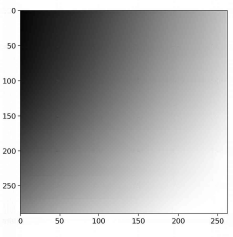

Fig. 14: $k = 50$

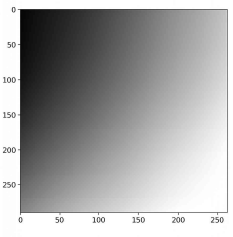

Fig. 15: $k = 100$

K Value	Size (kb)	Frobenius Norm	Time
5	32.6	12021.15923	$3min40s$
20	30.1	5115.276415	$5min20s$
50	28	1837.316469	$6m23s$
100	27.8	868.010322	$7min1.7s$

TABLE 6: Grey Scale Image

4.4 Globe Image

Fig. 16: Original Image - 18.9kB

The following figures show the reconstructed images for different values of k.

Fig. 17: $k = 5$

Fig. 18: $k = 20$

Fig. 19: $k = 50$

Fig. 20: $k = 100$

K Value	Size (kb)	Time
5	8.9	12.469
20	11.3	16.482
50	12.0	16.177
100	12.6	22.021

TABLE 6: Coloured Image

4.5 Einstein Image - Using Only One Side Jacobi Algorithm

The following figures show the reconstructed images for different values of k.

Fig. 21: k = 5

Fig. 22: k = 20

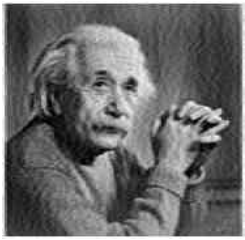

Fig. 23: k = 50

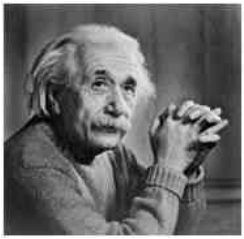

Fig. 24: $k = 100$

K Value	Size (kb)	Frobenius Norm
5	3.1	29022.37801
20	4.3	29597.99624
50	4.6	29856.26185
100	4.2	29922.05457

TABLE 6: Only Jacobi

4.6 Coloured Image To Grey Scale

Fig. 25: $k = 100$

4.7 Error Analysis

To quantify the compression quality, we compute the Frobenius norm of the error matrix, $\|A - A_k\|_F$. The Frobenius norm is the square root of the sum of the squares of all elements in the matrix.

The results for different values of k are compiled in Tables above.

As expected, the Frobenius error decreases as k increases, corresponding to the increase in visual quality. This can we seen only with the Randomized SVD , but when the image compression was perform using only One Sided Jacobi , the Frobenius value did not decrease.

5 Choice of Algorithm

For this project , the **Randomized Singular Value Decomposition (rSVD)** followed by a **one-sided Jacobi refinement** has been chosen over other algorithms since it offers an excellent trade-off between computational efficiency, numerical stability, and accuracy.

5.0.1 Big-O Summary:

$$\text{Random Projection}\mathbf{AP} : O(mn(k+p)) \tag{31}$$

$$\text{QR Decomposition} : O(m(k+p)^2) \tag{32}$$

$$\text{Small Svd } \mathbf{B} = \mathbf{Q}^\top \mathbf{A} : O((k+p)^2 n) \tag{33}$$

$$\text{Orthogonalization} : O(k^2 n) \tag{34}$$

$$\textbf{TOTAL} : O(mn(k+p)) \tag{35}$$

In comparison to using only One-Sided-Jacobi, this algorithm is better. This can be seen from the Frobenius values calculated for this algo. The values are pretty large and do not decrease with increase in k.

In comparison to the Power Iteration Algorithm , the convergence achived by rSVD is much faster.It significantly reduces computational complexity when dealing with dense large matrix.

6 Discussion and Trade-offs

The core trade-off in SVD compression is between compression size and image quality.

- **Storage:** The storage required for $\mathbf{A_k}$ is given by $(m \times k) + k + (n \times k)$ floating-point numbers. As k increases, the storage cost increases linearly.Thus as k increase quality of image becomes better since more pixel data is store.
- **Quality:** As k increases, $\mathbf{A_k}$ becomes a more accurate approximation of $\mathbf{A}$, results in better quality image and a lower Frobenius error.
- **Choosing k value :** A very small k (e.g., $k = 5$) results in a very high compression ratio but a blurry, blocky image that only captures the broadest structures.Not all the key columns that are required to display the image are not taken into consideration thus it looks blurry. A large k (e.g., $k = 100$) retains significant detail but offers less compression. The ”optimal” k depends on the application—a value that is small enough to provide meaningful compression but large enough to preserve the essential features of the image. The values $k = 20$ to $k = 50$ seem to provide a good balance for the test images.

7 Conclusion

This project successfully implemented a full C-based image compression program using truncated SVD. A modern hybrid algorithm combining Randomized SVD for dimensionality reduction and Jacobi SVD for accurate factorization was developed. The results confirm the theoretical trade-off between the approximation of rank k, storage requirements, and image quality.

8 References

1) https://youtu.be/fJ2EyvR85ro?si=o52TtthFeItDBYNu
2) https://youtu.be/VUktLhUiR7w?si=ZZJrfHYqI1G0IG61
3) https://youtu.be/6TIVIw4B5VA?si=1G8KXnmhnZLFvnnl - and the following parts.