Image Compression with the Singular Value Decomposition

Shreyas Goud Burra EE25BTECH11051

November 8, 2025

Contents

1 Introduction to SVD - Based on Gilbert Strang's Lecture

1.1 Overview

The Singular Value Decomposition (SVD) is a fundamental factorization for any real matrix (be it square or rectangular) A. It produces an expression

$$A = U\Sigma V^T,$$

where U and V are orthogonal matrices and Σ is diagonal with nonnegative entries. This factorization works for *every* matrix, unlike eigenvalue decompositions that require special structure. It is said to be the best decomposition of a matrix.

1.2 Geometric Meaning

The SVD identifies orthonormal bases in the row space and column space with a special property: each basis vector v_i in the row space is mapped by A to a scalar multiple of a corresponding basis vector u_i in the column space,

$$Av_i = \sigma_i u_i.$$

The scalars σ_i are the singular values. This means the action of A becomes diagonal when expressed in the right bases, with no coupling between components.

1.3 Relation to the Four Fundamental Subspaces

- $v_1, \ldots, v_r$ form an orthonormal basis for the row space.
- $u_1, \ldots, u_r$ form an orthonormal basis for the column space.
- The remaining $v_{r+1}, \ldots, v_n$ form an orthonormal basis for the null space of A.
- The remaining $u_{r+1}, \ldots, u_m$ form an orthonormal basis for the left null space of A.

The diagonal matrix Σ contains r positive singular values followed by zeros that correspond to the null-space dimensions.

1.4 How the SVD Is Computed

The key observation is that A^TA and AA^T are symmetric and positive semidefinite (all the matrix's eigenvalues are non-negative). Their eigenvectors produce V and U. Specifically:

$$A^TA = V\Sigma^2V^T, \qquad AA^T = U\Sigma^2U^T.$$

The positive eigenvalues of these matrices are $\sigma_1^2, \ldots, \sigma_r^2$. The singular values are the positive square roots.

Thus:

- Columns of V are eigenvectors of A^TA.
- Columns of U are eigenvectors of AA^T.
- Singular values are square roots of the eigenvalues.

1.5 Key Insight

The SVD chooses the "right" bases for all four fundamental subspaces. Once those bases are chosen, the matrix becomes diagonal with nonnegative entries. This connects orthogonality, eigenvalues, and the geometry of linear transformations in a unified framework.

2 The Jacobi Algorithm

The Jacobi method is an iterative algorithm that uses a sequence of planar rotations (Givens rotations) to systematically diagonalize a matrix. While it's famously used for finding eigenvalues of a symmetric matrix (the Jacobi eigenvalue algorithm), a variation called **one-sided Jacobi** is highly effective for computing the SVD.

The core idea of the one-sided Jacobi SVD is not to diagonalize **A** itself, but to apply rotations to the columns of **A** until they are all mutually orthogonal (upto a given tolerance).

Givens rotation is a rotation spanned over a plane of 2 coordinate axes within a given subspace.

2.1 The Goal

Let's start with our matrix **A**. We want to find an $n \times n$ orthogonal matrix **V** such that the matrix $A' = AV$ has orthogonal columns.

If the columns of $A' = [a'_1, a'_2, \ldots, a'_n]$ are orthogonal, then:

$$a'_p \cdot a'_q = 0 \quad \text{for all } p \neq q$$

Once we have this **A'**, we can easily find **U** and Σ.

1. The singular values σ_j are simply the lengths (2-norms) of the columns of **A'**:

$$\sigma_j = ||a'_j||_2$$

2. The left singular vectors u_j are the normalized columns of A':

$$u_j = \frac{a'_j}{\sigma_j}$$

If we assemble these into matrices, we have $U = [u_1, \ldots, u_n]$ and $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_n)$. This gives us:

$$A' = U\Sigma \Rightarrow AV = U\Sigma \Rightarrow A = U\Sigma V^T$$

So, the entire problem boils down to finding the orthogonal matrix **V** that makes the columns of **AV** orthogonal.

2.2 Why One-Sided Jacobi's Algorithm?

One sided Jacobi's algorithm is characterized by taking the Σ as a square matrix. Contrary to regular Singular Value Decomposition where Σ has the same dimension as **A**. This is done as the additional rows/columns in Σ are redundant and we would have to find the whole eigenvalues for **U** (this would take greater computation). To reduce the computation and get rid of redundancy, one-sided Jacobi's is highly efficient.

3 The Jacobi Iteration

We find $\mathbf{V}$ as a product of many simple Givens rotations, $V = J_1 J_2 J_3 \ldots$. Each rotation J_k is chosen to force just one pair of columns (p, q) to become orthogonal (applying a Givens rotation).

A Givens rotation $J(p, q, \theta)$ is an identity matrix except for four entries, which define a 2×2 rotation in the (p, q) plane:

$$J(p,q,\theta) = \begin{bmatrix} 1 & \ldots & 0 & \ldots & 0 & \ldots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \ldots & c & \ldots & -s & \ldots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \ldots & s & \ldots & c & \ldots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \ldots & 0 & \ldots & 0 & \ldots & 1 \end{bmatrix} \begin{matrix} \\ \\ \leftarrow p \\ \\ \leftarrow q \\ \\ \\ \end{matrix}$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$.

When we post-multiply $\mathbf{A}$ by $\mathbf{J}$, $A' = AJ(p, q, \theta)$, it only affects columns p and q:

$$[a'_p, a'_q] = [a_p, a_q] \begin{pmatrix} c & -s \\ s & c \end{pmatrix} = [ca_p + sa_q, -sa_p + ca_q]$$

Our goal is to choose θ (or rather, c and s) such that the new columns a'_p and a'_q are orthogonal:

$$a'_p \cdot a'_q = 0$$

Substituting the expressions for a'_p and a'_q leads to a 2×2 symmetric eigenvalue problem. Let's consider the dot products of the original columns, which we can compute:

- $\alpha = a_p \cdot a_p = ||a_p||_2^2$
- $\beta = a_q \cdot a_q = ||a_q||_2^2$
- $\gamma = a_p \cdot a_q$

The new dot product $a'_p \cdot a'_q$ will be zero if we choose c and s to diagonalize the 2×2 symmetric sub-problem:

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} \alpha & \gamma \\ \gamma & \beta \end{pmatrix} \begin{pmatrix} c & -s \\ s & c \end{pmatrix} = \begin{pmatrix} \alpha' & 0 \\ 0 & \beta' \end{pmatrix}$$

The rotation parameters c and s that achieve this can be computed robustly. We define a value τ:

$$\tau = \frac{\beta - \alpha}{2\gamma}$$

Then, we find $t = \tan(\theta)$:

$$t = \frac{\text{sign}(\tau)}{|\tau| + \sqrt{1 + \tau^2}} \quad (\text{if } \gamma \neq 0)$$

And from t, we get c and s:

$$c = \frac{1}{\sqrt{1 + t^2}} \quad \text{and} \quad s = c \cdot t$$

If $\gamma = 0$ the columns are already orthogonal, so we can set $c = 1, s = 0$.

3.1 The Algorithm

The full algorithm consists of iteratively "sweeping" through all possible pairs of columns (p, q) and applying the rotation to orthogonalize them. While rotating a pair (p, q) to be orthogonal, it may "un-orthogonalize" them with respect to other columns. However, the Jacobi method is guaranteed to converge because each step reduces the total sum of squares of the off-diagonal dot products.

4 Code for One-Sided Jacobi SVD

```
void jacobi(double **A, int max_sweeps, double toler, int m, int n, double
    **V, double **S, double **U){
    for(int sweep = 0; sweep < max_sweeps; sweep++) {
        int converged = 1;
        for(int p = 0; p <= n - 2; p++) {
            for(int q = p + 1; q < n; q++) {

                double alpha = 0.0, beta = 0.0, gamma = 0.0;
                for(int i = 0; i < m; i++) {
                    double ap = A[i][p];
                    double aq = A[i][q];
                    alpha += ap * ap;
                    beta  += aq * aq;
                    gamma += ap * aq;
                }

                if (fabs(gamma) <= toler * sqrt(alpha * beta + 1e-30))
                    continue;

                converged = 0;
                double tau = (beta - alpha) / (2.0 * gamma);
                double t   = signum(tau) / (absol(tau) + sqrt(1.0 + tau *
                   tau));
                double c   = 1.0 / sqrt(1.0 + t * t);
                double s   = c * t;

                for(int i = 0; i < m; i++) {
                    double aip = A[i][p];
                    double aiq = A[i][q];
                    double newp = c * aip - s * aiq;
                    double newq = s * aip + c * aiq;
                    A[i][p] = newp;
                    A[i][q] = newq;
                }

                for(int i = 0; i < n; i++) {
                    double vip = V[i][p];
                    double viq = V[i][q];
                    double newp = c * vip - s * viq;
                    double newq = s * vip + c * viq;
                    V[i][p] = newp;
                    V[i][q] = newq;
                }
            }
```

```
43          }
44          if(converged)
45              break;
46      }
47
48      for(int i = 0; i < n; i++)
49          for(int j = 0; j < n; j++)
50              S[i][j] = 0.0;
51      for(int i = 0; i < m; i++)
52          for(int j = 0; j < n; j++)
53              U[i][j] = 0.0;
54
55      for(int j = 0; j < n; j++) {
56          double sigma_sq = 0.0;
57          for(int i = 0; i < m; i++)
58              sigma_sq += A[i][j] * A[i][j];
59          double sigma = sqrt(sigma_sq);
60          S[j][j] = sigma;
61          if(sigma > toler) {
62              for(int i = 0; i < m; i++)
63                  U[i][j] = A[i][j] / sigma;
64          }
65      }
66  }
```

5 Analytical Comparison of Jacobi's Algorithm with other methods of Singular Value Decomposition

5.1 Jacobi's Algorithm

5.1.1 The Merits

1. Jacobi's algorithm has excellent numerical stability and can find all the eigenvalues up to a certain tolerance with high accuracy and less runtime.

2. Jacobi's algorithm is really simple in application. With the use of simple mathematical methods of Givens rotation and properties of symmetrical matrices, it can be easily applied.

3. It can be effectively modified to run different parts of an iteration simultaneously and independently. This can heavily reduce the runtime of the program.

5.1.2 The Demerits

1. It can be slow for smaller images as compared to other algorithms, but with larger images and larger chunks of data, in general, it outperforms the other algorithms.

2. It works on the full matrix in its raw form. This can be more time consuming as compared to methods that optimize the initial matrix before performing operations on it.

5.2 Golub-Kahn-Reinsch Algorithm

The Golub-Kahn-Reisch algorithm is a more standard algorithm used in a broader range of applications (in MATLAB and LAPACK library). It is capable of handling denser matrices with predictability and efficiency at the cost of its accuracy.
The following are the points of comparison between the aforementioned algorithms.

1. As compared to Jacobi's algorithm. It is algorithmically more complex making its implementation from scratch far more difficult.

2. The Golub-Kahn-Reisch algorithm creates data dependencies as it executes, that is, it cannot run in parallel like the Jacobi's algorithm.

3. Though the Golub-Kahn-Reisch algorithm is more efficient for large scale scientific computing, the Jacobi's algorithm is far more efficient in the case of Image compression.

5.3 Power Iteration Method

Power iteration is a simple iterative method used to find the largest singular value or eigenvalue of a matrix. It repeatedly multiplies a vector by the matrix (or A^TA for SVD) and normalizes the result. Over iterations, the vector aligns with the dominant singular vector, assuming that value is clearly larger than the rest (the algorithm has slow convergence in the case that the larger two values are close). This is then deflated (the contribution of this singular value is ignored for the matrix) and then power iterated again to get the next singular value and so on,

Given below are the points of comparison between the aforementioned algorithms.

1. It is simpler to implement as compared to Jacobi's requiring only a few lines of linear algebraic operations.

2. Unlike Jacobi's algorithm which optimizes the singular values at every step, Power Iteration finds the largest singular value (and the associated left and right eigenvectors) and works its way down from there.

3. It is less numerically stable as compared to Jacobi's as the rounding errors at each step can add up. Jacobi's takes care of this at every step.

4. If the singular values do not differ by much, power iteration can struggle as it would require a larger number of iterations to be able to find them.

6 Error Analysis by Frobenius Norm

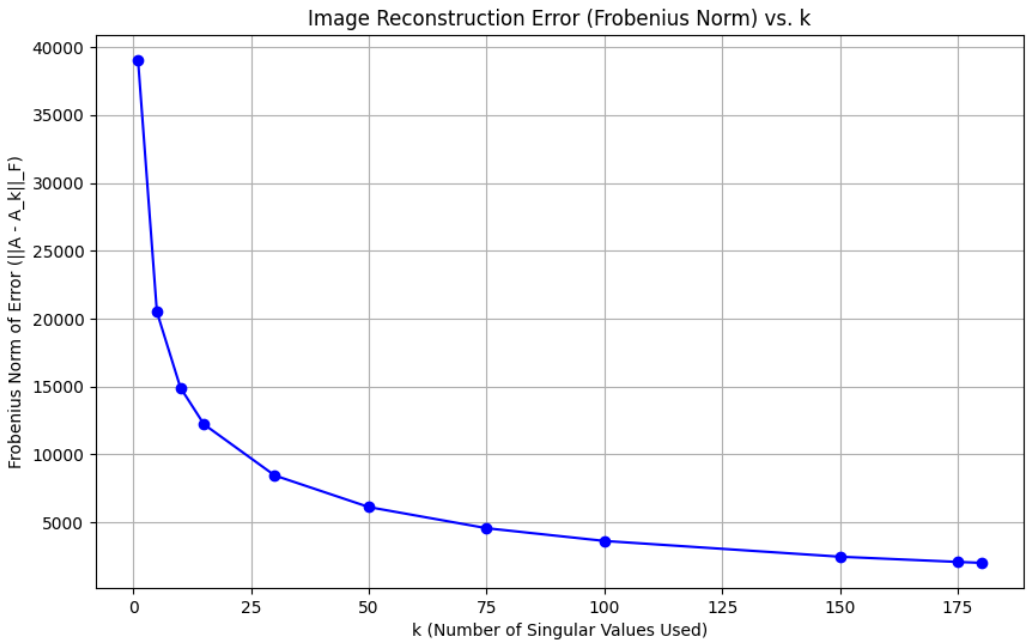
Frobenius norm is defined by

$$\|A - A_k\|_F$$

Where **A** is the initial matrix of luminosity values of the pixels of the given image. And $\mathbf{A_k}$ is the reconstructed matrix of luminosity values after the application of SVD.

For the Case of Globe.jpg

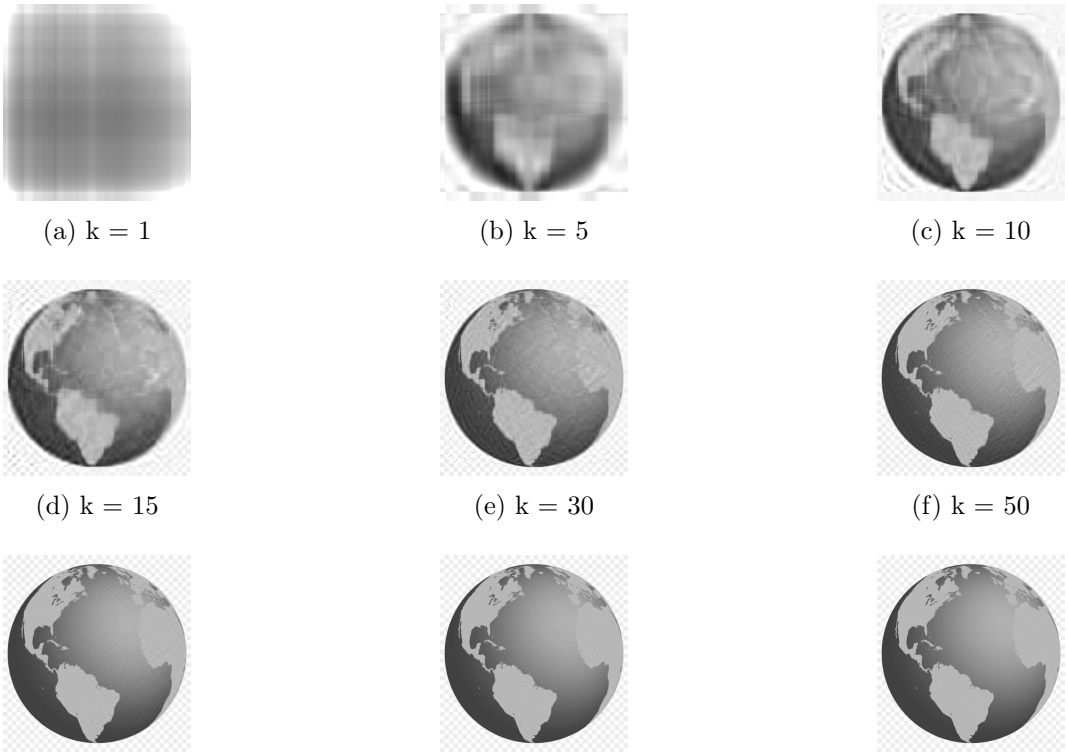
The graph obtained is shown as below :



k Value	Frobenius Norm
1	39045.483388
5	20501.518212
10	14919.064448
15	12218.031020
30	8456.577677
50	6124.659909
75	4567.945162
100	3630.536462
150	2469.341613
175	2085.406675
180	2018.163274

Table 1: Frobenius Norm of Reconstruction Error vs. k

Here also there a significant decrease in error from k = 1 to k = 150.
Now below are the Compressed images for the globe.jpg.



(a) k = 1 (b) k = 5 (c) k = 10

(d) k = 15 (e) k = 30 (f) k = 50

(g) k = 75 (h) k = 100 (i) k = 150

Figure 1: Compressed images for globe.jpg with varying k values.

7 Trade Off

In SVD image compression, an image A is approximated by a rank-k matrix

$$A_k = U_k \Sigma_k V_k^T.$$

The parameter k determines both compression size and reconstruction quality.

- **Compression Size:** A rank-k approximation requires storing $k(m+n+1)$ values, so smaller k gives stronger compression.
- **Image Quality:** Reconstruction error

$$\|A - A_k\|_F^2 = \sum_{i=k+1}^{r} \sigma_i^2$$

  decreases as k increases. Small k leads to blurred images, while large k produces near-original quality.

- **Tradeoff:** Low k offers compact storage but poor fidelity. Moderate k provides a balanced result. High k yields excellent quality with limited compression benefits.

Most practical images reach good quality for k between 50 and 100 for the given example.

8 Conclusion

This project successfully demonstrated the theory and practical application of Singular Value Decomposition (SVD) for image compression. By implementing the one-sided Jacobi algorithm, we were effectively able to decompose an image matrix.

The core of the experiment was to reconstruct the image using a truncated rank-k approximation, A_k. Our error analysis, based on the Frobenius norm, provided a quantitative measure of this approximation. The results for 'globe.jpg' clearly show that the reconstruction error $||A - A_k||_F$ decreases monotonically as k increases. More importantly, the most significant drops in error occur for relatively small values of k (e.g., from $k = 1$ to $k = 100$), after which the returns diminish.

This numerical finding was mirrored by the visual evidence. The grid of compressed images visually confirms that a vast majority of the image's structural information and "energy" is captured by the first 100-150 singular values. While the image at $k = 10$ is still abstract, the image at $k = 75$ or $k = 100$ is clearly recognizable and of high quality, despite using only a fraction of the original data.

Ultimately, this project validates SVD as a powerful technique in data analysis and compression. It confirms the central tradeoff: a small k provides excellent compression at the cost of fidelity, while a larger k provides high fidelity at the cost of compression. The optimal choice of k is not fixed but is a practical decision that depends on the specific requirements for storage space versus visual quality.