Image Compression using SVD

Shriyansh Chawda - EE25BTECH11052

November 8, 2025

Contents

1 Summary of Gilbert Strang's Lecture on Singular Value Decomposition (SVD)

1.1 Introduction

Gilbert Strang's lecture introduces the Singular Value Decomposition (SVD) as the "final and best factorization of a matrix." SVD is a fundamental matrix factorization technique applicable to any matrix A (square or rectangular, invertible or singular), decomposing it into three matrices: an orthogonal matrix U, a diagonal matrix Σ containing singular values, and another orthogonal matrix V^T. The decomposition is expressed as:

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

1.2 Computation of SVD Components

Strang demonstrates a systematic approach to compute U, Σ, and V, leveraging the properties of symmetric positive semi-definite matrices.

1.2.1 Determining V and Singular Values Σ

The matrix V and singular values σ_i are derived from A^TA:

$$A^TA = (U\Sigma V^T)^T(U\Sigma V^T) = V\Sigma^TU^TU\Sigma V^T = V\Sigma^2V^T$$

Thus:

- The columns of V are the orthonormal eigenvectors of A^TA.
- The diagonal entries of Σ^2 are the eigenvalues of A^TA, hence $\sigma_i = \sqrt{\lambda_i(A^TA)}$.

1.2.2 Determining U

Similarly,

$$AA^T = (U\Sigma V^T)(U\Sigma V^T)^T = U\Sigma\Sigma^TU^T = U\Sigma^2U^T$$

Hence:

- The columns of U are the orthonormal eigenvectors of AA^T.
- The eigenvalues of AA^T match those of A^TA (σ_i^2).

Alternatively, once V and Σ are known, the columns of U can be computed as:

$$u_i = \frac{1}{\sigma_i}Av_i$$

For zero singular values, additional u_i vectors complete an orthonormal basis for the left null space.

1.3 Significance and Application to Image Compression using Truncated SVD

The SVD is a cornerstone of linear algebra due to its universality and depth:

- **Universal Applicability:** Works for any matrix, regardless of shape or rank.
- **Four Fundamental Subspaces:** SVD provides orthonormal bases for row, null, column, and left null spaces.
- **Rank and Condition Number:** The number of nonzero singular values gives the rank, and their ratio gives the condition number.

In applications like image compression, a low-rank approximation of A is formed by keeping only the k largest singular values:

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

This truncated SVD efficiently captures dominant information using fewer parameters—the mathematical essence of image compression.

2 Code Analysis 1: The Power Iteration

This function is designed to find the **dominant** (largest) eigenvalue and its corresponding eigenvector for a symmetric matrix B.

2.1 The Mathematical Logic

you have a matrix B and a random vector v. If you keep multiplying v by B over and over $(Bv, B(Bv), B(B(Bv)), \dots)$, the vector will gradually start to point in the same direction as the eigenvector associated with the largest eigenvalue (λ_1). This is because any random vector can be thought of as a mix of all the eigenvectors. Each time you multiply by B, the part of the vector corresponding to the largest eigenvalue gets stretched **more** than all the other parts. After many iterations, this "largest" part completely dominates, and the vector becomes almost perfectly aligned with that dominant eigenvector.

3 Code Analysis 2: The Deflation Function

After we find the dominant eigenpair (λ_1, v_1), we need to "remove" it from B so we can find the *next* one, λ_2. This process is called **deflation**.

3.1 The Mathematical Logic

The code uses **Hotelling's Deflation**. The formula is very simple. To get the new matrix B_{new}, we do:

$$B_{new} = B_{old} - \lambda_1 v_1 v_1^T$$

Here, $v_1 v_1^T$ is an "outer product," which creates a matrix. Let's see why this works:

- **What happens to v_1?** Let's multiply our new matrix by the eigenvector v_1:

$$B_{new}v_1 = (B_{old} - \lambda_1 v_1 v_1^T)v_1$$

$$B_{new}v_1 = B_{old}v_1 - \lambda_1 v_1(v_1^T v_1)$$

  Since v_1 is a normalized eigenvector, $B_{old}v_1 = \lambda_1 v_1$ and its length is 1, so $(v_1^T v_1) = 1$.

$$B_{new}v_1 = \lambda_1 v_1 - \lambda_1 v_1(1) = 0$$

  This means v_1 is still an eigenvector of B_{new}, but its eigenvalue is now **0**.

4 Code Breakdown: `SVDResult` and `compute_svd`

The main SVD function acts as the director of the entire operation, bringing together all the helper functions to compute the final result. Here is a step-by-step breakdown of its logic:

1. **Step 1: Initial Setup.** The process begins by calculating a special symmetric matrix, B, from the original matrix A (specifically, B = $A\top A$). Since this matrix will be progressively modified, a temporary copy of it is created.

2. **Step 2: Iteratively Find Eigenpairs.** The algorithm enters a main loop that runs k times to find the top k singular values and right singular vectors. In each cycle of the loop, it performs three actions:

   - First, it calls the **Power Iteration** function on the temporary matrix B to find its single largest eigenvalue and the corresponding eigenvector.
   - Next, it stores this result. The found eigenvector is a right singular vector, so it is saved as a new column in the final V matrix. The square root of the eigenvalue is the corresponding singular value, which is saved in the S array.
   - Finally, it calls the **Deflation** function. This process mathematically ”removes” the eigenpair just found from the temporary matrix B.

3. **Step 3: Calculate the Left Singular Vectors.** After the loop finishes, the S array and the V matrix are complete. The last major task is to calculate the U matrix. This is also done iteratively in a loop, one column at a time. For each column of U, the algorithm uses the formula: $u_i = (A \cdot v_i)/\sigma_i$.

4. **Step 4: Cleanup and Return.** With U, S, and V all fully computed, the function frees all the temporary memory it used during the calculations.

```
    svdresult SVD_compute(matrix a, int k)
    {
      int m = a.row;
      int n = a.col;
      matrix b = At_multiply_A(a);
      matrix b_temp = copyMatrix(b);
```

```c
    svdresult svd;
    svd.k = k;
    svd.s = (double *)malloc(k * sizeof(double));
    svd.v = creatematrix(n, k);
    svd.u = creatematrix(m, k);
    double *v = (double *)malloc(n * sizeof(double));
    double *u_col = (double *)malloc(m * sizeof(double));
    printf("Starting Power Iteration loop for k=%d...\n", k);
    for (int i = 0; i < k; i++)
    {
      double l = poweriteration(b_temp, v);
      svd.s[i] = sqrt(l);

      for (int j = 0; j < n; j++)
      {
        svd.v.data[j][i] = v[j];
      }

      deflatematrix(b_temp, l, v);
    }

    for (int i = 0; i < k; i++)
    {
      for (int j = 0; j < n; j++)
      {
        v[j] = svd.v.data[j][i];
      }
      matrix_vector_multiplication(a, v, u_col);
      double sigma = svd.s[i];
      if (sigma > 1e-9)
      {
        for (int j = 0; j < m; j++)
        {
          svd.u.data[j][i] = u_col[j] / sigma;
        }
      }
      else
      {
        for (int j = 0; j < m; j++)
        {
          svd.u.data[j][i] = 0.0;
        }
      }
    }
    freematrix(b);
    freematrix(b_temp);
    free(v);
    free(u_col);
    printf("SVD computation complete.\n");
    return svd;
  }

```

Listing 1: Main SVD Function

5 A Comparative Analysis of Singular Value Decomposition Methods for Image Compression

5.1 The Power Iteration Method

5.1.1 Advantages of Power Iteration

- **Simplicity of Implementation:** The Power Iteration method is relatively straightforward to implement, as demonstrated in our project.
- **Efficiency for Truncated SVD:** For applications like image compression where only the top few singular values are needed, the Power Iteration method is highly efficient. It avoids the computational expense of calculating the full SVD.
- **Scalability:** It can be more scalable than direct methods for very large and sparse matrices, as it primarily involves matrix-vector multiplications.

5.1.2 Disadvantages of Power Iteration

- **Convergence Rate:** The rate of convergence depends on the ratio of the largest to the second-largest eigenvalue. If these are close, convergence can be slow.
- **Numerical Stability:** The method can be susceptible to numerical instability, especially without careful implementation of normalization and deflation steps.

5.2 Alternative SVD Computation Methods

5.2.1 Golub-Kahan-Reinsch Algorithm

This is a widely used and robust algorithm for computing the SVD of a dense matrix. **Comparison with Power Iteration:**

- **Full vs. Truncated SVD:** The Golub-Kahan-Reinsch algorithm is designed to compute the full SVD, making it less efficient if only a few singular values are needed.
- **Computational Complexity:** The complexity for a dense $m \times n$ matrix is typically $O(mn^2)$ for Golub-Kahan-Reinsch. For our Power Iteration method to find the top k singular values, the complexity is roughly $O(k \cdot \text{iterations} \cdot mn)$, which can be significantly lower for small k.
- **Numerical Stability:** The Golub-Kahan-Reinsch algorithm is known for its excellent numerical stability. While our Power Iteration implementation is effective, it is generally considered less robust than this industry-standard method.

5.2.2 Jacobi SVD Algorithm

Comparison with Power Iteration:

- **Accuracy:** The Jacobi method is often praised for its high accuracy, particularly in computing small singular values.
- **Parallelism:** The structure of the Jacobi algorithm lends itself well to parallel computation, which can be an advantage on modern multi-core processors.
- **Convergence:** The convergence of the Jacobi method can be slower than the Golub-Kahan-Reinsch algorithm, especially for matrices with no specific structure.
- **Suitability for Truncation:** Similar to Golub-Kahan-Reinsch, the standard Jacobi method computes the full SVD. While modifications for partial SVD exist, the Power Iteration method is more naturally suited for finding only the top singular values.

5.2.3 Randomized SVD

Comparison with Power Iteration:

- **Efficiency for Large Data:** For very large matrices where even matrix-vector products are expensive, randomized SVD can be significantly faster than Power Iteration.
- **Approximation Quality:** The quality of the approximation in randomized SVD depends on the choice of the random projection. While often very good, it is a probabilistic method. The Power Iteration method, being deterministic, will consistently converge to the same result.
- **Implementation Complexity:** Randomized SVD algorithms can be more complex to implement correctly compared to the basic Power Iteration.

6 Error Analysis: The Frobenius Norm

After reconstructing the image with a rank-k approximation, A_k, a crucial step is to quantify how much information was lost. The project requires using the Frobenius norm to measure this approximation error, defined as $\|A - A_k\|_F$.

6.1 The Mathematical Logic

The Frobenius norm is a matrix norm analogous to the Euclidean vector norm. For an $m \times n$ matrix M, it is defined as the square root of the sum of the absolute squares of its elements:

$$\|M\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |m_{ij}|^2}$$

A fundamental and highly useful property of the Frobenius norm is its direct relationship with the singular values (σ_i) of the matrix. The squared Frobenius norm is equal to the sum of the squares of all its singular values:

$$\|M\|_F^2 = \sum_{i=1}^{r} \sigma_i^2, \quad \text{where } r = \text{rank}(M)$$

This property provides a much more efficient way to calculate the approximation error.

6.2 Efficient Computation using Singular Values

A elegant and efficient method leverages the singular values we have already computed. The original matrix A can be expressed as a sum of r rank-1 matrices (where r is the rank of A):

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^T$$

Our rank-k approximation A_k is simply the sum of the first k terms:

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$$

The error matrix $E = A - A_k$ is therefore the sum of the terms we discarded:

$$A - A_k = \sum_{i=k+1}^{r} \sigma_i u_i v_i^T$$

The singular values of the error matrix $A - A_k$ are precisely the singular values we discarded: $\sigma_{k+1}, \sigma_{k+2}, \ldots, \sigma_r$. Using the property from the previous equation, the squared Frobenius norm of the error is the sum of the squares of its singular values:

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

This is the key insight: The approximation error can be calculated directly from the discarded singular values, without needing to reconstruct the image or reference the original matrix at all.

7 The Trade-off: Compression vs. Image Quality

7.1 Defining the Components

Image Quality: This can be measured objectively and subjectively.

- **Objective Quality:** A low approximation error (Frobenius norm) corresponds to high objective quality. As $k \to r$, the error $\|A - A_k\|_F \to 0$.

- **Subjective Quality:** This is how visually pleasing the reconstructed image is to a human observer.

Compression: This is the reduction in data needed to store the image.

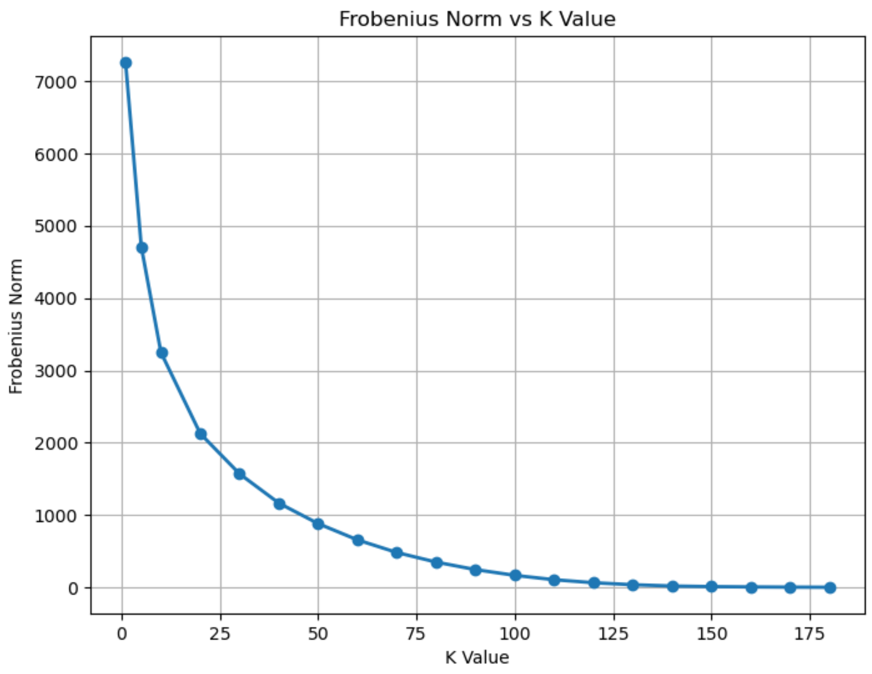
- Original storage: An $m \times n$ image requires storing $m \times n$ pixel values.
- Compressed storage: The rank-k approximation requires storing k singular values, k columns of U (each size m), and k columns of V (each size n). The total storage is $k(1 + m + n)$ values.
- Compression ratio:
$$\text{Compression Ratio} = \frac{k(1 + m + n)}{mn}$$
A lower ratio means better compression.

7.2 Analyzing the Relationship

- **When k is very small:**
  - Compression: Storage savings are enormous.
  - Quality: The error is very high, and the reconstructed image is blurry and blocky.
- **When k is moderate:**
  - Compression: Significant savings are still achieved.
  - Quality: The error is moderate; important features are preserved but textures are softened.
- **When k is large (close to r):**
  - Compression: Little or no savings.
  - Quality: The error is very low, and the image closely matches the original.

7.3 Results from Standard images

For the case of Einstein.jpg
When we calculated the Error using Frobenius Norm , the following graph was optained :



This Shows us that the error significantly reduces from K =1 to K =100 while there not much significant loss from k =100 to k =175 . The following are the images generated for Einstein.jpg

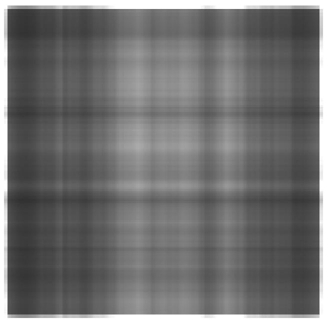


Figure 1: k = 1

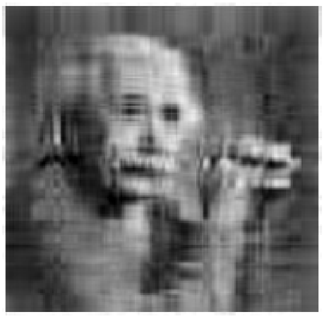


Figure 2: k = 10

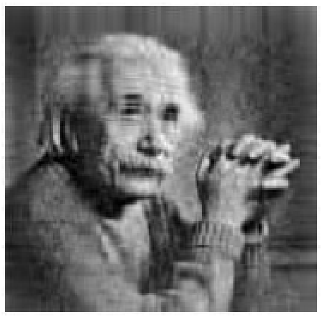


Figure 3: k = 20

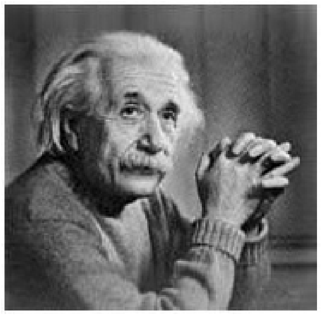


Figure 4: k = 50

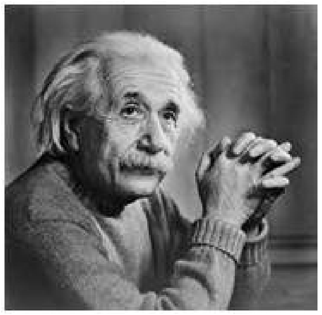


Figure 6: k = 130

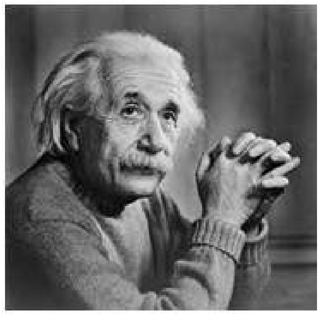


Figure 8: k = 170

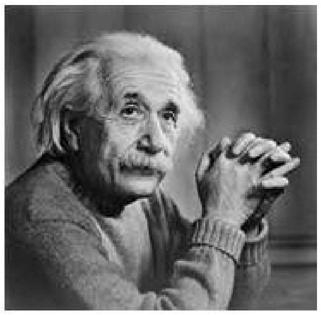


Figure 5: k = 100

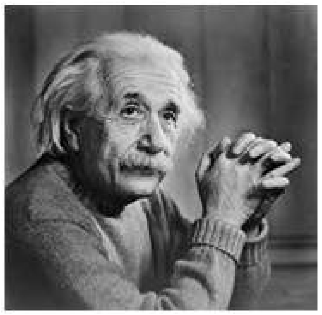


Figure 7: k = 160

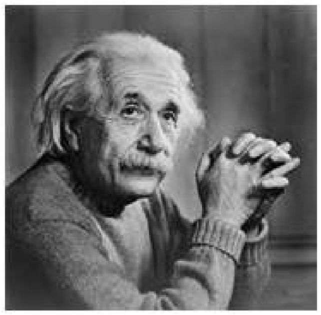


Figure 9: k = 180

Table 1: K vs Frobenius Norm for the Einstein Image

K	**Frobenius Norm**	**K**	**Frobenius Norm**	**K**	**Frobenius Norm**
1	7264.54	50	880.35	100	164.78
5	4713.34	60	655.04	110	104.38
10	3248.82	70	480.54	120	63.11
20	2126.44	80	347.96	130	35.14
30	1568.72	90	244.33	140	17.48
40	1163.65	150	9.62	160	5.26
170	2.21	180	0.26		

Similarily

For the Case of Globe.jpg

The graph obtained is shown as below :

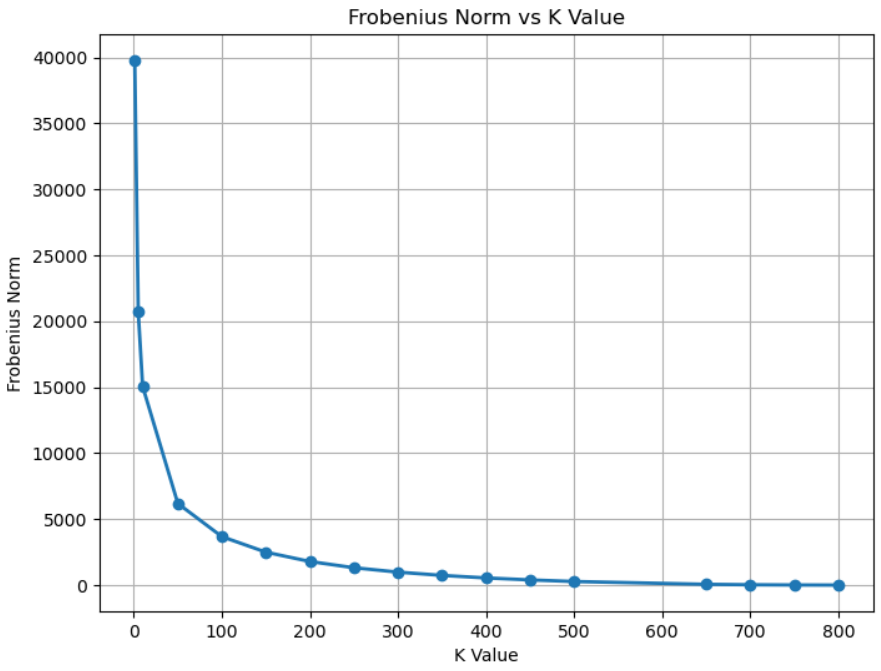


Table 2: K vs Frobenius Norm for the Globe Image

K	Frobenius Norm	K	Frobenius Norm	K	Frobenius Norm
1	39782.88	100	3672.67	400	542.13
5	20703.89	150	2494.99	450	391.98
10	15060.47	200	1787.55	500	271.91
50	6185.21	250	1315.17	650	58.42
300	982.41	350	734.16	700	26.37
750	8.73	800	0.58		

Here also there a significant decrease in error from k = 1 to k = 100.
Now below are the Compressed images for the globe.jpg.

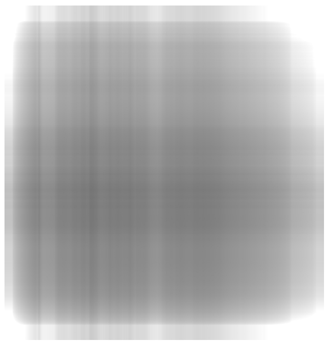


Figure 10: k = 1

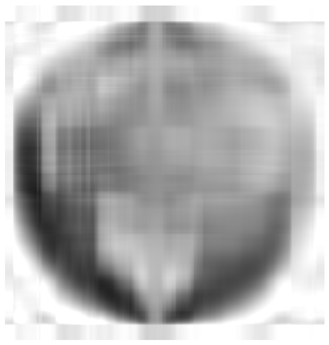


Figure 11: k = 5

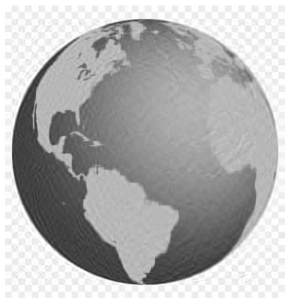


Figure 12: k = 50

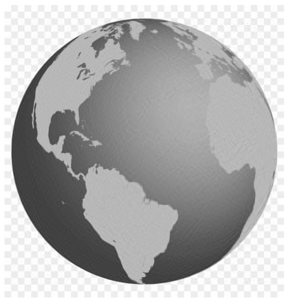


Figure 13: k = 100

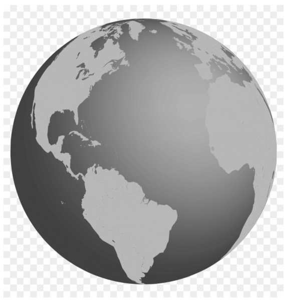


Figure 15: k = 300

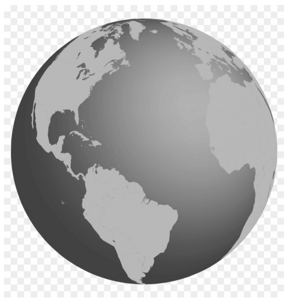


Figure 17: k = 500

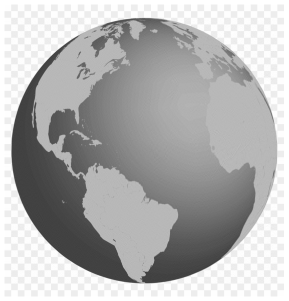


Figure 14: k = 200

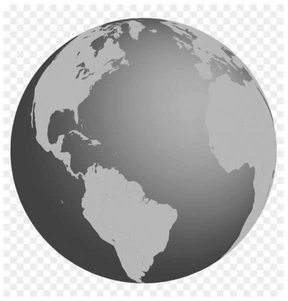


Figure 16: k = 400

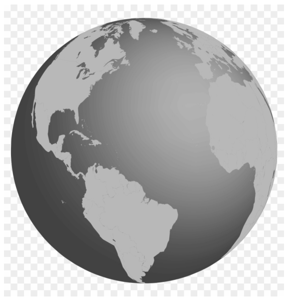


Figure 18: k = 800

8 Reflections on Implementation Choices

The primary reason for this choice is **simplicity and conceptual clarity**.This combination was chosen for several key reasons:

- **Focus on the Dominant Components:** Power Iteration is the simplest and most direct algorithm for finding the single largest (dominant) eigenvalue and its corresponding eigenvector.
- **Enabling Iteration:** Hotelling's Deflation provides a conceptually simple way to "remove" a found eigenpair from the matrix, allowing the Power Iteration method to be re-applied to find the *next* largest eigenpair. .
- **Efficiency for Truncation:** The core goal was to implement a **Truncated SVD**, which is what most practical applications like Principal Component Analysis (PCA), dimensionality reduction, and recommendation systems require. This iterative method is highly efficient for this purpose, as it avoids the significant computational cost of calculating the full SVD when only the top k components are needed.

In summary, this implementation prioritizes a clear, step-by-step logical flow over the raw performance and numerical stability of production-level numerical libraries.

9 Coloured Image Compression using SVD(Power iteration algorith)

I implemented the same algorithm individually to The R , G and B channel and then combined them to get a compressed coloured image. This is done using pyhton plus C implementation.
The entire C backend is the same as grayscale ; while the image handling is done using python libraries and python code.
Testcase Image:

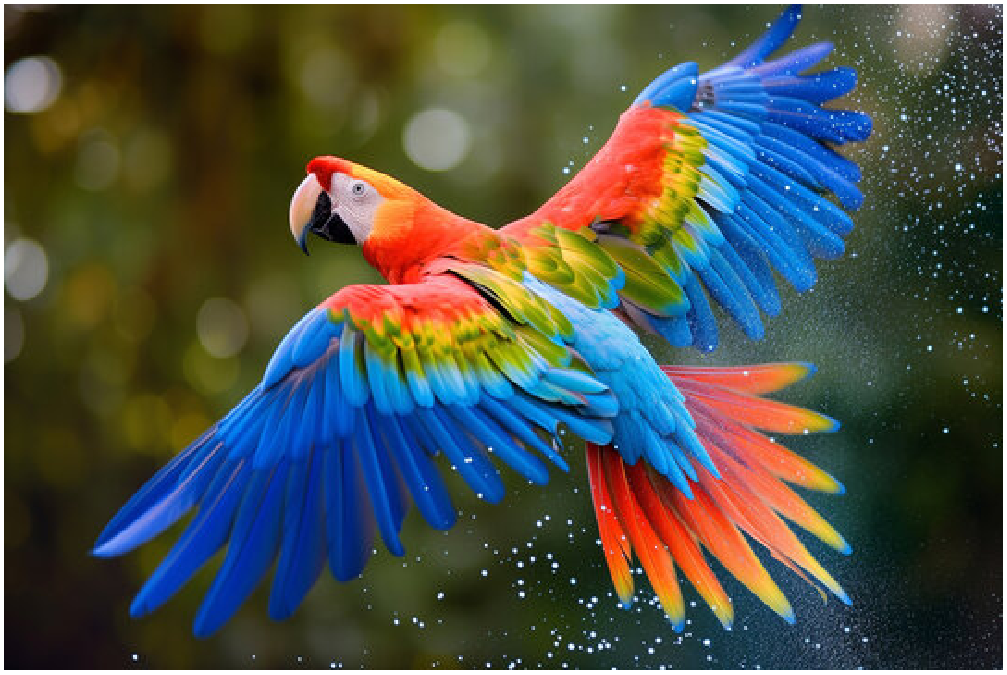


Figure 19:

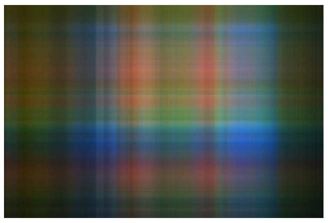


Figure 20: k = 1

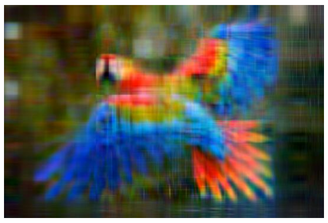


Figure 21: k = 10

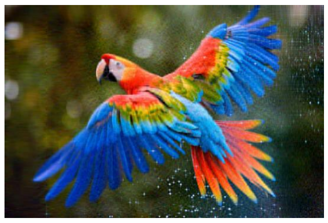


Figure 22: k = 50

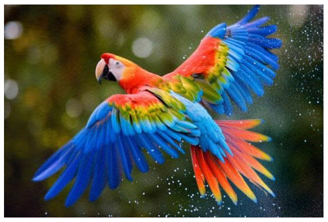


Figure 23: k = 100

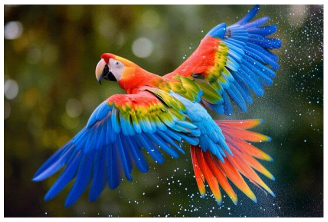


Figure 24: k = 150

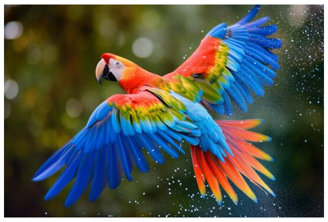


Figure 25: k = 200

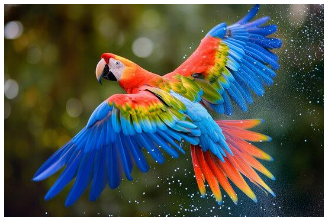


Figure 26: k = 250

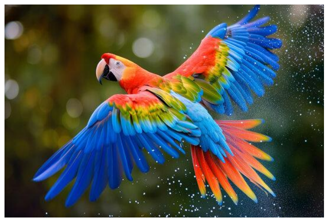


Figure 27: k = 300

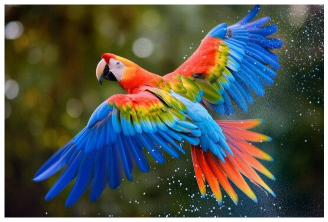


Figure 28: k = 350