SOFTWARE PROJECT-IMAGE COMPRESSION USING SVD

EE25BTECH11037- DIVYANSH

Introduction

Objective

The primary objective of this project is to implement the Singular Value Decomposition (SVD) algorithm from scratch and apply it to the task of lossy image compression. The goal is to analyze the trade-off between image quality and data compression by reconstructing images using a truncated SVD, specifically by retaining only the top k singular values.

Implementation Choice

For this project, **Option B (Full C Implementation)** was chosen. This approach involves writing the entire program—including image file I/O for JPEG and PNG formats, memory management, and all numerical computations—in the C programming language.

Report Structure

This report is organized as follows:

- **Section 1** provides a summary of Gilbert Strang's lecture on the theory and geometric intuition behind the SVD.
- **Section 2** discusses the various algorithms for SVD and justifies the choice of the Jacobi eigenvalue method for this project.
- **Section 3** details the challenges and benefits of the full C implementation, particularly concerning memory management and low-level library integration.
- **Section 4** presents the results of the compression on test images, analyzing the visual quality and Frobenius norm error for various k values.
- **Section 5** discusses the fundamental trade-offs between rank k, image quality, and compression ratio.
- **Section 6** briefly lists other applications of SVD, and the report concludes with a summary of the project's findings.
- **Section 7** concludes the report.

1 Summary of Video Lecture by Gilbert Strang

1.1 Summary

This section of the report summarizes the video lecture by Gilbert Strang on the topic of Singular Value Decomposition (SVD). The SVD provides a profound insight into the action of a linear transformation by decomposing any matrix $\mathbf{A}$ into product of two orthogonal matrices and one diagonal matrix: $\mathbf{A} = \mathbf{U\Sigma V}^\top$.

1.2 SVD definition and Geometric Intuition

The Singular Value Decomposition (SVD) is a fundamental factorization that exists for any arbitrary $m \times n$ matrix $\mathbf{A}$. It expresses $\mathbf{A}$ as a product of three matrices:

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

where:

- $\mathbf{U}$ is an $m \times m$ orthogonal matrix (columns are left singular vectors).
- $\mathbf{\Sigma}$ is an $m \times n$ rectangular diagonal matrix containing the non-negative *singular values* ($\sigma_1 \geq \sigma_2 \geq \cdots > 0$).
- $\mathbf{V}$ is an $n \times n$ orthogonal matrix (columns are right singular vectors).

Geometrically, the SVD finds a special orthonormal basis for the row space of $\mathbf{A}$ (the columns of $\mathbf{V}$) and an orthonormal basis for the column space (the columns of $\mathbf{U}$). The action of $\mathbf{A}$ on any vector $\mathbf{v}_i$ from the row space basis is a simple rotation and scaling to produce $\sigma_i \mathbf{u}_i$ in the column space.

$$\mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i \quad \text{for } i = 1, \ldots, r \tag{2}$$

Vectors from the null space (the remaining $\mathbf{v}_j$) are mapped to zero. This can be written in matrix form as $\mathbf{A}\mathbf{V} = \mathbf{U}\mathbf{\Sigma}$, which rearranges to the SVD.

1.3 Construction and Relation to Subspaces

1. Finding $\mathbf{V}$ and $\mathbf{\Sigma}$
   We use the $n \times n$ symmetric matrix $\mathbf{A}^T\mathbf{A}$. Substituting the SVD equation shows:

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

   This is the standard eigenvalue decomposition of $\mathbf{A}^T\mathbf{A}$. Therefore:

   - The columns of $\mathbf{V}$ (right singular vectors) are the orthonormal eigenvectors of $\mathbf{A}^T\mathbf{A}$.
   - The singular values σ_i are the positive square roots of the eigenvalues (λ_i) of $\mathbf{A}^T\mathbf{A}$, since $\mathbf{\Sigma}^T\mathbf{\Sigma}$ is a diagonal matrix of σ_i^2.

2. Finding $\mathbf{U}$
   Similarly, we use the $m \times m$ symmetric matrix $\mathbf{A}\mathbf{A}^T$:

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

   This shows that the columns of $\mathbf{U}$ (left singular vectors) are the orthonormal eigenvectors of $\mathbf{A}\mathbf{A}^T$.

3. The Four Fundamental Subspaces
   The SVD provides a complete, orthonormal basis for all four fundamental subspaces of a matrix $\mathbf{A}$ of rank r:

   - **Column Space,** $C(\mathbf{A})$: Basis given by the first r columns of $\mathbf{U}$, $\{\mathbf{u}_1, \ldots, \mathbf{u}_r\}$.
   - **Row Space,** $C(\mathbf{A}^T)$: Basis given by the first r columns of $\mathbf{V}$, $\{\mathbf{v}_1, \ldots, \mathbf{v}_r\}$.
   - **Null Space,** $N(\mathbf{A})$: Basis given by the remaining $n - r$ columns of $\mathbf{V}$, $\{\mathbf{v}_{r+1}, \ldots, \mathbf{v}_n\}$.
   - **Left Null Space,** $N(\mathbf{A}^T)$: Basis given by the remaining $m - r$ columns of $\mathbf{U}$, $\{\mathbf{u}_{r+1}, \ldots, \mathbf{u}_m\}$.

2 Methodology and Algorithm

This project required computing the Singular Value Decomposition (SVD) of a matrix $\mathbf{A}$, which is a factorization $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$, where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\boldsymbol{\Sigma}$ is a diagonal matrix of singular values.

2.1 Algorithm Comparison

Several SVD algorithms were researched:

1. **Golub-Kahan-Reinsch (GKR):** The industry standard for dense matrices, used in libraries like LAPACK. It first reduces A to a bidiagonal form using Householder reflections, then applies a QR-algorithm variant. It is highly efficient and stable, but its implementation is very complex.

2. **Jacobi Methods:** Iterative methods based on applying plane rotations.
   - **Two-Sided Jacobi:** Applies rotations to both sides of A to iteratively diagonalize it.
   - **One-Sided Jacobi:** Applies rotations only from the right, iteratively orthogonalizing the columns of A. This implicitly diagonalizes $A^T A$ without forming it explicitly, which avoids a potential loss of precision.

3. **Randomized SVD (R-SVD):** A probabilistic approach for very large, low-rank matrices. It projects the matrix to a smaller subspace and computes an approximate SVD, offering great speed for large-scale problems.

4. **Power Iterative Method:** The power iteration algorithm finds a matrix's largest singular value and vectors by repeatedly multiplying a vector by $\mathbf{A}^\top\mathbf{A}$. This process isolates the dominant right singular vector, $\mathbf{v_1}$, which is then used to quickly find the corresponding singular value σ_1 and left singular vector $\mathbf{u_1}$.

2.2 Chosen Algorithm: Jacobi Eigenvalue Method

The **classic Jacobi eigenvalue algorithm** was chosen for implementation, applied to the symmetric matrix $\mathbf{A}^T\mathbf{A}$.

Mathematical Explanation

As established in Section 1.2, the right singular vectors ($\mathbf{V}$) and the squared singular values (Σ^2) are the eigenvectors and eigenvalues, respectively, of the symmetric $n \times n$ matrix $\mathbf{A}^T\mathbf{A}$.

$$\mathbf{A}^T\mathbf{A} = \mathbf{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})\mathbf{V}^T = \mathbf{V}\Lambda\mathbf{V}^T$$

This project's approach is to first compute $\mathbf{B} = \mathbf{A}^T\mathbf{A}$, and then find its eigendecomposition using the classic Jacobi eigenvalue method.

The Jacobi method is an iterative algorithm that finds the eigenvalues and eigenvectors of a symmetric matrix. It works by applying a series of "Givens rotations" to the matrix $\mathbf{B}$. Each rotation, $\mathbf{R}(p, q, \theta)$, is a plane rotation designed to set a specific off-diagonal element, b_{pq}, to zero.

The algorithm proceeds in "sweeps," applying these rotations to every unique off-diagonal pair (p, q). While later rotations may undo the work of previous ones, each step reduces the sum of the squares of the off-diagonal elements. This process is repeated until the matrix is "diagonal enough" (i.e., all off-diagonal elements are below a small tolerance).

The final result is:

$$\mathbf{B}' = \ldots\mathbf{R}_2^T\mathbf{R}_1^T\mathbf{B}\mathbf{R}_1\mathbf{R}_2\cdots = \Lambda = \Sigma^2$$

- The diagonal elements of the resulting matrix $\mathbf{B}'$ are the eigenvalues, $\lambda_i = \sigma_i^2$. The singular values are their square roots.
- The matrix $\mathbf{V}$ is the product of all rotation matrices applied: $\mathbf{V} = \mathbf{R}_1\mathbf{R}_2\ldots$.

The matrix $\mathbf{U}$ can then be found using the relation $\mathbf{A}\mathbf{V} = \mathbf{U}\boldsymbol{\Sigma}$, which gives $\mathbf{u}_i = \frac{1}{\sigma_i}\mathbf{A}\mathbf{v}_i$.

Pseudocode

```
FUNCTION jacobi_eigen(n, a, V, singular_vals):
// n: integer, the size of the square matrix
// a: n x n matrix (input/output, assumed symmetric)
// V: n x n matrix (output, eigenvectors)
// singular_vals: array of size n (output)

// 1. Initialize V as an identity matrix
FOR i FROM 0 TO n-1:
    FOR j FROM 0 TO n-1:
        IF i == j THEN
            V[i][j] = 1.0
        ELSE
            V[i][j] = 0.0
        END IF
    END FOR
END FOR

// 2. Iteratively apply Jacobi rotations to diagonalize 'a'
SET computation_active = TRUE
WHILE computation_active IS TRUE:
SET computation_active = FALSE // Assume convergence for this pass

// Iterate over all unique off-diagonal elements (i, j) where i < j
FOR i FROM 0 TO n-2:
  FOR j FROM i+1 TO n-1:

     // Get the off-diagonal element
     SET Aij = a[i][j]

     // Check if the element is large enough to warrant rotation
     IF absolute_value(Aij) > 1e-9 THEN
       SET computation_active = TRUE // A rotation is needed, so not converged yet

       // Get diagonal elements
       SET Aii = a[i][i]
       SET Ajj = a[j][j]

       // Calculate rotation angle (theta)
       SET theta = 0.5 * arctan2(2 * Aij, Ajj - Aii)
       SET c = cos(theta)
       SET s = sin(theta)

       // --- Apply rotation ---
       // This transformation is A' = R^T * A * R
       // where R is the Givens rotation matrix.

       // Apply to rows (A' = R^T * A)
       // This updates rows i and j of matrix 'a'
       FOR k FROM 0 TO n-1:
         SET temp_Aki = c * a[i][k] - s * a[j][k]
         SET temp_Akj = s * a[i][k] + c * a[j][k]
         a[i][k] = temp_Aki
         a[j][k] = temp_Akj
       END FOR

       // Apply to columns (A'' = A' * R)
```

```
        // This updates columns i and j of the modified matrix 'a'
        FOR k FROM 0 TO n-1:
          SET temp_Aki = a[k][i]
          SET temp_Akj = a[k][j]
          a[k][i] = c * temp_Aki - s * temp_Akj
          a[k][j] = s * temp_Aki + c * temp_Akj
        END FOR

        // --- Update the eigenvector matrix V ---
        // V_new = V * R
        FOR k FROM 0 TO n-1:
          SET temp_Vki = V[k][i]
          SET temp_Vkj = V[k][j]
          V[k][i] = c * temp_Vki - s * temp_Vkj
          V[k][j] = s * temp_Vki + c * temp_Vkj
        END FOR

      END IF
    END FOR
  END FOR


END WHILE

// 3. Store the results
// The diagonal of 'a' now contains the eigenvalues
// The code calculates singular values as sqrt(abs(eigenvalues))
FOR i FROM 0 TO n-1:
singular_vals[i] = square_root(absolute_value(a[i][i]))
END FOR

RETURN
END FUNCTION
```

2.3 Justification for Choice

The **classic Jacobi eigenvalue algorithm** was selected primarily for its **relative simplicity of implementation**.

While the GKR algorithm is computationally faster in serial, it is significantly more complex to code. The Jacobi method, based on simple, repeated 2x2 rotations, is more conceptually straightforward, reducing development and debugging time.

Although forming $\mathbf{A}^T\mathbf{A}$ can introduce numerical precision issues (squaring the condition number), the Jacobi method itself is known for its high relative accuracy for the computed eigenvalues. Given the project's focus on implementation, the Jacobi method's clarity made it the most practical choice.

3 Implementation Details

For this project, a full C Implementation (Option B) was chosen. This approach integrates all aspects of the program—file I/O, memory management, and numerical computation—within a single C codebase. This decision presented a series of distinct challenges and benefits.

3.1 Challenges of Full C Implementation

The most significant challenge was the manual handling of image file I/O. Unlike high-level languages like Python, which offer simple, one-line commands for reading and saving images, C requires direct interaction with specialized, low-level libraries.

- **Complex Library Integration:** The project required integrating both `libjpeg` and `libpng`. As seen in the `read_jpeg` and `read_png` functions, this is not a trivial task. It involves initializing and managing numerous control structures (e.g., `jpeg_decompress_struct`), setting up error handlers, and meticulously managing memory buffers and row pointers.
- **Low-Level Data Manipulation:** The libraries do not simply return a matrix. They require the developer to configure parameters for color space, bit depth, and alpha channels. The code had to explicitly include logic to convert all input images to an 8-bit grayscale representation to be processed as a single numerical matrix.
- **Manual Memory Management:** A full C implementation places the entire burden of memory management on the developer. All matrices, including the initial matrix `a`, its transpose `at`, `ata`, and the SVD components `U` and `V`, required explicit allocation. Utility functions like `allocate_matrix` and `free_matrix` were created to manage this, but the risk of memory leaks or segmentation faults is a constant factor during development.

3.2 Benefits of a Full C Implementation

Despite the complexities, this approach offered substantial benefits that were crucial to the project's success.

- **Performance and Efficiency:** The primary advantage is execution speed. The core of the SVD computation, the `jacobi_eigen` function, is highly iterative, involving many nested loops. By implementing this in C, the algorithm is compiled to native machine code, allowing it to execute with maximum efficiency. This is particularly noticeable when compared to the performance of equivalent numerical loops in an interpreted language.
- **Fine-Grained Control:** This approach provides complete, low-level control over every aspect of the program. From the precise, contiguous memory layout of the matrices (as defined in `allocate_matrix`) to the specific rounding and clamping logic in the `write_jpeg` function, nothing is hidden behind an abstraction.
- **Project Scope and Learning:** Choosing the full C implementation demonstrates a comprehensive understanding of the entire data pipeline. It proves an ability to manage complex external libraries, perform manual memory management, and implement sophisticated numerical algorithms from scratch, fulfilling the most challenging requirements of the project.
- **Self-Contained Executable:** The final result is a single, compiled executable. It does not depend on a large runtime environment (like Python or MATLAB) and can be run efficiently on any system with the requisite C libraries.

3.3 Reflection on Choice

The decision to pursue a full C implementation was a trade-off between development complexity and runtime performance. The initial development phase, particularly the image I/O, was significantly more difficult and time-consuming than it would have been with a Python-based wrapper. However, the resulting performance of the numerical core and the deep level of control achieved were invaluable. This choice ultimately led to a more robust and efficient final program and learning a lot from this project.

3.4 Code structure:

In my code folder there are `c_lib`, `c_main` and `python` folders. In `c_lib`, there is `jacobi.h` which has the function definitions. In `c_main` there are my main codes `svd.c` which has the main function which runs my svd pipeline and `jacobi.c` which contains the svd codes to perform svd. In `python`, there is the `plot.py` which plots the error and singular value plots.

4 Results and Analysis:

4.1 Reconstructed Images

The SVD algorithm was applied to the three test images. For each image, the SVD was computed once, and the matrix was then reconstructed using several different k values to demonstrate the effect of low-rank approximation on image quality.

1. **Einstein:**
   Here are the original image and reconstructed images for `einstein.jpg`

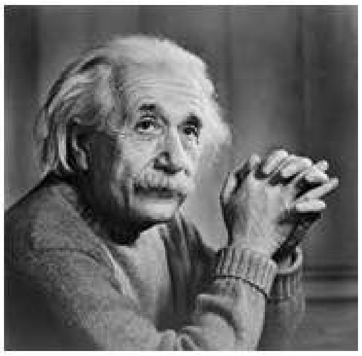

Figure 1: Original Einstein Image

Now, here are the reconstructed images:c

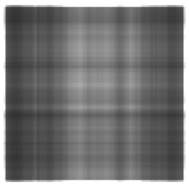

Figure 2: Einstein with k=1

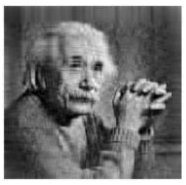

Figure 5: Einstein with k=20

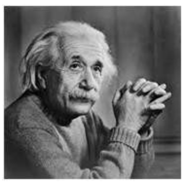

Figure 8: Einstein with k=150

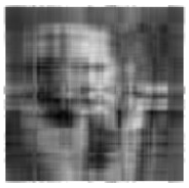

Figure 3: Einstein with k=5

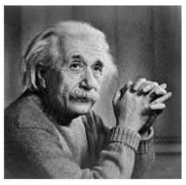

Figure 6: Einstein with k=50

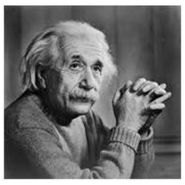

Figure 9: Einstein with k=170

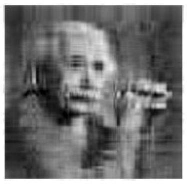

Figure 4: Einstein with k=10

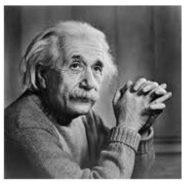

Figure 7: Einstein with k=100

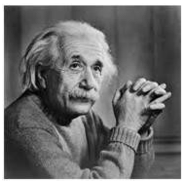

Figure 10: Einstein with k=186

k	Abs Frob Error	Rel Frob Error
1	7264.539 621	0.333 097
5	4713.338 525	0.216 118
10	3248.815 786	0.148 966
20	2126.439 116	0.097 502
30	1568.715 345	0.071 929
40	1163.648 112	0.053 356
50	880.351 151	0.040 366
60	655.041 379	0.030 035
70	480.536 699	0.022 034
80	347.961 779	0.015 955
90	244.329 015	0.011 203
100	164.781 709	0.007 556
110	104.376 714	0.004 786
120	63.109 796	0.002 894
130	35.135 193	0.001 611
140	17.484 407	0.000 802
150	9.620 052	0.000 441
160	5.257 037	0.000 241
170	2.211 619	0.000 101
180	0.258 229	0.000 012
186	0.000 000	0.000 000

Table 1: Einstein: k and Frobenius Norm

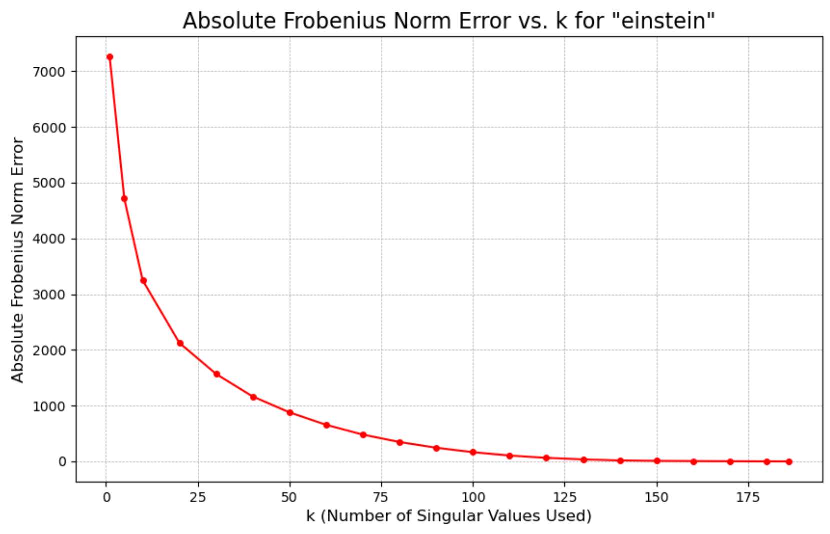

Figure 11: Einstein-Frobenius Norm Plot

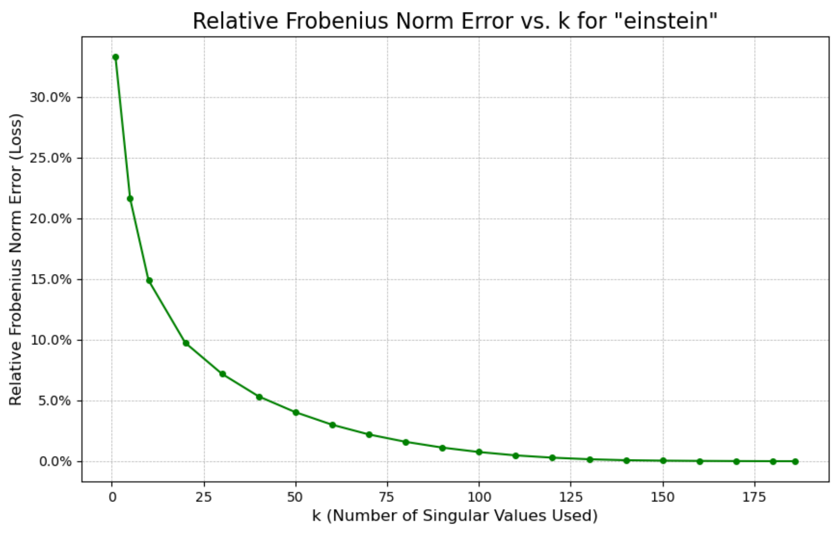

Figure 12: Einstein-Relative Frobenius Norm Plot

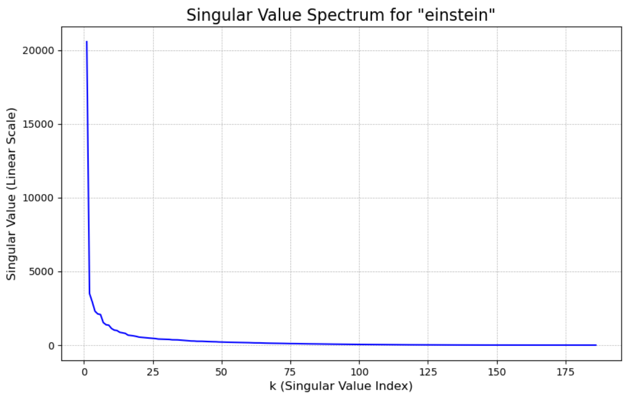

Figure 13: Einstein-Singular Values Plot

We can conclude from this data that

- The Frobenius norm approaches zero very fast for $k \geq 75$, which is also evident from the reconstructed images that there is very little deviation from the original images.
- The Relative Frobenius error becomes less than approximately 5 percent after k=50. This implies that reconstructed images for $k \geq 50$ are very close to the original image
- The Singular Values Spectrum is a very fast decaying graph, which implies that most of the data can be reconstructed using the first 50 to 60 singular values.

2. **Globe:**
Here are the original image and reconstructed images for `globe.jpg`

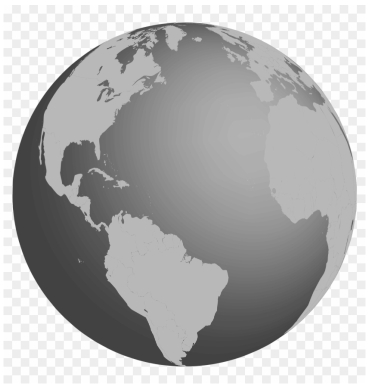

Figure 14: Original Globe Image

Now, here are the reconstructed images:

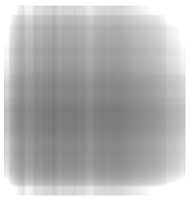

Figure 15: Globe with k=1

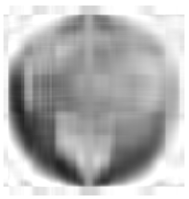

Figure 16: Globe with k=5

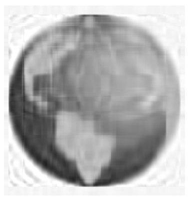

Figure 17: Globe with k=10

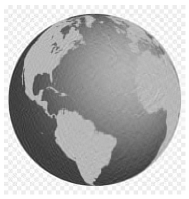
Figure 18: Globe with k=50

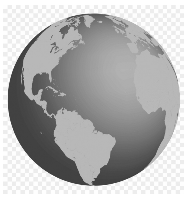
Figure 20: Globe with k=300

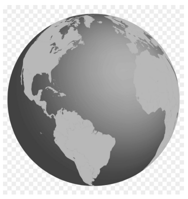
Figure 22: Globe with k=750

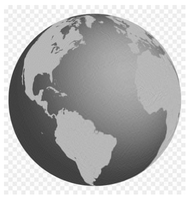
Figure 19: Globe with k=100

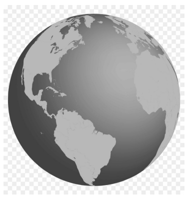
Figure 21: Globe with k=500

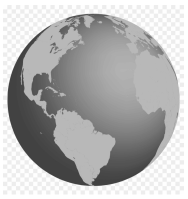
Figure 23: Globe with k=840

k	Abs Frob Error	Rel Frob Error	k	Abs Frob Error	Rel Frob Error
1	39 782.882 816	0.251 293	400	542.125 178	0.003 424
5	20 703.892 759	0.130 779	450	391.983 559	0.002 476
10	15 060.473 607	0.095 131	500	271.912 546	0.001 718
50	6185.213 475	0.039 070	550	178.540 157	0.001 128
100	3672.668 558	0.023 199	600	108.476 443	0.000 685
150	2494.993 225	0.015 760	650	58.422 222	0.000 369
200	1787.546 745	0.011 291	700	26.367 710	0.000 167
250	1315.165 371	0.008 307	750	8.726 019	0.000 055
300	982.408 353	0.006 206	800	0.570 731	0.000 004
350	734.157 969	0.004 637	840	0.000 000	0.000 000

Table 2: Globe: k and Frobenius Norm

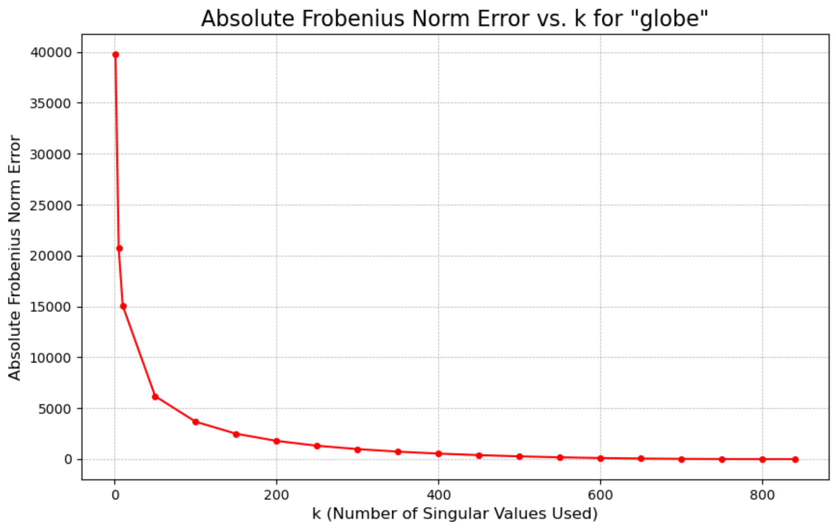

Figure 24: Globe-Absolute Frobenius Norm Plot

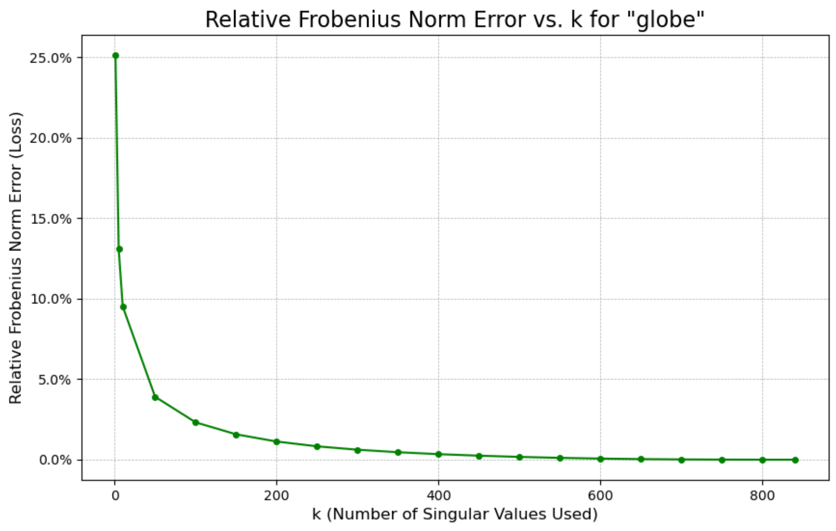

Figure 25: Globe-Relative Frobenius Norm Plot

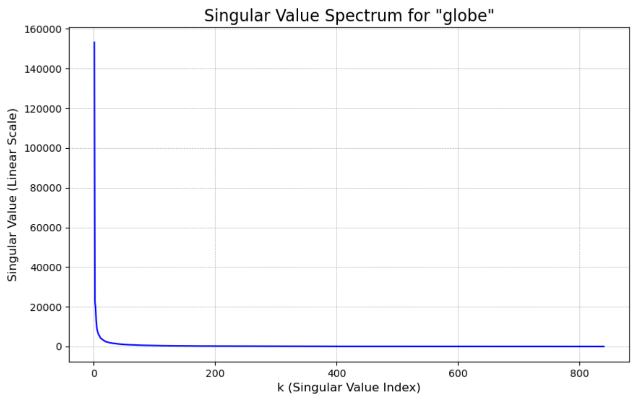

Figure 26: Globe-Singular Values Plot

We can conclude from this data that

- The Frobenius norm approaches zero very fast for $k \geq 150$, which is also evident from the reconstructed images that there is very little deviation from the original images.
- The Relative Frobenius norm becomes less than approximately 5 percent after k=60. This implies that reconstructed images for $k \geq 60$ are very close to the original image
- The Singular Values Spectrum is a very fast decaying graph, which implies that most of the data can be reconstructed using the first 50 to 60 singular values.

3. **Greyscale:**
   Here are the original image and reconstructed images for `greyscale.png`

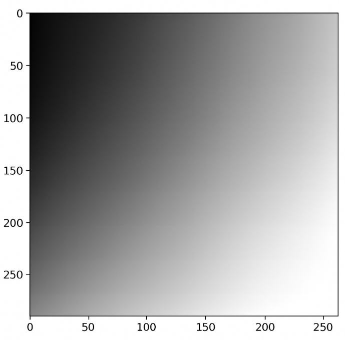

Figure 27: Original Greyscale Image

Now, here are the reconstructed images:

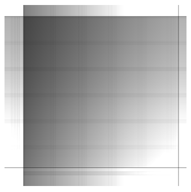

Figure 28: Greyscale with k=1

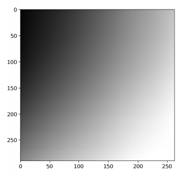

Figure 31: Greyscale with k=50

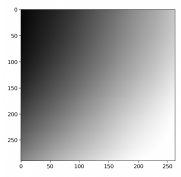

Figure 34: Greyscale with k=900

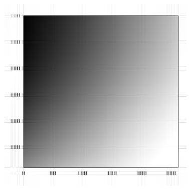

Figure 29: Greyscale with k=5

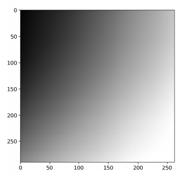

Figure 32: Greyscale with k=100

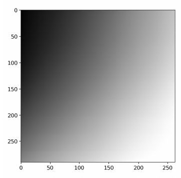

Figure 35: Greyscale with k=1000

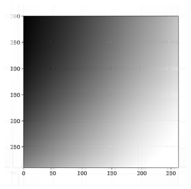

Figure 30: Greyscale with k=10

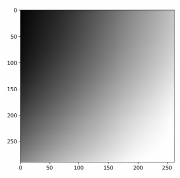

Figure 33: Greyscale with k=500

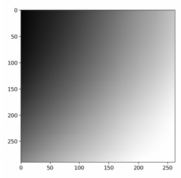

Figure 36: Greyscale with k=1024

k	Abs Frob Error	Rel Frob Error	k	Abs Frob Error Error	Rel Frob Error
1	41 571.602 531	0.214 774	500	193.428 152	0.000 999
5	11 146.309 445	0.057 586	550	165.108 456	0.000 853
10	7176.804 841	0.037 078	600	138.604 556	0.000 716
50	1159.888 175	0.005 992	650	114.002 610	0.000 589
100	512.345 679	0.002 647	700	91.399 091	0.000 472
150	456.295 438	0.002 357	750	70.703 883	0.000 365
200	409.463 145	0.002 115	800	51.907 156	0.000 268
250	366.822 944	0.001 895	850	35.323 655	0.000 182
300	327.292 094	0.001 691	900	21.187 044	0.000 109
350	290.394 556	0.001 500	950	9.727 900	0.000 050
400	255.950 829	0.001 322	1000	1.791 047	0.000 009
450	223.680 188	0.001 156	1024	0.000 000	0.000 000

Table 3: Greyscale: k and Frobenius Norm

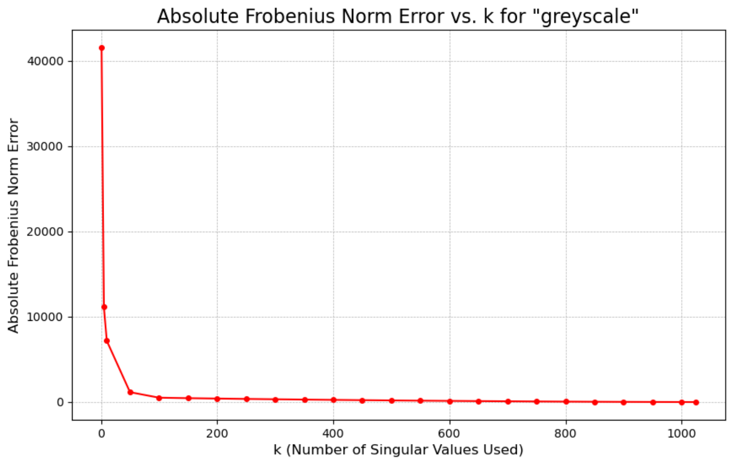

Figure 37: Greyscale-Absolute Frobenius Norm Plot

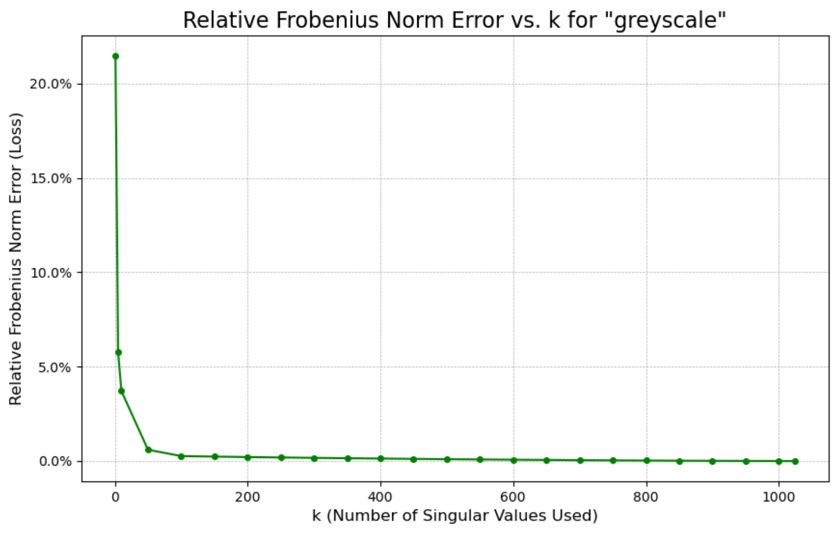

Figure 38: Greyscale-Relative Frobenius Norm Plot

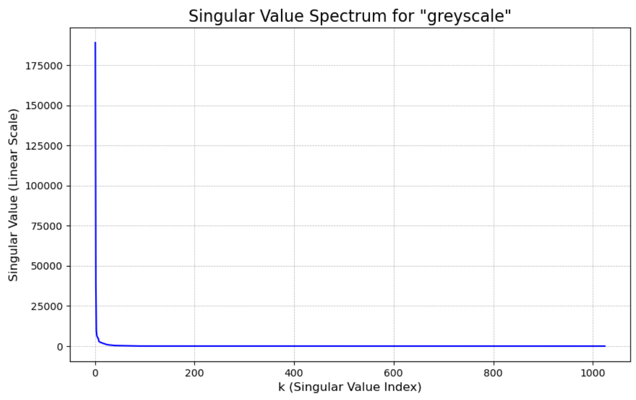

Figure 39: Greyscale-Singular Values Plot

We can conclude from this data that

- The Frobenius norm approaches zero very fast for $k \geq 300$, which is also evident from the reconstructed images that there is very little deviation from the original images.
- The Relative Frobenius norm becomes less than approximately 5 percent after k=30. This implies that reconstructed images for $k \geq 30$ are very close to the original image
- The Singular Values Spectrum is a very fast decaying graph, which implies that most of the data can be reconstructed using the first 150 to 200 singular values.

5 Discussion:

The results from the previous section highlight the practical application of SVD for image compression. This application is governed by two fundamental trade-offs involving the rank k: image quality versus the number of singular values, and compression ratio versus the number of singular values.

5.1 Trade-off: k vs. Image Quality

The SVD sorts the singular values σ_i in descending order. This means that σ_1 (and its associated vectors u_1, v_1) captures the most significant component of the image, σ_2 captures the next most significant, and so on.

- **Low k values** (e.g., $k = 5, 20$) represent the low-frequency data of the image. This includes the general structure, overall shapes, and broad areas of light and dark. As seen in the results, a small k produces a "blocky" or "blurry" image that is recognizable but lacks any fine detail.
- **High k values** (e.g., $k = 50, 100$) begin to incorporate the high-frequency data. This includes textures, sharp edges, and subtle gradations in shading.

As k increases, the reconstructed image $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$ becomes a more faithful approximation of the original matrix A. The error of this approximation can be quantified by the Frobenius norm of the difference, which is precisely the root of the sum of the squares of the singular values that were *omitted.*

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

As k increases, this error term decreases, and the image quality improves. This was confirmed by the Frobenius error calculations from the implementation, which showed a rapid drop in error for initial k values and a slower decline as k increased.

5.2 Trade-off: k vs. Compression

Image compression is achieved if the storage required for the SVD components is less than the storage for the original image.

- **Original Image Storage:** An $m \times n$ grayscale image requires $m \times n$ values to be stored.
- **Compressed Image Storage:** To reconstruct the k-rank approximation A_k, we only need to store the first k singular values, the first k columns of U, and the first k rows of V^T.
  - k columns of U: $m \times k$ values
  - k singular values (Σ_k): k values
  - k rows of V^T (or k columns of V): $n \times k$ values

  The total storage required is $mk + k + nk$, or $\mathbf{k(m+n+1)}$ values.

Compression is achieved when $k(m+n+1) < mn$. This creates a direct trade-off:

- **Low k:** Provides a high compression ratio (low storage cost), but as discussed, results in low image quality.
- **High k:** Provides high image quality, but results in a low compression ratio. As k approaches n, the storage cost $k(m+n+1)$ may even exceed the original mn, resulting in no compression at all.

The goal of SVD compression is to find a "sweet spot" for k that provides the best balance between acceptable image quality and a meaningful reduction in storage size.

6 Other uses:

SVD is a very versatile tool. It can be used for other things also like

1. **Principal Component Analysis(PCA):** SVD is the mathematical backbone of PCA, a core technique in data science and machine learning for dimensionality reduction.
2. **Recommender Systems:** Used in collaborative filtering (like on Netflix) to find latent factors in user-item rating matrices to predict what a user might like.
3. **Digital Signal Processing and Noise Reduction:** SVD can be used to isolate and remove noise from a signal or image by discarding the singular values associated with the noise (which are typically very small).

7 Conclusion

This project successfully demonstrated the implementation and application of the Singular Value Decomposition (SVD) for lossy image compression. The key finding is that the SVD is a highly effective technique for analyzing and approximating an image matrix, as it sorts the image components by their "energy" or significance. The results clearly show that a large portion of an image's visual information is concentrated within the first few singular values.

The chosen implementation, a full C program using the Jacobi algorithm to compute the SVD, proved to be both challenging and effective. While manual file I/O using `libjpeg` and `libpng` added significant development complexity, the resulting executable was highly efficient. The Jacobi method itself was found to be numerically stable and well-suited for the task, successfully producing the U, Σ, and V components needed for reconstruction.

Ultimately, the project highlights the fundamental trade-off in SVD compression, which is governed by the chosen rank, k.

- **Image Quality** is directly proportional to k. As k increases, the approximation error decreases and high-frequency details are restored.
- **Compression** is inversely proportional to k. The storage cost scales linearly as $k(m+n+1)$, meaning a higher k results in a larger file size.

The effectiveness of SVD lies in its ability to provide a "control knob" (k) that allows a user to find a practical balance between these competing demands, achieving a high compression ratio while retaining acceptable visual fidelity.