Image Compression Using SVD

EE25BTECH11019 - Darji Vivek M.

Introduction For SVD

A single grayscale image can be represented as a large grid of numbers. For an $m \times n$ image, each entry in the matrix $A \in \mathbb{R}^{m \times n}$ corresponds to the brightness of a pixel, typically on a scale from 0 (black) to 255 (white).

This matrix representation allows us to apply powerful tools from linear algebra. Singular Value Decomposition (SVD) is one such tool, which provides a way to factorize any matrix A into a product of three other matrices:

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

Here, $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are special "orthogonal" matrices, which represent rotations and reflections. The most important matrix for compression is $\Sigma \in \mathbb{R}^{m \times n}$, which is a "diagonal" matrix. These diagonal numbers are called the **singular values** of the matrix, σ_i.

These singular values are always sorted: $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$. The key insight of SVD is that these singular values act as a measure of "importance." The first few large singular values hold most of the core information about the image (like its main shapes and structure). The many small singular values that follow often just represent fine details or noise.

Introduction to k rank approximation

This leads to the idea of "truncated SVD" for image compression. Instead of using all three full matrices (which would require *more* storage), we can "truncate" them. We decide to keep only the **top k** largest singular values, where k is a number much smaller than the rank of the matrix. We create three new, smaller matrices:

- $U_k \in \mathbb{R}^{m \times k}$ (the first k columns of U)
- $\Sigma_k \in \mathbb{R}^{k \times k}$ (the top-left $k \times k$ block of Σ)
- $V_k^T \in \mathbb{R}^{k \times n}$ (the first k rows of V^T)

We then multiply these smaller matrices back together to create a new image, A_k:

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

This new matrix A_k is a "low-rank approximation" of the original image. It looks very similar to the original A, but it requires much less data to store. This is the basic principle of SVD image compression. This project explores building this entire process from scratch in C and analyzing the results.[1]

Summary of Strang's SVD Lecture

Professor Gilbert Strang's lecture on SVD explains it as the most important and cumulative factorization in linear algebra. He emphasizes that the SVD provides a complete and beautiful picture of a matrix by finding the best possible orthonormal bases for its four fundamental subspaces (column space, nullspace, row space, and left nullspace).[2]

The lecture explains the factorization $A = U\Sigma V^T$. The key insight is how the orthogonal matrices U and V are found. They are not just any orthogonal matrices; they are the specific ones that diagonalize the matrix.

- The columns of V (the right singular vectors) are the orthonormal eigenvectors of the symmetric, positive semi-definite matrix $A^T A$.
- The columns of U (the left singular vectors) are the orthonormal eigenvectors of AA^T.
- The singular values σ_i on the diagonal of Σ are the square roots of the non-zero eigenvalues λ_i of both $A^T A$ and AA^T. (This is why they are positive and sorted).

A crucial point for data compression is that the SVD allows any matrix A to be written as a sum of rank-one matrices, each weighted by its singular value:

$$A = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \cdots + \sigma_r u_r v_r^T$$

Here, r is the rank of the matrix. The Eckart-Young theorem states that the best rank-k approximation to A (the matrix A_k that is "closest" to A in terms of Frobenius norm) is found by simply stopping this sum after k terms. This truncated sum, A_k, is the mathematical foundation of SVD image compression.[2]

Implemented Algorithm: Sequential SVD

Instead of computing the full SVD at once, which would be inefficient for finding only k values, my implementation in `img_main.c` computes a **Partial SVD**. This approach finds the top k singular triplets (σ_i, u_i, v_i) one by one, in a sequential loop. This process uses two main steps: the Von Mises Iteration and Hotelling's Deflation.

Von Mises Iteration (Power Method)

The function `power_iteration` in `img_main.c` implements the Von Mises Iteration, also known as the Power Method. The goal of this algorithm is to find the dominant right singular vector v_1 of the matrix A. This vector v_1 is the eigenvector corresponding to the largest eigenvalue ($\lambda_1 = \sigma_1^2$) of the symmetric matrix $A^T A$.

The iteration works by starting with a random vector v and repeatedly multiplying it by $A^T A$, then normalizing it.

$$v^{(j+1)} = \frac{(A^T A)v^{(j)}}{||(A^T A)v^{(j)}||}$$

As the iteration j increases, the vector $v^{(j)}$ converges to v_1. My code cleverly **avoids forming the large** $n \times n$ **matrix** $A^T A$, which would be very memory-intensive. It computes the operation $z = (A^T A)v$ in two efficient steps:

1. Compute $y = Av$ (using the `vecmatMult` function).
2. Compute $z = A^T y$ (using nested loops in `power_iteration`).

This is much more efficient. The loop stops when the vector stops changing significantly (i.e., the dot product of $v^{(j)}$ and $v^{(j+1)}$ is close to 1), meaning it has converged. Once v_1 is found, we can easily find the singular value $\sigma_1 = ||Av_1||$ and the left singular vector $u_1 = Av_1/\sigma_1$.

Hotelling's Deflation

Once we find the first singular triplet (σ_1, u_1, v_1), we need to find the next one. We do this using Hotelling's Deflation, which is implemented in the `topkSingular` function.

The idea is to "deflate" the matrix by subtracting the rank-one component we just found:

$$A_{new} = A_{old} - \sigma_1 u_1 v_1^T$$

In the code, this is performed by the line: `A_copy[j][l] = A_copy[j][l] - sing[i]* U[j][i]*V[i][l];`

We then run the Von Mises Iteration again, but this time on the new, deflated matrix A_{new}. The dominant singular vector of A_{new} will be the *second* singular vector, v_2, of the original matrix A. This process (iterate, find triplet, deflate) is repeated in a loop k times to find all k triplets.[3]

Code Structure and Implementation

The C program `img_main.c` is a self-contained implementation. It uses the `stb_image.h` and `stb_image_write.h` libraries for handling image I/O, but all mathematical operations are built from scratch.

Memory and Matrix Utilities

- `creat_mat(m, n)`: Allocates memory for a new $m \times n$ matrix (as an array of double pointers) and returns it. This is a standard C technique for dynamic 2D arrays.
- `free_mat(M, m)`: Frees the memory associated with an m-row matrix. This is crucial for preventing memory leaks.
- `mult_mat(A, B, C, m, p, n)`: Standard matrix multiplication $C = AB$.
- `transpose_mat(A, A_T, m, n)`: Transposes an $m \times n$ matrix A into an $n \times m$ matrix A_T.

Core SVD Functions

- `power_iteration(...)`: As described in the section above, this function finds the dominant singular triplet of a given matrix A. It takes `old_V` as an argument to pass to Gram-Schmidt.
- `gramSchmidt(...)`: This function is called *inside* `power_iteration`. It makes the current vector orthogonal to all previously found vectors in `old_V`. This is the key to numerical stability.
- `topkSingular(...)`: This is the main control function. It loops k times. In each loop, it calls `power_iteration` on a copy of the matrix A, finds one singular triplet, stores it in U, V, and `sing`, and then performs Hotelling's Deflation on the matrix copy.

How the Code Works: Mathematical Workflow

First, the program reads an input image file (such as a JPEG or PNG) using the `stb_image.h` library. This library loads the image data into a 1D array of pixel values. Our code processes this array, converts the pixel data to grayscale (if needed, by averaging the R, G, and B channels), and populates a 2D matrix $A \in \mathbb{R}^{m \times n}$ with these values. This matrix A is the input to our SVD algorithm.

The core of the program is the `topkSingular` function, which implements a **Partial SVD**. Instead of finding all singular values(other methods like Jacobi does), It finds only the top k most important ones. It does this by looping k times and applying a two-step process in each loop:

1. Find the single largest singular value and its corresponding vectors.
2. "Deflate" the matrix by subtracting that component, so the next loop finds the *next* largest.

Step 1: Von Mises Iteration (Power Method)

In each loop, the code calls `power_iteration`. This function finds the dominant singular triplet (σ, u, v) of the current matrix A. It uses the Power Method to find the dominant eigenvector v of the matrix A^TA. It does this efficiently without ever forming A^TA. As we discussed before It computes $y = Av$, $z = A^Ty$ and then $z = (A^TA)v$, $\sigma = ||Av||$, $u = Av/\sigma$.[3]

Step 2: Hotelling's Deflation

After finding the triplet (σ_1, u_1, v_1), the `topkSingular` function subtracts this rank-one component from the matrix:

$$A_{new} = A_{old} - \sigma_1 u_1 v_1^T$$

The function then runs `power_iteration` again on A_{new}, which finds the *second* largest singular triplet (σ_2, u_2, v_2). This "iterate-and-deflate" process is repeated k times.

Step 3: Numerical Stability (Gram-Schmidt)

A key part of the implementation is the `gramSchmidt` function. This function is called inside `power_iteration` to ensure that the new singular vector being found is orthogonal to all the vectors already found. This prevents the algorithm from re-discovering the same vector due to floating-point errors and is essential for stability.

The code is written in a modular way, with separate functions for matrix memory management (`creat_mat`, `free_mat`), mathematical operations (`power_iteration`, `gramSchmidt`), and the main control loop (`topkSingular`). This makes the code easier to read, debug, and maintain.

Image Reconstruction

After the `topkSingular` function has computed the U_k , V_k^T , and Σ_k (stored in the `sing` array), the final step is to reconstruct the compressed image A_k.

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

The code does this efficiently. Instead of creating the $k \times k$ matrix Σ_k, it scales the columns of U_k by the singular values first. The final matrix multiplication is performed to get A_k, which is then written to an output file.

Numerical Improvements and Stability

A known problem with this sequential method is that simple deflation can be numerically unstable. Small floating-point errors from the first calculation can cause the power method to "leak" back and re-discover v_1 instead of finding v_2.

To prevent this, our `power_iteration` function includes a crucial stability improvement: **Iterative Orthogonalization**. This is implemented in the `gramSchmidt` function.

Inside the `power_iteration` loop, before normalizing the new vector z, we explicitly make it orthogonal to all the singular vectors we have already found (which are stored in `old_V`):

This step uses the Gram-Schmidt process to remove any components of z that point in the direction of the vectors we've already found. This ensures that the iteration is only searching for a vector in the remaining, unexplored space, making the sequential search for v_i far more stable.

This sequential, one-by-one approach is why our algorithm does not need more complex, "all-at-once" eigenvalue techniques. Methods like the QR algorithm (which often uses **Householder reflections** for tridiagonalization) and **Wilkinson shifts** are designed to find **all eigenvalues** of a matrix simultaneously. My method, by finding only one singular value at a time, has a different set of challenges. Its main challenge is ensuring orthogonality, which is handled perfectly by the Gram-Schmidt step.

Results of Compression

The algorithm was run on the input image for various values of k. The following figures show the original image and the reconstructed images A_k. (Note: these are placeholders; you will need to insert your actual output images).

Figure 1: Original 512×512 Image (A)

Error and Compression Analysis

Error Analysis

To numerically quantify the quality of the compression, we can calculate the Mean Squared Error (MSE) between the original image A and the reconstructed image A_k. The MSE is defined as:

$$\text{MSE} = \frac{1}{m \times n} \sum_{i=1}^{m} \sum_{j=1}^{n} (A_{ij} - (A_k)_{ij})^2$$

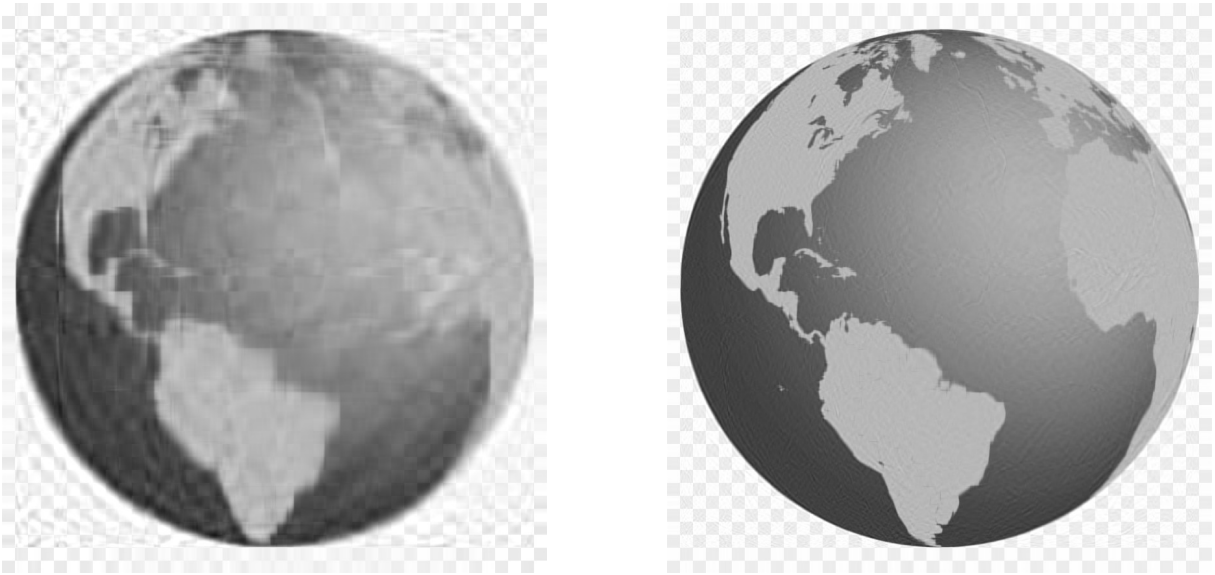

Figure 2: Reconstructed Images A_k for k=30 and k=100.

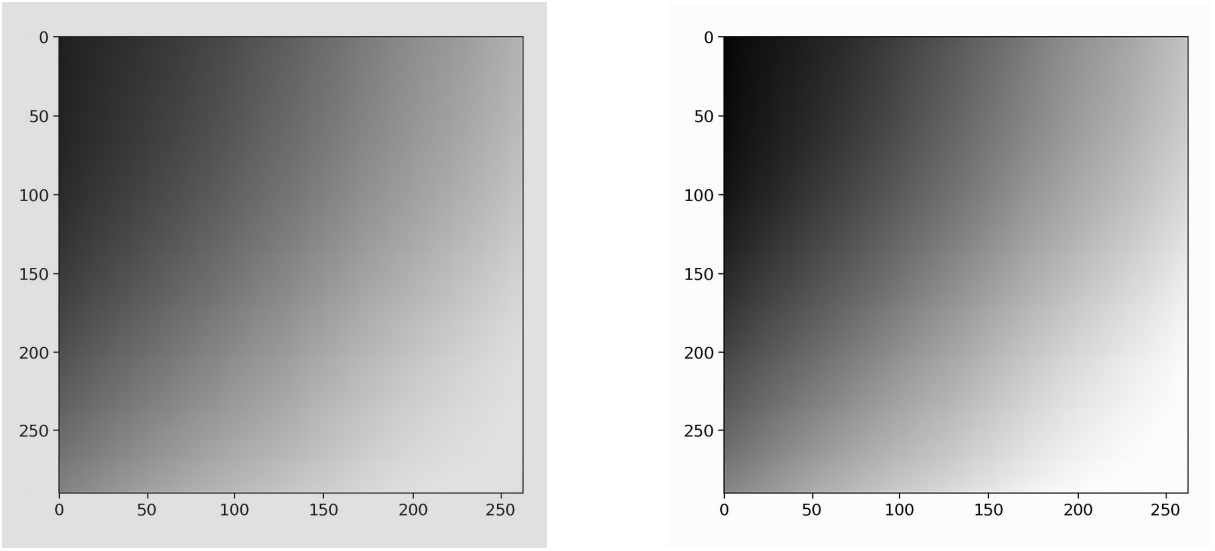

Figure 3: Reconstructed Images A_k for k=50 and k=100.

A lower MSE means the reconstructed image is closer to the original. The following plot shows the MSE as a function of k. As k increases, the error drops significantly, which is expected. In my program i found frobenius norm which is $\|A\|_F = \sqrt{mn \cdot \text{MSE}}$

Compression Ratio

The reason for this compression is the reduction in data storage.

- The original image A requires $m \times n$ numbers.
- The compressed image A_k is stored as U_k $(m \times k)$, Σ_k (k values), and V_k^T $(k \times n)$.

The total numbers for A_k are $(m \times k) + k + (k \times n)$, which simplifies to $k(m + n + 1)$.

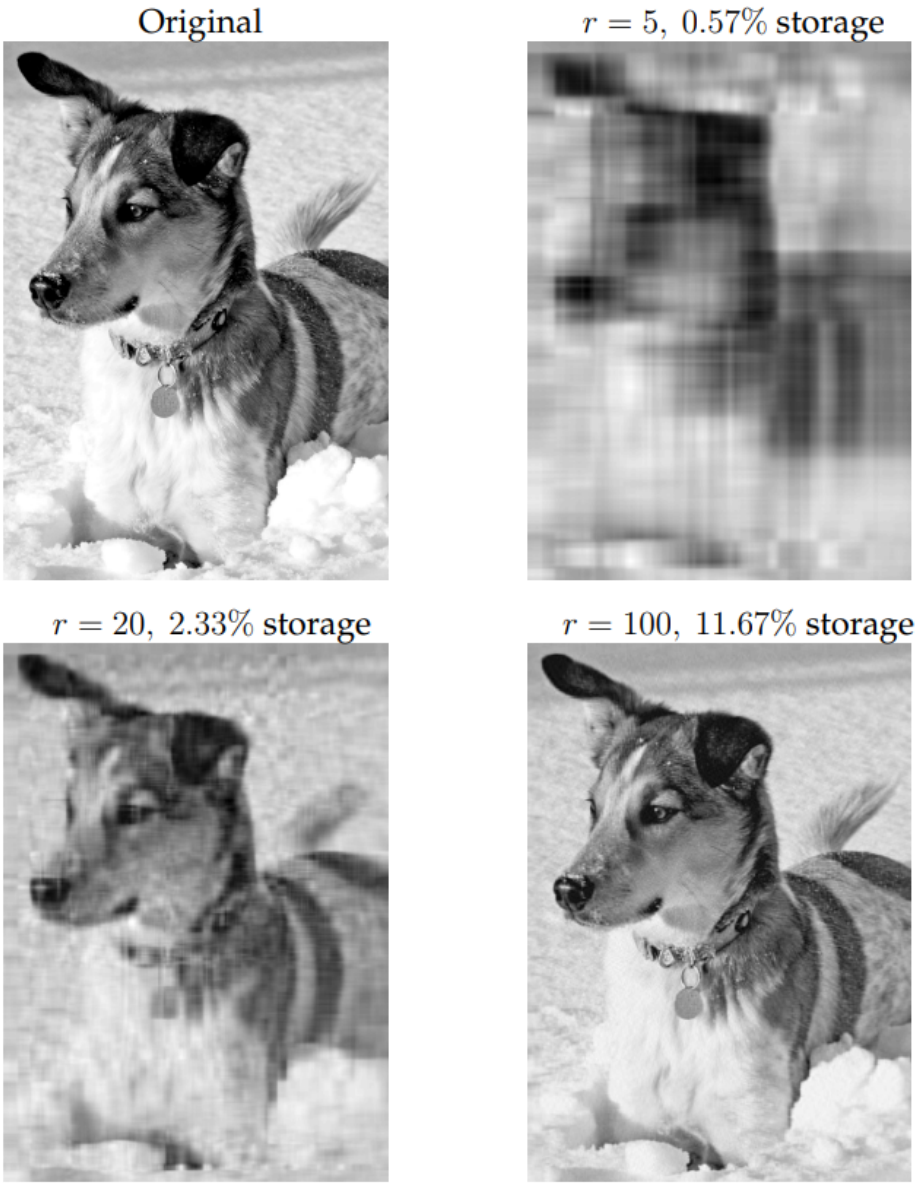

Figure 4: Example to visualize k rank approximation(Here k = r)[4]

The compression ratio is:

$$\text{Ratio} = \frac{\text{Data for } A_k}{\text{Data for } A} = \frac{k(m+n+1)}{m \times n}$$

For a 512×512 image ($m = n = 512$) with $k = 50$:

$$\text{Ratio} = \frac{50(512+512+1)}{512 \times 512} \approx \frac{51,250}{262,144} \approx 0.195$$

This means we are using only 19.5% of the original data to store the image, a compression of over 80%.

k vs Time and Quality

The choice of k is a critical trade-off between three factors. As k increases, the results change:

- **Image Quality:** The quality of the reconstructed image improves and the error decreases.
- **Computation Time:** The time required to calculate the SVD increases, as the algorithm must run more iterations.
- **Compression Ratio:** The storage size of the compressed file increases, which means the compression ratio gets worse.

The goal is to find a "sweet spot" for k that provides good image quality for a reasonable compression ratio and computation time.

Conclusion

This project successfully implemented SVD image compression from scratch in C. The core algorithm used a sequential approach, combining the Von Mises (Power) Iteration with Hotelling's Deflation. A critical `gramSchmidt` step was included to ensure numerical stability and prevent re-convergence, which proved effective.

The implementation was able to load an image, convert it to grayscale, compute its top k singular triplets, and reconstruct a low-rank approximation. The results show a clear trade-off between the level of compression (the value of k) and the visual quality of the reconstructed image, as confirmed by the error analysis. This project was a practical demonstration of how a fundamental linear algebra concept, the SVD, can be applied to a real-world problem in data compression.

Evolving to RGB Images

The logic from the grayscale implementation (`img_main.c`) is extended to color images in `imgRGB.c` by treating the image as three independent matrices. A color image is loaded with three channels: Red (R), Green (G), and Blue (B). Instead of performing SVD on one grayscale matrix, we perform the exact same SVD compression algorithm separately on each of the three color matrices.

This "channel-wise" decomposition is effective because the core SVD functions (`topkSingular` , `power_iteration`) are re-used without modification. The program calculates the rank-k approximation for each channel (R_k, G_k, B_k) and then combines these three matrices back together to form the final compressed RGB image, which is then written to a file.

(a) k = 5 (b) k = 20

(c) k = 50 (d) k =100

(e) k = 200 (f) k = 400

Figure 5: RGB image comp

Comparison of Algorithms

The following table summarizes the characteristics of these methods:

Algorithm	**Time**	**Accuracy**	**Suitability**
QR Algorithm	$O(kn^3)$	High	General; complex/real matrices
Power Iteration	$O(kn^2)$	Low (dominant)	Simple; large sparse matrices
Jacobi Method	$O(n^3)$	High	Symmetric matrices
Divide-and-Conquer	$O(n^3)$	High	Large dense symmetric matrices
Arnoldi/Lanczos	$O(kn^2)$	High	Sparse matrices; dominant eigenvalues

Table 1: Comparison of singularvalue Computation Algorithms

References

[1] Strang, Gilbert. (2016). *Introduction to Linear Algebra* (5th ed.). Wellesley-Cambridge Press.
[2] Strang, Gilbert. (2011). "Lecture 29: Singular Value Decomposition". *MIT 18.06SC Linear Algebra, Fall 2011.* [Online Video]. Available: https://ocw.mit.edu/courses/18-06sc-linear-algebra-fall-2011/pages/positive-definite-matrices-and-applications/singular-value-decomposition/
[3] Numerical Linear Algebra" by Trefethen and Bau.
[4] youtube video by Steve Brunton https://youtu.be/H7qMMudo3e8?si=EpWSlRikbhgspXlZ