

Software Assignment Report

Image compression using Truncated SVD

Harsha B J - EE25BTECH11026

November 8, 2025

Contents

1	Introduction	3
1.1	Summary of Gilbert Strang's Lecture on SVD	3
2	Mathematical Background	4
3	Motivation	5
4	Algorithm and Implementation	6
4.1	Selected Algorithm: Truncated Singular Value Decomposition	6
4.2	Implementation Overview	6
4.3	Pseudocode of the Implementation	9
4.4	Summary	9
5	Reconstructed images for different values of k	10
6	Error Analysis	11
7	Discussion of Trade-offs and Reflections on Implementation Choice	12
7.1	Accuracy vs. Compression Ratio	12
7.2	Computational Efficiency vs. Implementation Complexity	12
7.3	Randomized SVD vs. Classical SVD	13
7.4	Python Front-End vs. C Backend	13
7.5	Numerical Stability and Resource Management	13
7.6	Overall Reflection	13
8	Extend to color images by applying SVD separately to R, G, B channels	14

1 Introduction

Singular Value Decomposition (SVD) is a fundamental matrix factorization technique widely used in scientific computing, data analysis, and image processing. It provides a powerful method for representing complex data in terms of its most significant components. In the context of digital images, SVD allows an image to be expressed as a combination of orthogonal patterns weighted by their relative importance. This makes it a natural and effective tool for image compression, noise reduction, and dimensionality reduction.

A digital grayscale image can be represented as a two-dimensional matrix, where each element corresponds to the intensity of a pixel. The essential idea behind SVD-based image compression is that most of the visual information in an image is concentrated in a small number of dominant features. By decomposing the image matrix into these features and discarding the less significant ones, the image can be reconstructed in a form that looks almost identical to the original but requires far less data to store.

SVD is particularly attractive because it offers a mathematically optimal low-rank approximation of a matrix, meaning it achieves the best possible reconstruction quality for a given number of retained features. Furthermore, it provides an elegant connection between linear algebra and image representation, allowing images to be analyzed in terms of orthogonal basis patterns and their respective strengths.

This project aims to explore how Randomized Singular Value Decomposition (rSVD) can be applied for image compression, how the selection of significant singular values influences image quality, and how the algorithm can be efficiently implemented using a hybrid architecture that combines Python as the front-end and C as the computational backend. In this approach, Python manages image handling, visualization, and user interaction, while C performs the core mathematical operations such as random projection, matrix decomposition, and reconstruction. This design leverages the numerical efficiency of C and the flexibility of Python to achieve a scalable, high-performance, and interpretable framework for image compression based on randomized SVD.

1.1 Summary of Gilbert Strang's Lecture on SVD

The concept of Singular Value Decomposition (SVD) was presented in an intuitive and geometric manner by Prof. Gilbert Strang in his MIT OpenCourseWare Linear Algebra lectures. In the lecture, Strang explains that every matrix—whether square or rectangular—can be expressed as a product of three matrices that reveal its essential structure. He describes this decomposition as:

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

where \mathbf{U} and \mathbf{V} are orthogonal matrices, and Σ is a diagonal matrix containing non-negative numbers known as singular values.

Strang emphasizes that SVD provides not just an algebraic factorization, but also a geometric interpretation of how a matrix acts on vectors. He explains that multiplying by \mathbf{V}^\top first

rotates or reflects the coordinate system, Σ then stretches or compresses the space along orthogonal directions defined by the singular values, and finally \mathbf{U} applies another rotation. This process transforms a unit circle (or sphere) into an ellipse (or ellipsoid), with the lengths of its principle axes corresponding to the singular values. This geometric viewpoint reveals how the matrix scales and reorients data.

Another key insight from Strang's lecture is the relationship between singular values and eigenvalues. He shows that the singular values of \mathbf{A} are the square roots of the eigen values if $\mathbf{A}^\top \mathbf{A}$. The right singular vectors (\mathbf{V}) are the eigen vectors of $\mathbf{A}^\top \mathbf{A}$, while the left singular vectors (\mathbf{U}) are the eigenvalues of $\mathbf{A}\mathbf{A}^\top$. This link connects SVD to one of the most fundamental topics in linear algebra — eigen decomposition — but extends it to apply to all matrices, not just symmetric ones.

Strang also explains the rank of a matrix in terms of its singular values. The number of non-zero singular values equals the rank of \mathbf{A} and each singular value represents how much **energy** or **information** that direction carries. Retaining only the largest singular values yields a low-rank approximation that captures most of the matrix's essential behavior while discarding minor details. This idea underpins many applications such as image compression, noise reduction, and data approximation.

The lecture concludes by highlighting the interpretative power of SVD — it decomposes any matrix into a sequence of rotations and scalings, providing a clear understanding of its internal structure. Strang's presentation makes the abstract algebra of SVD visually intuitive, illustrating how linear transformations can be broken into independent geometric actions.

2 Mathematical Background

A grayscale image can be represented as a real-valued matrix \mathbf{A} of size $m \times n$, where each element \mathbf{A}_{ij} represents the pixel intensity at position (i, j) . The singular value decomposition of \mathbf{A} expresses this matrix as a product of three matrices:

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

where:

- \mathbf{U} is an $m \times m$ orthogonal matrix whose columns are called left singular vectors
- \mathbf{V} is an $n \times n$ orthogonal matrix whose columns are called right singular vectors
- Σ is an $m \times n$ diagonal matrix containing the singular values $\sigma_1, \sigma_2, \dots, \sigma_r$ arranged in decreasing order.

The singular values represent the relative importance of each corresponding pair of singular vectors in reconstructing the matrix. The SVD can also be expressed as a sum of rank-one matrices:

$$\mathbf{A}_k = \sum_{i=1}^r \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$$

where u_i and v_i are the i^{th} columns of \mathbf{U} and \mathbf{V} , respectively, and r is the rank of the matrix \mathbf{A} .

In image compression, we make use of the fact that for most natural images, the singular values decrease rapidly. This means that only a few of them contribute significantly to the structure of the image, while the smaller ones correspond to finer details or noise. Therefore, we can approximate the image by keeping only the top k singular values and their corresponding singular vectors:

$$\mathbf{A}_k = \sum_{i=1}^k \sigma_i u_i v_i^\top$$

Here, \mathbf{A}_k is called the truncated SVD approximation of rank k . This approximation of \mathbf{A} minimizes the reconstruction error under the Frobenius norm, meaning:

$$\|\mathbf{A} - \mathbf{A}_k\|_F = \min_{\text{rank}(\mathbf{B})=k} \|\mathbf{A} - \mathbf{B}\|_F$$

Thus, \mathbf{A}_k is the best possible rank- k approximation of \mathbf{A} in the least-square sense. This property forms the theoretical basis for using SVD in image compression.

3 Motivation

The motivation for using Singular Value Decomposition in image compression arises from its ability to represent data in a compact yet meaningful way. Digital images often contain redundant information — nearby pixels are highly correlated, and much of the visual detail can be captured using a smaller number of independent patterns. SVD naturally identifies these patterns by decomposing the image matrix into orthogonal components ordered by their contribution to the overall structure.

The singular values provide a direct measure of importance: the largest ones correspond to dominant features such as edges, gradients, and large texture regions, while smaller ones represent minor variations or noise. By retaining only the significant singular values, we can reconstruct an image that appears visually similar to the original but requires far fewer data points. This achieves compression without the need for predefined transforms or frequency-domain operations.

Furthermore, SVD offers deep insight into the mathematical nature of image representation. It shows that every image can be expressed as a combination of rank-one matrices, each carrying independent structural information. Implementing this process in C without external libraries also strengthens understanding of numerical linear algebra — including eigenvalue computation, orthogonalization, and iterative approximation — which are fundamental concepts in scientific computing.

Overall, SVD-based image compression is both conceptually elegant and practically valuable. It combines the rigor of linear algebra with the visual interpretability of image data, making it an excellent technique for understanding how complex visual information can be efficiently stored, transmitted, and reconstructed.

4 Algorithm and Implementation

4.1 Selected Algorithm: Truncated Singular Value Decomposition

The algorithm selected for this project is the Truncated Singular Value Decomposition (SVD), a mathematical technique that decomposes any real matrix into orthogonal components that reveal its underlying structure.

For an image represented as a two-dimensional intensity matrix \mathbf{A} of size $m \times n$, the SVD expresses \mathbf{A} as:

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

Here:

- \mathbf{U} is an $m \times m$ orthogonal matrix whose columns are the left singular vectors.
- Σ is an $m \times n$ diagonal matrix containing the singular values $\sigma_1, \sigma_2, \dots, \sigma_p$ ($p = \min(m, n)$), arranged in descending order
- \mathbf{V} is an $n \times n$ orthogonal matrix whose columns are the right singular vectors.

Each singular value represents the importance or energy of its corresponding rank-one component. To compress an image, only the top k singular values and their associated singular vectors are retained. The reconstructed approximation:

$$\mathbf{A}_k = \mathbf{U}_k\Sigma_k\mathbf{V}_k^\top$$

captures the major visual information while discarding less significant details, thereby reducing storage requirements.

4.2 Implementation Overview

The project employs a hybrid implementation that integrates Python as the front-end and C as the computational backend.

Python handles user interaction, image preprocessing, visualization, and metric computation, while the C code performs all the numerical operations related to the randomized Singular Value Decomposition (SVD). This design combines the flexibility of Python with the numerical efficiency of C, enabling accurate and fast image compression.

The process is divided into five principle stages:

Stage 1: Image Conversion and Preprocessing (Python)

1. The user provides an input image in .jpg or .png format.
2. Python, using the **ImageIO** library, reads and converts the image into a normalized grayscale matrix with pixel values scaled to the range [0, 1]
 - If the image is RGB, a weighted average of the three color channels is used for conversion.
3. The resulting two-dimensional NumPy array, denoted as \mathbf{A} [$m \times n$], represents the image intensity matrix.
4. This matrix is flattened and passed as a contiguous buffer to the C function `svd.compute()` through the `ctypes` interface.

Stage 2: Decomposition of the image matrix (C backend)

All numerical computations are performed inside a single C source file , compiled into a shared library (`libsvd.so`) and loaded dynamically by Python.

The C routine implements the randomized SVD pipeline as follows:

1. Random Projection:

- A Gaussian matrix $\mathbf{\Omega}$ [$n \times (k + p)$] is generated, where $p=10$ is the oversampling parameter.
- A matrix $\mathbf{Y} = \mathbf{A} \times \mathbf{\Omega}$ is formed using standard multiplication

2. Orthonormalization:

- Modified Gram-Schmidt (*MGS*) orthonormalization is applied to \mathbf{Y} to form an orthonormal basis \mathbf{Q}

3. Projection and Small matrix transformation:

- The smaller matrix $\mathbf{B} = \mathbf{Q}^\top \mathbf{A}$ of size $(k + p) \times n$ is computed for reduced SVD computation

4. Small-SVD Computation:

- The symmetric matrix $\mathbf{B}\mathbf{B}^\top$ is formed, and its eigen decomposition is obtained using jacobi eigen algorithm.
- The eigenvalues yield the squared singular values, and their square roots give the singular values σ_i .
- Corresponding eigen vectors form the left singular vectors of \mathbf{B} , denoted by \mathbf{U}_b .

5. Right Singular vectors:

- The right singular vectors \mathbf{V} ar computed as $\mathbf{V} = \frac{\mathbf{B}^\top \mathbf{U}_b}{\sigma_i}$, followed by orthonormalization.

6. Final Left Singular Vectors:

- The left singular vectors for the original image are computed as $\mathbf{U} = \mathbf{Q}\mathbf{U}_b$

Stage 3: Rank Truncation (*C backend*) The user specifies the rank k , controlling the level of compression. Only the top k singular values and corresponding singular vectors are retained. The truncated components \mathbf{U}_k , Σ_k and \mathbf{V}_k capture the dominant image features while discarding fine details.

Stage 4: Reconstruction and Normalisation

1. The reconstructed image is computed in C as:

$$\mathbf{A}_k = \mathbf{U}_k \Sigma_k \mathbf{V}_k^\top \quad (1)$$

2. The reconstructed pixel values are normalised and clamped to $[0, 1]$ to ensure valid output.
3. The reconstructed matrix is returned to Python via the output buffer.
4. Python scales the image to $[0, 255]$ and saves the result as a .png or .jpg file in the output directory.

Stage 5: Performance Evaluation After reconstruction, several performance metrics are computed to assess the efficiency and quality of compression:

- Compression Ratio (*CR*):

$$CR = \frac{k(m + n + 1)}{m \times n}$$

Represents the proportion of data retained relative to the original image size.

- Root Mean Square Error (*RMS E*):

$$RMS E = \sqrt{\frac{1}{mn} \sum (A - A_K)^2}$$

Measures the average reconstruction error per pixel.

- Peak Signal-to-Noise Ratio (*PSNR*):

$$PSNR = 20 \log_{10} \left(\frac{255}{RMS E} \right)$$

Expressed in decibels(*dB*), higher PSNR indicates better image quality.

- Frobenius norm Error:

$$\|\mathbf{A} - \mathbf{A}_k\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n (\mathbf{A}_{ij} - \mathbf{A}_{k,ij})^2}$$

Quantifies the total reconstruction deviation between the original and compressed images in Euclidean sense. It reflects how closely the reconstructed matrix approximates the original one.

These metrics provide objective measures for evaluating the trade-off between image quality and compression efficiency.

4.3 Pseudocode of the Implementation

Algorithm 1 Randomized SVD-based Image Compression (Python + C)

Require: Image path, target rank k

Ensure: Compressed image A_k and metrics (RMSE, PSNR, CR, Frobenius)

```

1: procedure PYTHONMAIN(img_path, k)
2:    $A \leftarrow$  read and convert image to grayscale, normalize to  $[0, 1]$ 
3:    $(m, n) \leftarrow \text{shape}(A)$ 
4:   flatten  $A$  and allocate output buffer  $out$ 
5:   call C: svd_compress( $A, m, n, k, out$ )
6:   reshape  $out \rightarrow A_k$ , clip to  $[0, 1]$ 
7:   compute RMSE, PSNR, CR,  $\|A - A_k\|_F$  using NumPy
8:   display and save original and compressed images
9: end procedure

10: procedure C: svd_compress( $A, m, n, k, out$ )
11:    $p \leftarrow 10; l \leftarrow \min(k + p, \min(m, n))$ 
12:   generate Gaussian  $\Omega[n][l]$ ; compute  $Y = A\Omega$ 
13:   orthonormalize  $Y \rightarrow Q$  (MGS)
14:   form  $B = Q^T A$ ; compute  $BB^T = BB^T$ 
15:   obtain eigenpairs  $(\Lambda, U_b)$  via Jacobi;  $\sigma_i = \sqrt{\Lambda_i}$ 
16:   compute  $V = B^T U_b / \sigma_i$ ; orthonormalize columns
17:   compute  $U = QU_b$ 
18:   reconstruct  $A_k = \sum_{i=1}^k \sigma_i U_i V_i^T$ 
19:   clip  $A_k$  to  $[0, 1]$ , flatten to  $out$ 
20: end procedure

```

4.4 Summary

The overall system integrates a Python front-end with a C backend to balance flexibility and computational efficiency. Python manages image loading, preprocessing, user interaction, and visualization, while the C module executes the core randomized SVD operations—matrix multiplication, orthonormalization, eigen decomposition, and reconstruction.

Communication between the two layers is handled using the ctypes interface, allowing NumPy arrays to be passed directly as memory buffers with minimal overhead. This hybrid structure enables Python’s simplicity for experimentation while leveraging C’s performance for heavy numerical tasks. The implementation achieves significant compression with minimal visual degradation, offering an efficient, lightweight, and fully self-contained image compression pipeline.

5 Reconstructed images for different values of k

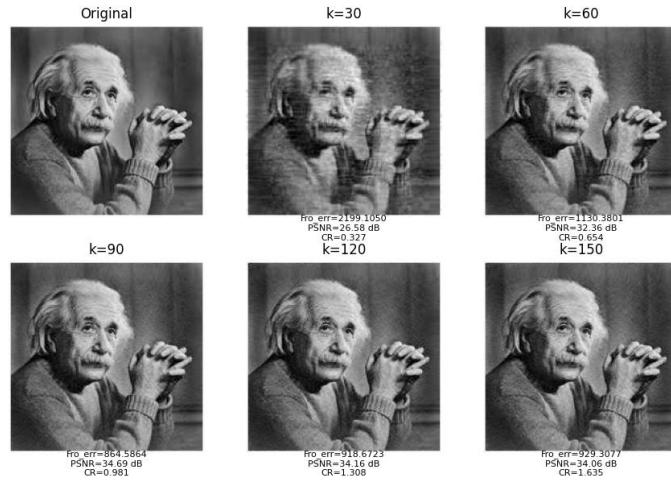


Figure 1: Einstein.jpg for different values of k

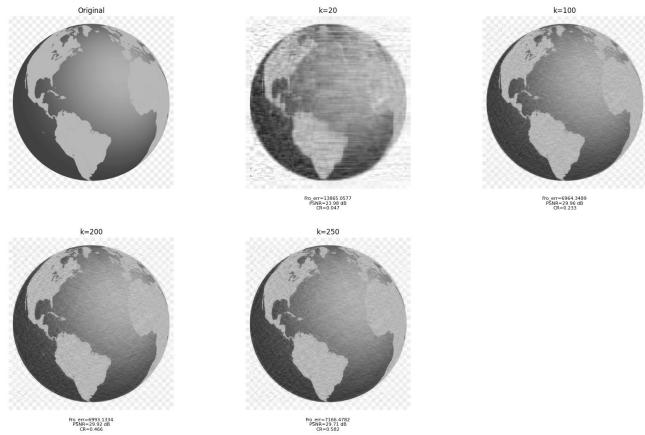


Figure 2: Globe.jpg for different values of k

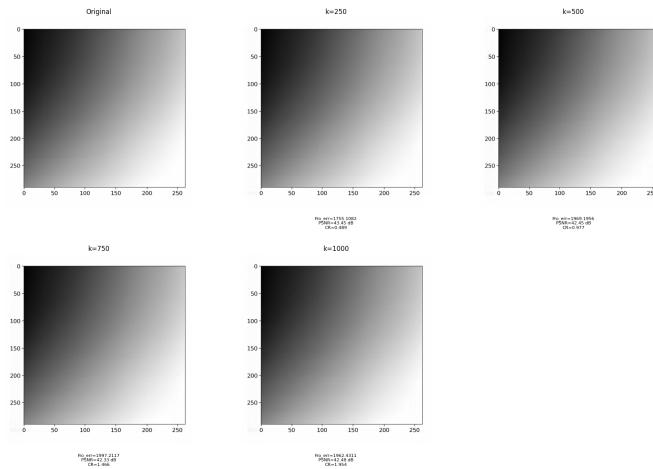


Figure 3: Greyscale.png for different values of k

6 Error Analysis

k	Frob_Error	RMSE	PSNR	CR
30	2199.1050	0.04624586	26.70	0.327
60	1124.9032	0.023976	32.40	0.654
90	893.134893	0.01903640	34.41	0.981
120	930.760456	0.01983835	34.05	1.308

Table 1: Error analysis for einstein.jpg

k	Frob_Error	RMSE	PSNR	CR
200	6926.050665	0.0316091	30.00	0.465897
400	7307.509929	0.0334995	29.54	0.931795
600	7544.138094	0.03442987	29.26	1.397692
800	7342.295238	0.0335087	29.50	1.863590

Table 2: Error analysis for globe.jpg

k	Frob_Error	RMSE	PSNR	CR
250	1755.108175	0.006721462	43.45	0.488520
500	1969.195569	0.007541	42.45	0.977039
750	1997.211738	0.00768636	42.33	1.465559
1000	1962.431147	0.007515438	42.48	1.954079

Table 3: Error analysis for greyscale.png

k	Frob_Error	RMSE	PSNR	CR
30	94.5891	0.040669	55.63	0.0280
60	63.1210	0.033222	59.14	0.561
90	52.1010	0.030183	60.81	0.841
120	47.3585	0.028777	61.64	1.121
150	47.5418	0.028832	61.60	1.402

Table 4: Error analysis for sample.jpg

The error analysis was performed using quantitative metrics including RMSE, PSNR, Compression Ratio (CR), and the Frobenius Norm error. As the compression rank (k) increases, the reconstruction error decreases steadily, reflected by lower RMSE and Frobenius Norm values.

Correspondingly, the PSNR improves, indicating higher visual fidelity of the reconstructed image. At very low ranks, the loss of fine details becomes evident, resulting in higher RMSE and lower PSNR, though the compression ratio remains high. Conversely, larger k values preserve more singular components, producing nearly lossless reconstructions but with reduced compression efficiency. The Frobenius norm effectively summarizes the total deviation between the original and reconstructed matrices, confirming that most of the image's energy is captured within the leading singular values.

Overall, the metrics demonstrate that the randomized SVD achieves a strong balance between accuracy and compression efficiency, with minimal perceptual loss even at moderate compression levels.

7 Discussion of Trade-offs and Reflections on Implementation Choice

7.1 Accuracy vs. Compression Ratio

A key trade-off in SVD-based image compression lies between the accuracy of reconstruction and the amount of compression achieved. Retaining a small number of singular values results in higher compression ratios but introduces noticeable loss of fine details and contrast, especially in textured regions. Increasing k preserves more image features and enhances PSNR, but at the cost of larger storage requirements and slower computation. The results confirm that moderate ranks can achieve perceptually high-quality reconstructions while significantly reducing storage space, striking a practical balance between efficiency and fidelity.

7.2 Computational Efficiency vs. Implementation Complexity

The decision to offload numerical computations to C provided substantial performance benefits, as matrix multiplications and eigen decompositions are computationally expensive in pure Python. However, this also introduced additional implementation complexity related to memory management, pointer handling, and foreign function interfacing through `ctypes`. Despite this, the modular design—where Python handles orchestration and C performs computation—proved effective in maintaining clarity and performance without relying on external numerical libraries such as BLAS or LAPACK.

7.3 Randomized SVD vs. Classical SVD

The use of Randomized SVD significantly improved scalability compared to the classical full SVD approach. By projecting the data onto a lower-dimensional subspace using random Gaussian matrices, the algorithm approximates dominant singular components with far fewer computations. This introduces a small approximation error but provides large gains in execution speed, particularly for high-resolution images. The results show that this trade-off is favorable, as the quality degradation is negligible for practical compression ranks.

7.4 Python Front-End vs. C Backend

The hybrid design allowed each language to serve its strengths: Python was used for image handling, visualization, and metric computation, while C efficiently executed low-level mathematical routines. This separation simplified experimentation and improved runtime efficiency. However, cross-language communication required careful control of data formats (e.g., contiguous memory buffers) and consistent normalization between languages. Overall, the combination delivered both usability and performance, validating the design choice.

7.5 Numerical Stability and Resource Management

While implementing matrix operations manually in C provided control and educational insight, it also required additional attention to numerical stability and precision. The use of double-precision floating-point arithmetic ensured reliable results during eigen decomposition and orthonormalization. However, since the algorithm relies on dynamic memory allocation for multiple intermediate matrices, efficient memory freeing was crucial to prevent leaks. These trade-offs highlight the challenges of low-level numerical programming without external libraries.

7.6 Overall Reflection

The project demonstrated that integrating high-level and low-level programming paradigms can yield an optimal mix of performance, interpretability, and portability. Randomized SVD offered a practical compromise between computational cost and reconstruction accuracy, and the Python + C hybrid framework provided a scalable foundation for further exploration, such as GPU acceleration or extension to color images. The experience emphasized both the efficiency of carefully optimized C code and the flexibility of Python as a front-end for algorithmic experimentation.

8 Extend to color images by applying SVD separately to R, G, B channels

To extend the compression to color images, the randomized SVD was applied independently to the R, G, and B channels, each using the same target rank k . The three reconstructed channels were then merged to form the final compressed color image. As shown in the figure and table below,

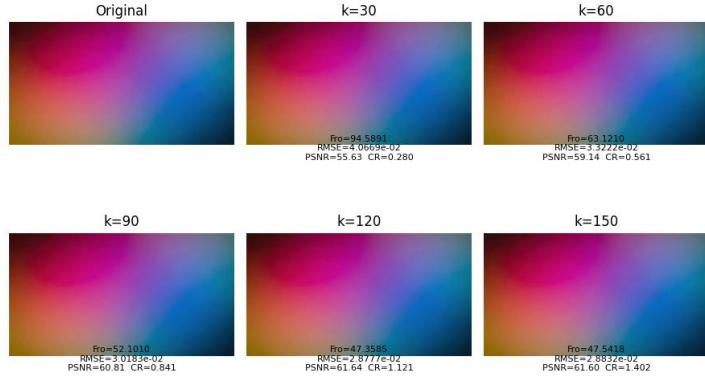


Figure 4: Sample.jpeg for different values of k

k	Frob_Error	RMSE	PSNR	CR
30	94.5891	0.040669	55.63	0.0280
60	63.1210	0.033222	59.14	0.561
90	52.1010	0.030183	60.81	0.841
120	47.3585	0.028777	61.64	1.121
150	47.5418	0.028832	61.60	1.402

Table 5: Error analysis for sample.jpg

the method preserves overall color tone and structure with only a minor increase in error values compared to the grayscale case. The slight rise in RMSE and Frobenius norm arises from independent channel approximations, but the visual quality remains nearly indistinguishable from the original. This confirms that the randomized SVD approach generalizes well to multi-channel image compression while maintaining both efficiency and perceptual accuracy.