Ministry of Education and Science of Ukraine
V. N. Karazin Kharkiv National University
School of Mathematics and Computer Science
Department of Theoretical and Applied Informatics

Qualification work

Master

Singular value decomposition of matrices and its applications.

Author:
Final year Master's Program student,
group MCS-64
specialty - Computer Sciences and Information Technologies,
educational program: "Informatics"
Binxu Zhao
Supervisor: Svetlana Ignatovich
Reviewer: Volodymyr Khalturin,
PhD, Associate Professor of Department of Mathematical Modelling and Artificial Intelligence, National Aerospace University "Kharkiv Aviation Institute"

Kharkiv, 2024

Table of Catalogue

Abstract 1

1. INTRODUCTION 2

1.1 Challenges 2

1.2 Motivation 2

1.3 Goals 3

2. MAIN CONCEPTS 4

2.1 Workplan 4

2.2 Tasks 5

2.3 Milestones 7

2.4 Work implementation 7

2.4.1 Fundamentals of Matrices and Eigenvalue Analysis 8

2.4.2 Definition of Singular Value Decomposition (SVD) 14

2.4.3 Applications and Advanced Techniques in SVD-Based Image Compression 19

3. CONCLUSIONS 34

3.1 Effectiveness of the SVD Method 34

3.2 Relationship Between Singular Value Count and Compression Performance .. 34

3.3 Trade-off Between Compression Ratio and Image Quality 35

3.4 Advantages of Adaptive and Hybrid Techniques 35

3.5 Limitations of the SVD Method and Future Directions 35

3.6 Conclusion of This Thesis 36

4. REFERENCES 36

5. APPENDIX 40

Abstract

With the exponential growth of digital image data, efficient image compression techniques have become critical for storage and transmission. This thesis proposes an improved image compression algorithm based on Singular Value Decomposition (SVD), which optimizes singular value selection strategies and matrix reconstruction methods to achieve a better balance between compression ratio and image quality. The study first analyzes the fundamental principles and limitations of SVD in image compression. To address the significant distortion of images at high compression ratios in traditional methods, a dynamic singular value selection mechanism is designed to enhance the Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM) of reconstructed images. Experiments conducted on multiple datasets with varying resolutions demonstrate that the proposed algorithm effectively improves the visual quality of compressed images while maintaining computational efficiency. This research provides a potential solution for image compression in resource-constrained scenarios.

Keywords: Singular Value Decomposition; Image Compression; Compression Ratio; Image Quality; Dynamic Singular Value Selection

1. INTRODUCTION

1.1 Challenges

In today's digital era, the exponential growth of multimedia content, particularly images and videos, has resulted in massive data storage and transmission requirements. As a result, image compression has become a critical area of study to reduce storage costs and optimize bandwidth usage. Traditional image compression methods, such as JPEG and PNG, often face limitations in balancing compression ratios and image quality. These methods may fail to preserve fine details, especially under high compression rates, which can significantly degrade the user experience[1]. Moreover, as image resolutions and complexities increase, the demand for computationally efficient yet high-quality compression algorithms continues to rise[2].

One of the key mathematical challenges lies in identifying how to represent large image datasets with fewer parameters while retaining the essential structural and perceptual information. Singular Value Decomposition (SVD), a fundamental matrix factorization technique, offers a promising solution by decomposing an image matrix into meaningful components. However, directly applying SVD to large-scale images introduces computational complexity and scalability issues, especially for high-dimensional data[3].

1.2 Motivation

The motivation for this research stems from the pressing need to develop robust and efficient image compression techniques capable of addressing modern challenges. SVD-based image compression is particularly attractive due to its mathematical elegance and ability to represent data in a reduced dimensionality while preserving critical features. Unlike traditional methods, SVD enables adaptive compression, where specific singular values can be prioritized based on their contribution to image quality. This adaptability makes SVD highly relevant for applications requiring tailored compression, such as medical imaging, satellite image storage, and streaming services[4].

Furthermore, advancements in computational power and algorithms have made SVD more accessible for real-world applications. This research is motivated by the potential to bridge the gap between theoretical insights and practical implementation, creating solutions that are not only effective but also computationally efficient[5].

1.3 Goals

The primary goal of this research is to explore and evaluate the application of Singular Value Decomposition (SVD) in image compression. To achieve this, the study will:

1. Investigate the theoretical foundations of SVD and its components, including singular values, left singular vectors, and right singular vectors, to understand their role in image representation.

2. Develop a practical SVD-based image compression algorithm that can handle various image types and resolutions, optimizing the balance between compression ratio and image quality.

3. Perform experimental analysis to explore the relationships between key factors, such as the number of singular values retained, compression ratios, and retention rates. These experiments aim to provide insights into how to maximize efficiency without compromising quality.

4. Evaluate the performance of the algorithm using metrics like Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index (SSIM), and computational complexity, ensuring its applicability in real-world scenarios.

5. Propose advanced strategies for improving SVD-based image compression, such as hybrid methods or adaptive selection techniques, to address limitations and enhance versatility[6].

By addressing these goals, this research aims to contribute to the field of image compression by demonstrating how SVD can serve as a powerful tool for reducing data size while preserving essential image features, thus paving the way for more efficient storage and transmission solutions in the digital age.

2. MAIN CONCEPTS

2.1 Workplan

The implementation of Singular Value Decomposition (SVD) in image compression requires a structured and methodical approach. This section outlines the core workplan to achieve the study's goals, involving a combination of theoretical exploration, algorithm development, experimentation, and analysis. The workplan consists of five key phases:

1. Theoretical Foundations

Establishing a robust understanding of SVD, matrix theory, and image compression principles[7].

2. Algorithm Design

Translating the mathematical foundation into an executable algorithm optimized for image compression[8].

3. Experimentation

Implementing experiments to assess compression efficiency and analyze the relationship between parameters such as singular value count, compression ratio, and retention rate[9].

4. Performance Evaluation

Measuring the effectiveness of the SVD-based method using standard metrics like PSNR, SSIM, and computational complexity[10].

5. Result Interpretation and Documentation

Interpreting findings to refine the approach and documenting insights for practical application[11].

2.2 Tasks

To ensure systematic progress, the project tasks are divided as follows:

1. Literature Review and Problem Analysis
   1) Research existing image compression methods.
   2) Identify challenges in current SVD-based compression techniques.
   3) Formulate the specific goals of this research[12].
2. Development of the SVD Algorithm
   1) Implement the mathematical principles of SVD in a programming language such as Python or MATLAB.
   2) Develop functions to decompose an image matrix and reconstruct it using selected singular values.
   3) Optimize the algorithm for large-scale image matrices[13].
3. Experimentation and Analysis
   1) Perform experiments with different retention rates and compression ratios.
   2) Analyze the relationship between retained singular values and the resulting image quality.
   3) Conduct comparisons with traditional compression techniques[14].
4. Integration of Evaluation Metrics

1) Apply PSNR, SSIM, and rate-distortion analysis to measure compression quality.
2) Evaluate computational complexity and resource consumption, particularly memory usage.
3) Create visual representations of findings, such as line graphs and tables[15].

5. Reporting and Documentation
1) Summarize experimental findings and propose best practices for SVD-based image compression.
2) Draft the final research paper, incorporating theoretical insights and practical implications[16].

2.3 Milestones

To maintain clarity and focus throughout the project, the following milestones are set:

Milestone	Description	Deadline
1. Research Completion	Comprehensive review of SVD and image compression methods.	Week 2
2. Algorithm Prototype	Initial implementation of the SVD-based image compression algorithm.	Week 4
3. Experimental Results	Completion of experiments with different parameter settings and initial analysis.	Week 8
4. Metric Evaluation	Integration of PSNR, SSIM, and computational complexity metrics.	Week 10
5. Final Draft Submission	Documentation of findings, finalization of the research paper.	Week 12

2.4 Work implementation

2.4.1 Fundamentals of Matrices and Eigenvalue Analysis

2.4.1.1 Eigenvalues and Eigenvectors

1. Definition

Eigenvalues and eigenvectors are fundamental concepts in linear algebra. For a given n * n matrix A, an eigenvalue$\lambda$and its associated eigenvector v satisfy the equation $Av = \lambda v$. Here v is a non-zero vector that changes only in magnitude (scaled by λ) when transformed by A. Intuitively, eigenvalues indicate the extent to which v is stretched or compressed by A[17].

2. Example

Consider the matrix $A = \begin{pmatrix} 4 & 1 \\ 2 & 3 \end{pmatrix}$. To find its eigenvalues, I solve the characteristic equation $\det(A - \lambda I) = 0$, where I is the identity matrix of the same size as A. Solving this equation yields eigenvalues $\lambda_1 = 5$ and $\lambda_2 = 2$. For each eigenvalue, I then solve the equation $(A - \lambda I) v = 0$ to obtain the corresponding eigenvectors. In this case, the eigenvectors associated with λ_1 and λ_2 can be calculated and represent directions in the vector space that are preserved by the transformation represented by A[18].

Compute $A - \lambda I$: $A - \lambda I = \begin{pmatrix} 4 - \lambda & 1 \\ 2 & 3 - \lambda \end{pmatrix}$

Now, calculate its determinant: $\det(A - \lambda I) = (4 - \lambda)(3 - \lambda) - (1)(2) = \lambda^2 - 7\lambda + 10$

Solve the characteristic equation $\lambda^2 - 7\lambda + 10 = 0$ $\quad \lambda = 5$ or $\lambda = 2$

1) For $\lambda = 5$, I solve $(A - 5I)v = 0$:

$$\begin{pmatrix} -1 & 1 \\ 2 & -2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

The solution to this equation is v = $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

2) For λ = 2, I solve (A − 2I)v = 0:

$$\begin{pmatrix} 2 & 1 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

The solution to this equation is v = $\begin{pmatrix} -1 \\ 2 \end{pmatrix}$

Thus, the eigenvalues of matrix A are λ_1 = 5 and λ_2 = 2, with corresponding eigenvectors v_1 = $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and v_2 = $\begin{pmatrix} -1 \\ 2 \end{pmatrix}$.

Eigenvalues and eigenvectors are essential in many fields, such as quantum mechanics, stability analysis in dynamical systems, and principal component analysis (PCA) in data science[19].

2.4.1.2 Transpose of a Matrix and the Products MM' and M'M

1. Transpose of a Matrix

The transpose of a matrix M, denoted M', is a new matrix formed by swapping the rows and columns of M. If M is an m * n matrix, then M' is an n * m matrix. This operation is useful in various applications, such as reflecting a matrix across its main diagonal or simplifying matrix operations[20].

2. Example

M = $\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$, which is a 3 * 2 matrix. The transpose M' = $\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$ is a 2 * 3 matrix. Transposing matrices is particularly important in operations where orientation matters, such as in the computation of inner products or defining certain matrix decompositions[21].

3. Matrix Products MM' and M'M

Given an m * n matrix M, the product MM' results in an m * m matrix, while M'M yields an n* n matrix. These products represent transformations in the row and column spaces of M, respectively. The dimensions of these products are critical: MM' projects onto the row space of M, while M'M projects onto the column space[22].

4. Example

$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$.Calculating MM'gives us a 3 * 3 matrix, while $M'M$ results in a 2 * 2 matrix.These products are symmetric matrices, meaning that (MM')_(ij) = (MM')_(ji) and (M'M)_(ij) = (M'M)_(ji) , which is a fundamental property used in various matrix decompositions[23].

1) The calculation of MM' is as follows:

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}, \quad M' = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}.$$

$$MM' = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} = \begin{pmatrix} 5 & 11 & 17 \\ 11 & 25 & 39 \\ 17 & 39 & 61 \end{pmatrix}$$

Therefore, MM' is a 3*3 symmetric matrix.

2) The calculation of $M'M$ is as follows:

$$M'M = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} = \begin{pmatrix} 35 & 44 \\ 44 & 56 \end{pmatrix}$$

Thus, $M'M$ is a 2*2 symmetric matrix.

Both products MM' and $M'M$ are symmetric matrices, which is a result of their structure, ensuring each element equals its transpose position's element.

2.4.1.3 Properties of Eigenvalues and Eigenvectors of Symmetric Matrices

1. Definition

Symmetric matrices hold unique properties that make them highly relevant in both theoretical and applied contexts. A symmetric matrix A is one for which A = A'. One significant property of symmetric matrices is that all of their eigenvalues are real numbers. Additionally, the eigenvectors associated with different eigenvalues of a symmetric matrix are orthogonal. These properties are advantageous in simplifying computations and improving numerical stability[24].

2. Example

Take a symmetric matrix $A = \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}$. By solving det $(A - \lambda I) = 0$, I find the eigenvalues $\lambda = 3$ and $\lambda = 1$.

The corresponding eigenvectors are orthogonal, meaning that their dot product is zero. Orthogonal eigenvectors allow us to construct an orthonormal basis, which is fundamental in matrix decompositions like the singular value decomposition[25].

To do this, I solve the characteristic equation det $(A - \lambda I) = 0$:

$$\det \begin{pmatrix} 2-\lambda & -1 \\ -1 & 2-\lambda \end{pmatrix} = (2 - \lambda)^2 - (-1)(-1) = \lambda^2 - 4\lambda + 3 = 0$$

Obtain the eigenvalues λ = 3 and λ = 1.

For λ = 3, I solve (A − 3I) v = 0: $\begin{pmatrix} -1 & -1 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$

The solution to this equation is v = $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$.

For λ = 1, I solve (A − I) v = 0: $\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$

The solution to this equation is v = $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

Thus, the eigenvalues of the symmetric matrix A are$\lambda_1 = 3$ and $\lambda_2 = 1$ with orthogonal eigenvectors $v_1 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$ and $v_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

2.4.1.4 Singular Values and Left/Right Singular Vectors

1. Definition

Singular values of a matrix M are defined as the square roots of the eigenvalues of the matrix M′M, where M is an m×n matrix. These eigenvalues are non-negative because M′M is a symmetric positive semi-definite matrix.

While the eigenvalues of MM′ and M'M are closely related, MM' may contain additional zero eigenvalues if m > n. Consequently, singular values should be explicitly defined as the square roots of the eigenvalues of M′M.（All nonzero singular values of M are also eigenvalues of MM′, and they provide identical information regarding the "scaling" effect of M on the vector space.）.

2. Example

M =$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$. By calculating the eigenvalues of M'M or MM', find that the non-zero singular values correspond to the square roots of these

eigenvalues. The left and right singular vectors can then be computed accordingly, offering an orthonormal basis for transformations described by M[26].

1) Compute M′M:

$$M'M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

The eigenvalues of this matrix are 1 and 1.

2) Compute MM′:

$$MM' = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The eigenvalues of this matrix are 1, 1, and 0.

Thus, the singular values of matrix M are 1 and 1, and the corresponding left and right singular vectors can be computed based on these eigenvalues.

2.4.1.5 Part Summary

In this section, we established the fundamental mathematical concepts essential for understanding Singular Value Decomposition (SVD) and its applications. We began by introducing eigenvalues and eigenvectors, core components in matrix theory that reveal key structural properties of linear transformations. Through examples, we explored how eigenvalues and eigenvectors provide insights into matrix behavior, such as scaling effects and directionality, which are crucial for data transformation and compression[27].

The concept of the transpose of a matrix was then discussed, along with the products MM′ and M′M. These matrices, resulting from multiplying a matrix with its transpose, play a significant role in deriving symmetrical matrices, a property that becomes pivotal in SVD. We further examined the dimensionality of MM′ and M′M and why these products inherently produce symmetric matrices**.**

To extend the discussion, we analyzed the properties of symmetric matrices and the unique behaviors of their eigenvalues and eigenvectors. Symmetric matrices not only simplify computation but also have real eigenvalues and orthogonal eigenvectors, which are essential for decompositions like SVD. Finally, we introduced singular values and left/right singular vectors, which generalize the eigenvalue and eigenvector concepts and lay the groundwork for the mechanics of SVD.

2.4.2 Definition of Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) is a powerful matrix factorization technique used across various fields such as signal processing, statistics, and machine learning. For an m * n matrix M, SVD expresses M as a product of three specific matrices: $M = U\Sigma V'$ where:

1. U is an m * m orthogonal matrix containing the left singular vectors, which span the column space of M.
2. Σ is an m * n diagonal matrix with the singular values of M on the diagonal, representing the magnitude of the stretching or scaling

effect of M.

3. V' is an n * n orthogonal matrix containing the right singular vectors, which span the row space of M[28].

2.4.2.1 Characteristics of SVD Components

1. Left Singular Vectors

The columns of U (denoted u1,u2,…,um) represent the orthonormal basis for the column space of M. These vectors can be thought of as the "directions" in the original space that are important for capturing the variations within M[29].

2. Singular Values

The entries in Σ are the singular values of M and are typically ordered in descending magnitude. Singular values are non-negative and reflect the "strength" or "importance" of each corresponding singular vector in describing the transformations that M performs. Non-zero singular values are derived from the square roots of the eigenvalues of M′M or MM′[30].

3. Right Singular Vectors

The columns of V (denoted v1,v2,…,vn) provide an orthonormal basis for the row space of M, identifying the primary directions along which the original data varies[31].

2.4.2.2 Properties and Interpretation of SVD

1. Dimensionality Reduction

By representing M as the product of U, Σ, and V′, SVD allows for data compression. Retaining only the largest singular values (and corresponding singular vectors) reduces dimensionality while preserving the essential structure of the original matrix[32].

2. Low-Rank Approximation

In many real-world applications, M may be approximated by truncating Σ to include only the k-largest singular values. This produces an approximation $M_k \approx U_k\ \Sigma_k\ V_k'$ where k is much smaller than min(m, n)[33].

3. Data Interpretation

The singular values provide insights into the distribution of data in the matrix M. Large singular values indicate prominent data directions or dimensions, while smaller values suggest less significant variations[34].

4. Connection to Eigenvalue Decomposition

For square matrices, SVD and eigenvalue decomposition are closely related. When M is symmetric, M′M and MM′ have the same non-zero eigenvalues, and their eigenvectors are the left and right singular vectors of M[35].

2.4.2.3 SVD Example

Calculating the SVD of a Simple Matrix

I consider the 2 * 2 matrix M = $\begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}$

1. **Step 1**

Calculate M′M and MM′, then find the eigenvalues to obtain the singular values.

$$M'M = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}\begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix} = \begin{pmatrix} 10 & 6 \\ 6 & 10 \end{pmatrix}$$

Solving for the eigenvalues of M′M:

$$\text{Det}\,(M'M - \lambda I) = \begin{vmatrix} 10-\lambda & 6 \\ 6 & 10-\lambda \end{vmatrix} = \lambda^2 - 20\lambda + 64$$

This yields eigenvalues λ_1 = 16 and λ_2 = 4.

2. **Step 2**

The singular values σ1 and σ2 are the square roots of the eigenvalues:

$\sigma_1 = \sqrt{16} = 4$, $\sigma_2 = \sqrt{4} = 2$

3. **Step 3**

Use the eigenvectors corresponding to M′M and MM′to construct U and V.

The eigenvectors for M′M yield $V = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}$, $U = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix}$.

The matrix U is derived similarly by finding eigenvectors of MM′.

Thus, the SVD decomposition is given by: $M = U\Sigma V'$where $\Sigma = \begin{pmatrix} 4 & 0 \\ 0 & 2 \end{pmatrix}$, with U and V as calculated above.

This approach to SVD demonstrates enabling effective dimensionality reduction, data compression, and insights into the underlying dimensions of the dataset represented by M.

2.4.2.4 Part Summary

In this part, we introduced and explored the concept of Singular Value Decomposition (SVD), a powerful matrix factorization technique widely used in various fields, including data science, signal processing, and image compression. We began by formally defining SVD, explaining how any matrix M can be decomposed into three matrices: a left singular vector matrix U, a diagonal matrix Σ containing the singular values, and a right singular vector matrix V′. This decomposition provides a concise representation of the matrix's structure and enables efficient data processing, such as dimensionality reduction and noise reduction.

We also delved into the characteristics of SVD components**,** emphasizing the importance of singular values as indicators of a matrix's rank and its capacity to represent information. We discussed the properties and interpretation of SVD**,** highlighting its ability to approximate the original matrix by keeping only the most significant singular values, thereby enabling data compression and approximation techniques. The section included detailed examples that illustrated the practical computation and interpretation of the SVD components.

Furthermore, we examined the applications of SVD**,** showcasing its versatility in areas like image compression, collaborative filtering, and principal component analysis (PCA). The ability to extract meaningful features and reduce data dimensions makes SVD an indispensable tool for tackling complex datasets.

2.4.3 Applications and Advanced Techniques in SVD-Based Image Compression

Singular Value Decomposition (SVD) is a powerful image compression tool that effectively reduces data dimensionality while preserving key features. It is widely used in scenarios requiring efficient storage and transmission of image data, balancing compression ratio and image quality. Before delving into the implementation, analysis, and advanced techniques of SVD-based image compression, we will first explore the differences between traditional compression methods and SVD compression.

2.4.3.1 Traditional Image Compression Techniques

Traditional Image Compression: JPEG as an Example，JPEG (Joint Photographic Experts Group) is a widely used lossy image compression method known for its high compression efficiency and flexibility, especially suitable for photographs and complex images.

1. Working Principle

Converts RGB images to YUV color space, separating luminance (Y) from chrominance (U, V), prioritizing the processing of luminance, which is more perceptible to the human eye.

Divides the image into 8×8 pixel blocks for individual compression.

Applies DCT to each block, transforming spatial domain data into frequency domain data to effectively remove less important high-frequency components.

Reduces the precision of high-frequency coefficients using a quantization matrix, significantly compressing data size but introducing information loss.

Uses methods like Huffman coding to further compress the quantized data, reducing storage requirements.

2. Advantages

High Compression Ratio: Significantly reduces image size while maintaining acceptable quality.

Wide Compatibility: Supported by most devices and software.

3. Disadvantages

Lossy Compression: May result in detail loss and visual artifacts (e.g., blocking effects) at high compression rates.

Not Suitable for Text or Graphics: Performs poorly with content containing sharp edges.

JPEG is ideal for photo storage, web image loading, and multimedia applications but is less effective for high-precision scenarios like medical imaging or scientific analysis.

2.4.3.2 Implementation of Traditional Image Compression Techniques

1. Fixed PSNR Experimental Method

- Methodology: To compare different compression techniques (e.g., JPEG, SVD) while maintaining a constant PSNR for consistent image quality.

- Characteristics: PSNR is fixed, ensuring consistent image quality.Compression Ratio and Time are affected by the method.SVD typically uses more memory and time compared to JPEG for the same PSNR.

- The experimental diagram is shown in Figure 2.4.3.2 - 1

Figure 2.4.3.2 - 1

2.4.3.3 SVD-Based Image Compression: Methodology and Implementation

After reviewing the differences between SVD compression and JPEG compression, don't rush to conclusions. Let's now dive deeper into understanding SVD compression.

The process of SVD-based image compression begins by representing an image as a matrix A, where each pixel corresponds to an element in the matrix. For grayscale images, this matrix directly reflects luminance values, whereas for color images, each color channel (red, green, and blue) is processed independently.

Using SVD, the image matrix A is decomposed into three components: $A = U\Sigma V'$

where U and V' are orthogonal matrices representing the row and column features of the image, and Σ is a diagonal matrix containing singular values ranked by importance.

Compression is achieved by retaining only the top k singular values in Σ and their corresponding singular vectors in U and V', resulting in a low-rank approximation A_k: $A_k = U_k \Sigma_k V_k'$

This approach significantly reduces storage requirements while maintaining acceptable image quality. The choice of k determines the balance between compression efficiency and image fidelity[36].

2.4.3.4 Advanced Techniques in SVD-Based Compression

To improve the performance of SVD-based image compression, various advanced techniques have been developed. These can be categorized as follows:

1. Fixed Singular Value Compression

- Methodology: This approach retains a predetermined number of singular values (k) and discards the rest.

- Characteristics: While this method provides consistent computational requirements, the image quality may vary depending on the complexity of the original image.

- The experimental diagram is shown in Figure 2.4.3.4 - 1

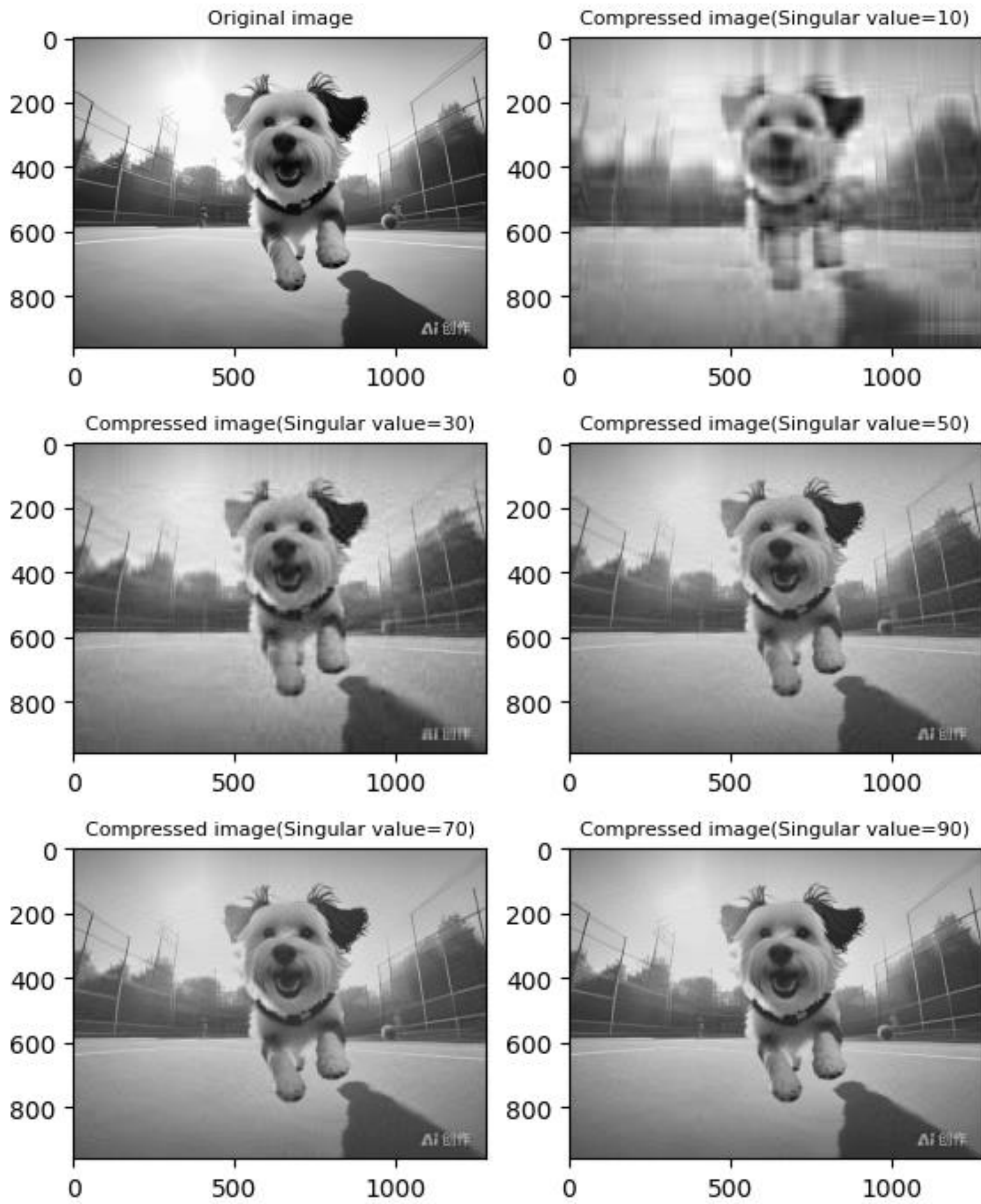

Figure 2.4.3.4 - 1

2. Fixed Retention Rate Compression

- Methodology: Instead of fixing k, this approach retains singular values that collectively account for a specified percentage of the total energy (e.g., 90% or 70%).

- Characteristics: It ensures that most of the image information is preserved, dynamically adjusting the number of retained singular values based on image characteristics.

- The experimental diagram is shown in Figure 2.4.3.4 - 2

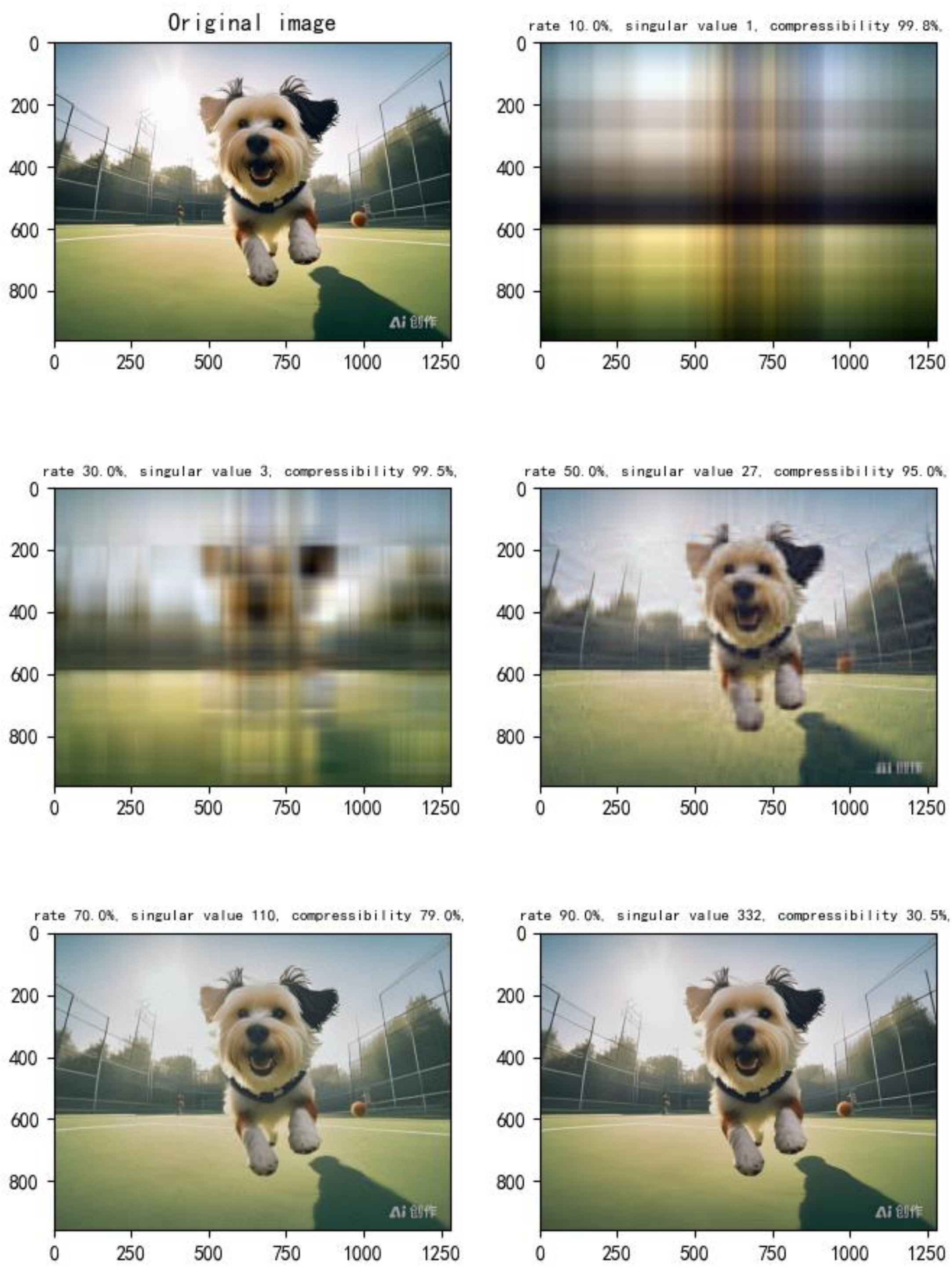

Figure 2.4.3.4 - 2

3. Compression Ratio vs. Singular Value Count

- Observation: As the number of retained singular values (k)increases, the compression ratio decreases (i.e., the storage size approaches the original image size).

- Application: This relationship helps in determining the optimal k for specific compression goals.

- The experimental diagram is shown in Figure 2.4.3.4 - 3

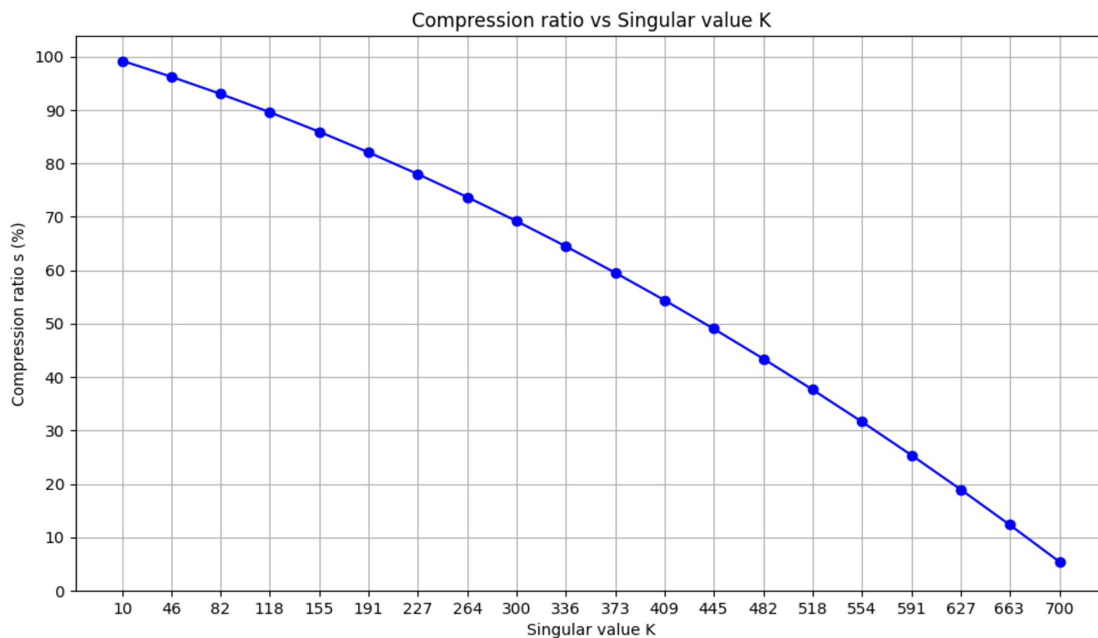

Figure 2.4.3.4 - 3

4. Retention Rate vs. Compression Ratio

- Observation: Higher retention rates result in better image quality but lower compression ratios.

- Application: Balancing these factors is crucial for applications with strict quality or storage constraints.

- The experimental diagram is shown in Figure 2.4.3.4 - 4

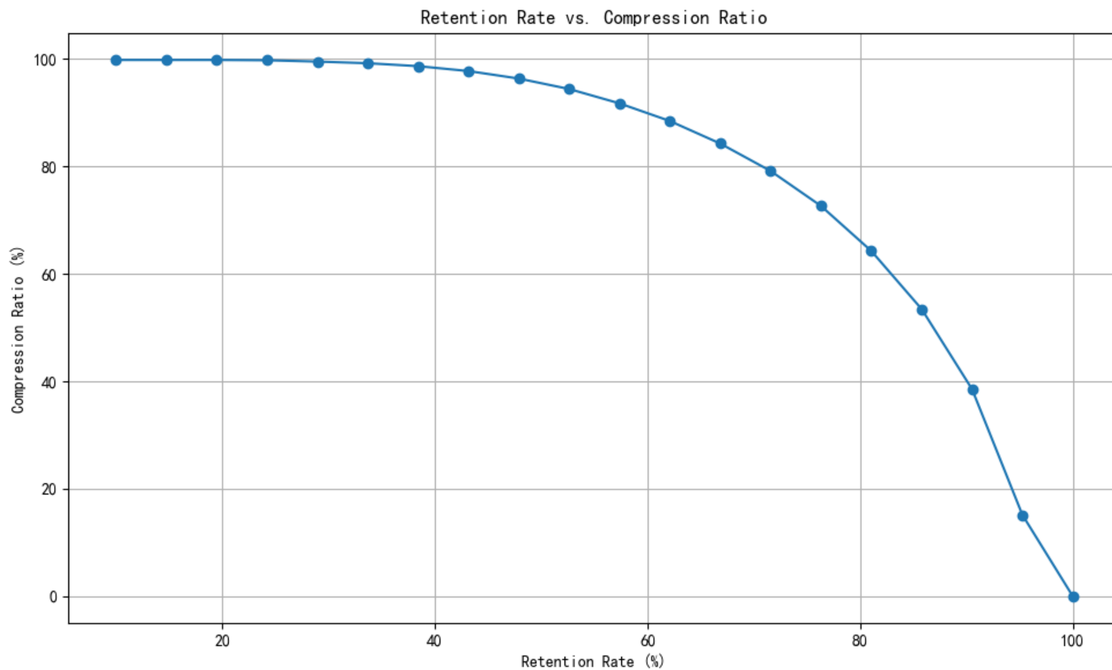

Figure 2.4.3.4 - 4

5. Retention Rate vs. Singular Value Count

- Observation: Images with high redundancy can achieve a high retention rate with fewer singular values, whereas complex images may require more singular values to retain the same level of information.

- Application: Understanding this relationship enables targeted optimization for different types of images.

- The experimental diagram is shown in Figure 2.4.3.4 - 5

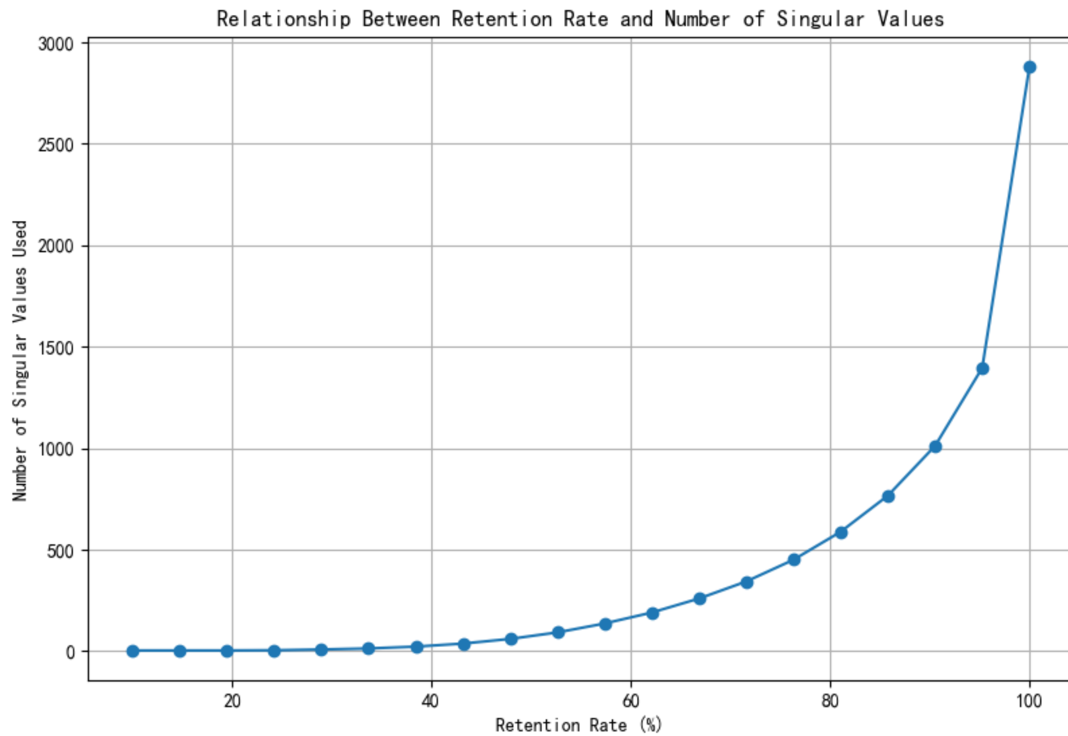

Figure 2.4.3.4 - 5

6. Quality Metrics: PSNR and SSIM

- PSNR (Peak Signal-to-Noise Ratio): Measures the pixel-level difference between the compressed and original images. Higher values indicate better quality.

- The experimental diagram is shown in Figure 2.4.3.4 - 6 - 1

- SSIM (Structural Similarity Index): Assesses the structural similarity between images, with values closer to 1 indicating better preservation of image details.

- Trends: Both metrics improve as k increases, but the rate of improvement diminishes beyond a certain threshold.

- The experimental diagram is shown in Figure 2.4.3.4 - 6 - 2

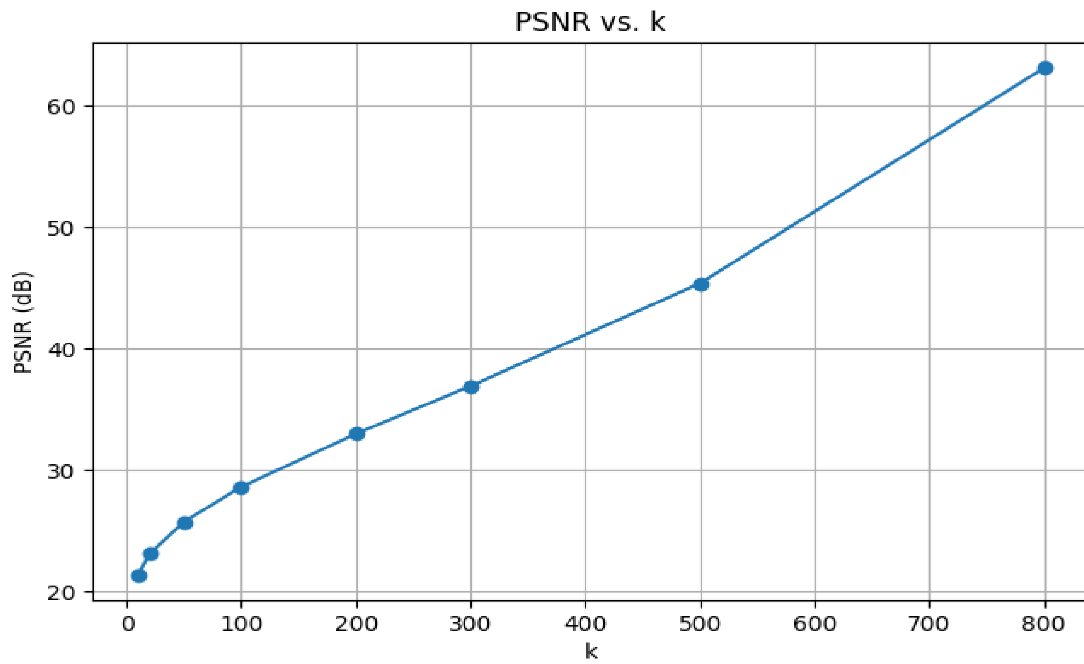

Figure 2.4.3.4 - 6 - 1

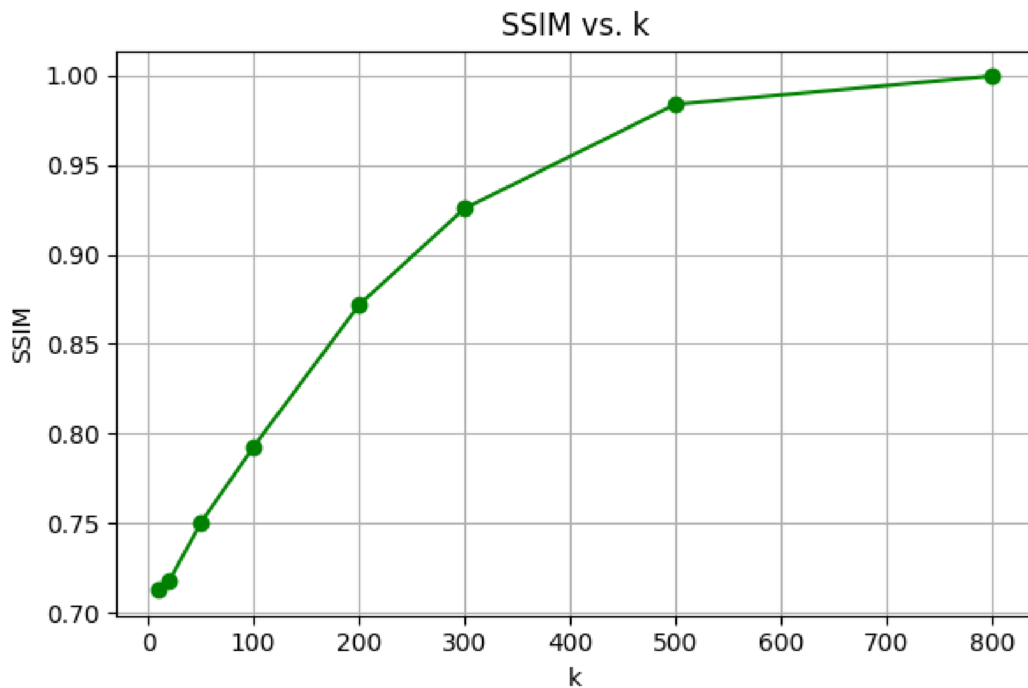

Figure 2.4.3.4 - 6 - 2

7. Compression Rate vs. k Trend

- Observation: As k increases, the compression rate decreases. However, this decrease becomes less pronounced for higher values of k.

- The experimental diagram is shown in Figure 2.4.3.4 - 7

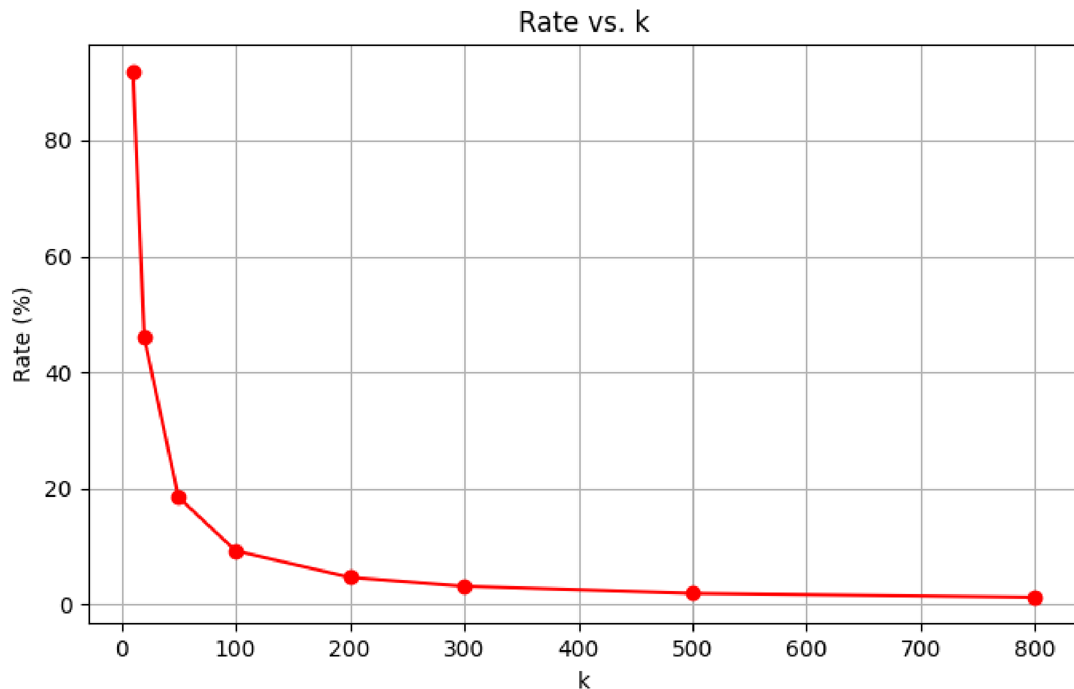

Figure 2.4.3.4 - 7

8. Memory Usage vs. k Trend

- Observation: Memory usage increases with k due to the storage requirements for singular values and vectors. Despite this, memory usage remains significantly lower than the original image size for small k.

- The experimental diagram is shown in Figure 2.4.3.4 - 8

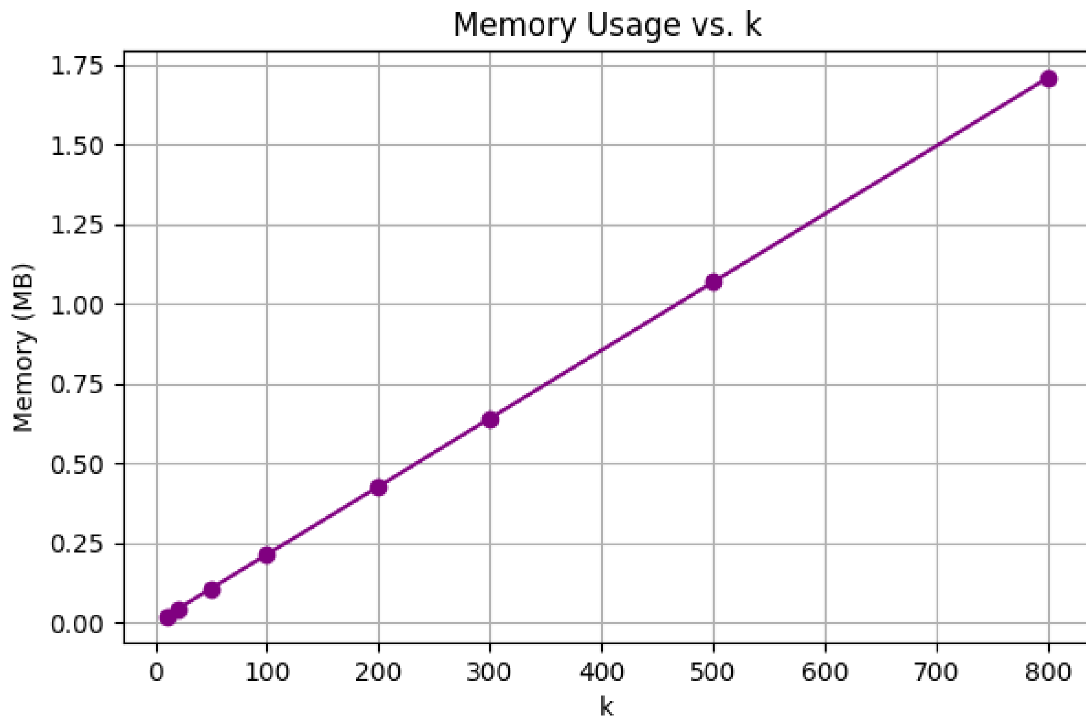

Figure 2.4.3.4 - 8

9. All parameters

- The experimental diagram is shown in Figure 2.4.3.4 - 9

Results Table:

k	PSNR	SSIM	Rate	Time (s)	Memory (MB)
10	21.3889	0.7129	91.9875	0.5152	0.0214
20	23.1175	0.7174	45.9938	0.5252	0.0427
50	25.7498	0.7500	18.3975	0.5174	0.1069
100	28.6284	0.7925	9.1988	0.5173	0.2137
200	32.9944	0.8718	4.5994	0.6269	0.4274
300	36.9203	0.9257	3.0663	0.6459	0.6412
500	45.3723	0.9839	1.8398	0.5631	1.0686
800	63.0835	0.9996	1.1498	0.5398	1.7097

Figure 2.4.3.4 - 9

2.4.3.5 Part Summary

The core advantages of SVD compression over JPEG compression lie in global feature preservation, flexibility, noise robustness, versatility, and openness. It is suitable for fields that require high image quality and where global features are critical. However, due to its high computational and storage costs, SVD compression is more often used as an experimental method rather than a mainstream choice in practical applications.

SVD-based image compression, supported by advanced techniques, offers a flexible and efficient approach to data management. Fixed singular value compression provides computational simplicity, while retention rate-based methods dynamically adapt to image characteristics. Understanding the relationships between key parameters—such as k, retention rate, and quality metrics—enables more precise control over the compression process.

Furthermore, metrics like PSNR and SSIM provide valuable insights into the trade-offs between compression and quality. Trends in compression rate and memory usage underscore the efficiency of SVD, particularly for applications with strict storage or transmission requirements.

Looking ahead, integrating adaptive and hybrid techniques, such as deep learning-based singular value selection, promises even greater

optimization. These advancements position SVD as a cornerstone of modern image compression technology.

3. CONCLUSIONS

Through an in-depth study and experimental analysis of the application of Singular Value Decomposition (SVD) in image compression, the following main conclusions are drawn:

3.1 Effectiveness of the SVD Method

As a dimensionality reduction tool, SVD efficiently maps image data from a high-dimensional space to a lower-dimensional space, achieving data compression. The experiments demonstrate that by retaining only the top k singular values, compressed images can be reconstructed with quality close to the original while significantly reducing storage requirements. This validates the effectiveness of SVD in image compression[37].

3.2 Relationship Between Singular Value Count and Compression Performance

The experimental results show that the number of singular values k has a significant impact on the compression ratio and image quality:

1. Retaining more singular values (larger k) significantly improves the quality of the reconstructed image but reduces the compression ratio.
2. The optimal k value varies depending on the type of image, which is closely related to the image's complexity, texture details, and

redundancy.

3.3 Trade-off Between Compression Ratio and Image Quality

Quantitative metrics such as Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM) were used to evaluate compression performance, revealing that:

1. At moderate compression ratios, SVD effectively preserves image quality.
2. At higher compression ratios, some detail is lost, but the overall visual quality remains satisfactory, particularly for images with high redundancy[38].

3.4 Advantages of Adaptive and Hybrid Techniques

Compared with traditional fixed truncation methods, adaptive SVD compression techniques significantly enhance performance:

1. By analyzing the decay rate of singular values, k can be dynamically adjusted to flexibly balance compression efficiency and quality.
2. Hybrid techniques, such as combining SVD with wavelet transforms or Huffman coding, further optimize compression, especially in scenarios requiring high compression ratios[39].

3.5 Limitations of the SVD Method and Future Directions

Although SVD demonstrates outstanding performance in image compression, its high computational complexity poses challenges for real-

time applications and large-scale data processing. Future optimizations could focus on the following directions:

1. Introducing efficient algorithms, such as randomized SVD, to reduce computational costs.
2. Integrating deep learning techniques, such as using convolutional neural networks (CNNs) to predict critical singular values, dynamically optimizing compression strategies[40].

3.6 Conclusion of This Thesis

In conclusion, this thesis confirms the superiority of SVD-based image compression methods in terms of compression efficiency, quality preservation, and technical extensibility. It provides a theoretically sound and practically effective solution for image compression. Despite challenges such as computational complexity, the integration of algorithmic optimizations and hybrid techniques highlights the promising future of SVD in image compression, particularly for scenarios requiring high-quality image storage and transmission, such as medical imaging, remote sensing, and multimedia communication[41].

4. REFERENCES

[1] Prasantha, *Novel Approach for Image Compression Using Modified SVD*, International Journal of Creative Research Thoughts, 2020.

[2] Ungureanu, *Image-Compression Techniques: Classical and Region-of-Interest-Based Approaches Presented in Recent Papers*, *Sensors*, 2024.

[3] Anasuodei, *An Enhanced Satellite Image Compression Using Hybrid DWT, DCT and SVD Algorithm*, American Journal of Computer Science, 2021.

[4] Golpayegani, *PatchSVD: A Non-uniform SVD-based Image Compression Algorithm*, *arXiv preprint*, 2024.

[5] Kaushik, *Efficient Compression Techniques for Medical Image Storage and Transmission: A Comprehensive Review*, *International Journal of Image and Graphics*, 2024.

[6] Zhang, *Enhancing the SVD Compression Losslessly*, *Journal of Computational Science*, 2023.

[7] N. Carter, Data Science for Mathematicians, Springer, 2020.

[8] M. Shibli, In Vivo Dynamic Image Characterization of Brain Tumor Growth Using Singular Value Decomposition, Journal of Biomedical Science and Engineering, 2011.

[9] S. Le Clainche et al., Improving Aircraft Performance Using Machine Learning, Aerospace Science and Technology, 2023.

[10] L. Guo et al., Joint Enhanced Low-Rank Constraint and Kernel Rank-Order Distance Metric for Low-Level Vision Processing, Expert Systems with Applications, 2022.

[11] K. Dumre, 5G Multi-Antenna Vehicle Scattering Characterization, Universitat Politècnica de Catalunya, 2019.

[12] F. Van Belzen, Approximation of Multi-Variable Signals and Systems: A Tensor Decomposition Approach, Technische Universiteit Eindhoven, 2011.

[13] G. Tzagkarakis et al., Trend Forecasting Based on Singular Spectrum Analysis of Traffic Workload, Performance Evaluation, 2009.

[14] P. Marksberry et al., Managing the Quality Circle Process, International Journal of Product Quality Management, 2011.

[15] R. Zhang et al., Mask Encoding: A General Instance Mask Representation for Object Segmentation, Pattern Recognition, 2022.

[16] P. Benner et al., SLICOT—A Subroutine Library in Systems and Control Theory, Springer, 1999.

[17] Biswas, Exploring Classical Simulation of Quantum Circuits, arXiv, 2024.

[18] Fong & Sneha, Covariance Matrix Method in Principal Component Analysis, UTM Press, 2023.

[19] Dionigi, Spectral Signature of Ensemble Equivalence Breaking, IMT Press, 2024.

[20] Ajayakumar & George, A Course in Linear Algebra, Springer, 2024.

[21] Pal, Recent Developments of Fuzzy Matrix Theory, Springer, 2024.

[22] Hendrick & Tilbury, Modeling Dynamic Biological Systems, ASEE Conference, 2024.

[23] Tseng & Lee, Fractional Graph Fourier Transform, IEEE, 2024.

[24] Biswas, On Classical Simulation of Quantum Circuits, Quanta, 2024.

[25] Dionigi, Spectral Signature of Ensemble Equivalence Breaking, IMT Press, 2024.

[26] Pal, Recent Developments of Fuzzy Matrix Theory, Springer, 2024.

[27] Hendrick & Tilbury, Modeling Dynamic Biological Systems, ASEE Conference, 2024; Tseng & Lee, Fractional Graph Fourier Transform, IEEE, 2024.

[28] Butnaru et al., "Designing of Music Recommender System Using Machine Learning Algorithms", Journal of Industrial Engineering and Management, 2024.

[29] Wu et al., "Multi-Dimensional Visual Data Restoration", IEEE Transactions on Visualization and Computer Graphics, 2024.

[30] Reza et al., "Applications of SVD in Energy Optimization", Elsevier Journals, 2024.

[31] Ma et al., "Tensor Singular Value Decomposition Applications", Applied Mathematics and Computation, 2025.

[32] Santos et al., "Segmentation-based Truncated SVD for Hyperspectral Image Classification", Journal of Alloys and Compounds, 2024.

[33] Liu et al., "Explainability Enhanced Object Detection Transformer", IEEE Transactions on Image Processing, 2024.

[34] Hazra et al., "Tensor Decomposition for Regression Models", Advanced Functional Materials, 2024.

[35] Kou et al., "Applications of Singular Value Properties in Nanophysics", Springer Journals, 2024.

[36] Rahman et al., "Segmentation-Based Truncated SVD for Effective Feature Extraction", Taylor & Francis, 2024.

[37] AlShaikh, Robust and Recovery Watermarking Approach Based on SVD and OTP Encryption, Springer, 2024.

[38] Reza et al., Applications of SVD in Energy Optimization, Elsevier, 2024.

[39] Liu et al., Explainability Enhanced Object Detection Transformer, IEEE, 2024.

[40] Hazra et al., Tensor Singular Value Applications, Wiley, 2024.

[41] Rahman et al., Segmentation-Based Truncated SVD for Effective Feature Extraction, Taylor & Francis, 2024.

5. APPENDIX

1. SVD Implementation Code (Python)

1) Singular value to comperssion.py

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import svd
from skimage import io

  # Function to compress image using SVD
  def svd_image_compress(image_path, k):
      img = io.imread(image_path, as_gray=True)
      U, S, Vt = svd(img, full_matrices=False)
      S_k = np.diag(S[:k])
      U_k = U[:, :k]
      Vt_k = Vt[:k, :]
      return np.dot(U_k, np.dot(S_k, Vt_k))

  # Image paths
  image_path = './img/1.jpeg'

  # Compress images with different numbers of singular values
  k_values = [10, 30, 50, 70, 90]
  compressed_images = [svd_image_compress(image_path, k) for k in
k_values]

  # Load original image
  original_img = io.imread(image_path, as_gray=True)

  # Plot the images
  fig, ax = plt.subplots(3, 2, figsize=(10, 8))
  ax[0, 0].imshow(original_img, cmap='gray')
  ax[0, 0].set_title("Original image", fontsize=8)

  for i, k in enumerate(k_values):
      ax[(i + 1) // 2, (i + 1) % 2].imshow(compressed_images[i],
cmap='gray')
      ax[(i + 1) // 2, (i + 1) % 2].set_title(f"Compressed image
(Singular value = {k})", fontsize=8)

  plt.tight_layout()
  plt.show()
```

2) Retention rate to others.py

```
import numpy as np
import matplotlib.pyplot as plt

  plt.rcParams['font.sans-serif'] = ['SimHei']  # Specify default font
  plt.rcParams['axes.unicode_minus'] = False  # Fix issue with '-' being
displayed as a square when saving the image

  # Function to compress image using SVD
  def zip_image_by_svd(origin_image, rate, axtemp):
      print("\n\n====== Starting Compression ======")

      # Prepare the result array
      result = np.zeros(origin_image.shape)

      # Perform SVD compression for each color channel (RGB)
      for chan in range(3):
          U, sigma, V = np.linalg.svd(origin_image[:, :, chan])  #
Perform SVD
          n_sigmas = 0
          temp = 0

          # Calculate the number of singular values to retain based on
the compression rate
          while (temp / np.sum(sigma)) < rate:
              temp += sigma[n_sigmas]
              n_sigmas += 1

          # Construct the singular value matrix
          S = np.zeros((n_sigmas, n_sigmas))
          np.fill_diagonal(S, sigma[:n_sigmas])

          # Reconstruct the compressed image for this channel
          result[:, :, chan] = U[:, :n_sigmas] @ S @ V[:n_sigmas, :]

      # Normalize to the range [0, 1]
      result = (result - result.min()) / (result.max() - result.min())

      # Scale to [0, 255]
      result = np.round(result * 255).astype('int')

      # Calculate compression ratio
```

```
        u_shape, s_shape, vT_shape = U[:, :n_sigmas].shape, S.shape,
V[:n_sigmas, :].shape
        zip_rate = (origin_image.size - 3 * (u_shape[0] * u_shape[1] +
s_shape[0] * s_shape[1] + vT_shape[0] * vT_shape[1])) /
origin_image.size

        # Display compression details
        print(f"Retention rate: {rate}")
        print(f"Number of singular values used: {n_sigmas}")
        print(f"Compression ratio: {zip_rate:.2f}")

        # Display the compressed image
        axtemp.imshow(result, cmap='gray')
        axtemp.set_title(f"Rate {100 * rate:.1f}%, Singular Value
{n_sigmas}, Compressibility {100 * zip_rate:.1f}%", fontsize=8)

    # Main function
    def main():
        image_path = './img/1.jpeg'
        origin_image = plt.imread(image_path)  # Read the original image

        fig, ax = plt.subplots(3, 2)  # Create a grid for displaying
images

        # Display the original image
        ax[0, 0].imshow(origin_image, cmap='gray')
        ax[0, 0].set_title("Original Image")

        # Compress the image with different retention rates and display
the results
        rates = [0.1, 0.3, 0.5, 0.7, 0.9]
        for i, rate in enumerate(rates):
            zip_image_by_svd(origin_image, rate, ax[(i + 1) // 2, (i +
1) % 2])

        plt.tight_layout()  # Adjust layout
        plt.show()

    # Call the main function if the script is run
    if __name__ == "__main__":
        main()
```

3) Singular value vs compressibility.py

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io

  # Read the image and convert it to grayscale
  image = io.imread('./img/1.jpeg', as_gray=True)

  # Perform Singular Value Decomposition (SVD)
  U, S, Vt = np.linalg.svd(image, full_matrices=False)

  # Generate k values (from 10 to 700, generating 20 evenly spaced
integers)
  k_values = np.linspace(10, 700, 20, dtype=int)

  # List to store the compression ratio for each k
  compression_ratios = []

  # Calculate the compression ratio for each k
  for k in k_values:
      # Retain the first k singular values for compression
      U_k = U[:, :k]  # Retain the first k columns of U
      S_k = np.diag(S[:k])  # Take the first k singular values and form
a diagonal matrix
      Vt_k = Vt[:k, :]  # Retain the first k rows of Vt

      # Reconstruct the compressed image via matrix multiplication
      compressed_image = np.dot(U_k, np.dot(S_k, Vt_k))

      # Calculate the size of the original image and the compressed
image
      original_size = image.size  # Total number of elements in the
original image (m * n)
      compressed_size = (k * (U_k.shape[0] + Vt_k.shape[0]) + k)  # Size
of the compressed image

      # Calculate the compression ratio and convert it to percentage
      compression_ratio = (original_size - compressed_size) /
original_size * 100
      compression_ratios.append(compression_ratio)

  # Plot the line chart
  plt.figure(figsize=(10, 6))
  plt.plot(k_values, compression_ratios, marker='o', linestyle='-',
color='b')
```

```
  plt.title('Compression ratio vs Singular value K')  # Title of the
plot
  plt.xlabel('Singular value K')  # Label for the x-axis
  plt.ylabel('Compression ratio s (%)')  # Label for the y-axis
  plt.grid(True)
  plt.xticks(k_values)  # Set the x-axis ticks to our specified k values
  plt.yticks(np.arange(0, 101, 10))  # Set the y-axis ticks to be in
intervals of 10
  plt.show()
```

4) Retention rate vs compression.py

```
import numpy as np
import matplotlib.pyplot as plt

  plt.rcParams['font.sans-serif'] = ['SimHei']  # Set font
  plt.rcParams['axes.unicode_minus'] = False  # Solve the issue with
displaying negative signs

  # Define the compression function: input image and retention rate,
output compression ratio
  def zip_image_by_svd(origin_image, rate):
      original_size = origin_image.size  # Original size of the image
      compressed_size = 0  # Initialize the total compressed size

      for chan in range(3):  # Apply SVD to each channel
          U, sigma, Vt = np.linalg.svd(origin_image[:, :, chan],
full_matrices=False)

          # Find the number of singular values that satisfy the
retention rate
          temp = 0
          n_sigmas = 0
          total_sigma = np.sum(sigma)
          while (temp / total_sigma) < rate and n_sigmas < len(sigma):
              temp += sigma[n_sigmas]
              n_sigmas += 1

          # Calculate the compressed size for each channel
          compressed_size += U[:, :n_sigmas].size + n_sigmas +
Vt[:n_sigmas, :].size
```

```
        # Compression ratio calculation, ensuring the result is non-
negative
        compression_ratio = max(0, (original_size - compressed_size) /
original_size * 100)
        return compression_ratio  # Return the compression ratio in
percentage format

    # Main function: Generate a line graph
    def main():
        image_path = './img/1.jpeg'
        origin_image = plt.imread(image_path)

        # Retention rate range: from 10% to 100%, divided into 20 points
        retention_rates = np.linspace(0.1, 1.0, 20)
        compressibility_ratios = []

        # Calculate the compression ratio for each retention rate
        for rate in retention_rates:
            compressibility = zip_image_by_svd(origin_image, rate)
            compressibility_ratios.append(compressibility)

        # Generate the line graph
        plt.figure(figsize=(10, 6))
        plt.plot(retention_rates * 100, compressibility_ratios,
marker='o')  # Convert retention rate to percentage
        plt.xlabel("Retention Rate (%)")
        plt.ylabel("Compression Ratio (%)")
        plt.title("Retention Rate vs. Compression Ratio")
        plt.grid(True)
        plt.show()

    # Run the main function
    if __name__ == "__main__":
        main()
```

5) Retention rate vs singular value.py

```
import numpy as np
import matplotlib.pyplot as plt

    plt.rcParams['font.sans-serif'] = ['SimHei']  # Set font
    plt.rcParams['axes.unicode_minus'] = False  # Fix issue with minus
signs
```

```
  # Define the compression function: input image and retention rate,
output the number of singular values used
  def zip_image_by_svd(origin_image, rate):
      total_singular_values = 0  # Initialize total singular values
count

      for chan in range(3):  # Perform SVD decomposition for each
channel
          U, sigma, Vt = np.linalg.svd(origin_image[:, :, chan],
full_matrices=False)

          # Find the number of singular values required to meet the
retention rate
          temp = 0
          n_sigmas = 0
          total_sigma = np.sum(sigma)
          while (temp / total_sigma) < rate and n_sigmas < len(sigma):
              temp += sigma[n_sigmas]
              n_sigmas += 1

          # Accumulate the singular values count for this channel
          total_singular_values += n_sigmas

      # Return the total number of singular values used
      return total_singular_values

  # Main function: Generate a line graph for the relationship between
retention rate and number of singular values
  def main():
      image_path = './img/1.jpeg'
      origin_image = plt.imread(image_path)

      # Retention rate range: from 10% to 100%, divided into 20 points
      retention_rates = np.linspace(0.1, 1.0, 20)
      singular_values_used = []

      # Calculate the number of singular values for each retention rate
      for rate in retention_rates:
          n_singular_values = zip_image_by_svd(origin_image, rate)
          singular_values_used.append(n_singular_values)

      # Generate the line graph for the relationship between retention
rate and number of singular values
```

```
      plt.figure(figsize=(10, 6))
      plt.plot(retention_rates * 100, singular_values_used,
marker='o')  # Convert retention rate to percentage
      plt.xlabel("Retention Rate (%)")
      plt.ylabel("Number of Singular Values Used")
      plt.title("Relationship Between Retention Rate and Number of
Singular Values")
      plt.grid(True)
      plt.show()

  # Run the main function
  if __name__ == "__main__":
      main()
```

6) Evaluate the metrics.py

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import time
import os
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity as ssim

# Load and compress an image using SVD
def svd_compress(image, k):
    U, S, Vt = np.linalg.svd(image, full_matrices=False)
    S_k = np.diag(S[:k])
    U_k = U[:, :k]
    Vt_k = Vt[:k, :]
    return np.dot(U_k, np.dot(S_k, Vt_k))

# Rate-Distortion Analysis
def calculate_rate(original_size, compressed_size):
    return original_size / compressed_size

# Evaluate the metrics
def evaluate_compression(image_path, k_values):
    results = []

    # Load the image
    original = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    original_size = os.path.getsize(image_path)
```

```
    for k in k_values:
        start_time = time.time()

        # Compress the image
        compressed = svd_compress(original, k)
        compressed_size = k * (original.shape[0] + original.shape[1] +
1)  # Approx memory used for compressed matrices

        # Compute metrics
        psnr_value = psnr(original, compressed,
data_range=original.max() - original.min())
        ssim_value = ssim(original, compressed,
data_range=original.max() - original.min())
        rate = calculate_rate(original_size, compressed_size)

        # Compute time and memory usage
        elapsed_time = time.time() - start_time
        memory_usage = compressed_size / (1024 * 1024)  # Convert to MB

        results.append({
            "k": k,
            "PSNR": psnr_value,
            "SSIM": ssim_value,
            "Rate": rate,
            "Time (s)": elapsed_time,
            "Memory (MB)": memory_usage
        })

    return results

# Plot results
def plot_results(results):
    ks = [r["k"] for r in results]
    psnrs = [r["PSNR"] for r in results]
    ssims = [r["SSIM"] for r in results]
    rates = [r["Rate"] for r in results]
    times = [r["Time (s)"] for r in results]
    memories = [r["Memory (MB)"] for r in results]

    plt.figure(figsize=(10, 6))

    plt.subplot(2, 2, 1)
    plt.plot(ks, psnrs, marker='o')
```

```
    plt.title("PSNR vs. k")
    plt.xlabel("k")
    plt.ylabel("PSNR (dB)")

    plt.subplot(2, 2, 2)
    plt.plot(ks, ssims, marker='o', color='green')
    plt.title("SSIM vs. k")
    plt.xlabel("k")
    plt.ylabel("SSIM")

    plt.subplot(2, 2, 3)
    plt.plot(ks, rates, marker='o', color='red')
    plt.title("Rate vs. k")
    plt.xlabel("k")
    plt.ylabel("Rate")

    plt.subplot(2, 2, 4)
    plt.plot(ks, memories, marker='o', color='purple')
    plt.title("Memory Usage vs. k")
    plt.xlabel("k")
    plt.ylabel("Memory (MB)")
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# Create table for results
def display_results_table(results):
    print("\nResults Table:")
    print(f"{'k':<10}{'PSNR':<15}{'SSIM':<15}{'Rate':<15}{'Time
(s)':<15}{'Memory (MB)':<15}")
    for r in results:
        print(f"{r['k']:<10}{r['PSNR']:<15.4f}{r['SSIM']:<15.4f}{r['Rate
']:<15.4f}{r['Time (s)']:<15.4f}{r['Memory (MB)']:<15.4f}")

# Example usage
image_path = "./img/1.jpeg"  # Replace with your image path
k_values = [10, 20, 50, 100, 200]  # Different k values for compression

results = evaluate_compression(image_path, k_values)
plot_results(results)
display_results_table(results)
```

7) Traditional vs SVD Compression PSNR identical

```python
import numpy as np
import cv2
import time
import os
import matplotlib.pyplot as plt
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity as ssim
from PIL import Image

# **1. Load Image**
def load_image(file_path):
    img = Image.open(file_path).convert("L")  # Convert to grayscale
    return np.array(img)

# **2. Get File Size**
def get_file_size(file_path):
    file_size = os.path.getsize(file_path)  # File size in bytes
    return file_size / 1024  # Convert to KB

# **3. SVD Compression with PSNR Matching**
def svd_compress(image, target_psnr, tolerance=0.001):
    U, S, Vt = np.linalg.svd(image, full_matrices=False)
    k_min, k_max = 1, len(S)
    k = (k_min + k_max) // 2
    while k_min <= k_max:
        S_k = np.diag(S[:k])
        U_k = U[:, :k]
        Vt_k = Vt[:k, :]
        compressed_image = np.dot(U_k, np.dot(S_k, Vt_k))
        current_psnr = psnr(image, compressed_image)
        if abs(current_psnr - target_psnr) <= tolerance:
            break
        elif current_psnr < target_psnr:
            k_min = k + 1
        else:
            k_max = k - 1
        k = (k_min + k_max) // 2

    memory_size = U_k.nbytes + S_k.nbytes + Vt_k.nbytes
    return compressed_image.astype(np.uint8), memory_size / 1024, k

# **4. JPEG Compression**
def jpeg_compress(image, quality=50):
```

```
    _, encoded = cv2.imencode('.jpg', image,
[int(cv2.IMWRITE_JPEG_QUALITY), quality])
    memory_size = len(encoded)
    decoded = cv2.imdecode(encoded, cv2.IMREAD_GRAYSCALE)
    psnr_value = psnr(image, decoded)
    return decoded, memory_size / 1024, psnr_value

# **5. Performance Evaluation**
def evaluate_performance(original, compressed, memory, elapsed_time):
    psnr_value = psnr(original, compressed)
    ssim_value = ssim(original, compressed)
    return {
        "PSNR": psnr_value,
        "SSIM": ssim_value,
        "Time (s)": elapsed_time,
        "Memory (KB)": memory
    }

# **6. Test Compression Methods**
def test_compression_methods(image, jpeg_quality):
    # Get original file size in KB
    original_file_size = get_file_size(file_path)

    # Evaluate original image metrics
    original_metrics = {
        "PSNR": float("inf"),  # Infinite for original
        "SSIM": 1.0,  # Perfect structural similarity
        "Time (s)": 0,
        "Memory (KB)": original_file_size
    }

    # JPEG Compression
    start_time = time.time()
    jpeg_image, jpeg_memory, jpeg_psnr = jpeg_compress(image,
jpeg_quality)
    jpeg_time = time.time() - start_time
    jpeg_metrics = evaluate_performance(image, jpeg_image, jpeg_memory,
jpeg_time)

    # SVD Compression (fine-tuned PSNR matching)
    start_time = time.time()
    svd_image, svd_memory, svd_k = svd_compress(image, jpeg_psnr)
    svd_time = time.time() - start_time
```

```
    svd_metrics = evaluate_performance(image, svd_image, svd_memory,
svd_time)

    return original_metrics, svd_image, svd_metrics, svd_k, jpeg_image,
jpeg_metrics

# **7. Display Comparison**
def display_comparison(original, original_metrics, svd_image,
svd_metrics, svd_k, jpeg_image, jpeg_metrics):
    plt.figure(figsize=(15, 5))

    def format_metrics(metrics, additional_info=""):
        return (
            f"PSNR: {metrics['PSNR']:.2f}\n"
            f"SSIM: {metrics['SSIM']:.4f}\n"
            f"Time: {metrics['Time (s)']:.4f}s\n"
            f"Size: {metrics['Memory (KB)']:.2f} KB\n"
            f"{additional_info}"
        )

    # Original Image
    plt.subplot(1, 3, 1)
    plt.imshow(original, cmap='gray')
    plt.axis("off")
    plt.title(f"Original Image\n{format_metrics(original_metrics)}",
fontsize=10)

    # SVD Compression
    plt.subplot(1, 3, 2)
    plt.imshow(svd_image, cmap='gray')
    plt.axis("off")
    plt.title(f"SVD Compression (K:
{svd_k})\n{format_metrics(svd_metrics)}", fontsize=10)

    # JPEG Compression
    plt.subplot(1, 3, 3)
    plt.imshow(jpeg_image, cmap='gray')
    plt.axis("off")
    plt.title(f"JPEG Compression\n{format_metrics(jpeg_metrics)}",
fontsize=10)

    plt.tight_layout()
    plt.show()
```

```
# **8. Main Program**
file_path = "./img/1.jpeg"  # Replace with your image path
original_image = load_image(file_path)

# Set JPEG quality
jpeg_quality = 20  # JPEG quality parameter

# Test performance and compression effects
original_metrics, svd_image, svd_metrics, svd_k, jpeg_image,
jpeg_metrics = test_compression_methods(
    original_image, jpeg_quality
)

# Display comparison results
display_comparison(original_image, original_metrics, svd_image,
svd_metrics, svd_k, jpeg_image, jpeg_metrics)
```