Image Compression using Linear Algebra

EN.535.641.81 : Mathematical Methods For Engineers
Dr. Benjamin Minnick
Dilip Kumar and Charles Davis
Oct 25, 2025

Abstract

A digital image, to a computer, is just a massive matrix (a grid) of numbers.

- A **grayscale image** is one m×n matrix, where each number represents a pixel's brightness (e.g., 0 for black, 255 for white).
- A **color image** is typically three separate m*n matrices: one for Red, one for Green, and one for Blue.

These matrices are huge. A single 12-megapixel photo (4000 × 3000 pixels) has 12 million pixels. For a color photo, that's 12 million × 3 = 36 million numbers that need to be stored and transmitted. Without compression, storing a few photos would fill your storage, and sending one high-quality image over the network would take a very long time. The goal of compression is to *reduce* the amount of data needed to represent an image while keeping the visual quality as high as possible.

In this report, we demonstrate image compression using Linear Algebra, specifically, **Singular Value Decomposition (SVD)**. By keeping only the top k singular values of an image matrix, we obtain a rank-k approximation that preserves most visual information with far fewer stored parameters. Our Python implementation (Tkinter + NumPy + PIL + Matplotlib) supports **grayscale and RGB images**, offers an **energy-based rule** to select k, reports **PSNR** and **storage-based compression ratio**, and saves a side-by-side visualization. Experiments on multiple images show that retaining 95% spectral energy typically yields perceptually faithful reconstructions with substantial storage savings.

Introduction

Digital images contain a lot of repeated information because nearby pixels often have similar colors or brightness. Linear algebra helps us find and reduce this repetition by turning the image into a matrix and then breaking it down into smaller parts. We can represent an image as an m×n matrix A, where each value in the matrix is a pixel intensity.

Using a method called **Singular Value Decomposition (SVD)**, we can factor the matrix into three new matrices:

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

The matrices U and V describe patterns in the rows and columns of the image, and Σ is a diagonal matrix that contains the **singular values**. These values tell us how much each pattern contributes to the overall image.

Most of the important information in an image is stored in just a few of the largest singular values. The smaller ones add very little detail. If we keep only the top k singular values and their related vectors, we can make a new version of the image that looks almost the same but takes up much less space. This is called a **rank-k approximation**, and it is a simple way to compress images using linear algebra.

Our Python program uses this idea through the following steps:

1. Load a user-selected image (PNG, JPG, BMP, TIFF, or WEBP).
2. Detect if it is grayscale or color (RGB).
3. Choose k either by user input or automatically to keep 95% of the image's energy.
4. Compress each channel using SVD.
5. Measure the image quality (PSNR) and calculate the compression ratio.
6. Show and save the original and compressed images.

Each of these steps connects directly to the code functions: `svd_truncate_channel`, `choose_k_by_energy`, `compute_storage_ratio`, `psnr`

Methodology

By keeping only the first k singular values and ignoring the smaller ones, we create a **rank-k approximation** of the original image. This new matrix,

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

has far fewer stored values but still looks very similar to the original image when displayed. The reason this works so well is that most real-world images have a lot of internal similarity, such as smooth gradients or repeated colors, which means that much of the image information can be captured by just a few dominant patterns. In our program, this is done by computing the SVD of each image channel (or just one for grayscale) and reconstructing it using only the top k components. This method directly uses the **Eckart–Young–Mirsky theorem**, which guarantees that this rank-k version is the best possible low-rank approximation of the image in terms of minimizing the overall error. In other words, SVD gives us the mathematically optimal way to reduce image data while keeping it as close as possible to the original.

Implementation

The Python program begins by allowing the user to select an image file through a simple graphical interface built with Tkinter. Once an image is chosen, the code checks whether it is grayscale or color. This is done by examining the number of dimensions in the image array: a 2-D array means the image is grayscale, while a 3-D array with three identical color channels is also converted to grayscale. Otherwise, the program treats it as an RGB image. After detecting the image type, the program converts the pixel values into floating-point numbers between 0 and 1 to prepare them for mathematical processing.

Next, the program asks the user to enter a rank k, which controls the level of compression. If the user leaves this blank, the program automatically chooses k based on the energy of the image, keeping enough singular values to retain 95% of the total energy. This step is done using the `choose_k_by_energy` function, which computes the cumulative sum of the squared singular values and finds the smallest k that meets the 95% energy threshold.

The main compression process is carried out using the **Singular Value Decomposition (SVD)**. In mathematical terms, this means that for each color channel (or for the grayscale matrix), the program factors the image matrix A into three parts

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

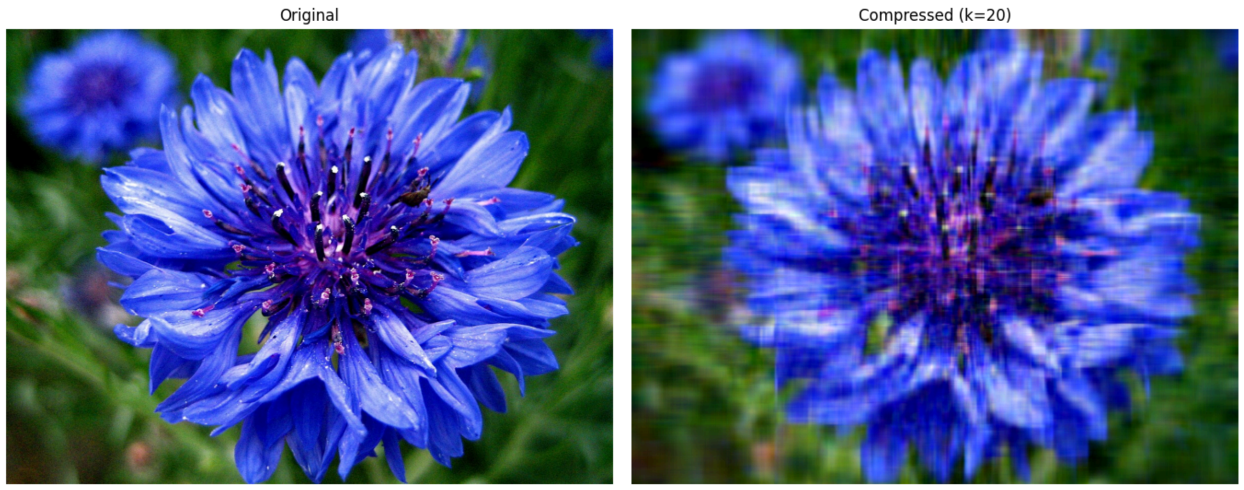
The matrices U and V contain orthogonal basis vectors that represent patterns in the rows and columns of the image, while Σ contains the singular values that measure the importance of each pattern. The program then keeps only the first kkk singular values and their corresponding columns in U and V, forming a **rank-k approximation** of the original image. In code, this step is performed by `svd_truncate_channel`.

After the SVD compression, the program evaluates how much information was preserved by calculating two key metrics: **PSNR (Peak Signal-to-Noise Ratio)** and the **compression ratio**. PSNR measures the similarity between the original and compressed images, where a higher value means better quality. The compression ratio is computed by comparing the number of elements in the original image matrix to the number of stored values in the SVD representation, which is based on k(m+n+1)k(m + n + 1)k(m+n+1). These calculations are handled by the `psnr` and `compute_storage_ratio` functions.

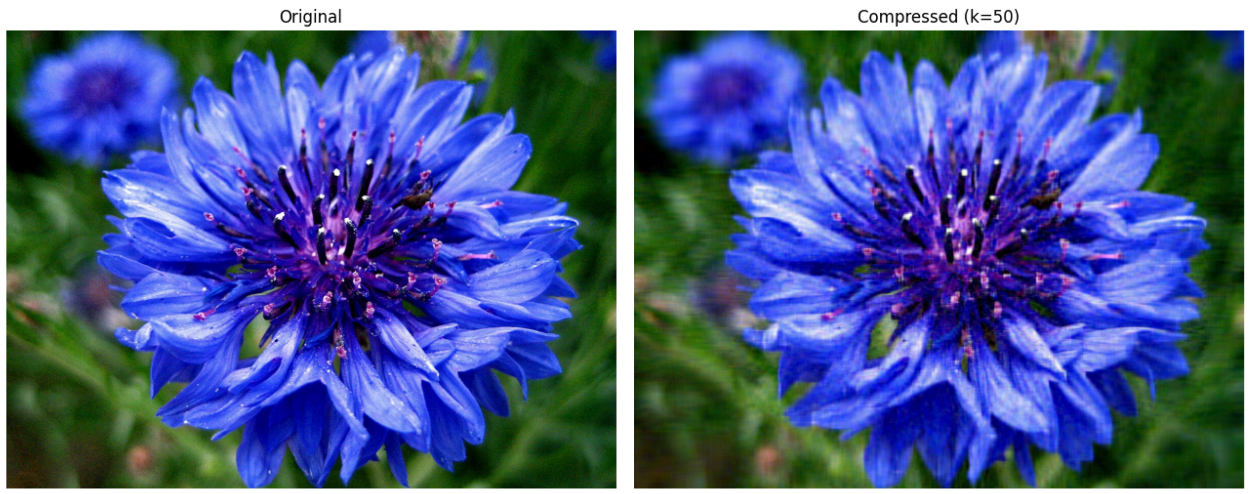
Finally, the program displays the original and compressed images side by side using Matplotlib and saves the new image in the same folder as the original, with a filename that includes the rank or energy used. The printed summary includes the image type, the rank k used, the compression ratio, and the PSNR value, giving a clear overview of how effective the compression was. This entire process combines linear algebra concepts, such as matrix factorization and rank approximation, with practical coding steps that make the math come alive through real image compression.

Results

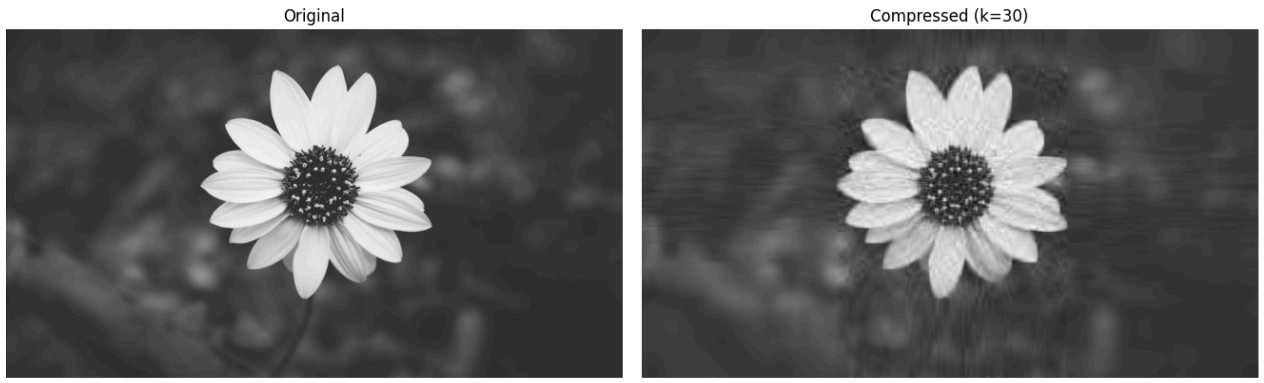
Three images were tested to observe how the rank k value affects compression quality and visual appearance. The first test used a **color (RGB) image with k=20**. This yielded a compression ratio (orig/SVD entries) of 25.70 and a PSNR (dB) of 21.88. In this case, the image appeared noticeably blurrier, with fine details and sharp edges smoothed out. However, the main shapes and colors were still easy to recognize, showing that even a small number of singular values can capture most of the important structure of the image. The compression ratio was high, meaning a large reduction in data size was achieved, though at the cost of some clarity.



The second test used the same **RGB image with k=50**. This yielded a compression ratio (orig/SVD entries) of 10.28 and a PSNR (dB) of 26.01 This produced a much clearer and sharper image, with textures and edges better preserved compared to k=20. The colors also appeared more natural and closer to the original. While this required storing more singular values, it still represented a strong level of compression compared to the uncompressed version. The improvement in visual quality from k=20 to k=50 shows how increasing k adds more detail back into the reconstruction.



Finally, the **grayscale image with k=30** produced good results as well. This yielded a compression ratio (orig/SVD entries) of 36.00 and a PSNR (dB) of 30.97. Since grayscale images contain only one matrix of values instead of three color channels, the compression was more efficient. This test confirms that SVD works especially well for grayscale images because there is less variation across channels, allowing good compression even at relatively low ranks.



Overall, the results demonstrate the trade-off between compression and image quality: smaller k values save more space but lose detail, while larger k values preserve visual quality but require more data. In all cases, the SVD method successfully reduced storage needs while maintaining a visually similar version of the original image.

Appendix

```python
# Run this whole file in VS Code
# 1) Opens file picker to choose an image
# 2) Ask for rank k (or use 95% energy if left blank)
# 3) Compress via SVD, show before/after, and save output

import os
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

import tkinter as tk
from tkinter import filedialog, simpledialog, messagebox


def to_float01(arr):
    if arr.dtype == np.uint8:
        return arr.astype(np.float64) / 255.0
    arr = arr.astype(np.float64)
    if arr.max() > 1.0:
        arr = arr / arr.max()
    return arr


def to_uint8(arr):
    return np.clip(arr, 0.0, 1.0) * 255.0


def psnr(orig, recon):
    mse = np.mean((orig - recon) ** 2)
    if mse == 0:
        return float("inf")
    return 10 * np.log10(1.0 / mse)


def choose_k_by_energy(s, energy=0.95):
    # --- Linear Algebra (energy criterion):
    # ||A||_F^2 = Σ_i s_i^2. We pick smallest k so Σ_{i=1..k} s_i^2 ≥ energy *
Σ_i s_i^2
    s2 = s**2
    cum = np.cumsum(s2)
    total = cum[-1]
    k = int(np.searchsorted(cum, energy * total)) + 1
    return k


def svd_truncate_channel(channel, k=None, energy=None):
    # --- Linear Algebra (SVD factorization and rank-k approximation):
    # A = U Σ V^T,  A_k = U_k Σ_k V_k^T  (Eckart-Young-Mirsky theorem)
    U, s, Vt = np.linalg.svd(channel, full_matrices=False)

    if energy is not None and k is None:
        k = choose_k_by_energy(s, energy)

    if k is None:
        raise ValueError("Provide k or energy.")

    k = max(1, min(k, len(s)))
    Uk = U[:, :k]
    sk = s[:k]
    Vtk = Vt[:k, :]
    recon = Uk @ (sk[:, None] * Vtk)
    return recon, k, s


def compute_storage_ratio(shape, k, channels=1):
    # --- Linear Algebra (parameter counting):
    # Original: m*n entries.  Rank-k: U_k (m*k) + Σ_k (k) + V_k (n*k)
    m, n = shape
    orig = m * n * channels
    svd = channels * (k * (m + n + 1))
    return orig / svd


def compute_storage_ratio_rgb_exact(shape, k_list):
    m, n = shape
    orig = 3 * m * n
    svd = sum(kc * (m + n + 1) for kc in k_list)
    return orig / svd


def main():
    root = tk.Tk()
    root.withdraw()

    # Choose image file
    path = filedialog.askopenfilename(
        title="Choose an image",
        filetypes=[
            ("Images", "*.png;*.jpg;*.jpeg;*.bmp;*.tif;*.tiff;*.webp"),
            ("All files", "*.*"),
        ],
    )
    if not path:
        messagebox.showinfo("SVD Compression", "No file selected. Exiting.")
        return

    # Ask for rank (k) or use energy-based cutoff
    k_text = simpledialog.askstring(
        "Compression Rank (k)",
        "Enter rank k (e.g., 50). Leave blank to target 95% energy:",
    )

    k_val = None
    energy_val = None
    if k_text and k_text.strip():
        try:
            k_val = int(k_text)
            if k_val <= 0:
                raise ValueError
        except Exception:
            messagebox.showerror("Invalid Input", "k must be a positive
integer.")
            return
    else:
        energy_val = 0.95  # default 95% energy retention

    # Load image
    img = Image.open(path)
    if img.mode == "RGBA":
        img = img.convert("RGB")

    arr = np.array(img)

    # Hybrid grayscale detection
    if arr.ndim == 2:
        is_gray = True
    elif arr.ndim == 3 and arr.shape[2] == 3 \
         and np.array_equal(arr[:, :, 0], arr[:, :, 1]) \
         and np.array_equal(arr[:, :, 1], arr[:, :, 2]):
        print("Detected RGB image with identical channels → converting to
grayscale.")
        arr = arr[:, :, 0]
        is_gray = True
    else:
        is_gray = False

    print("Detected image mode:", "Grayscale" if is_gray else "Color (RGB)")

    # Perform SVD compression
    if is_gray:
        arr_f = to_float01(arr)
        recon, k_used, svals = svd_truncate_channel(arr_f, k=k_val,
energy=energy_val)
        psnr_val = psnr(arr_f, np.clip(recon, 0, 1))
        cr = compute_storage_ratio(arr_f.shape, k_used, channels=1)
        recon_img = to_uint8(recon).astype(np.uint8)
        k_report = k_used
    else:
        arr_f = to_float01(arr.astype(np.float64))
        m, n, _ = arr_f.shape
        recon = np.zeros_like(arr_f)
        k_used_list = []
        psnr_list = []

        for c in range(3):
            rec_c, k_used_c, _ = svd_truncate_channel(arr_f[:, :, c], k=k_val,
energy=energy_val)
            recon[:, :, c] = rec_c
            k_used_list.append(k_used_c)
            psnr_list.append(psnr(arr_f[:, :, c], np.clip(rec_c, 0, 1)))

        k_report = max(k_used_list)
        psnr_val = float(np.mean(psnr_list))
        cr = compute_storage_ratio((m, n), k_report, channels=3)
        recon_img = to_uint8(recon).astype(np.uint8)

    # Save output
    base, ext = os.path.splitext(os.path.basename(path))
    suffix = f"_svd_k{k_report}" if energy_val is None else
f"_svd_energy{int(energy_val*100)}_k{k_report}"
    out_path = os.path.join(os.path.dirname(path), f"{base}{suffix}{ext}")
    Image.fromarray(recon_img).save(out_path)

    # Show before/after comparison
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.imshow(arr if not is_gray else arr, cmap=None if not is_gray else
"gray")
    plt.title("Original")
    plt.axis("off")

    plt.subplot(1, 2, 2)
    plt.imshow(recon_img if not is_gray else recon_img, cmap=None if not
is_gray else "gray")
    plt.title(f"Compressed (k={k_report})")
    plt.axis("off")

    plt.tight_layout()
    plt.show()

    # Summary output
    msg = (
        f"Saved: {out_path}\n"
        f"Mode: {'Grayscale' if is_gray else 'RGB'}\n"
        f"Rank used: {k_report}\n"
        f"Compression ratio (orig/SVD entries): {cr:.2f}\n"
        f"PSNR (dB): {psnr_val:.2f}"
    )
    print(msg)
    messagebox.showinfo("SVD Compression Complete", msg)


if __name__ == "__main__":
    main()
```

References

https://math.mit.edu/~gs/everyone/everyone_svd01.pdf

https://web.stanford.edu/class/cs168/l/l9.pdf