Kathryn Saunders - Section 002
Keegan Chatham - Section 002
APPM 3310
04/30/2025

The Application of Low-Rank Matrix Approximation in Image Compression

Abstract

This paper details the applications of singular value decomposition (SVD) in lossy image compression. Lossy compression is a way to reduce the data size of digital media by removing information to make it easier to store, display, or transmit. Lossy compression removes some of the detailed data from the image to reduce the image's data size, which reduces image quality. In this project, we show how images can be interpreted as matrices of pixels and factored using SVD. We use low-rank matrix approximation to reduce the size and quality of an image measurably and provide examples of images compressed using this method using Python coding. Lossy image compression is used in a wide variety of industries and services, so it is a useful tool to understand.

Attribution

Kathryn wrote the abstract, introduction, and the mathematical formulation section. Keegan did all of the Python coding and wrote the examples section. Both Kathryn and Keegan contributed to editing and the conclusion section.

Introduction

This project will show the effectiveness of low-rank matrix approximation as a method for conducting lossy image compression. Our research for this project was primarily based on the excellent paper "Rapid Interlacing and High Compression Rates for Images Using Singular Value Decomposition" by Anas Y. Boubas and Maamar Bettayeb. We were interested in Boubas and Bettayeb's work in image compression because of its practicality and variety of applications. This project will focus on the use of singular value decomposition (SVD), which decomposes a matrix into three other matrices – two unitary (rotation) matrices and a diagonal (scaling) matrix. Although there are many applications of SVD, we focus on the application of low-rank matrix approximations in image compression. Lossy image compression is used in social media, video streaming, web optimization, and many more services to make image transfer and display faster and more efficient. In a world with more visual media than ever, image compression is used everywhere. Additionally, we were further interested in the ability to control how much images are compressed and the resulting change in quality. Lossy image compression is an interesting application of SVD because it makes use of the way SVD ranks singular values in order of the largest to the smallest size. By disregarding the singular values at the end, one can substantially decrease an image's data size while still maintaining high image quality.

In APPM 3310, we discussed SVD and the various applications thereof – finding the condition number, the pseudo-inverse, and low-rank approximations. In class, we mostly focused on the uses of singular values in comparison with the more limited application of eigenvalues. This project made us realize the value of low-rank approximations. We also used the Frobenius norm to evaluate the difference between the compressed image and the original image. Before this project, we thought the Frobenius norm was similar to the 2-norm but easier to calculate. Now, we know that the Frobenius norm is incredibly useful for determining the difference between matrices. In general, this project has greatly expanded our understanding of the applications of mathematical processes and formulae we learned in class.

Mathematical Formulation

Singular value decomposition is a method of matrix factorization. For $A \in \mathbb{R}_r^{m\times n}$, there exists orthogonal matrices $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ such that $A = U\Sigma V^T = \sum_{i=1}^{r} \sigma_i u_i v_i^{\mathrm{T}}$ where $\Sigma = \begin{bmatrix} S & 0 \\ 0 & 0 \end{bmatrix}$ and $S = diag(\sigma_1, \sigma_1, \cdots, \sigma)$ with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$. Singular values, σ_i, are the square roots of the eigenvalues of $A^T A$. As $A^T A$ is positive semidefinite, both its eigenvalues and its singular values are real and non-negative.

$$\mathrm{A} = \underset{m\times n}{\begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nm} \end{bmatrix}} = \underset{m\times m}{\begin{bmatrix} | & | & & | \\ u_1 & u_2 & \dots & u_m \\ | & | & & | \end{bmatrix}} \underset{m\times n}{\begin{bmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_n \\ 0 & 0 & \dots & 0 \end{bmatrix}} \underset{n\times n}{\begin{bmatrix} - & v_1^T & - \\ - & v_2^T & - \\ & \vdots & \\ - & v_n^T & - \end{bmatrix}}$$

Lossy compression can be accomplished using SVD by interpreting an image as a matrix of pixels. A pixel's color can be broken down into red, green, and blue (RGB) components, where each component ranges in value from 0 to 255. Each combination of RGB values represents a unique color. Before starting SVD, the image must be separated into three separate matrices, one for each color component.

$$\begin{bmatrix} r_{11} & r_{12} & \dots & r_{1m} \\ r_{21} & r_{22} & \dots & r_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ r_{n1} & r_{n2} & \dots & r_{nm} \end{bmatrix} \begin{bmatrix} g_{11} & g_{12} & \dots & g_{1m} \\ g_{21} & g_{22} & \dots & g_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ g_{n1} & g_{n2} & \dots & g_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nm} \end{bmatrix}$$

Each matrix must then be decomposed using SVD. SVD sorts the singular values, σ_i, in decreasing order, so the singular values that correspond to the greatest amount of information are at the beginning. Low-rank matrix approximation involves disregarding the singular values at the end of the matrix to reduce the image's size while retaining the most significant information. It creates a new matrix that is a close approximation of the original but much smaller in dimension. Starting from the end, the more dimensions disregarded, the smaller the image's size will be, and the worse the image's quality will be. Changing the number of disregarded singular values changes the amount of compression.

One can specify a desired percentage compression where values closer to 1 result in more degraded images and values closer to 0 result in better quality images using the following (where r is the rank of the matrix and w is the number of dropped singular values).

$$\text{Quality} = \sum_{i=r-w+1}^{r} \sigma_i$$

The version of the SVD after w singular values are dropped is called the truncated SVD and is represented by

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$$

More simply, the truncated SVD of A is $A_k = \tilde{U}\tilde{\Sigma}\tilde{V}^T$ where $\tilde{U} \in \mathbb{R}^{m\times k}$, $\tilde{V} \in \mathbb{R}^{n\times k}$, and $\tilde{\Sigma} \in \mathbb{R}^{k\times k}$. A_k is an approximation of A with rank k instead of rank r. According to the Eckart-Young Theorem, the k-rank SVD approximation of A is the best rank k approximation of A with respect to the Frobenius norm. Thus, the truncated SVD results in the best possible compression algorithm for the specified amount of compression. In image compression, each of the three SVD decompositions of the color value matrices must be compressed into their truncated SVD form, where each matrix has the same w singular values dropped.

Once the three A_k matrices are calculated, they must be recombined into a single matrix A_c with full RGB color values. To illustrate this, we've stacked the pixel values into a three-dimensional matrix along a new third axis. This function works by taking three two-dimensional matrices (R, G, and B) and combining them into one three-dimensional matrix by stacking each entry into an array of three values, one for each R, G, and B value ranging from 0 to 255 inclusive. This new matrix acts as a two-dimensional matrix with each entry taking the form:

$$c_{i,j} = R_{i,j}, G_{i,j}, B_{i,j}$$

The Frobenius norm is similar to the 2-norm and is often used to measure similarity between matrices.

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}$$

It can be used to evaluate the difference between A and A_c, which allows one to approximate A within a specified error tolerance by choosing an ε and running an algorithm to find the largest possible A_c where $\|A - A_c\|_F^2 < \epsilon$.

$$\|A - A_c\|_F^2 = \|\sum_{i=k+1}^{r} \sigma_i u_i v_i^T\|_F^2 = \sum_{i=k+1}^{r} \sigma_i^2$$

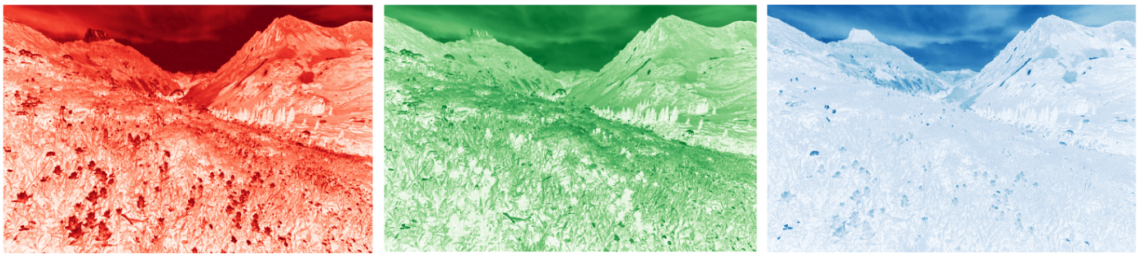
The normalized reconstruction error of different levels of SVD image compression can be measured by dividing the Frobenius norm by the image's size. Both A and A_c should have the same number of pixels, but to compare images of different sizes, the Frobenius norm should be divided by the total number of pixels in the image to normalize it.

$$\text{Normalized Reconstruction Error} = \frac{\|A - A_c\|_F}{\text{number of pixels}}$$

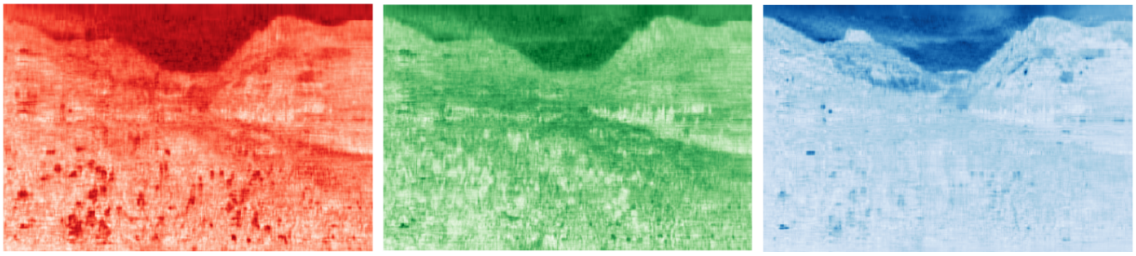
Examples

Using the image below, photographed by Jason Hatfield, we will demonstrate this process.

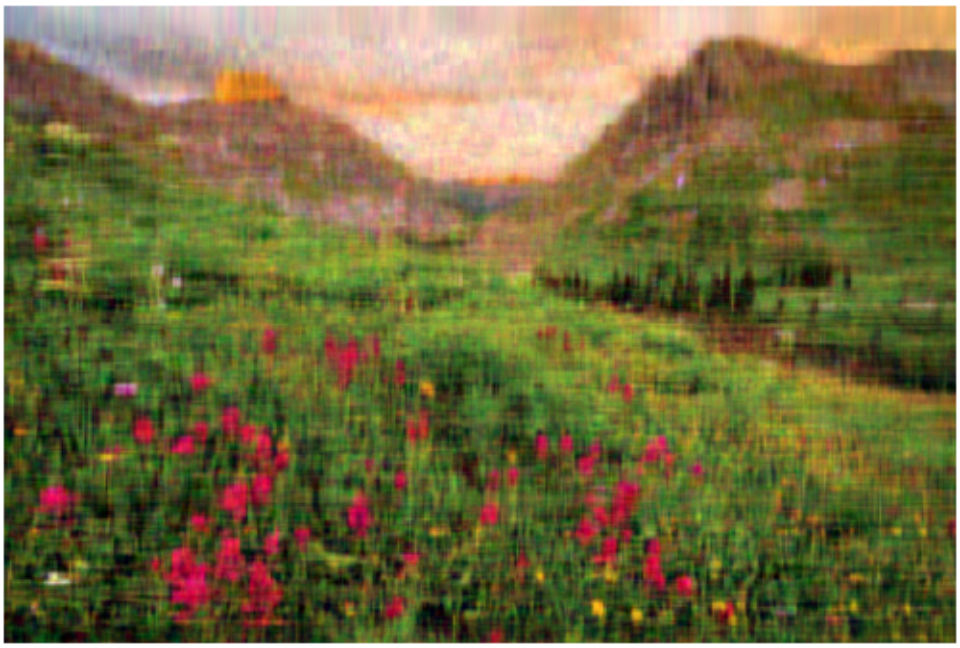
The image must be split into the three color matrices like so:



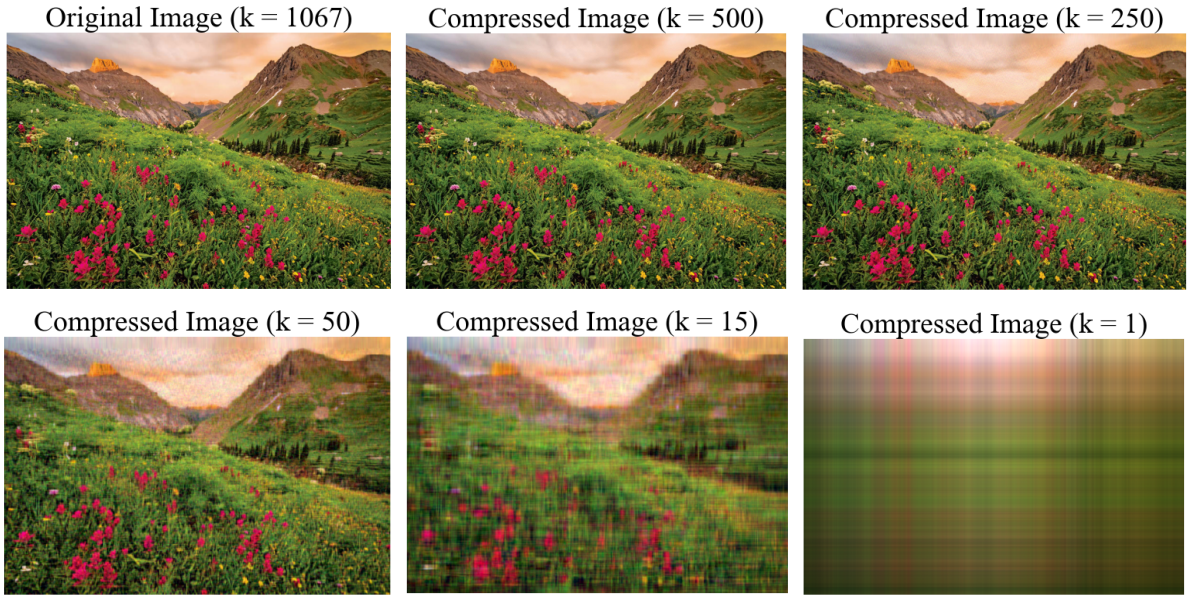
From there, we calculated the SVD for each matrix and disregarded w singular values. The number of values disregarded depends on the goal of the compression. Below is an example of the compressed RGB color matrices with 25 singular values remaining.



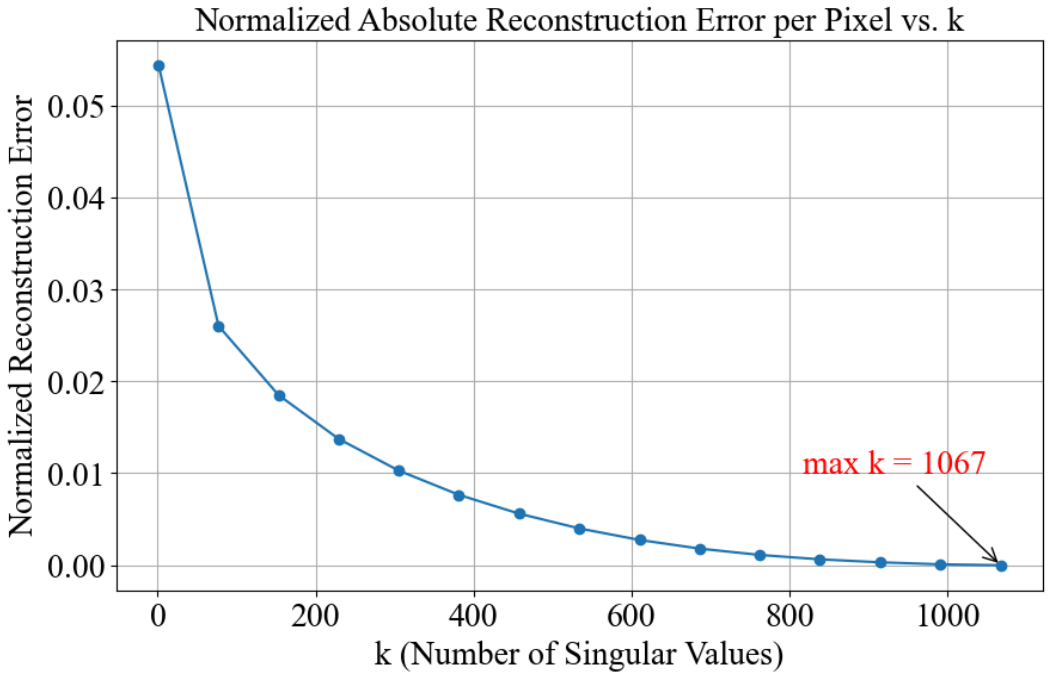
When the three color matrices are interlaced, the compressed image reveals itself. In this example, both the original and compressed images are 40.973 megabytes in terms of RAM memory. Both the original and compressed images are of the same size because RAM is measured by the memory occupied by the size of the array, and both images are the same size array. Later, we will estimate the file size in RAM memory based on the ratio of singular values retained in the compressed image to illustrate an idealized linear relationship between singular values retained and file size. We will also model the observed tradeoff between the number of singular values retained and the JPG file size after lossy image compression.



How much one should compress an image depends on one's motivation. To maintain as high an image quality as possible, one would disregard fewer singular values. To reduce data size, one would disregard more singular values. The correct balance between the data quality and image size is left to the individual. Below are examples of the same image at various levels of compression:

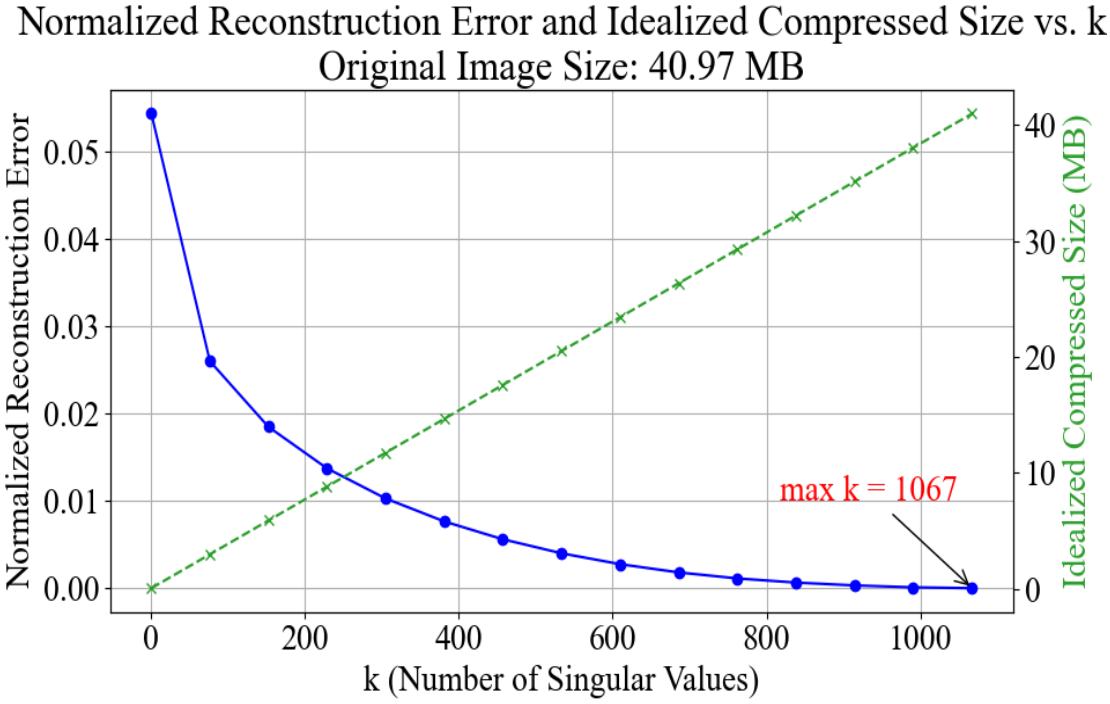


As seen above, one can dramatically reduce the data size of an image without a major change in image quality. A drop in image quality only becomes visually apparent in the third image, where $w = 817 = 1067 - 250$ singular values were disregarded. Despite the loss of singular values, visual parity remains intact. Even when only fifteen of the original one thousand and sixty-seven singular values remain (the fifth image), the subject of the image can still be interpreted. The most compressed image possible has a single singular value remaining (the sixth image), as an image with $k = 0$ singular values is entirely black. An image retaining $k = 0$ singular values has RGB values of zero in each pixel, and the color indicated by the RGB code 0, 0, 0 is black.



As the graph above illustrates, the more singular values are disregarded, the more the resulting compressed image varies from the original according to the Frobenius norm of the difference. The norm is standardized so that one can compare images with different numbers of singular values. The normalized reconstruction error, in layman's terms, is the average difference in R, G, and B values for a given pixel. For example, the compressed image retaining $k = 500$ singular values has a normalized reconstruction error of 0.0047, whereas the compressed image retaining $k = 15$ has a normalized reconstruction error of 0.0397. It is interesting to note that one can disregard nearly half of the original values before seeing an increase in error greater than 0.005. In fact, the error remains relatively low compared to the size of the image until fewer than 100 singular values remain in the compressed image.

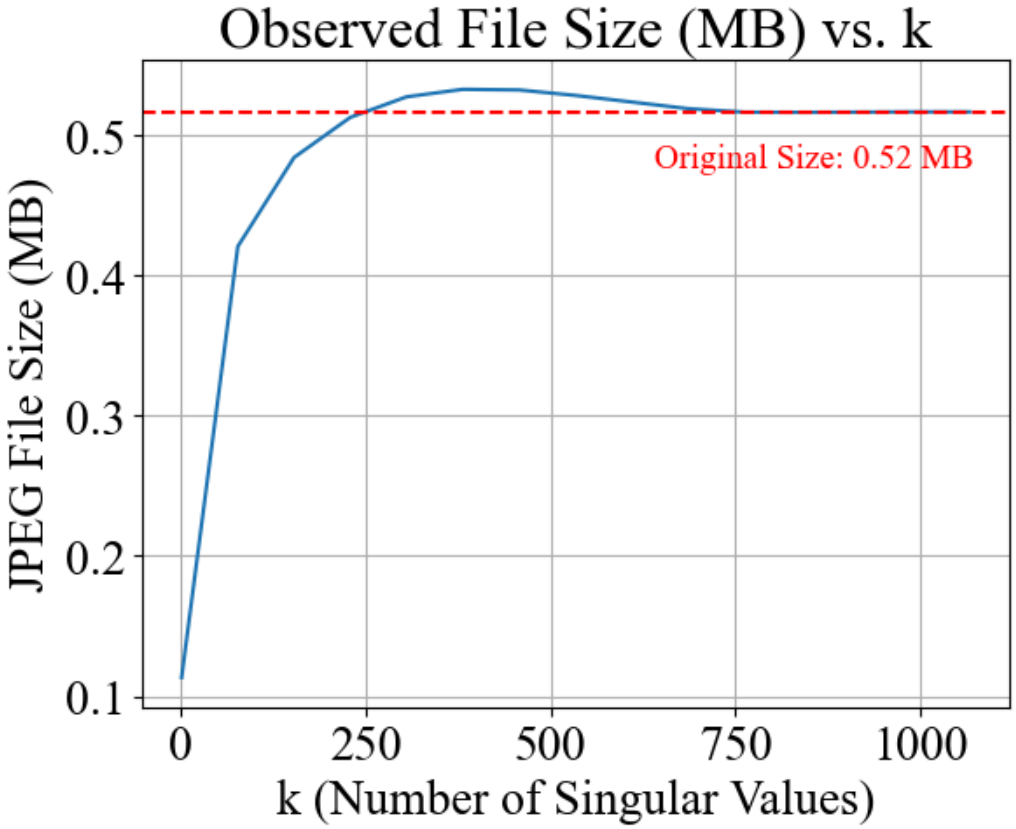
Now, we will illustrate the trade-off between image size and visual quality.



One would assume, intuitively, that the relationship between the number of singular values retained and image size in megabytes is a decreasing linear relationship, meaning that as you compress the image by reducing the number of singular values kept, the image size decreases linearly. One can contrast this notion with the relationship between singular values kept and normalized reconstruction error, which has a decreasing, non-linear relationship. The original image is 40.97 megabytes of RAM memory, the compressed image with $k = 500$ has an idealized size of 19.2 megabytes, and the compressed image with $k = 15$ has an idealized size of 0.58 megabytes. The idealized compressed size in RAM memory was calculated by multiplying the ratio of singular values retained by the original image size:

$$\text{Idealized Compressed Size} = \frac{k}{k_{max}}(40.97)$$

Below, we have modeled the number of singular values retained versus observed JPEG file size in megabytes.



In contrast to the idealized linear relationship between singular values retained and file size, when the images are saved in JPEG format, the observed relationship is non-linear and non-monotonic for JPEG lossy image compression. The original image is 0.517 megabytes in JPEG format, but the compressed image with $k = 500$ is 0.531 megabytes, and the compressed image with $k = 15$ is 0.266 megabytes. The file size with all 1067 singular values is smaller than the file sizes of images with significantly fewer singular values. This phenomenon can be observed by comparing the blue line of observed JPEG file sizes to the red dashed line, which is the size of the original image.

This observed relationship is likely due to characteristics of JPEG compression. In contrast to RAM memory, JPEGs do not store raw image data. Instead, images are compressed based on image smoothness and redundancy, meaning that blurrier images and images with predictable patterns are stored more easily. When an image is not compressed very much, the difference in singular values does not dramatically reduce image detail, so the JPEG compression optimization algorithm does not necessarily perceive it as a smaller file size because the pattern of data is similar. However, if an image is compressed enough that the difference in singular values noticeably reduces image detail, the JPEG compression algorithm does reduce the image's data size.

Now we'll illustrate how three different images appear when 2.5% of the maximum singular values are kept. The purpose of this is to illustrate differences in reconstruction error

between images with different numbers of singular values. First is a portrait of Carl Friedrich Gauss by Christian Albrecht Jensen:

Original Image (k = 640)

Compressed Image (k = 16)

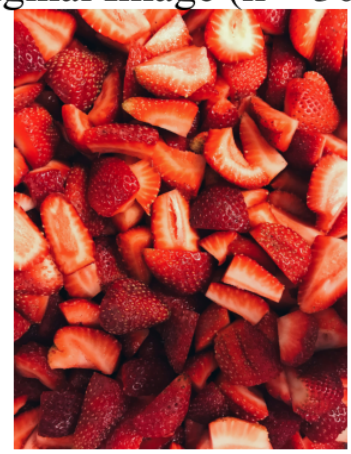
Next is an image of a Blue Morpho butterfly (photographer unknown):

Original Image (k = 763)

Compressed Image (k = 19)

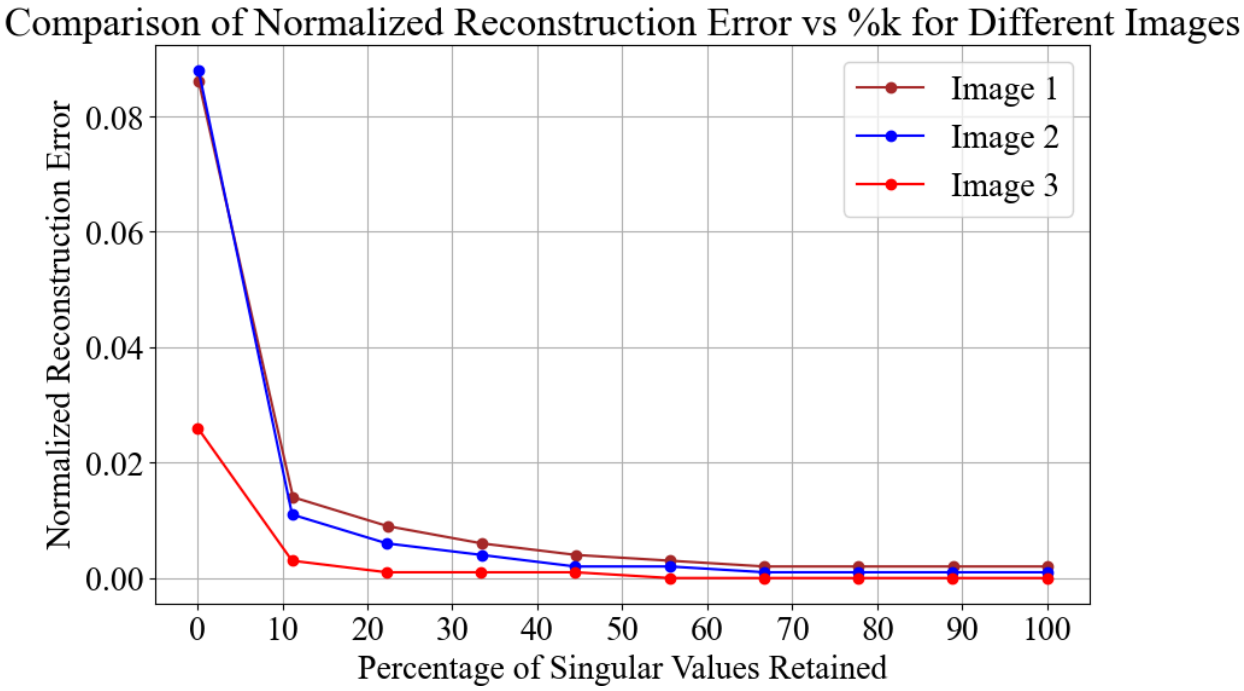
Finally, an image of sliced strawberries (photographer unknown):

Original Image (k = 3000)



Compressed Image (k = 75)

Below, we've modeled the differences in the relationship between percentages of singular values retained and normalized reconstruction error for the above three images. The brown line represents the Carl Friedrich Gauss portrait, the blue line represents the Blue Morpho butterfly image, and the red line represents the strawberries image.



Based on the plot, images 1 and 2 (Gauss portrait and blue butterfly) had very similar normalized reconstruction errors for a given percentage of singular values retained. The third image (strawberries) had a smaller normalized reconstruction error compared to the other images as the percentage of singular values retained approaches zero. We hypothesize that this is due to low variability in color because, unlike the other two images, the image of strawberries maintains strong visual parity. However, we are unable to confirm it without further analysis.

Conclusion

Low-rank matrix approximation is an effective way to compress images. We accomplished a significant reduction in image size while maintaining image quality. We were able to demonstrate and quantify different levels of image compression using a variety of different images. In the above sections, we have discussed the trade-off between image quality and data size but have not proposed a universal solution. If one is posting an image on social media, for example, one would probably want the compressed image to be as visually similar to the original, uncompressed image as possible, so the image size, while still reduced, may be larger than in other applications. By contrast, if one is sending an image over email or text message, perhaps image quality matters less than the ability to send the image quickly, so a more compressed image with a smaller data size is preferred.

One of the most surprising things we discovered during this project was just how much an image can be compressed without any significant increase in error. The compressed and

uncompressed images look nearly identical to the naked eye, but the compressed version has a significantly smaller file size.

We illustrated the process of lossy image compression by removing w singular values from an image. However, although singular value decomposition is the backbone of lossy image compression, we have only demonstrated a simple compression technique. Actual applications of lossy image compression, like JPEG image compression, are more complex than our process. JPEG compression depends less on the number of singular values in the image and more on the image's smoothness and redundancy.

All of our work in this paper has been on compressing still images. In the modern era, many of the most widespread applications of lossy compression concern the compression of video and audio media. Lossy compression is used in video/music streaming, video surveillance, video calling, broadcast media, and many other services. If we were to expand our project further, we would examine how the process of low-rank matrix approximation for image compression changes when processing static data versus dynamic data. Although this would be a fascinating expansion of this process, it is beyond the scope of this project.

References

Boubas, Anas Y., and Maamar Bettayeb. “Rapid Interlacing and High Compression Rates for Images using Singular Value Decomposition.” *2008 3rd International Conference on Information and Communication Technologies: From Theory to Applications*, Apr. 2008, pp. 1–4, https://doi.org/10.1109/ictta.2008.4530090.

Caplan, Paul. “What is a JPEG? The Invisible Object You See Every Day.” *The Atlantic*, September 24, 2013. https://www.theatlantic.com/technology/archive/2013/09/what-is-a-jpeg-the-invisible-object-you-see-every-day/279954/.

Eckart, Carl, and Gale Young. “The Approximation of One Matrix by Another of Lower Rank.” *Psychometrika*, vol. 1, no. 3, Sept. 1936, pp. 211–218, https://doi.org/10.1007/bf02288367.

Hou, Junhui, et al. “Sparse Low-Rank Matrix Approximation for Data Compression.” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 5, May 2017, pp. 1043–1054, https://doi.org/10.1109/tcsvt.2015.2513698.

Appendix

Python code used in the examples section:

```
# Importing necessary packages for matrices, plotting, image importation, and image saving
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import os
```

```
# Defining function to compress an individual matrix
def compress_channel(channel, k): #takes one 2D channel (R, G, or B) and compresses it
    P, E, QT = np.linalg.svd(channel, full_matrices=False) #uses numpy linalg.svd function to compute the SVD
    E = np.diag(E[:k]) # keeps only the first k singular values in the diagonal matrix
    return P[:, :k] @ E @ QT[:k, :] # reconstructs the new compressed matrix by truncating singular values
```

```
# loads & deconstructs an image into R, G, and B matrices, compresses each, then reassembles
def compress_image(img_path, k):
    img = np.array(Image.open(img_path).convert('RGB'), dtype=np.float64) #loads image and converts into RGB arrays
    R, G, B = img[:,:,0], img[:,:,1], img[:,:,2] # assigning the split into the 3 matrices
    R_compressed = compress_channel(R, k) # compresses each matrix
    G_compressed = compress_channel(G, k)
    B_compressed = compress_channel(B, k)
    img_comp = np.stack([R_compressed, G_compressed, B_compressed], axis=2) #reassembles image by layering compressed RGB
    return np.clip(img_comp, 0, 255).astype(np.uint8) #ensures pixel values stay between 0 and 255
```

```
# Most accurate compression
def compress_image_float(img_path, k):
    img = np.array(Image.open(img_path).convert('RGB'), dtype=np.float64) #loads image and converts into RGB arrays
    R, G, B = img[:,:,0], img[:,:,1], img[:,:,2] # assigning the split into the 3 matrices
    R_compressed = compress_channel(R, k) # compresses each matrix
    G_compressed = compress_channel(G, k)
    B_compressed = compress_channel(B, k)
    img_comp = np.stack([R_compressed, G_compressed, B_compressed], axis=2)
```

```
#reassembles image by layering compressed RGB
    return img_comp
```

```
# Tells you how different the compressed image is fthe rom original image by
computing the difference in Frobenius norm
def reconstruction_error(original, compressed):
    return np.linalg.norm(original - compressed).round(3)
```

```
# Lists reconstruction error between the original and compressed images for a given
number of singular values kept
k_values = [5, 100, 500, 1067]
errors = []

for k in k_values:
    compressed = compress_image_float("wildflower.jpg", k)
    err = reconstruction_error(original_img, compressed)
    errors.append(err)
    print(f"k = {k}, Reconstruction error = {err:.2f}")
```

```
# Calculates the maximum k value for an image
height, width, _ = original_img.shape
k_max = min(height, width)
print(f"Maximum possible k (no compression): {k_max}")
```

```
# Function to compress and visualize process, displays original, compressed R, G,
and B, and the reassembled compressed matrix
def compress_and_visualize(img_path, k):
    # Load image and split into RGB
    img = np.array(Image.open(img_path).convert('RGB'), dtype=np.float64)
    R, G, B = img[:,:,0], img[:,:,1], img[:,:,2]

    # Compress each channel
    R_comp = compress_channel(R, k)
    G_comp = compress_channel(G, k)
    B_comp = compress_channel(B, k)

    # Reconstruct final RGB image
    img_comp = np.stack([R_comp, G_comp, B_comp], axis=2)
    img_comp_clipped = np.clip(img_comp, 0, 255).astype(np.uint8)

    # Plotting
    fig, axs = plt.subplots(2, 3, figsize=(15, 10))

    # Original image
```

```
    axs[0, 0].imshow(img.astype(np.uint8))
    axs[0, 0].set_title('Original Image')
    axs[0, 0].axis('off')

    # Compressed R Image
    axs[0, 1].imshow(R_comp, cmap='Reds')
    axs[0, 1].set_title(f'Compressed Red Channel (k={k})')
    axs[0, 1].axis('off')

    # Compressed G Image
    axs[0, 2].imshow(G_comp, cmap='Greens')
    axs[0, 2].set_title(f'Compressed Green Channel (k={k})')
    axs[0, 2].axis('off')

    # Compressed B Image
    axs[1, 0].imshow(B_comp, cmap='Blues')
    axs[1, 0].set_title(f'Compressed Blue Channel (k={k})')
    axs[1, 0].axis('off')

    # Reconstructs compressed image
    axs[1, 1].imshow(img_comp_clipped)
    axs[1, 1].set_title('Reconstructed Compressed Image')
    axs[1, 1].axis('off')

    axs[1, 2].axis('off')

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

```
# Calculates a standardized difference in Frobenius norm, standardized by the
number of total pixels in the image
# Tells you how different, on average, each pixel is from the original
def standard_norm_diff(img_path, k):
    compressed_image = compress_image_float(img_path, k)
    original_image = np.array(Image.open(img_path).convert("RGB"),
dtype=np.float64)

    # Finds maximum number of singular values and pixels for a given image
    height, width, _ = original_image.shape
    k_max = min(height, width)
    num_pixels = height*width

    # This is the output, the standardized difference in Frobenius norm
    avg_diff = (reconstruction_error(original_image, compressed_image) /
num_pixels)
```

```
    return avg_diff

    # print(f"Normalized absolute reconstruction error per pixel is {avg_diff} for
k={k}, compared to the original which has k={k_max}.")
```

```
# Plots normalized reconstruction error per pixel vs singular values retained
def plot_norm_diff(img_path, num_k_values):
    original_image = np.array(Image.open(img_path).convert("RGB"),
dtype=np.float64)

    # Calculates maximum number of singular values
    height, width, _ = original_image.shape
    k_max = min(height, width)

    # Defines vectors for plotting, with num_k_values being how many points are
plotted
    k_values = np.linspace(1, k_max, num_k_values, dtype=int)
    std_norms = [standard_norm_diff(img_path, k) for k in k_values]

    # Plotting
    plt.figure(figsize=(10, 6))
    plt.plot(k_values, std_norms, marker='o', linestyle='-')
    plt.annotate(f"max k = {k_values[-1]}",
             xy=(k_values[-1], std_norms[-1]),
             xytext=(k_values[-1] - 250, std_norms[-1] + 0.01),
             arrowprops=dict(facecolor='black', arrowstyle='->'),
             fontsize=20,
             color='red', fontname="Times New Roman")
    plt.xlabel("k (Number of Singular Values)", fontsize=20, fontname="Times New
Roman")
    plt.ylabel("Normalized Reconstruction Error", fontsize=20, fontname="Times New
Roman")
    plt.title("Normalized Absolute Reconstruction Error per Pixel vs. k",
fontsize=20, fontname="Times New Roman")
    plt.xticks(fontsize=20, fontname="Times New Roman")
    plt.yticks(fontsize=20, fontname="Times New Roman")
    plt.grid(True)
    plt.show()
```

```
# Sets default font globally for plot format consistency
plt.rcParams['font.family'] = 'Times New Roman'
plt.rcParams['font.size'] = 20

# Same as last plot except the line is plotted against an idealized compressed size
in RAM memory
def plot_norm_diff_and_size(img_path, num_k_values):
```

```
    original_image = np.array(Image.open(img_path).convert("RGB"),
dtype=np.float64)

    height, width, _ = original_image.shape
    k_max = min(height, width)

    original_size_bytes = original_image.nbytes  # Raw original size (in bytes)
    original_size_MB = original_size_bytes / 1e6 # Raw original size (in MB)

    k_values = np.linspace(1, k_max, num_k_values, dtype=int)

    # Computes normalized errors
    std_norms = [standard_norm_diff(img_path, k) for k in k_values]

    # Idealized compressed size is proportional to k based on the number of bytes
(RAM)
    compressed_sizes_MB = [(k / k_max) * original_size_MB for k in k_values]

    fig, ax1 = plt.subplots(figsize=(10, 6))

    # Axis 1 - left side
    ax1.set_xlabel("k (Number of Singular Values)")
    ax1.set_ylabel("Normalized Reconstruction Error", color="black")
    ax1.plot(k_values, std_norms, marker='o', linestyle='-', color="blue",
label="Reconstruction Error")
    ax1.grid(True)

    ax1.annotate(f"max k = {k_values[-1]}",
                 xy=(k_values[-1], std_norms[-1]),
                 xytext=(k_values[-1] - 250, std_norms[-1] + 0.01),
                 arrowprops=dict(facecolor='black', arrowstyle='->'),
                 color='red')

    # Axis 2 - right side
    ax2 = ax1.twinx()

    ax2.set_ylabel("Idealized Compressed Size (MB)", color='tab:green')

    # Plots compressed size proportional to k
    ax2.plot(k_values, compressed_sizes_MB, marker='x', linestyle='--',
color='tab:green', label="Compressed Size")

    plt.title(f"Normalized Reconstruction Error and Idealized Compressed Size vs.
k\nOriginal Image Size: {original_size_MB:.2f} MB")
    fig.tight_layout()

    plt.show()
```

```
# Not used in paper, but this plots 1 minus the reconstruction error to illustrate
the idealized trade-off between image quality and image size
def plot_norm_diff_and_size_nbytes(img_path, num_k_values):

    original_image = np.array(Image.open(img_path).convert("RGB"),
dtype=np.float64)
    height, width, _ = original_image.shape
    k_max = min(height, width)
    k_values = np.linspace(1, k_max, num_k_values, dtype=int)

    # Compute SVDs for each channel
    R, G, B = original_image[:,:,0], original_image[:,:,1], original_image[:,:,2]
    UR, SR, VTR = np.linalg.svd(R, full_matrices=False)
    UG, SG, VTG = np.linalg.svd(G, full_matrices=False)
    UB, SB, VTB = np.linalg.svd(B, full_matrices=False)

    std_norms = []
    compressed_sizes = []

    for k in k_values:
        # Compressed matrices
        Pr, Sr, Qr = UR[:, :k], SR[:k], VTR[:k, :]
        Pg, Sg, Qg = UG[:, :k], SG[:k], VTG[:k, :]
        Pb, Sb, Qb = UB[:, :k], SB[:k], VTB[:k, :]

        # Size in RAM memory (nbytes)
        size_r = Pr.nbytes + Sr.nbytes + Qr.nbytes
        size_g = Pg.nbytes + Sg.nbytes + Qg.nbytes
        size_b = Pb.nbytes + Sb.nbytes + Qb.nbytes

        total_size_bytes = size_r + size_g + size_b

        total_size_mb = total_size_bytes / 1e6

        compressed_sizes.append(total_size_mb)

        # Standard normalized error
        std_norms.append(standard_norm_diff(img_path, k))

    # Plots k vs. error
    fig, ax1 = plt.subplots(figsize=(10, 6))

    # Plotting
    color = 'tab:blue'
    ax1.set_xlabel("k (Number of Singular Values)")
    ax1.set_ylabel("1 - Normalized Reconstruction Error", color=color)
```

```
    ax1.plot(k_values, [1 - e for e in std_norms], marker='o', linestyle='-',
color=color)
    ax1.tick_params(axis='y', labelcolor=color)
    ax1.grid(True)

    # This part just adds an arrow to note max k (original image)
    ax1.annotate(f"max k = {k_values[-1]}",
                 xy=(k_values[-1], 1 - std_norms[-1]),
                 xytext=(k_values[-1] - 100, 1 - std_norms[-1] + 0.01),
                 arrowprops=dict(facecolor='black', arrowstyle='->'),
                 fontsize=10,
                 color='red')

    # Adds second y-axis for idealized compressed size
    ax2 = ax1.twinx()

    color = 'tab:green'
    ax2.set_ylabel("Idealized Compressed Size (MB)", color=color)
    ax2.plot(k_values, compressed_sizes, marker='x', linestyle='--', color=color)
    ax2.tick_params(axis='y', labelcolor=color)

    plt.title("Normalized Reconstruction Error and Idealized Compressed Size (MB)
vs. k")
    fig.tight_layout()
    plt.show()
```

```
# Defines vectors of image sizes for 15 evenly spaced k values
k_values2 = k_values = np.linspace(1, k_max, 15, dtype=int)
sizes = []
for k in k_values2:
    compressed_k = compress_image('wildflower.jpg', k)
    Image.fromarray(compressed_k).save(f"compressed_{k}.jpg", quality = 85)
    sizes.append(os.path.getsize(f"compressed_{k}.jpg") / 1e6)

print(sizes)
```

```
# Plots JPEG file size versus the number of singular values kept
original_size_MB = os.path.getsize("compressed_1067.jpg") / 1e6

plt.plot(k_values2, sizes)
plt.axhline(y=original_size_MB, color='red', linestyle='--', label=f"Original Size:
{original_size_MB:.2f} MB")
plt.text(x=k_values2[-1]*0.6, y=original_size_MB - 0.04,
         s=f"Original Size: {original_size_MB:.2f} MB",
         color='red', fontsize=14) # adds annotation to plot for y intercept
plt.grid(True)
```

```
plt.xlabel("k (Number of Singular Values)")
plt.ylabel("JPEG File Size (MB)")
plt.title("Observed File Size (MB) vs. k")
plt.show()
```

```
#Optimized function to display normalized error
def precompute_svd(img_path):
    img = np.array(Image.open(img_path).convert("RGB"), dtype=np.float64)
    R, G, B = img[:,:,0], img[:,:,1], img[:,:,2]

    # Full SVD for each color
    U_r, S_r, Vt_r = np.linalg.svd(R, full_matrices=False)
    U_g, S_g, Vt_g = np.linalg.svd(G, full_matrices=False)
    U_b, S_b, Vt_b = np.linalg.svd(B, full_matrices=False)

    return (U_r, S_r, Vt_r), (U_g, S_g, Vt_g), (U_b, S_b, Vt_b), img.shape
```

```
# More efficient compression and reconstruction function
def reconstruct_from_svd(svd_channels, shape, k):
    (U_r, S_r, Vt_r), (U_g, S_g, Vt_g), (U_b, S_b, Vt_b) = svd_channels
    h, w, _ = shape

    # Keeps top k singular values
    R = (U_r[:, :k] @ np.diag(S_r[:k]) @ Vt_r[:k, :])
    G = (U_g[:, :k] @ np.diag(S_g[:k]) @ Vt_g[:k, :])
    B = (U_b[:, :k] @ np.diag(S_b[:k]) @ Vt_b[:k, :])

    # Stacks RGB back together
    img_comp = np.stack([R, G, B], axis=2)
    return np.clip(img_comp, 0, 255).astype(np.uint8)
```

```
# More efficient standardized norm difference function
def fast_standard_norm_diff(original_img, compressed_img):
    error = np.linalg.norm(original_img - compressed_img)
    return (error / (original_img.shape[0] * original_img.shape[1])).round(3)
```

```
# Three new image paths
three_img_paths = ["gauss.jpg", "butterfly.jpg", "strawberries.jpg"]

# Uses a nested for loop to plot the three lines representing the relationship
between the percentage of k-values retained and the normalized reconstruction error

def plot_reconstruction_comparison(img_paths, num_k_values=10): # Inputs: list of
file strings, number of partitions of total k values
```

```
    plt.figure(figsize=(10,6))
    colors = ['brown', 'blue', 'red']

    for idx, img_path in enumerate(img_paths):

        # Loads image
        original_image = np.array(Image.open(img_path).convert("RGB"),
dtype=np.float64)

        # Precomputes full SVD once
        svd_r, svd_g, svd_b, shape = precompute_svd(img_path)
        svd_channels = (svd_r, svd_g, svd_b)

        # Calculates maximum number of singular values for each image
        height, width, _ = shape
        k_max = min(height, width)

        k_values = np.linspace(1, k_max, num_k_values, dtype=int)

        errors = []

        # Creates error vector
        for k in k_values:
            compressed = reconstruct_from_svd(svd_channels, shape, k)
            error = fast_standard_norm_diff(original_image, compressed)
            errors.append(error)

        # Converts k to percentages (standardizes) for comparison of images of
different sizes
        k_percentages = (k_values / k_max) * 100

        # Plotting for each image
        plt.plot(k_percentages, errors, marker='o', linestyle='-',
                 label=f"Image {idx+1}", color=colors[idx % len(colors)])

    # Plotting
    plt.xlabel("Percentage of Singular Values Retained")
    plt.ylabel("Normalized Reconstruction Error")
    plt.title("Comparison of Normalized Reconstruction Error vs %k for Different
Images")
    plt.grid(True)
    plt.xticks(np.linspace(0, 100, 11))
    plt.legend()
    plt.show()
```