University of Colorado - Boulder

APPM 3310

Matrix Methods — Fall 2024

Final Report - SVD Image Compression

Sam Walker
Pete Walker

Ian Grooms
Section: 002 (11:15-12:05)

Wednesday, December 11, 2024

College of Engineering & Applied Science
UNIVERSITY OF COLORADO BOULDER

Contents

Abstract

Singular Value Decomposition (SVD) is a powerful mathematical tool used in various fields for data analysis, dimensionality reduction, and image compression. In this report, we explore the application of SVD for compressing images by isolating and retaining the most significant information in RGB image matrices while discarding less essential components. Although SVD initially increases storage requirements by decomposing each color channel into three matrices (left singular, singular values, and right singular), the technique enables significant size reduction by approximating the original image with fewer singular values. Our study compares SVD-based compression with conventional image formats such as BMP, PNG, and JPG to evaluate its efficiency and effectiveness. Beyond image compression, SVD has numerous applications in fields like natural language processing, facial recognition, and noise reduction, Murthy, 2020. The results demonstrate the trade-offs between compression quality and data storage, offering insights into SVD's potential in image processing and beyond.

Attribution

Throughout the project, Pete Walker performed all mathematical derivations, while Sam Walker implemented the coding for the project. Both contributed equally to conducting research, preparing the report, and collaboratively communicating ideas throughout the project.

Introduction

Singular Value Decomposition (SVD) is a fundamental concept in linear algebra, widely used for analyzing and simplifying data in various applications. SVD decomposes a matrix into three components: the left singular matrix, the singular value matrix, and the right singular matrix. This decomposition reveals the underlying structure of the data and allows for efficient manipulation, such as dimensionality reduction and noise filtering.

In this report, we focus on applying SVD to image compression. Digital images are typically represented as matrices of pixel values for each of the three color channels: red, green, and blue (RGB). By applying SVD to these RGB matrices, we can isolate the most critical singular values, which contain the majority of the image's information. By discarding the less significant singular values, we can approximate the original image with reduced storage requirements, albeit with some loss in quality. However, it is important to note that SVD initially increases storage size by transforming the original three RGB matrices into nine matrices—three for each color channel.

Our goal is to evaluate SVD-based compression by comparing it with standard image compression methods such as BMP, PNG, and JPG. The following sections will delve into the methodology, results, and conclusions of our investigation into SVD-based image compression.

Understanding SVD

Performing SVD Example

We found the following example online for our own guidance through SVD, "SVD Example", n.d.:
Given the matrix A:

$$A = \begin{bmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{bmatrix}$$

We want to find the SVD decomposition of the above matrix, which will have the form $A = U\Sigma V^T$, where U and V^T are both orthogonal matrices, and Σ is a diagonal matrix containing the singular values of A.

We start by finding the eigenvalues of AA^T

$$AA^T = \begin{bmatrix} 17 & 8 \\ 8 & 17 \end{bmatrix}$$

So now:

$$(17-\lambda)^2 - 64 = 0$$
$$(\lambda - 9)(\lambda - 25) = 0$$

And the singular values are the square roots of the eigenvalues of this AA^T matrix, so we now have $\sigma_1 = 5$ and $\sigma_2 = 3$. Note that by convention, we order the singular values by size here, with the largest first. This will become relevant later when we start using the SVD to reduce the amount of data involved in storage.

With the singular values, we now have Σ:

$$\Sigma = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{bmatrix}$$

The next step in getting the full SVD is to get the right singular vectors, (the orthonormal vectors of A^TA) which are obtained by finding the orthonormal vectors A^TA. See Appendix A.1.1 for this full process.

This returns the following matrix:

$$V^T = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{18}} & \frac{2}{3} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{18}} & \frac{-2}{3} \\ 0 & \frac{4}{\sqrt{18}} & \frac{-1}{3} \end{bmatrix}$$

Now, we just need to find the left singular vectors. At this point, these can be found using the following formula:

$$u_i = \tfrac{1}{\sigma} A v_i$$

This then yields the following:

$$U = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix}$$

And now we have the SVD: $A = U\Sigma V^T$, or in its full glory:

$$A = \begin{bmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{18}} & \frac{2}{3} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{18}} & \frac{-2}{3} \\ 0 & \frac{4}{\sqrt{18}} & \frac{-1}{3} \end{bmatrix}$$

Programming SVD

For this part of the project, we developed our own implementation of the Singular Value Decomposition (SVD) function, drawing inspiration from the example outlined in the previous section. The custom SVD function is detailed in Appendix A.3.1. After implementing the function, we compared its input and output to the results produced by NumPy's built-in `linalg.svd` function, using the code provided in Appendix A.3.4.

While our implementation closely follows the theoretical steps of SVD, some modifications were made to account for computational limitations and efficiencies. In theoretical SVD, singular values (σ_i) are derived by solving for the eigenvalues of AA^T or A^TA, which involve determinant calculations. However, direct computation of determinants or eigenvalues for large matrices can be computationally expensive and numerically unstable, especially for high-dimensional data. To address this, our implementation avoids direct determinant calculations and instead relies on NumPy's `np.linalg.eig` function to compute eigenvalues and eigenvectors efficiently.

Another consideration is the sorting and normalization of singular values and their corresponding singular vectors. Our code explicitly sorts the singular values in descending order using NumPy's `argsort` function and reorders the columns of the U and V matrices accordingly. This step ensures consistency in the output format and aligns with the results of NumPy's `linalg.svd` function.

Additionally, the computation of the left singular vectors (U) leverages the relationship $U = \frac{1}{\sigma_i}Av_i$, where v_i is a right singular vector. This avoids directly constructing U from the eigenvectors of AA^T, which would require additional computation. The normalization of the singular vectors further ensures that the results conform to the expected properties of orthogonal matrices.

The comparison between our custom implementation and NumPy's built-in `linalg.svd` function showed that both methods produced identical results in terms of accuracy. However, our implementation was approximately five times slower than NumPy's version, as NumPy leverages optimized low-level libraries (such as LAPACK) for linear algebra computations. Despite the performance difference, the correctness of our implementation demonstrates a solid understanding of the underlying mathematics and how computational workarounds can be used to achieve the same results efficiently.

Given these results, we felt confident in leveraging NumPy's `linalg.svd` function for the remainder of the project. This allowed us to focus on analyzing the results of SVD rather than the computational overhead of performing the decomposition itself.

Small Singular Value Removal

Now that we better understand SVD, we can move on to the aspects of SVD that make it useful for data compression. Namely, due to the way that SVD recreates the original matrix, if small singular values are assumed to be 0, the overall effect on the original matrix is minimal. We will examine this here:

Let us look at the singular value matrix we found above: The smallest singular value here was $\sigma_2 = 3$, which was not much smaller than the largest, $\sigma_1 = 5$. However, we can still analyze the formatting after setting $\sigma_2 = 0$:

$$\sigma_2 = 0 \rightarrow \Sigma' = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

And now, $A' = U\Sigma' V^T$

Where A' is used to denote the fact that we are now approximating the original matrix A, using less data to do so. In the expanded form, the prime symbol goes on the Σ matrix to denote the same thing.

$$A' = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 5 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{18}} & \frac{2}{3} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{18}} & \frac{-2}{3} \\ 0 & \frac{4}{\sqrt{18}} & \frac{-1}{3} \end{bmatrix}$$

But now, the strength of removing small singular values can be seen: many of the entries in U and V^T can be clearly seen to only ever multiply by zero, and therefore are useless. In fact, the construction of A' can be entirely rewritten, simplified down to:

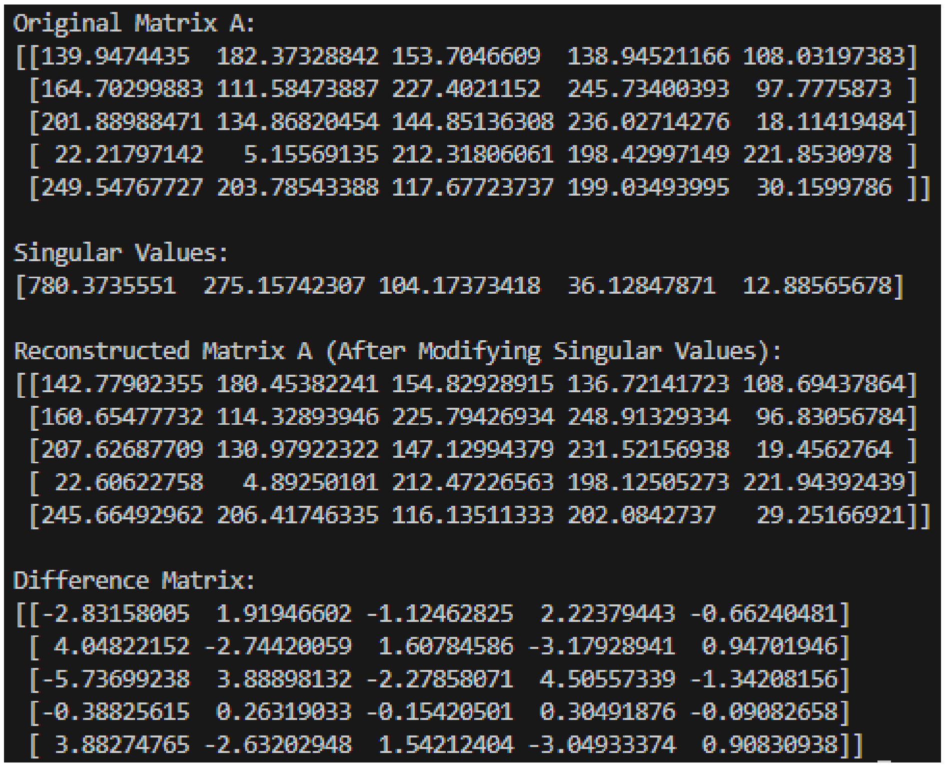
$$A' = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 5 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{18}} & \frac{2}{3} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{18}} & \frac{-2}{3} \end{bmatrix}$$

In doing so, we are able to attempt to recreate the true A matrix, although with some significant error in this case, due to the relatively large size of the singular value we set to zero. We can now compare these two matrices, A and A':

$$A = \begin{bmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{bmatrix}$$
$$A' = \begin{bmatrix} 2.5 & 2.5 & 0 \\ 2.5 & 2.5 & 0 \end{bmatrix}$$

Clearly, the amount of error here makes this almost useless in comparison.

We will now examine small value removal in a more helpful scenario. By creating a random 5x5 matrix with values from 0 to 255 (roughly simulating a third of an RGB matrix), we will see that the last singular value here is actually significantly less than the largest of the singular values.



```
Original Matrix A:
[[139.9474435  182.37328842 153.7046609  138.94521166 108.03197383]
 [164.70299883 111.58473887 227.4021152  245.73400393  97.7775873 ]
 [201.88988471 134.86820454 144.85136308 236.02714276  18.11419484]
 [ 22.21797142   5.15569135 212.31806061 198.42997149 221.8530978 ]
 [249.54767727 203.78543388 117.67723737 199.03493995  30.1599786 ]]

Singular Values:
[780.3735551  275.15742307 104.17373418  36.12847871  12.88565678]

Reconstructed Matrix A (After Modifying Singular Values):
[[142.77902355 180.45382241 154.82928915 136.72141723 108.69437864]
 [160.65477732 114.32893946 225.79426934 248.91329334  96.83056784]
 [207.62687709 130.97922322 147.12994379 231.52156938  19.4562764 ]
 [ 22.60622758   4.89250101 212.47226563 198.12505273 221.94392439]
 [245.66492962 206.41746335 116.13511333 202.0842737   29.25166921]]

Difference Matrix:
[[-2.83158005  1.91946602 -1.12462825  2.22379443 -0.66240481]
 [ 4.04822152 -2.74420059  1.60784586 -3.17928941  0.94701946]
 [-5.73699238  3.88898132 -2.27858071  4.50557339 -1.34208156]
 [-0.38825615  0.26319033 -0.15420501  0.30491876 -0.09082658]
 [ 3.88274765 -2.63202948  1.54212404 -3.04933374  0.90830938]]
```

Figure 1 SVD Reconstruction example from Appendix A.3.2

Here, the largest singular value is 780, whereas the smallest is around 12.9. The ratio between these singular values is around 60:1, whereas in the previous example, it was 5:3. Let us now examine the effect this lower ratio has on the accuracy of A', labeled as the difference matrix (A - A') in Figure 1:

The maximum difference was just over 5.7, with over half of the difference values being less than 2 in magnitude. Seeing as how the values ranged from 0 to 255, these are relatively small changes caused by removing one of the singular values.

This example demonstrates the power of SVD when it comes to figuring out how to determine which of the data is the least important in reconstructing the original matrix, A.

Singular Value Removal Effect on File Size

We will now examine the relation between the amount of data in A, and the amount of data required to store A using k singular values in SVD. As previously mentioned, because SVD turns one matrix into 3, without reducing some of the singular values to 0 it actually is more data intensive. For the sake of the math in this section, we will just consider the amount of non-zero numbers that are required to form the matrices.

To start, let us assume A is an m x n matrix that we will then apply SVD to. From this, the formatting of SVD will cause us to have 3 matrices instead of 1, where if $A = U\Sigma V^T$, then U is m x m, Σ is m x n, and V^T is n x n.

From this, it can be seen that, assuming we keep every single value in the matrix, we now have:

$$T(U\Sigma V^T) = mm + mn + nn$$

Where T(x) is an arbitrary function, taking in some matrix, and outputting the quantity of relevant numbers contained.

This is clearly much more than just storing the entirety of A ($T(A) = mn$), and so we are starting from behind in terms of saving on data with SVD.

However, we already have some ways in which we can save data. To start, we do not need to keep m x n values for Σ, and instead can just keep the rank of the matrix, because all of the off diagonal values are zero. Now:

$$\text{a} \equiv \text{min(m, n)} \rightarrow T(\Sigma) = a$$
$$\text{T}(\text{U}\Sigma V^T) = (ma + a + n)$$

This $T(U\Sigma V^T)$ equation works because a equals either m or n, and then for the larger of the two, many entries of that matrix are inevitably going to become irrelevant, due to the Σ matrix being filled with zeroes. Look at a 2 x 5 matrix for example:

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 & 0 \end{bmatrix}$$

Here, clearly the U matrix will be 2 x 2, and so will not be reduced at all, since a = 2, so 2 x a = 2 x 2. In comparison, the V^T matrix will be 5 x 5. However, since all of the entries in the last 3 columns of Σ are zero, none of the entries in the last 3 rows of V^T will be multiplied by anything but zero. Therefore, if we are going to store all of the values that actually matter in V^T, we will only need each of the values in the first 2 rows, which is then 5(2) = n(a) = 10 numbers to store. Likewise, this works in reverse for the m ¿ a case, since all of the values in the later columns will then only be multiplied by zeroes in the bottom of the Σ matrix. This logic is how the equation for $T(U\Sigma V^T)$ above was derived, with the knowledge that by knowing the necessary shape of all matrices, and which portions of them must be filled by zeroes (the off diagonal elements of Σ and either the right side of U or the bottom of V^T will be filled with zeroes).

Now, we need to consider the case of reducing the number of singular values by setting any number of them to 0. We will define k as the number of singular values used, k $\leq$ a.

We will now have Σ' of the form (for k = 3 in this case):

$$\Sigma' = \begin{bmatrix} \sigma_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Therefore, from this we will now only have k non-zero values from the Σ matrix to store, $m \cdot k$ values from the U matrix, and $n \cdot k$ values from the V^T matrix by similar logic as above. This gives the following equation for T:

$$T(U\Sigma' V^T) = km + k + kn = k(m + n + 1)$$

This result is the same as what we saw from Baumann, 2018. From this, we can then make the interesting comparison of $T(A)$ to $T(A')$, in particular for the n $\approx$ m >> 1 case:

$$\frac{T(A')}{T(A)} = \frac{k(n+m+1)}{nm} \rightarrow \frac{2k}{n}$$

This shows that, roughly speaking, you will need to remove at least half of the singular values in order to start saving on file size versus just storing the A matrix in its entirety.

In the n >> m case:

$$\frac{k(n+m+1)}{nm} \rightarrow \frac{k}{m}$$

Here, you need to have roughly m singular values to begin saving on file size. Because you start with only as many singular values as the minimum of m and n, for every singular value you remove you save on more data than in the m $\approx$ n case.

The conclusion here is that generally speaking, in order to catch up to just storing the matrix A normally, you have to remove up to roughly $\frac{n}{2}$ singular values in the worst case scenario of A being square (n x n), and having to remove very few singular values in the best case scenario, where m and n are very far apart.

Our Implementation

Understanding File Size

To understand file sizes in the context of image storage, it's important to recognize that images are easily represented as RGB matrices, with each pixel value ranging from 0 to 255. This range corresponds to $2^8 = 256$, meaning each pixel value requires 8 bits, or 1 byte, of storage. Since each image has three color channels (Red, Green, and Blue), the total size of the image in bytes can be calculated as:

$$\text{Total Bytes} = \text{Height} \times \text{Width} \times 3$$

Additionally, note that $1024\,\text{bytes} = 1\,\text{kilobyte (KB)}$, not $1000\,\text{bytes}$. For example, for a 10×15 pixel image, the raw file size is:

$$10 \times 15 \times 3 = 450\,\text{bytes}, \text{ or approximately } 0.439\,\text{KB}.$$

File Types

Throughout the later sections of this report different versions of image file types will become relevant.

The most important file type to this project is the .bmp image file. This stores the entire data of the image with an R, G, and B value from 0 to 255 for each pixel. This is a form of lossless compression, where all of the data is preserved, at the cost of a larger file size. Because of the lack of compression involved in .bmp files, they are typically not used in any everyday life context. However, they are useful for their raw data in the context of this project.

Next, .png image files are another, more advanced version of lossless compression. Similar to .bmp files, they will preserve every last bit of integrity of the image, although they will also reduce the file size significantly while doing so. This is possible through complicated algorithims for the case of .png files. However, lossless compression in general can be understood in the context of an image that is entirely white on one side, and entirely black on the other - there is no need to store the color for each pixel in the image, when instead it can be much more concisely stated that this half of the image is black, and the other white. In this vein, .png files will compress images without undergoing any loss, at the cost of slightly higher file sizes.

The last commonly seen file type we will compare to are .jpeg image files. These image files are known as lossy, because the compression involved ends up losing some of the image quality throughout its process. However, this also means that generally speaking, .jpeg image files are able to achieve higher compression ratios, which is often desirable for the almost unnoticeable amount of loss in quality.

Compared to all of these advanced image files, SVD is the worst of both worlds - it is always lossy as soon as you remove even just the smallest singular value. Additionally, because it starts out at even higher file sizes than .bmp, it is required to lose a lot of data before the file sizes can be comparable to the more advanced file types such as .jpeg or .png files.

Calculating File Size Programmatically

To programmatically calculate the file size after applying SVD-based compression, the following Python script was used (see Appendix A.3.3). The script calculates the total data required to store an image using a reduced number of singular values for each channel. The formula accounts for the dimensions of the U, Σ, and V^T matrices:

- $U : m \times k$ (Height × Number of Singular Values)
- $\Sigma : k$ (Number of Singular Values)
- $V^T : k \times n$ (Number of Singular Values × Width)

For instance, using a 10×15 pixel image, the program calculates the data size as shown in the output table (Figure 2). For comparison, storing the same image in raw format without compression requires $10 \times 15 \times 3 = 450$ bytes, or approximately 0.439 KB. From the 10 singular values produced by a matrix this size, we have to remove half of the singular values before we see any storage size benefit, this makes sense with our previous assumption of n/2 removal for an almost square matrix, the true benefits of SVD are much more apparent when m ¿¿ n or n ¿¿ m. This comparison illustrates how SVD compression can significantly impact storage requirements for the better, or for the worse, depending on the number of singular values retained.

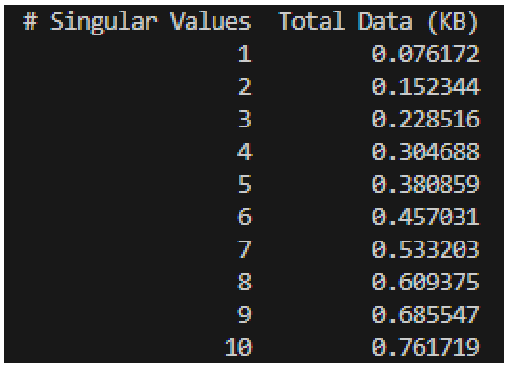

```
# Singular Values  Total Data (KB)
                1         0.076172
                2         0.152344
                3         0.228516
                4         0.304688
                5         0.380859
                6         0.457031
                7         0.533203
                8         0.609375
                9         0.685547
               10         0.761719
```

Figure 2 SVD Storage Analysis for a 10x15 Image

Kodak Image Set

The Kodak Image Set, which is frequently used for benchmarking image processing algorithms, consists of 768×512 resolution images. These images are valuable for evaluating performance because of their consistent quality and dimensions. The raw file size for such an image can be calculated as:

$$768 \times 512 \times 3 = 1,179,648 \,\text{bytes}, \text{ or approximately } 1152 \,\text{KB}.$$

In our benchmarks, we compare the raw file size with compressed formats such as BMP, PNG, and JPG, referencing benchmarks from Brocchi, n.d. The compression achieved with SVD is evaluated relative to these common formats. For our implementation, we downloaded the images from Tepid, 1999.

Performing Image Reconstruction with SVD

In this section, we demonstrate the process of image reconstruction using Singular Value Decomposition (SVD). For this paper, we will use Kodak Image 1 as the example. The script that performs the SVD-based image reconstruction can be found in Appendix A.3.5. This code allows us to specify the number of singular values to retain during reconstruction, enabling variable compression of the image.

The output from this process is illustrated in Figure 8 located at Appendix A.2.2. The output shows several critical pieces of information:

- The original BMP image, which has a file size of 1152 KB in this case.
- The compressed image, whose file size is dependent on the number of singular values retained.
- The number of singular values used for reconstruction.
- The percentage of singular values retained relative to the total.

As an example, the output image demonstrates the reconstruction of Kodak Image 1 using only 100 singular values. This showcases the balance between compression and image quality, as higher compression levels (fewer singular values) reduce the file size but may also lead to a loss of detail in the reconstructed image.

Results

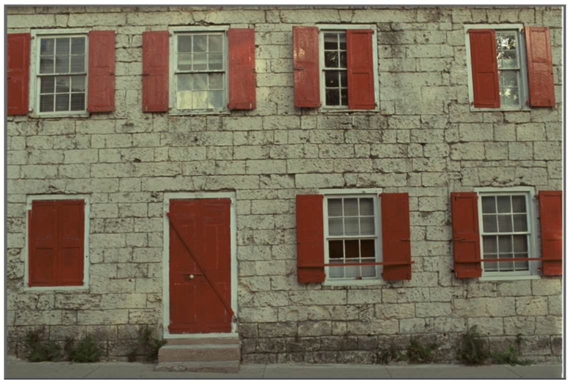
Across all of these categories, we will compare the image of a certain file type to our own compressed image at the same file size. Across the board, we expect to have worse quality, because SVD is a very elementary method of file compression. However, SVD in this format does have one strength - because we can easily dictate how many singular values we are using, we can very easily adjust the file size to as low as we want, including way lower than any automatic compression would ever normally go. The loss of image quality in these regimes is very obvious, but depending on application it could in theory be an acceptable trade off.

Additionally, it should be noted that in all of the below images, it is hard to say how much loss of quality has occurred in the transferring of these files into this pdf through overleaf. Because this is hard to quantify, much of the discussion relevant to this report will be about the qualitative aspects of the images, with the quantitative side of things mostly being relegated to the file sizes. In general though, it can be said that across the board, the SVD image is the worst quality at a given file size, and that this effect was more noticeable with the true image files to look at, instead of being forced to look through the lens of the pdf.

Comparing SVD to BMP

Here, it should be noted that the BMP image is actually a PNG, because both are lossless images, and overleaf does not support BMP image files.

It should be noted that 307 singular values were used to form this image.


(a) Original BMP Image (Kodak 1, 1152KB)

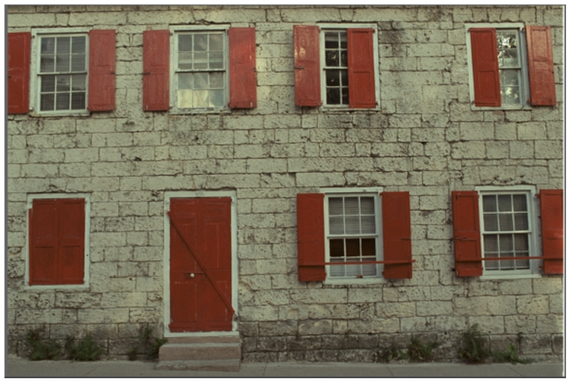
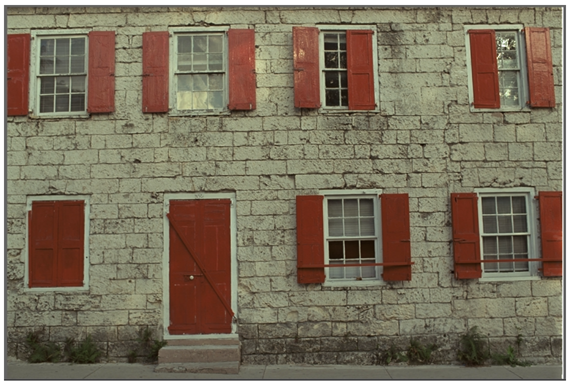

(b) SVD Compressed (Kodak 1, 1152KB)

Figure 3 Comparison of Original and Compressed BMP Images

These files are at an image size of 1,152 kB. At this size, they are over 60% larger than the commonly accepted lossless file type of PNG. It should be said, however, that at this size even the SVD image looks to be of comparable quality. Even still, slight problems do arise when looking closely at the SVD image - fine details, in particular any kind of line formed in the image is less distinct than it should be. This is the start of the effects of lossy compression.

Comparing SVD to PNG

In this image, the corresponding SVD image uses 192 singular values to get down to this file size of 719 kB.


(a) Original PNG Image (Kodak 1, 719KB)

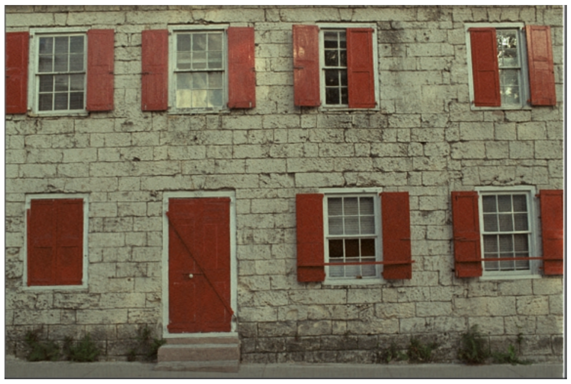
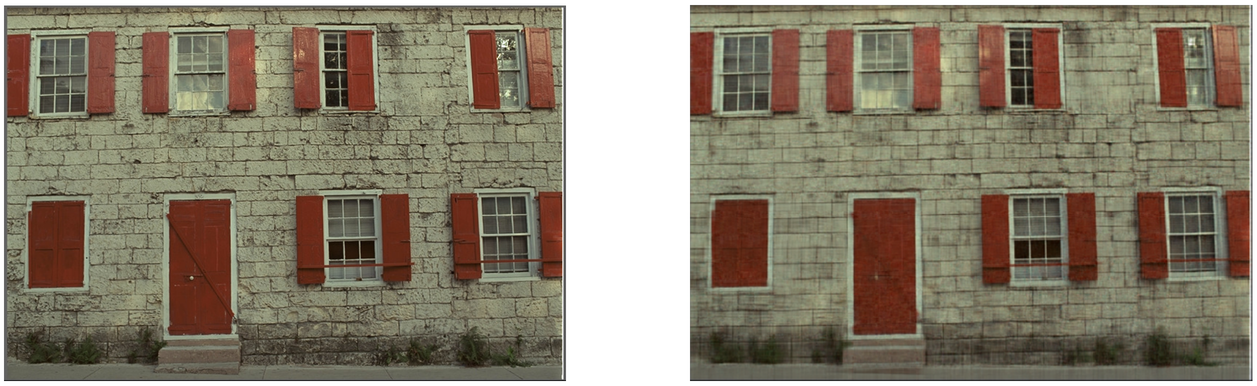

(b) SVD Compressed (Kodak 1, 719KB)

Figure 4 Comparison of Original and Compressed PNG Images

For this image, the issues that started popping up in the SVD image at 1,152 kB are even more pronounced in this 719 kB version. Every line between the bricks is blurrier and less pronounced than it is in the lossless version. On top of that, even the solid colors feel 'fuzzier', in particular the main door is clearly not quite right. The effects of losing more and more data to SVD is showing here, although of more interest is its manner of doing so: It is not like there is some overall color shift occurring, or that it generally seems too bright or too dark in comparison to the 'true' image, as much as it is the finer details and blending of colors which is being lost.

Comparing SVD to JPG

In this image, the corresponding SVD image uses 50 singular values to get down to this file size of 187 kB.



(a) Original JPG Image (Kodak 1, 187KB) **(b)** SVD Compressed (Kodak 1, 187KB)

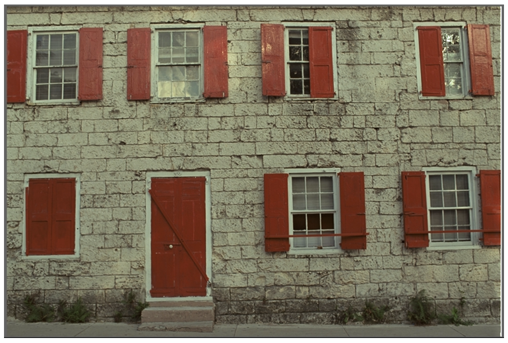
Figure 5 Comparison of Original and Compressed JPG Images

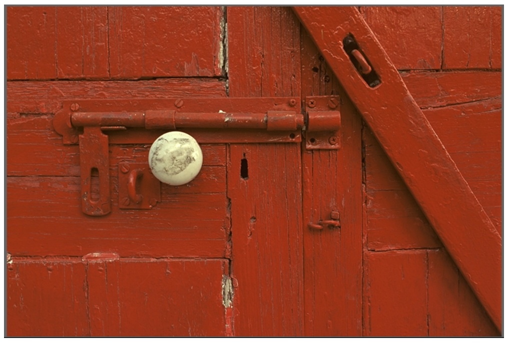
Notably in this comparison, despite the extremely small file size of the jpeg image, it is still of relatively high quality. When looking at the original image files, it was hard to find any obvious differences between this jpeg and the lossless alternatives. In contrast, at this file size the SVD image is at this point noticeably of poorer quality. The most obvious thing here is that the image appears to be getting 'streaky' in some ways, with lines of color being drawn out past where they seem to belong.

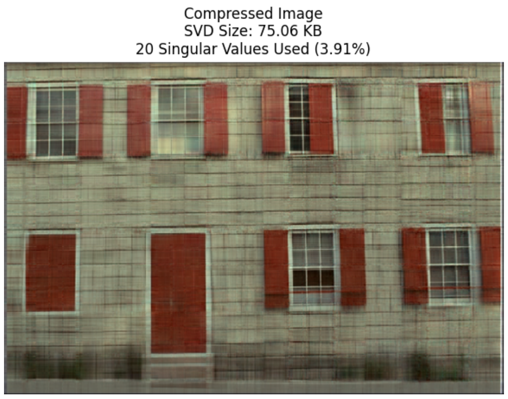
Limits of SVD Image Compression

An interesting fact about SVD compression is that the compression ratio is not directly correlated with the end quality of the image, and instead is entirely to do with the number of singular values being used. In this manner, how effective this compression is in terms of image quality has little to do with the arbitrary number of singular values used, and instead has more to do with the magnitude of the singular values kept vs the magnitude of those being dropped.

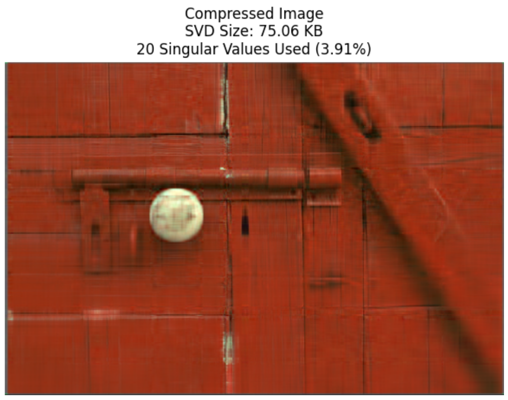
However, this section is more to talk about something we can do with SVD that is not automatically done with typical image files - getting the file size down to ridiculously small amounts.


(a) Uncompressed Kodak 1

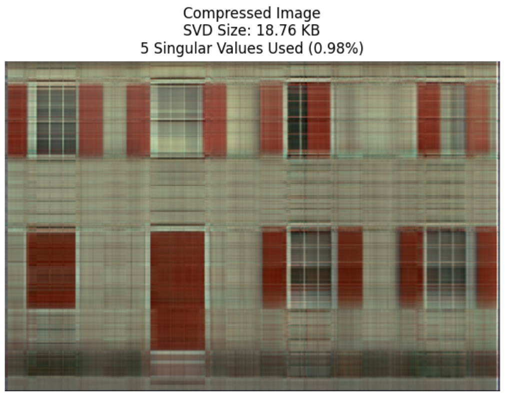

(b) Uncompressed Kodak 2



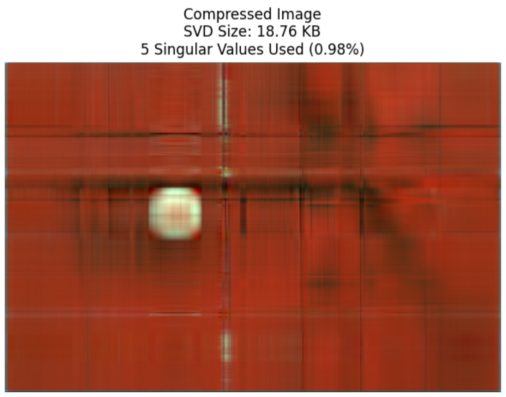
(c) SVD Compressed Kodak 1 (20 Singular Values)



(d) SVD Compressed Kodak 2 (20 Singular Values)



(e) SVD Compressed Kodak 1 (5 Singular Values)



(f) SVD Compressed Kodak 2 (5 Singular Values)

Figure 6 Comparison of Uncompressed and SVD Compressed Images for Kodak 1 and Kodak 2

These files above can be seen getting increasingly 'streaky' as the number of singular values used gets lower. However, even at these extremely quantities of singular values (5 out of the original 512), the pictures are still recognizable. While the streakiness of the images is making them increasingly unpractical, and use for practical purposes would require their fixing, the very low data usage here (compression ratio of close to 60:1) makes the loss in quality understandable.

Conclusion

Through this project, we discovered several key insights about the use of Singular Value Decomposition (SVD) for image compression. One major takeaway was that SVD performs better as a compression algorithm when applied to datasets that are not square. As outlined in the report, the point at which SVD becomes effective occurs much earlier in such cases, requiring fewer singular values to achieve meaningful compression. Additionally, SVD works more effectively on simpler images; detailed images tend to become blurry with significant compression, as evidenced by Chernyshev, 2023, where a mostly white cat maintained quality far better than more complex scenes.

Our results largely aligned with our expectations. We anticipated that a basic implementation of SVD would not outperform industry-standard formats like PNG and JPG, which are optimized for efficiency and quality. However, it was rewarding to create an image compression algorithm from scratch and observe its functionality. The ability to compress an image significantly while retaining its overall recognizability demonstrates the flexibility and utility of SVD as a tool for exploring data compression.

The results also provided a deeper appreciation for the complexity of existing compression methods. Comparing SVD to formats such as BMP, PNG, and JPG highlighted the lengths these formats go to in achieving lossless or high-quality lossy compression with remarkable efficiency. SVD served as a powerful lens through which we could understand the foundational principles of image compression, even if it is not a practical alternative for real-world applications.

In terms of future work, there is limited scope for improving SVD-based image compression as a viable method for real-world implementation. However, additional exploration into hybrid approaches, combining SVD with other compression techniques, could yield interesting results. In terms of furthering our understanding of SVD, it would be interesting to deep dive into what specifically causes artifacts in images when removing singular values in the decomposition. This would require analyzing possibly custom made images that isolate situations where SVD would either thrive or struggle with and explicitly analyzing the root cause of these artifacts.

References

Baumann, T. (2018). *Svd image compression demo.* https://timbaumann.info/svd-image-compression-demo/

Brocchi, S. (n.d.). *Kodak image benchmarking.* http://www.researchandtechnology.net/pcif/kodak_benchmarks.php?i=1

Chernyshev, D. (2023). *Svd image compression.* https://dmicz.github.io/machine-learning/svd-image-compression/

Murthy, M. (2020). *A beginner's guide to singular value decomposition (svd).* https://mukundh-murthy.medium.com/a-beginners-guide-to-singular-value-decomposition-svd-97581e080c11

Svd example. (n.d.). https://www.d.umn.edu/~mhampton/m4326svd_example.pdf

Tepid, R. (1999). *Kodak image set.* https://r0k.us/graphics/kodak/

A Appendix

A.1 Math

A.1.1 Finding Eigenvalues for SVD Example

First, find A^TA, and then row reduce for each of its eigenvalues:

$$A^TA = \begin{bmatrix} 13 & 12 & 2 \\ 12 & 13 & -2 \\ 2 & -2 & 8 \end{bmatrix}$$

This has eigenvalues of $\lambda = 25$, 9, and 0

$$A^TA - 25I = \begin{bmatrix} -12 & 12 & 2 \\ 12 & -12 & -2 \\ 2 & -2 & -17 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

This has a unit vector of $v_1 = (\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0)^T$
Similarly, for $\lambda = 9$:

$$A^TA - 9I = \begin{bmatrix} 4 & 12 & 2 \\ 12 & 4 & -2 \\ 2 & -2 & -1 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 6 & 1 \\ 0 & -16 & -4 \\ 0 & -2 & -1 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 6 & 1 \\ 0 & 4 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

This has a unit vector of $v_2 = (\frac{1}{\sqrt{18}}, \frac{-1}{\sqrt{18}}, \frac{4}{\sqrt{18}})^T$
Finally, v_3 can be found using orthognality to both v_1 and v_2 by solving the following equations:

$$v_1^T v_3 = 0$$
$$v_2^T v_3 = 0$$

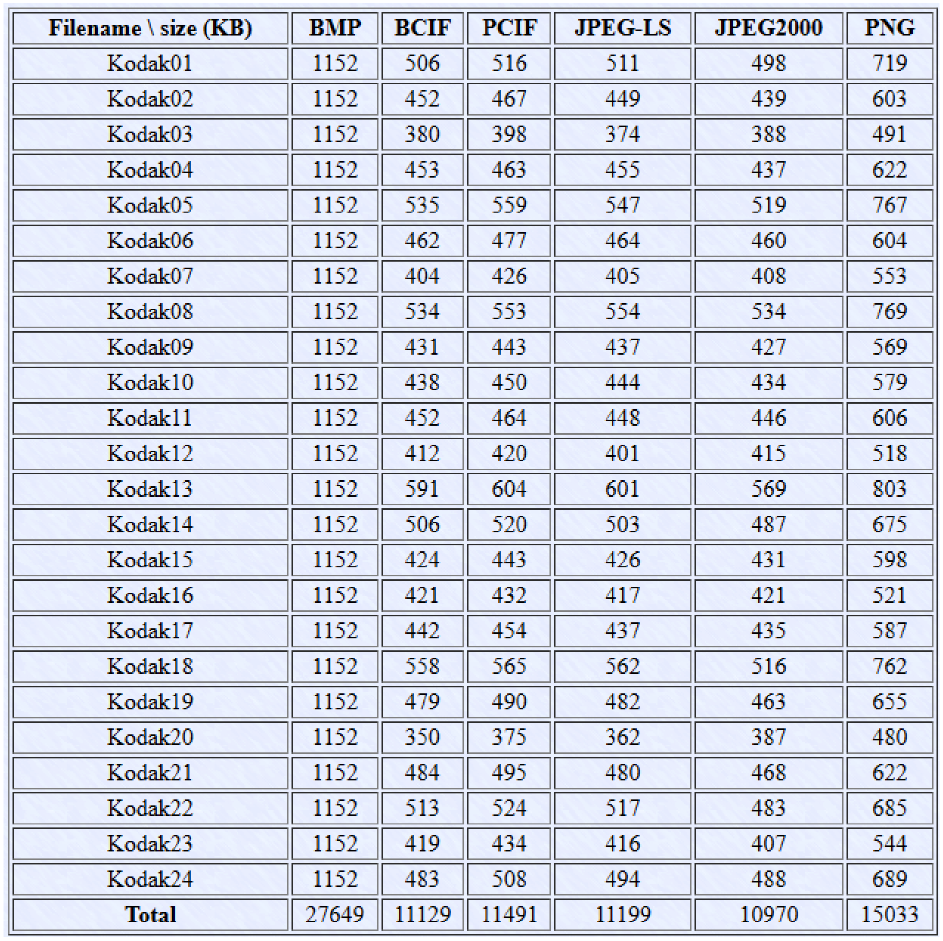
Define v_3 as (a, b, c)^T gives way to the system of equations:

$$a + b = 0$$
$$a - b + 4c = 0$$

Solving the system of equations then normalizing the vector yields $v_3 = (\frac{2}{3}, \frac{-2}{3}, \frac{-1}{3})^T$

A.2 Figures

A.2.1 Kodak Benchmark Numbers



Filename \ size (KB)	**BMP**	**BCIF**	**PCIF**	**JPEG-LS**	**JPEG2000**	**PNG**
Kodak01	1152	506	516	511	498	719
Kodak02	1152	452	467	449	439	603
Kodak03	1152	380	398	374	388	491
Kodak04	1152	453	463	455	437	622
Kodak05	1152	535	559	547	519	767
Kodak06	1152	462	477	464	460	604
Kodak07	1152	404	426	405	408	553
Kodak08	1152	534	553	554	534	769
Kodak09	1152	431	443	437	427	569
Kodak10	1152	438	450	444	434	579
Kodak11	1152	452	464	448	446	606
Kodak12	1152	412	420	401	415	518
Kodak13	1152	591	604	601	569	803
Kodak14	1152	506	520	503	487	675
Kodak15	1152	424	443	426	431	598
Kodak16	1152	421	432	417	421	521
Kodak17	1152	442	454	437	435	587
Kodak18	1152	558	565	562	516	762
Kodak19	1152	479	490	482	463	655
Kodak20	1152	350	375	362	387	480
Kodak21	1152	484	495	480	468	622
Kodak22	1152	513	524	517	483	685
Kodak23	1152	419	434	416	407	544
Kodak24	1152	483	508	494	488	689
Total	27649	11129	11491	11199	10970	15033

Figure 7 Kodak Image Benchmarking File Sizes per File Type

A.2.2 Image Reconstruction Example Output

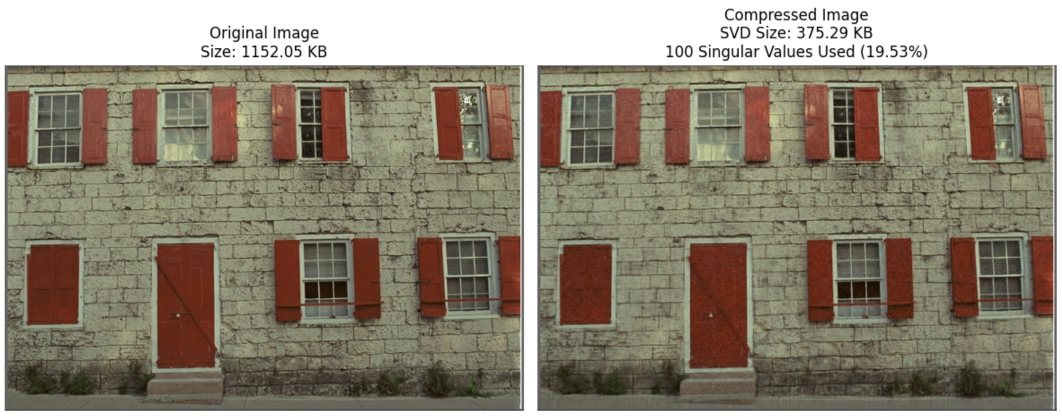


Figure 8 Image Reconstruction Example Output

A.3 Code

A.3.1 Custom SVD Implementation

```python
import numpy as np

def svd(A):
    # Step 1: Compute A * A^T to find the singular values (σi)
    AAT = np.dot(A, A.T)
    eigenvalues_AAT, eigenvectors_AAT = np.linalg.eig(AAT)
    singular_values = np.sqrt(eigenvalues_AAT)

    # Step 2: Compute AT * A to find the right singular vectors (columns of V)
    ATA = np.dot(A.T, A)
    eigenvalues_ATA, eigenvectors_ATA = np.linalg.eig(ATA)

    # Normalize eigenvectors of AT * A to form V (right singular vectors)
    V = eigenvectors_ATA / np.linalg.norm(eigenvectors_ATA, axis=0)

    # Step 3: Sort singular values and reorder U and V accordingly
    sorted_indices = np.argsort(singular_values)[::-1]  # Indices to sort in descending order
    singular_values = singular_values[sorted_indices]  # Sort singular values
    V = V[:, sorted_indices]  # Reorder columns of V
    eigenvectors_AAT = eigenvectors_AAT[:, sorted_indices]  # Reorder columns of U (AAT eigenvectors)

    # Step 4: Form Σ (diagonal matrix of singular values, same size as A)
    Σ = np.zeros_like(A, dtype=float)
    np.fill_diagonal(Σ, singular_values)

    # Step 5: Compute U (left singular vectors) using U = (1/σ) * A * v
    U = np.zeros((A.shape[0], A.shape[0]))
    for i in range(len(singular_values)):
        U[:, i] = np.dot(A, V[:, i]) / singular_values[i]

    # Normalize U
    U = U / np.linalg.norm(U, axis=0)

    # Verification: Reconstruct A from U, Σ, and V^T
    A_reconstructed = np.dot(U, np.dot(Σ, V.T))

    # Return results
    results = {
        "AAT": AAT,
        "Eigenvalues of AAT": eigenvalues_AAT,
        "Singular values": singular_values,
        "ATA": ATA,
        "Eigenvalues of ATA": eigenvalues_ATA,
        "Right singular vectors (V)": V,
        "Σ": Σ,
        "Left singular vectors (U)": U,
        "Reconstructed A": A_reconstructed
    }

    return results

if __name__ == "__main__":
    # Example usage
    A = np.random.rand(4, 4)  # Replace with any matrix
    results = svd(A)
    for key, value in results.items():
        print(f"{key}:\n{value}\n")
```

A.3.2 Reconstruction Example

```python
import numpy as np

# Example matrix A
np.random.seed(0)
A = 255*np.random.rand(5, 5)

# Step 1: Compute SVD decomposition using NumPy
U, S, Vt = np.linalg.svd(A)

# Print the original matrix A
print("Original Matrix A:")
print(A)

# Print the singular values
print("\nSingular Values:")
print(S)

# Step 2: Modify the singular values by setting the smallest to 0
S_modified = S.copy()
S_modified[-1] = 0  # Set the smallest singular value to 0

# Step 3: Reconstruct the matrix with modified singular values
Σ_modified = np.zeros_like(A, dtype=float)  # Create a Σ matrix of the same size as A
np.fill_diagonal(Σ_modified, S_modified)  # Place modified singular values on the diagonal
A_reconstructed = np.dot(U, np.dot(Σ_modified, Vt))  # Reconstruct A

# Print the reconstructed matrix
print("\nReconstructed Matrix A (After Modifying Singular Values):")
print(A_reconstructed)

# Print the difference matrix
print("\nDifference Matrix:")
print(A-A_reconstructed)
```

A.3.3 Expected File Size

```python
import pandas as pd

# Function to calculate data storage for reduced SVD in KB per channel
def calculate_storage(height, width, num_singular_values):
    U_size = height * num_singular_values / 1024  # U: m x k
    S_size = num_singular_values / 1024           # S: k
    Vt_size = num_singular_values * width / 1024  # Vt: k x n
    total_data = U_size + S_size + Vt_size
    return total_data

# Input dimensions
height = 512
width = 768

# Store results for different numbers of singular values used
results = []
max_singular_values = min(height, width)  # Max singular values is the minimum of dimensions
for num_singular_values in range(1, max_singular_values + 1):  # Use at least 1 singular value
    total_data_per_channel = calculate_storage(height, width, num_singular_values)
    total_data_rgb = total_data_per_channel * 3  # Multiply by 3 for RGB channels
    results.append({
        "# Singular Values": num_singular_values,
        "Total Data (KB)": total_data_rgb
    })

# Create a DataFrame to display results
df = pd.DataFrame(results)

# Adjust pandas display settings to show all rows and columns
pd.set_option("display.max_rows", None)  # Show all rows
pd.set_option("display.max_columns", None)  # Show all columns
pd.set_option("display.width", None)  # Don't wrap columns

# Display results
print(df.to_string(index=False))
```

A.3.4 Compare SVD Implementations

```python
from svd import svd
import numpy as np
# Generate a random matrix A
np.random.seed(0)
A = np.random.rand(5, 5)

# Compute SVD using custom svd function
results = svd(A)
U_custom, S_custom, Vt_custom = results["Left singular vectors (U)"], np.diag(results["Σ"]), results["Right singular vectors (V

# Compute SVD using numpy's svd function
U_np, S_np, Vt_np = np.linalg.svd(A, full_matrices=False)

# Print the decompositions
print("Custom SVD Decomposition:")
print(U_custom)
print(np.diag(S_custom))
print(Vt_custom)

print("\nNumpy SVD Decomposition:")
print(U_np)
print(np.diag(S_np))
print(Vt_np)
```

A.3.5 Main Image Reconstruction

```python
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
```

```python
import os

def compress_image_svd(image_path, num_singular_values):
    # Load the image
    img = Image.open(image_path)
    img_array = np.array(img)

    # Ensure the image has RGB channels
    if img_array.ndim != 3 or img_array.shape[2] != 3:
        raise ValueError("Image must have three color channels (RGB).")

    # Extract the R, G, B channels
    R, G, B = img_array[:, :, 0], img_array[:, :, 1], img_array[:, :, 2]

    # Function to perform SVD and calculate size of reduced decomposition
    def svd_reconstruct(channel, num_singular_values):
        U, S, Vt = np.linalg.svd(channel, full_matrices=False)
        total_singular_values = len(S)

        # Reconstruct the channel with reduced SVD
        S_reduced = np.diag(S[:num_singular_values])
        reconstructed_channel = np.dot(U[:, :num_singular_values], np.dot(S_reduced, Vt[:num_singular_values, :]))

        # Quantize values to 8-bit integers (0-255)
        quantized_channel = np.clip(reconstructed_channel, 0, 255).astype(np.uint8)

        # Calculate size of reduced matrices (in bytes) after quantization
        reduced_size = (
            (U[:, :num_singular_values].shape[0] * U[:, :num_singular_values].shape[1])  # U
            + num_singular_values  # S
            + (Vt[:num_singular_values, :].shape[0] * Vt[:num_singular_values, :].shape[1])  # Vt
        )  # All in bytes per element (1 byte after quantization)

        return quantized_channel, total_singular_values, reduced_size

    # Compress each channel
    R_compressed, R_total_sv, R_reduced_size = svd_reconstruct(R, num_singular_values)
    G_compressed, G_total_sv, G_reduced_size = svd_reconstruct(G, num_singular_values)
    B_compressed, B_total_sv, B_reduced_size = svd_reconstruct(B, num_singular_values)

    # Ensure all channels have the same total singular values
    assert R_total_sv == G_total_sv == B_total_sv, "Total singular values mismatch across channels."
    total_singular_values = R_total_sv

    # Combine the channels to form the compressed image
    compressed_image = np.stack([R_compressed, G_compressed, B_compressed], axis=2)

    # Total reduced size for all channels (in bytes)
    total_reduced_size = (R_reduced_size + G_reduced_size + B_reduced_size)

    # Calculate sizes of the original image and the reduced decomposition
    original_size = os.path.getsize(image_path)

    return img, compressed_image, original_size, total_reduced_size, total_singular_values

# Parameters
image_path = "images/1.bmp"
num_singular_values = 30  # Adjust for desired compression level

# Compress the image
original_img, compressed_img, original_size, reduced_size, total_singular_values = compress_image_svd(image_path, num_singular_v

# Calculate percentage of singular values used
percentage_singular_values = (num_singular_values / total_singular_values) * 100

# Display the images
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title(f"Original Image\nSize: {original_size / 1024:.2f} KB")
plt.imshow(original_img)
plt.axis("off")

plt.subplot(1, 2, 2)
plt.title(f"Compressed Image\nSVD Size: {reduced_size / 1024:.2f} KB\n{num_singular_values} Singular Values Used ({percentage_si
plt.imshow(compressed_img)
plt.axis("off")

plt.tight_layout()
plt.show()

# Output the sizes and percentage in the console
print(f"Original image size: {original_size / 1024:.2f} KB")
print(f"Reduced SVD decomposition size: {reduced_size / 1024:.2f} KB")
print(f"Number of singular values used: {num_singular_values} ({percentage_singular_values:.2f}%)")
```