Tufts University

Department of Mathematics

Compression of Large Images via the Random Singular Value Decomposition

Authors:
Thomas Anzalone
Elijah Sanderson

Supervisor:
Dr. Abiy Tasissa

A final paper submitted for Tufts University:
Math 123: Mathematical Aspects of Data Analysis
December 24, 2021

Contents

Abstract

This project aims to compare the performance of four randomized singular value decomposition algorithms on very large grayscale images, roughly 6000 pixels by 4000 pixels in dimensions. We first outline the mathematics of our initial rSVD algorithm and employ two techniques, oversampling and power iteration, to form our other algorithms. We score the scripts based on two criteria, namely time complexity and error complexity, to determine which form of modified rSVD is 'most' efficient. Subsequently, we elect one of these algorithms in attempt to compress color images, specifically RGB, by performing the rSVD on each color channel, recomposing the image, and then analyzing the results.

1 Introduction

In an always evolving world, big data is used more than ever as a means to improve the lifestyles of human-kind. One of the critical topics in the forefront of Numerical Linear Algebra (NLA), is how to compress data both quickly and effectively. First conceived in the late 1800s and formally proved by Carl Eckart and Gale Young in 1936, the Singular Value Decomposition (SVD) is a technique that allows an m by n matrix, X, to be factorized into the product of an m by m unitary matrix, U, a diagonal matrix, S, and an n by n unitary matrix, V. For simplicity of research, this paper will assume X is composed of strictly real entries. In this case, the matrices, U and V, are guaranteed to be orthogonal matrices such that the following properties hold:

$$U^T U = U U^T = I_m \tag{1}$$

$$V^T V = V V^T = I_n \tag{2}$$

It is also the case that the entries of the diagonal matrix, S, are the singular values of X. Similar to the standard eigen-decomposition, the columns of U are known as the left-singular vectors of X, with the corresponding singular value in the same column of S. The columns of V are then known as the right-singular vectors of X, with the corresponding singular value in the same column of S. Given these definitions, the SVD representation of X is as follows:

$$X = USV^T \tag{3}$$

One special property regarding the singular values of X, is that they will always be non-negative. A non-negative real number, σ, is a singular value of X if and only if there exists vectors u and v such that:

$$Xv = \sigma u \tag{4}$$

$$X^T u = \sigma v \tag{5}$$

The transpose of X is an important factor in determining the SVD of X. Given the SVD of X, the SVD of X^T is simply defined as:

$$X^T = (USV^T)^T = VSU^T \tag{6}$$

The SVD of the transpose could then be used to generalize two powerful equations:

$$XX^T = (USV^T)(VSU^T) = US(V^TV)SU^T = US(I)SU^T = US^2U^T \tag{7}$$
$$X^TX = (VSU^T)(USV^T) = VS(U^TU)SV^T = VS(I)SV^T = VS^2V^T \tag{8}$$

This tells us that the eigen-decomposition of XX^T and X^TX will give us the unitary matrices U and V respectively, and the eigenvalues corresponding to the eigenvectors in U and V are precisely the squared singular values. Another fact about the singular values of X, is that in order to completely represent X, we only require the non-zero singular values of X, as the zero singular values would not contribute to the computation. Considering the above equations (7) and (8), and the fact that a matrix's rank is equal to its number of non-zero eigenvalues, the number of nonzero singular values is precisely equal to both rank(XX^T) and rank(X^TX) which are proven to be equivalent to rank(X) through the Rank-Nullity Theorem. If we consider rank$(X) = r$, one useful way to represent the standard form of the SVD is to consider it as a sum of the first r vector outer products using the i^{th} column of U, the i^{th} singular value, and the i^{th} column of V:

$$X = USV^T = \sum_{i=1}^{r} s_i u_i v_i^T \tag{9}$$

This means that we don't need to retain the corresponding columns of U and V, and can thereby reduce the sizes of U, S, and V and call these new matrices: U_r, S_r, and V_r, respectively, where $U_r \in \mathbb{R}^{m\times r}$, $S_r \in \mathbb{R}^{r\times r}$, and $V_r \in \mathbb{R}^{r\times n}$. When it comes to storage of high dimensional data this is extremely beneficial in reducing the necessary space.

2 Problem

Although the SVD is powerful in its ability to create an accurate low-rank approximation of some data matrix, the computational complexity is far too great to justify using it on very large data sets. Eigen-decomposition is a very computationally expensive procedure on cubic order, which diminishes the benefits of SVD for such data sets. This lead mathematicians to explore methods of optimizing the SVD to account for the ever-growing world of big data during the 21st century.

2.1 Methodology

After much research on the subject, mathematicians Halko, Martinsson, and Tropp developed a new algorithm to compute the SVD, using a random projection.[4] This algorithm, aptly named the Random SVD (rSVD), projects a very tall[1] matrix down to a much smaller subspace, in which computing the SVD is extremely efficient. Outlined in Halko et al's

[1]A tall matrix is one that has more rows than columns. When dealing with matrices that have more columns than rows, we can simply work with the transpose of that matrix.

2011 paper, Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions, one can still preserve much of the structure of the original data matrix, $X \in \mathbb{R}^{m\times n}$, with only needing to compute a small fraction of the singular values. To do this, they propose generating random projection matrix, $P \in \mathbb{R}^{n\times p}$, whose entries are sampled from some random probabilistic distribution. The value of r is equivalent to the desired rank of X, or the number of singular values of X that forms an approximation of X fitting for the particular application. In this paper, the distribution chosen will be the Gaussian, with mean 0 and standard deviation 1. X is then projected into the random subspace by the matrix product:

$$Z = XP, \qquad Z \in \mathbb{R}^{m\times p} \tag{10}$$

The next step is to find the orthogonal basis for Z, by computing the QR decomposition of Z, and denoting the resulting semi-orthogonal matrix Q.

$$Z = QR, \qquad Q \in \mathbb{R}^{m\times p} \tag{11}$$

The original data matrix is then projected into the orthogonal subspace by the matrix product:

$$Y = Q^T X, \qquad Y \in \mathbb{R}^{p\times n} \tag{12}$$

In almost all cases, r is significantly smaller than the original dimension, m, which means that computing the SVD of Y will be much less expensive than that of the original data matrix, X. The SVD of Y yields the resulting matrices:

$$Y = \hat{U_Y}\hat{S}\hat{V}^T, \qquad \hat{U_Y} \in \mathbb{R}^{r\times r}, \quad \hat{S} \in \mathbb{R}^{r\times r}, \quad \hat{V} \in \mathbb{R}^{n\times r} \tag{13}$$

Note that we use $\hat{U}$ here instead of U, and so on, since we are only computing a truncated approximation of the SVD to the desired rank, r, which will be significantly smaller than rank(X) in almost every real world application. In the above case $\hat{S}$ and $\hat{V}$ are approximately those of the actual SVD of X, composed of the first r singular values of X in descending order. To obtain the approximation of $\hat{U}$, a simple computation will attempt to unravel the orthogonal projection to a degree:

$$\hat{U} = Q\hat{U_Y} \tag{14}$$

The results are exceedingly impressive in reconstructing a low-rank approximation of the original data matrix. Although some error exists, as we explore later, there are techniques to reduce that error while not increasing the computational complexity by an excessive amount.[2]

2.2 Oversampling

Oversampling is a rather simple yet useful technique to reduce the error of the standard rSVD algorithm. This is accomplished by adding some small parameter, o, ideally smaller than 25 and dependent upon the data set, to the desired rank when forming the random projection matrix, P. This yields some $P \in \mathbb{R}^{n\times(r+o)}$. At the end of the rSVD algorithm, we can then perform a truncation of the computed $\hat{U}$, $\hat{S}$, and $\hat{V}$ matrices by shaving off the last o columns of $\hat{U}$, $\hat{S}$, and $\hat{V}$, and the last o rows of $\hat{S}$. By doing this, we can slightly reduce some of the error introduced by the random subspace, giving X a bit more of it's deterministic influence on the SVD computation.

2.3 Power Iterations

Applying Power Iterations to the algorithm is an even stronger technique to reduce the error associated with the rSVD. To do this, one would multiply the random projection matrix, Z, by "powers" of X. Since X is rectangular we define the q^{th} power of X as follows:

$$X^{(q)} = (XX^T)^q \tag{15}$$

This accomplishes two important functions: it greatly increases the influence of the original data matrix onto the random projection, and it allows the singular values to decay more rapidly, allowing for a better low-rank approximation than previously achievable. For the latter part of the previous statement, we can think as though we are looking at singular values raised to the q^{th} power. If $\sigma_i > \sigma_j$, by one order of magnitude, then that implies that $\sigma_i^q >> \sigma_j^q$, by q orders of magnitude. Using power iterations, the algorithm can be tuned by this parameter, q, as follows:

$$\begin{align}
Z^{(q)} &= (XX^T)^q Z \tag{16}\\
&= (XX^T)(XX^T)^{q-1} Z \tag{17}\\
&= (XX^T) Z^{(q-1)} \tag{18}\\
&= X(X^T Z^{(q-1)}) \tag{19}
\end{align}$$

The important thing to note about the standard form of the power iterations above, is that the order of operations is critical in saving computational cost. Since X is a large and tall matrix, $(XX^T)Z$ is on the order of $O(m^2n)$, even if you were to store XX^T into memory prior to the computation. However, $X(X^TZ)$ is on the order of $O(mnr)$, and since r is much smaller than m, this is extremely efficient when performing this computation q times.[1]

3 Results

For this project, we explored the accuracy and efficiency of rSVD algorithms against those of the deterministic truncated SVD. The data chosen for the analysis were professionally photographed images of very high resolution, approximately 6000 by 4000, or 24 megapixels.[5] We compared four methods for computing the SVD of the gray-scaled image. Comparisons were based off of two key criteria:

1. A generalized error formula:

$$E = \frac{\|X - \hat{U}\hat{S}\hat{V}^T\|_2}{\|X\|_2} \tag{20}$$

2. The time it took to perform the computation.[2]

The truncation parameter[3] used in this experiment is held constant for each of the tested algorithms; therefore the storage size of the compressed data had negligible variance between the tested algorithms.

3.1 Deterministic SVD

We started by comparing the deterministic truncated SVD to the original image. For the chosen grayscaled image, MATLAB's inbuilt SVD solver with the 'econ' parameter enabled yielded results in 17.8441 seconds. Using our generalized error formula, this truncated SVD yielded an error of around 0.0013, or 0.13% off of the original image. The results are hardly noticeable to the naked eye, unless you zoom into details such as folds on the fabric, etc. The original image, when loaded into MATLAB, occupied 24.33 million bytes. By storing the individual components of the SVD, the data occupies 3.11 million bytes, or only 12.8% of the original image's information. Of course, this would vary depending on the chosen data set, and the optimal truncation parameter, but for our experiment, all of our results will contain this proportion of the data.

3.2 Basic rSVD

The basic rSVD is the algorithm defined within section 2.1, and is the simplest version of the rSVD. Using the MATLAB code that we created for the rSVD (see Appendix A), we were easily able to compute the SVD approximation with the same truncation parameter (desired rank, r) as was used in the deterministic SVD. The time it took MATLAB to complete the computation was only 0.3171 seconds, or about 317 milliseconds. Exceptionally fast, when compared to MATLAB's economy SVD function. The generalized error for performing this computation was 0.0038, or 0.38% off of the original image. Although the error, roughly three times that of the deterministic SVD, is slightly more noticeable to the naked eye, it is still subtle. Although not entirely accurate as one would desire, all of the key features of the image remain in tact with negligible loss. You may not want to frame this image over the original, but performing certain analysis on this data, such as Principal Component Analysis (PCA), would likely yield results that closely mirror that of the original data. Considering the storage benefits, and the fact that this only cost less 1.8% of the time compared to the deterministic SVD, the implications for how this can be used in the real world are huge.

[2]Computation time would vary depending on the hardware used. For this project the hardware was consistent through all trials.

[3]Truncation parameter $k = 300$ was determined through observation of the image's complete singular value decay.

3.3 Oversampled rSVD

The oversampling method was designed to reduce the rSVD's error while only adding on a relatively minor computational cost. Since the random projection matrix has additional number of columns, the associated matrix multiplications are costlier. For our application, we tuned the oversampling parameter in order to obtain optimal results, resulting in $o = 20$. With the truncation parameter of 300, our random projection matrix is now formed with 320 columns. Using this new random projection matrix, we perform the rSVD and then truncate the extra 20 columns of the resulting $\hat{U}$ and $\hat{V}$, as well as the extra 20 rows and columns of $\hat{S}$. The time it took MATLAB to compute this algorithm was 0.3573 seconds, only 40 milliseconds more than the basic rSVD. The error associated with the result came to 0.0035, or 0.35% off of the original image. Again the difference is marginal to the naked eye with almost a negligible increase in time; however, mathematically, it yields about an 8% improvement from the basic rSVD. This means that oversampling provides enough benefit to warrant use in almost all rSVD application, the exception being extremely time-sensitive constraints on the order of milliseconds.

3.4 Oversampled rSVD with Power Iterations

As described in section 2.3, the power iterations method was designed for two major reasons. It both acts as a way to converge the randomness incorporated by the random projection in order to reduce the rSVD's error, and it simulates more rapid singular value decay which will allow greater representation of the entire data while storing less singular values. For this trial, we maintain the oversampling parameter within our algorithm, and add in one step of the power iteration:

$$\begin{aligned} Z^{(0)} &= XP && (21) \\ Z^{(1)} &= X(X^T Z^{(0)}) && (22) \\ &= X(X^T(XP)) && (23) \end{aligned}$$

The time it took MATLAB to compute this algorithm was 0.5288 seconds, around 210 milliseconds more than the basic rSVD. It makes sense that this requires more time than the previous rSVD algorithms, by nature of adding a couple of large matrix multiplications. However, we still see this as only 3.0% of the time it took to compute the deterministic SVD. The error was calculated as 0.0016, or 0.16% off of the original image. The resulting image looks nearly identical to that of the deterministic SVD, and only took a fraction of the time. By far, this is the optimal choice in efficiently computing the SVD of the given data set, while still maintaining all of the key features of the data.

3.5 Applications to RGB Images

While working mostly with grayscale images in the past, we decided to experiment with a colored image. In MATLAB, a colored image can be loaded in as several different formats, including RGB and CYMK. RGB matrices are stored as 3-D matrices, of size $m \times n \times 3$, whose 3 "layers" are the Red, Green, and Blue color components of the image. The individual color layers are similar to how MATLAB treats a grayscale image, where rather than being a proportion of the amount of gray in the image, it is the proportion of the amount of red, for example. CMYK is treated the same way, except with 4 layers of color components. For our experimentation, we used RGB. Intuition tells us that since each color component is on a scale similar to how we dealt with grayscale images in the previous experiments, perhaps we could compute the SVD for each layer. We used a new image in this experiment, whose resolution is approximately 4000 by 6000. Here the image is still around 24 Megapixels, however, in MATLAB it occupies 73.00 million bytes. In this case, we have a matrix that has more columns than rows, so we adapted the algorithm to detect these instances and work on the transpose and then return the proper SVD. Upon computing the deterministic SVD of each color component, we are left with 9 matrices:

$$\begin{array}{lll} \hat{U}_{red}, & \hat{S}_{red}, & \hat{V}_{red}; \\ \hat{U}_{green}, & \hat{S}_{green}, & \hat{V}_{green}; \\ \hat{U}_{blue}, & \hat{S}_{blue}, & \hat{V}_{blue}. \end{array}$$

Since these matrices are significantly reduced in size compared to the original image, they only occupy 9.33 million bytes in total, which is still 12.8% that of the original image. This was no small task for our computer, however, and took 63.0543 seconds to compute, with an approximate error of 0.0082, or 0.82% off of the original image. This makes sense, since we run the algorithm 3 times, we would expect to see the computation take 3 times as long and produce around 3 times the error. Applying our rSVD algorithm into the procedure, our computer took 2.2933 seconds to compute the SVD with an error of 0.0105, or 1.05% off of the original image. While the error is notably higher, it still remains hardly noticeable to the naked eye compared to that of the deterministic SVD, and the computation time is only about 3.6% that of the deterministic SVD. At a similar efficiency rate, if one is working with a large enough data set that requires, say 20 minutes to compute, the rSVD algorithm could complete the task in under one minute.

4 Discussion

While the sentiment of the results were expected as increased efficiency is the main objective of the algorithm, the magnitude by which the efficiency improves was quite surprising. Even the basic rSVD algorithm was able to produce relatively low error results in a fraction of the time. Adding in the oversampling parameter and a power iteration worked to improve the error drastically, with only a small increase in cost relative to the deterministic SVD's cost. The table below outlines the overall results with side-by-side comparison:

	Truncated SVD	rSVD	Oversampling	Power Iteration
Time Complexity	17.8441	0.3171	0.3573	0.5288
Error Complexity	0.0013	0.0038	0.0035	0.0016

Table 1: *Time and error complexity results for the performance of each algorithm on our sample image.*

As mentioned earlier, the implications of the results are impressive. Thinking about the amount of data used in known algorithms, such as the Google Page Ranker and the Netflix Recommendation, this method for computing the SVD is ideal in that it efficiently captures almost all of the key features of the data, with little loss. With the SVD's applications to Least Square Regression, the Pseudoinverse, and PCA, rSVD can ultimately produce desired results at a much smaller cost. Although we only studied the benefits to the rSVD, we also learned when this method may not be ideal. Sometimes in nature, a data matrix may not have a rapid singular value decay. While this is still a hindrance to the deterministic SVD, it may actually lead to sub-par results within the rSVD. In cases such as these, the introduction of the random projection may produce excess variability, think similar to amplifying oscillations. This could significantly increase the error, and while computation time is still optimal, the key features may have been lost or distorted from the rSVD. Knowledge of this fact can actually be used to denoise a particular data set, however, since the data may have a rapid singular value decay, and the noise component would will not.[3] Regarding the RGB algorithm that we came up with, although our findings were positive, it seems that the method would only be optimal for 3-D matrices with a small number of layers, such as RGB images. In some applications, the data could be of even higher dimension and our typical SVD would not be applicable. Through some research, we learned about the Higher Order Singular Value Decomposition (HOSVD) which operates on a generalized form known as tensors, or multilinear mappings in higher dimensional vector spaces. Finding low-rank tensor decompositions is the topic of many recent research papers in Randomized NLA (RandNLA).

References

[1] Steven L. Brunton and J. Nathan Kutz. “Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control.” In: (2019), pp. 37–41.

[2] Xinyu Chen. *Intuitive Understanding of Randomized Singular Value Decomposition.* URL: https://towardsdatascience.com/intuitive-understanding-of-randomized-singular-value-decomposition-9389e27cb9de.

[3] Matan Gavish and David L. Donoho. “The Optimal Hard Threshold for Singular Values is $4/\sqrt{3}$”. In: *IEEE Transactions on Information Theory* 60 (2014).

[4] N. Halko, P. Martinsson, and J. Tropp. “Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions.” In: *SIAM Review* (2011).

[5] Kegan Stellato. URL: https://www.mambaa.gg/.

A Algorithms for rSVD, Oversampling, and Power Iterations

```
%-----------------------------------------------------------------------------
% MATH 123 project script
%
% usage :
%
%  input:
%  * file : the input image file to analyze
%
%  output:
%  * writes out the SVD and rSVD images to new files
%-----------------------------------------------------------------------------
% Thomas Anzalone and Elijah Sanderson, 2021

file = 'walking.jpg';

[im_name,im_format] = strtok(file,'.');

if strcmpi(im_format,'.tiff') || strcmpi(im_format,'.png')
    bits = 16;
else
    bits = 8;
end

X = double(rgb2gray(imread([im_name,im_format])))/(2^bits);

k = 300;

% Compute the deterministic SVD of X
tic; [U,S,V] = svd(X,'econ'); t_det = toc;

% rSVD #1 - Compute the random SVD with no techniques
tic; [rU1,rS1,rV1] = rsvd(X,k); t_rsvd1 = toc;

% rSVD #2 - Compute the random SVD with just oversampling the intrinsic rank
    by 5
tic; [rU2,rS2,rV2] = rsvd(X,k,20); t_rsvd2 = toc;

% rSVD #3 - Compute the random SVD with oversampling the intrinsic rank by 5
    and doing 2 power iterations
tic; [rU3,rS3,rV3] = rsvd(X,k,20,1); t_rsvd3 = toc;

% Observe the errors in each method from the actual image
err_det = norm(X-U(:,1:k)*S(1:k,1:k)*V(:,1:k)',2)/norm(X,2);
err_rsvd1 = norm(X-rU1*rS1*rV1',2)/norm(X,2);
err_rsvd2 = norm(X-rU2*rS2*rV2',2)/norm(X,2);
err_rsvd3 = norm(X-rU3*rS3*rV3',2)/norm(X,2);

% Write the new image files
```

```
if bits == 16
    imwrite(im2uint16(U(:,1:k)*S(1:k,1:k)*V(:,1:k)'),[im_name,'svd',im_format
   ]);
    imwrite(im2uint16(rU1*rS1*rV1'),[im_name,'rsvd1',im_format]);
    imwrite(im2uint16(rU2*rS2*rV2'),[im_name,'rsvd2',im_format]);
    imwrite(im2uint16(rU3*rS3*rV3'),[im_name,'rsvd3',im_format]);
else
    imwrite(im2uint8(U(:,1:k)*S(1:k,1:k)*V(:,1:k)'),[im_name,'svd',im_format])
   ;
    imwrite(im2uint8(rU1*rS1*rV1'),[im_name,'rsvd1',im_format]);
    imwrite(im2uint8(rU2*rS2*rV2'),[im_name,'rsvd2',im_format]);
    imwrite(im2uint8(rU3*rS3*rV3'),[im_name,'rsvd3',im_format]);
end


%% rSVD function

function [U,S,V] = rsvd(X,k,o,q)
%-----------------------------------------------------------------------------
% random SVD
% Extremely fast computation of the truncated Singular Value Decomposition,
   using
% randomized algorithms as described in Halko et al. 'finding structure with
   randomness'
%
% usage :
%
%  input:
%  * X : matrix whose SVD we want in R(m by n)
%  * k : target rank of X
%  * o : oversampling parameter (optional)
%  * q : # of power iterations (optional)
%
%  output:
%  * rU,rS,rV : SVD of our randomly projected matrix, truncated to the
%               target rank k
%-----------------------------------------------------------------------------
% Thomas Anzalone and Elijah Sanderson, 2021
% Citations:    Antoine Liutkus, Inria 2014
%               Steve Brunton, 2020

[m,n] = size(X);

transpose_flag = false;
if m < n
    % Fat matrix - compute rSVD on X' and swap U and V in result
    transpose_flag = true;
    X = X';
    [~,n] = size(X);
end

switch nargin
```

```
    case 2
        o = 0;
        q = 0;
    case 3
        q = 0;
end

% Step 1: Random Projection Matrix P in R(n by r)
r = k;
P = randn(n,r+o);
Z = X*P;

% Perform the Power iterations - order of operations matters!
% X'*Z is much less computational than X*X'
for i = 1:q
    Z = X*(X'*Z);
end

% Step 2: Find an orthogonal basis, Q, for Z (and X) using QR decomposition
[Q,~] = qr(Z,0);

% Step 3: Project X into the orthogonal basis, Q
Y = Q'*X;
[U,S,V] = svd(Y,'econ');

U = Q*U;
U = U(:,1:r);
S = S(1:r,1:r);
V = V(:,1:r);

if transpose_flag
    U_mem = U;
    U = V;
    V = U_mem;
end

end
```

B Modification for RBG Images

```
%-----------------------------------------------------------------------------
% MATH 123 project script
%
% usage :
%
%  input:
%  * file : the input image file to analyze
%
%  output:
%  * writes out the SVD and rSVD images to new files
%-----------------------------------------------------------------------------
% Thomas Anzalone and Elijah Sanderson, 2021

file = 'walking.tiff';

[im_name,im_format] = strtok(file,'.');

if strcmpi(im_format,'.tiff') || strcmpi(im_format,'.png')
    bits = 16;
else
    bits = 8;
end

X = double(imread([im_name,im_format]))/(2^bits);
R = X(:,:,1);
G = X(:,:,2);
B = X(:,:,3);

k = 300;

%% SVD
tic;
% Compute the deterministic SVD of X
[UR,SR,VR] = svd(R,'econ');

% Compute the deterministic SVD of X
[UG,SG,VG] = svd(G,'econ');

% Compute the deterministic SVD of X
[UB,SB,VB] = svd(B,'econ');

svd_construction = cat(3,...
    UR(:,1:k)*SR(1:k,1:k)*VR(:,1:k)',...
    UG(:,1:k)*SG(1:k,1:k)*VG(:,1:k)',...
    UB(:,1:k)*SB(1:k,1:k)*VB(:,1:k)');

t_det = toc;

%% rSVD
tic;
```

```
% Compute the random SVD on R component
[rUR,rSR,rVR] = rsvd(R,k,20,1);

% Compute the random SVD on G component
[rUG,rSG,rVG] = rsvd(G,k,20,1);

% Compute the random SVD on B component
[rUB,rSB,rVB] = rsvd(B,k,20,1);

rsvd_construction = cat(3,...
    rUR*rSR*rVR',...
    rUG*rSG*rVG',...
    rUB*rSB*rVB');

t_rsvd = toc;

%% Write the new image files
if bits == 16
    imwrite(im2uint16(svd_construction),[im_name,'svd_rgb',im_format]);
    imwrite(im2uint16(rsvd_construction),[im_name,'rsvd_rgb',im_format]);
else
    imwrite(im2uint8(svd_construction),[im_name,'svd_rgb',im_format]);
    imwrite(im2uint8(rsvd_construction),[im_name,'rsvd_rgb',im_format]);
end
```

C Contributions

Both team members worked in conjunction to research, brainstorm, and compile the project deliverables. Thomas was responsible for running the MATLAB scripts on his PC to capture the results, as well as providing the overview of the rSVD algorithm in the presentation and report. Elijah was responsible for researching and formulating the method for analyzing RGB images using SVD as well as comparing to other image compression algorithms. Both members worked on tuning the parameters for optimizing the algorithm as well as delivering the presentation equally and compiling the report using Overleaf.