Applications of Linear Algebra in Facial Recognition

Michael Somkuti

Department of Mathematics, Bates College, Lewiston, ME 04240

Acknowledgements

I would like to thank the Bates Math Department for helping me discover my passion for mathematics and computation, my advisor Grace Coulombe for helping me pursue my interests and grow as a student throughout my undergraduate experience, my thesis advisor Henry Boateng for helping me research and practice the intersection of both fields, professor Katy Ott for introducing me to MATLAB and programming as a whole, and Meredith Greer for helping me discover the joys of mathematical modeling. These professors pushed me to be the best student I could be, and this thesis would not have happened without their support. From faulty proofs to personal struggles, the help you gave me was invaluable. Thank you.

I would also like to thank my family and friends for supporting me throughout this process. All of you inspire me to be my best self and student, especially my fellow math majors.

I would also like to thank Bates Video Game Club (members present and past) for rooting for me, creating a space where I could de-stress, and for always having my back. We started as a small group and have grown so much! Nevo and I could not have founded a better club. I cannot wait to see what comes next for all of you. Thank you Martin Montgomery for joining us a faculty advisor, and for being a great professor!

Finally, I would like to thank professor Chip Ross for supplying images from Math Camp for me to experiment with, and every one who gave me permission to work with their respective images.

1. Introduction

In 2018, facial recognition is used in a variety of applications. These range from a consumer unlocking a device with their face to a security camera checking for a wanted criminal in the same area. While many believe that the challenge of facial recognition is a computer science problem, it is deeply rooted in mathematics, specifically linear algebra. This paper will dissect a specific approach to facial recognition through the application of SVD (*Singular Value Decomposition*), culminating in the creation of a functioning program. Writing an algorithm to consistently identify a face in a still image can be a challenging problem in and of itself. Therefore, the program written will use a controlled set of training images in order to accomplish this goal. Please note that the code snippets included throughout this text have been modified for readability, and that the full algorithm is included in the Appendix.

2. Image Processing Background

Before diving into Linear Algebra, one must first understand some concepts in Digital Image Processing. The matrices that will be used throughout this paper are not abstract, but rather tangible images that can be viewed on most devices. In this case, the image type used in this paper is JPEG. The core concept of this paper is the ability to represent images as large matrices, whose dimensions are equal to their sizes in pixels. Since this program will deal extensively with matrices, vectors, and the operations that can be performed on them, it was decided that this program would be written in Matlab. MATLAB has the ability to processes matrix manipulation and operations quickly, and create intuitive visualizations. It even has an image processing package that contains many useful functions.

2.1. Grayscale. Pixels in color images consist of the RGB (red, green, blue) values of each respective pixel. For example, a color pixel p can be represented as follows:

$$p = \begin{bmatrix} r & g & b \end{bmatrix}.$$

To increase efficiency, images used in this paper are grayscaled before any mathematical operations are performed on them. This reduces the number of values in each entry of the image matrix from 3 to 1, making each image matrix 2 dimensional. Each pixel's grayscale value can be found using the following equation [**3**].

$$p = 0.2989 * r + 0.5870 * b + 0.1140 * g$$

The calculated grayscale value of a pixel is intended to preserve the original luminescence, allowing the detection of significant features of the face. Images used in each training set were converted to grayscale using MATLAB's built in function, `rgb2gray(Image)`, taking a color image as its input. The example below depicts a n by m gray scale image, I, represented as a matrix of pixels, p, whose values are integers between 0 and 255.

$$I = \begin{bmatrix} p_{1,1} & p_{1,2} & \dots & p_{1,m} \\ p_{2,1} & p_{2,2} & \dots & p_{2,m} \\ p_{3,1} & p_{3,2} & \dots & p_{3,m} \\ \dots & \dots & \dots & \dots \\ p_{n,1} & p_{n,2} & \dots & p_{n,m} \end{bmatrix}$$

2.2. Scaling Images. Due to limitations of available computational power and memory, it was necessary to scale down all training images used in this thesis. Images were re sized to smaller dimensions using the MATLAB command, `imresize(Image,scale)`. MATLAB's default method for re sizing images is Bi-Cubic interpolation. Other methods include Nearest Neighbor and Bi-Linear Interpolation, which are executed at different speeds and yield lower quality outputs [**4**]. However, these methods will not be explored in depth, and are left as a possible future exploration. Due to the small scale of the dimensions and number of training images used, the time spent to execute re size is near negligible. With higher resolution images, it would be necessary to explore these methods.

3. Concepts in Linear Algebra

3.1. Vector Spaces. The goal of this facial recognition program is to find the closest match between an input image and and a face that already exists in the program's training set. An arbitrary training set used in this algorithm S consists of a set of j training images $\{I_1, I_2, \dots, I_j\}$. Now, consider any single image $I \in S$ whose gray scale values are represented by its m x n matrix, where $m, n \in \mathbb{N}$. It is known that I consists of a set of vectors $\{v_1, v_2, \dots, v_n\}$. Since the entries in every vector in I only consist of integers, I is well defined under addition and multiplication. Therefore, I is a vector space by definition [**1**]. One is able to represent any image I in the training set as a 2-D vector space, so it must be possible to represent the entire

training set in some manner. This idea will be addressed later through the creation of "face spaces" [**9**].

3.2. Singular Value Decomposition. SVD helps fill in the gaps where diagonalization is not possible [**1**]. This concept will serve as the foundation for a variety of applications necessary for facial recognition.

Consider an m x n matrix A with rank r. There exists an m by n matrix Σ, represented below.

$$\Sigma = \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}$$

The matrix Σ contains an r by r matrix D, whose diagonal entries are the first r singular values of A with all other entries equaling 0 [**1**].

$$D = \begin{bmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \dots & \dots & \dots & 0 \\ 0 & 0 & \dots & \sigma_r \end{bmatrix}$$

For these singular values, it is observed that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$. Additionally, there exists an orthogonal m by m matrix U and orthogonal n by n matrix V, such that A can be factored as follows [**1**].

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

Additionally, since A has rank r, $\{\boldsymbol{u}_1, \dots, \boldsymbol{u}_r\}$ is an orthonormal bases for the column space of A, Col A [**1**]. This fact be utilized later in the algorithm.

The aforementioned positive singular values are crucial to the image processing performed in facial recognition. Each one holds data about the image being decomposed. The larger the singular value, the more significant image data it holds [**5**]. Therefore, it is known that the largest singular values are the most important when representing an image. Using this knowledge, it becomes possible to compress images effectively by analyzing the distribution of its singular values. The elimination of specific values can result in an effective representation of the original image, increasing efficiency in a script and lowering the amount of storage necessary to run it.

3.3. Re-sizing. One aspect of an image that directly assures the success of the facial recognition algorithm is the size of the images being processed. In order to find matches, a variety of matrix operations will be performed on the data from a training set of images. Re-sizing each image ensures that there will be no issues regarding the dimensions of each matrix. Scaling each image to smaller dimensions initially also provides faster run time, as less entries will need to be computed.

When it comes to designing an algorithm that uses SVD, one must heavily consider the sizes of their input matrices in regard to their computing power. The calculation of the aforementioned matrix Σ relies on the orthogonal diagonalization of the matrix A^TA [**1**]. The standard image size used before re sizing was 580 by 730 pixels. While these dimensions may not seem large for an image, consider a training set T. The matrix representing T will consist of every image in the training set transformed into a column vector. If there are 10 images whose dimensions are 580 by 730 in our training set, then the dimensions of T will be 423400 by 10. It follows that the dimensions of T^TT will be a 423400 by 423400 matrix. It would be extremely costly to diagonalize a matrix of this size, especially in terms of memory. Therefore, it is extremely important to re-size training images to workable dimensions.

The re-sizing of images can be executed in Matlab through the usage of `imresize(image, scale)`. This function will return a new image that is the size of the old image times the specified scale. The value for scale can also be substituted with an array containing the desired dimensions for output. For example, `imresize(image, [10 20])` will return a 10 by 20 version of the input image [**4**]. This implementation of scaling images is an example of compression through bi-cubic interpolation.

4. Application of SVD in Compression

Since it is known that the largest singular values hold the most valuable image data, it is possible to compress images by eliminating multiple ones whose magnitudes are close to zero. Consider the following image of a Bates student named Eric. Reading the image in MATLAB yields a 580 by 730 matrix, E. Below, the original and gray scaled images are shown.

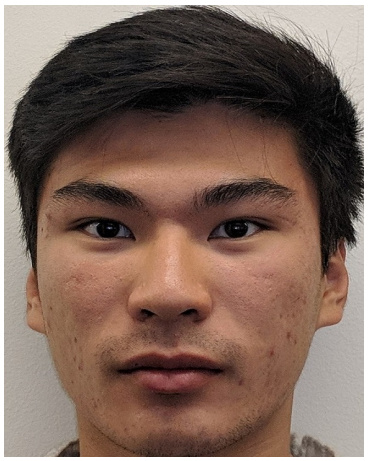

FIGURE 1.
Original Image of Eric

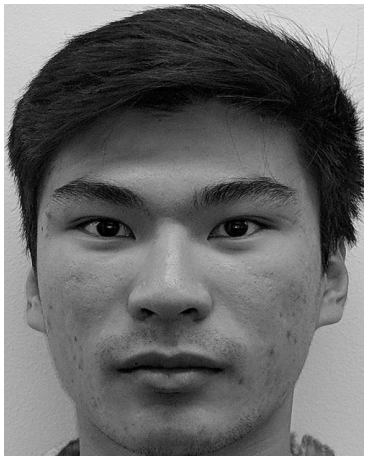

FIGURE 2.
Gray Scaled Image

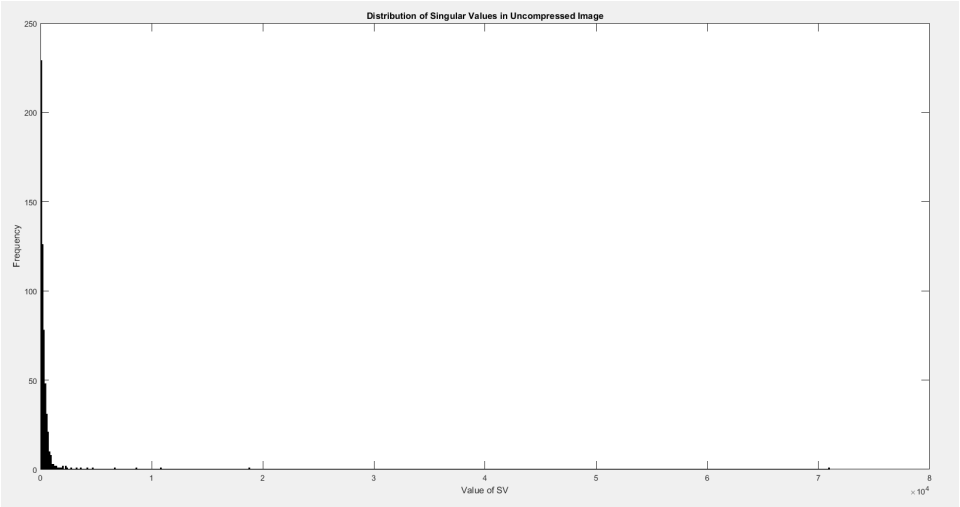

FIGURE 3.
Distribution of Singular Values in Uncompressed Image (Frequency vs Magnitude)

When SVD is performed on the uncompressed grayscale image, Figure 3 represents the distribution of all of its singular values. It is evident that the majority of the singular values for the image are close to zero, and therefore contain negligible image data. Through further experimentation, it was found that only keeping 15 singular values provided a good representation of the image, shown below.

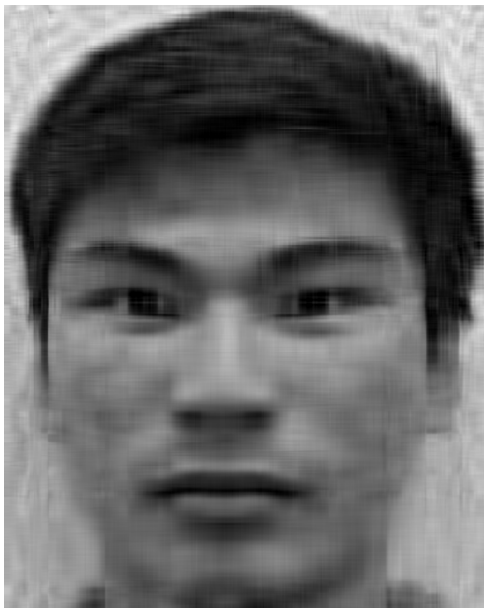

FIGURE 4.
Compressed Image of Eric, where K = 15

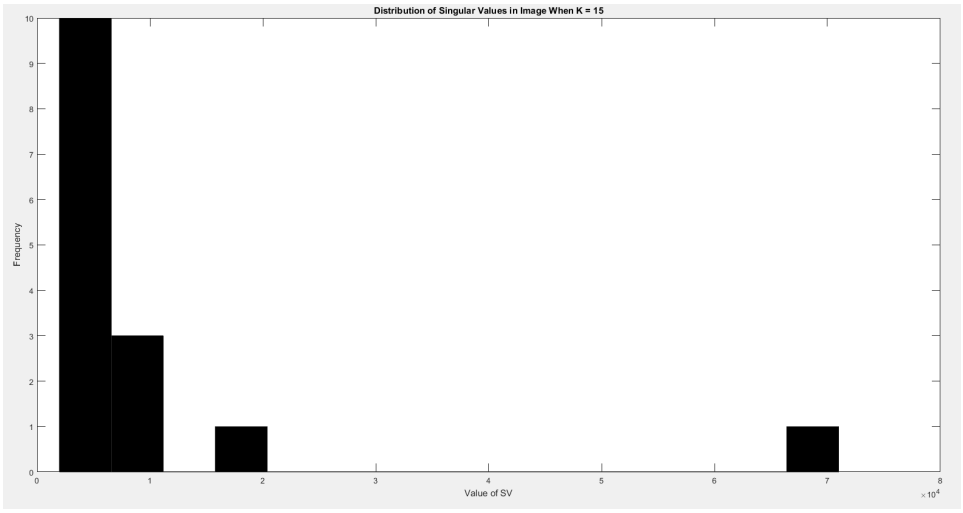

FIGURE 5.
Singular Value Distribution in Compressed Image
(K = 15)

Figure 5 provides a visual representation of the distribution of Figure 4's singular values.

4.1. Experimentation and Error. A large reduction in rank of an image was possible while retaining quality [**5**]. Therefore, experimentation began with the previously shown image of a student named Eric. How many singular values could one remove while retaining a solid representation of the original image? What would the resulting compressed image's storage size be in relation to the gray scaled original? In order to answer these initial question, the files were compressed with different chosen numbers of singular values, K. The new rank r of the m x n compressed image, C, is equal to K [**2**]. Finally, C would be reconstructed for viewing.

The compression process was implemented in Listing 1.

Listing 1. Implementation of Compression

```
function AK = svdPartialSum(E,K)
    E = double(E); % Convert image data to be numerical
    [U,S,V] = svd(E); % Perform SVD and output to 3 matrices

    % Compression takes place
    % Only keep K singular values
    % Ensuring that matrix dims are correct
    C = U(:,1:K) * S(1:K,1:K) * V(:,1:K)';

    C = uint8(C; % Convert back to image data
    figure; imshow(C); % Display compressed image
end
```

Table 1 displays the findings of this experiment with the chosen size for K, the resulting size of the compressed image in bytes, and the ratio of storage size for the compressed image in relation to the original.

Table 1. Compression Experimentation

K	**Bytes**	**Ratio**
580	70451	1.0000
100	58806	0.8347
30	39556	0.5615
15	32580	0.4624
5	25063	0.3558

Notice that the first row in the above table uses $K = 580$, preserving all singular values. An illuminating aspect of this experiment was the

results for $K = 15$. Reconstructing the image with 15 singular values provided a good representation of the image, and also reduced the file size by 53.76%.

However, simply reducing the file size is not enough. One must also consider the error between the original and the compressed version. One way to measure this is through the usage of MSE (*Mean Squared Error*). MSE measures the summed squared difference between an estimator and what is being estimated [**6**]. In this case, it compares the difference in pixel value at a specific coordinate from both the compressed and original image. This equation is defined as

$$MSE = 1/(mn) * \sum_{y=1}^{m} \sum_{x=1}^{n} (O(x,y) - C(x,y))^2,$$

where O is the original image, C is the compressed image, x, y are the coordinates of the specified pixel, and m, n are the dimensions of each image. [**2**]. The difference between the two values is squared, rather than taking the absolute value, to penalize outliers. Table 2 represents the MSE correlated with each K value of a compressed image. The closer the MSE value is to 0, the less error there is.

TABLE 2. MSE Experimentation

K	**MSE**
580	0
100	21.9934
30	33.5670
15	43.4128
5	65.3984

The findings of this experiment were expected. As more singular values are cut out, more image data is lost. This results in a larger margin of error between the compressed and the original. MSE was implemented into the algorithm using the line of code in Listing 2.

LISTING 2. Implementation of MSE

```
MSE = 1/(m * n) * sum(sum((Image - compressedImage).^2))
```

This line of code takes the difference between the matrices representing the original and the compressed, squares each value component wise, and sums and scales each resulting error value. This line builds on the

implementation in Lije Cao's code by replacing double for loops with matrix subtraction and component wise operations [**2**].

5. Facial Recognition

There are two training sets that will be used for this initial facial recognition algorithm. The first, T_1, is a series of images taken using a smartphone camera against a fixed background. The second, T_2, is a collection of head shots taken when prospective math majors take the famous prerequisite course "math camp." The first version of this algorithm will take an input image from T_1 and be able to identify the student in the set. Using images from the same training set ensures that a standard is set across all images processed. This standardization ensures that one is able to troubleshoot effectively in the preliminary stages of writing code. It removes the worry that any errors or lack of matches are due to variation in the images used.

5.1. Face Spaces. Consider a new image of a face q that will be input into the script, and a training set T containing z images of faces t, such that $T = \{t_1, t_2, \ldots, t_z\}$ and $z \in \mathbb{N}$. The representation of T can be seen below.

$$T = \begin{bmatrix} \begin{bmatrix} t_{1_{1,1}} & t_{1_{1,2}} & \ldots & t_{1_{1,m}} \\ t_{1_{2,1}} & t_{1_{2,2}} & \ldots & t_{1_{2,m}} \\ \ldots & \ldots & \ldots & \ldots \\ t_{1_{n,1}} & t_{1_{n,2}} & \ldots & t_{1_{n,m}} \end{bmatrix} & \ldots & \begin{bmatrix} t_{x_{1,1}} & t_{x_{1,2}} & \ldots & t_{z_{1,m}} \\ t_{x_{2,1}} & t_{x_{2,2}} & \ldots & t_{z_{2,m}} \\ \ldots & \ldots & \ldots & \ldots \\ t_{x_{n,1}} & t_{x_{n,2}} & \ldots & t_{z_{n,m}} \end{bmatrix} \end{bmatrix}$$

To ensure that matrix operations occur smoothly, every m by n image will be re-sized to to the same dimensions. In this case the chosen dimensions are 73 by 58 pixels. Then, preprocessing takes place, converting each image to gray scale. Finally, one must transform all $t_i \in T$ to column vectors. This transforms T from a multidimensional vector space to a 2-D vector space, represented below.

$$T = \begin{bmatrix} t_{1_{1,1}} & t_{2_{1,2}} & \ldots & t_{z_{1,m}} \\ t_{1_{1,2}} & t_{2_{2,2}} & \ldots & t_{z_{2,m}} \\ \ldots & \ldots & \ldots & \ldots \\ t_{1_{1,m}} & t_{2_{1,m}} & \ldots & t_{z_{1,m}} \\ \ldots & \ldots & \ldots & \ldots \\ t_{1_{n,m}} & t_{2_{n,2}} & \ldots & t_{z_{n,m}} \end{bmatrix}$$

Now, T is a vector space with dimensions $(m \cdot n)$ by z, and will be referenced as the face space [**9**]. The creation of the face space of T is implemented in MATLAB in Listing 3.

LISTING 3. Creation of Face Space

```
% Creation of Face Space in MATLAB
% Pre-allocate face space
T = zeros(dims(1) * dims(2), im_count);

for j = 1:im_count
    fileName = char(training_sets(i,j)); % Read our new
        image
    inputImage = preProcessing(fileName, 1, dims, 0);
    % Compress our images
    inputImage = double(svdPartialSum(inputImage, K));
    % Turn image into column vector
    columnf = inputImage(:);
    % Append our column
    T(:,j) = columnf;
end
```

5.2. Shifting by the Mean Image. Though the initial face space has been created for q to be projected onto, it must be shifted in relation to the mean image of T in order to compute a match. The mean image of T, $\bar{t}$, can be computed through the equation below [**2**].

$$\bar{t} = \frac{1}{z} * \sum_{i=1}^{z} t_i$$

The mean image, $\bar{t}$, is one of the most essential parts of the recognition algorithm. Matches are found by comparing the coordinates of the projection of q and those of each training image. All projections are in a new face space $\overline{T}$, which is T shifted by its mean image [**2**]. This face space, $\overline{T}$, can be found by subtracting $\bar{t}$ from each column of T [**2**]. Since $\bar{t}$ has the same dimensions as any image vector in T, a new matrix A can be created, where each column is equal to $\bar{t}$. The resulting mn by z face space $\overline{T}$ shares the same dimensions as T and A. Thus,

$$A = [\bar{t}_1, \bar{t}_2, \ldots, \bar{t}_z]$$

and

$$\overline{T} = T - A.$$

The creation of $\overline{T}$ is implemented in MATLAB in Listing 4.

LISTING 4. Creation of Tbar

```
% Creation of Tbar in MATLAB
 for j = 1:im_count
    . . .
    columnf = inputImage(:); % Turn image into column
        vector
    T(:,j) = columnf; % Append our column
    . . .
end

tBar = sum(T,2) / im_count; % Calculate the mean face
A = repmat(tBar,[1,size(S,2)]); % Matrix of mean face
Tbar = T - A; % Shift face space

mean_faces{i} = tBar; % Append mean face to cell array
face_spaces{i} = Tbar; % Append face space to cell array
```

5.3. Coordinate Matrices. The following step is to find the projection of each training image, t_i, in $\overline{T}$. Projecting t_i to its own face space $\overline{T}$ allows one to find the distances between it and all other images in T. This is possible since all images and their respective face spaces are shifted equally. To accomplish this, one must first compute the coordinate matrix of $\overline{T}$. This coordinate (position) matrix X, relative to the basis U of the column space of $\overline{T}$ is found through the equations below [**1, 2**].

$$UX = \overline{T} \tag{1}$$

$$U^{-1}UX = U^{-1}\,\overline{T} \tag{2}$$

$$X = U^{T}\,\overline{T} \tag{3}$$

Remember that $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_r\}$ is an orthonormal bases for Col $\overline{T}$ [**1**]. Thus, $U^{-1} = U^{T}$. The rank of U is equal to the number of singular values kept after compression, K. Thus, X will have K rows and z columns, where z is the number of images in $\overline{T}$.

The creation of X is implemented in MATLAB in Listing 5.

LISTING 5. Creation of X

```
[U,~,~] = svd(Tbar); % SVD of Tbar
Ur = U(:, 1:K); % Compression
X = Ur' * Tbar; % Compute coordinate matrix
```

5.4. Computing Distance. Next, the coordinates for the projection of q to $\overline{T}$ must be computed. This is found similarly to the way each image in T was projected. First, q is converted into a column vector. Then, $\overline{q}$ is calculated using the following equation.

$$\overline{q} = U^T \cdot (q - \bar{t})$$

Here, the mean image of T is subtracted from q to shift its coordinates. Then, the shifted vector is multiplied by U^T, compressing it to have r rows. The result is the coordinate vector $\overline{q}$ for the projection of q onto the basis of Col $\overline{T}$. Next, a new r by z matrix $\overline{Q}$ is created with each column equal to $\overline{q}$.

$$\overline{Q} = \begin{bmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,z} \\ q_{2,1} & q_{2,2} & \cdots & q_{2,z} \\ \cdots & \cdots & \cdots & \cdots \\ q_{r,1} & q_{2,r} & \cdots & q_{r,z} \end{bmatrix}$$

Notice that the dimension of $\overline{Q}$ mirror that of X. Now, the best possible match can be found. To accomplish this, one must find the minimum distance between the coordinate projection of q and those of all $t_i \in T$. This can be computed by taking the norm of $\overline{X}$ and $\overline{Q}$.

To compute the norm of $\overline{X}$ and $\overline{Q}$, one must first take the difference of the two coordinate matrices. This difference will be stored in a new matrix, D, where $D = X - Q$. Then, the norm of D is taken using the following equation.

$$D = Norm(D) = \sqrt{D^T D}$$

The values in D are the distances between q and each image in T. However, these distances are not yet sorted. The minimized distances will be stored in a new vector d. Notice that

$$d = diag(D) = \sqrt{[D_1^T D_1, D_2^T D_2, \ldots, D_n^T D_n]}$$

yields the minimized distances between between q and each $t_i \in T$ [Cao]. The vector d has z rows, equal to the number of images in T.

Finally, the best match can be computed by finding the minimum distance in the vector d. This can be computed in MATLAB using the `min()` function. This function returns the smallest element of the vector and its index, using the following syntax: `[min, index] = min(d)` [**7**]. The resulting variables `min` and `index` hold the distance between the projection of q and the best match, p, and the index of p in T. For example, the recognition script may return `index = 6`, signifying that q is most similar to $t_6 \in T$. If q is an exact match, the returned distance, `min`, will equal 0. If q is already in T or is a copy of an image $t_i \in T$, then the script will always return `min = 0` and an `index` equal to the index of $t_i \in T$.

The computation of d and the resulting match is implemented in MATLAB in Listing 6.

Listing 6. Finding d

```
% Difference of coordinate matrices
D = X - q * ones(1, size(q, 2))

d = sqrt(diag(D' * D)) % Norm

 % Finding the minimum distance and index of match
 [dmin, indx] = min(d)
```

5.5. Multiple Face Spaces. It is possible to compare an input image across multiple training sets, each of which will have its own face space. However, it would waste time to check the unknown image q against every face space. Computing the distance between q and each face space allows us to find the closest one to q, which is the one most likely to contain q's match. The distance between q and each face space is found by taking the norm of the difference of q and its coordinate vector, relative to each face space, respectively. The index of the closest face space is returned, so it can be used later in the recognition script.

The calculation of the closest face space is implemented in the modified MATLAB code snippet in Listing 7.

LISTING 7. Closest Face Space

```
% For each face space store the distance between it and q
    in the array dist_space

for i = 1:size(face_spaces, 1)
    q = q(:); % Vectorize input image
    [U,~,~] = svd(face_spaces{i}); % SVD of Tbar
    Ur = U(:, 1:K); % Compression

    q_shifted = double((q - mean_faces{i})); % Shift input
        image
    x = Ur' * q_shifted; % Project shifted input onto
        basis of Tbar
    q_projected = Ur * x; % New vector in face space

% magnitude of distance(face space and projection of input)
    distance = norm(q_shifted - q_projected);
    dist_space{i} = distance;
end

[dist, index] = min(dist_space); % Find closest face space
```

6. Confidence

While it is exciting to see that SVD can be used to execute facial recognition, another question is raised. How accurate are the results of the algorithm? This question led to the creation of a new parameter: the confidence of each match. Confidence can be calculated through the usage of data collected during recognition. The foundation of this parameter is the coordinate projections of each training image $t_i \in T$.

Consider a training face $t_6 \in T$. If one were to input t_6 into the facial recognition algorithm, it would match with itself with a distance equal to zero. Inputting images into the facial recognition algorithm reveals useful information about the training set. The variable d contains the distance between the coordinate projection of the face, t_6, and all the other training images in in the set. One is able to tell which face/image in T is least like t_6 using the information in d. The coordinate projection of the image that is least similar to t_6, now referenced as t_{worst}, will be the furthest away from t_6 and can be found using the MATLAB function `max`. The implementation of `[max, index] = max(d)` [8] returns the index of t_{worst} and the distance between t_6 and t_{worst} [8].

Assume that the images q and t_6 were taken of the same person. It would be ideal for q to match with t_6, but a variety of parameters may prevent the distance between the two from being close to or equal to 0. When q is input, the script will run and return the minimum distance between q and some $t_i \in T$. Next, assume the image furthest from t_6 is t_7. Thus, $t_{worst} = t_7$. Since t_{worst} is in T, it provides context for the worst possible image q could be matched with. The confidence of the match, c, between q and t_6 can now be calculated using the following equation.

$$c = \left(1 - \frac{p_{new}}{p_{worst}}\right) \cdot 100$$

Note that p_{new} is the distance between the projection of q and t_6, and p_{worst} is the distance between the projection of t_6 and t_{worst}. When $p_{new} \leq p_{worst}$, it is known that the match is valid to some degree. Calculating c gives the percentage accuracy for how good the match is (in relation to t_{worst}.) When $p_{new} > p_{worst}$, it is known that the projection of q is outside of T. In this case, the match would be invalid. The resulting accuracy of a match, known as the percent confidence, is implemented in MATLAB in Listing 8.

Listing 8. Confidence

```
% Calculate confidence of match

% Divide dist(input and match) by
%        dist(match and furthest in training set)

relation_to_furthest = min_dist / max_poss_dist;
confidence = (1 - relation_to_furthest) * 100;

% Not shown:
%    Usage of indices to get data from cell arrays
%    Printing results to command line
```

After thorough experimentation, the algorithm was able to match two pictures of a student, Liam. The images are shown below with the command line output. Fig 6 was the input image, and Figure 7 was included in the MATLAB directory with the rest of the training set, made up of 10 total students.

FIGURE 6.
Image 1 of Liam

FIGURE 7.
Image 2 of Liam

```
The input was matched with -> liam <-
The match has a confidence of 73.6714%.
The input is least similar to -> cristine <-
>>
```

FIGURE 8.
Command Line Output of Successful Match

7. Best Practices in Computation

A large portion of this thesis focused on computation. Therefore, it is prevalent to include information on good coding practices. Students should strive to write legible, concise, and dynamic code. Programmers should aim to write code that both the author and their peers can understand. This can be accomplished by thoroughly commenting each file, providing concise explanations that give context to the function of the file and its sections of complexity. Some places to include these comments include the beginning of the file, at function definitions, and before loops.

Students should also become comfortable with reducing redundancy in their files by creating functions to execute lines that show up multiple times and accomplish a similar task. One should also use variables instead of hard coding values that may change later during the development process. This allows one to change the value of an input without

having to search and replace each one, saving an immense amount of time. Naming variables contextually also ensures that students do not forget the purpose of each one as they read their files. Additionally, one should become comfortable with preallocating memory for the variables they create. If it is known that an array will hold ten numbers, preallocate it beforehand. This allows for faster computation, as the computer will know exactly where to store each number in memory. Finally, one should know how to search for questions online and use forums, such as stackoverflow or Mathworks. Asking a specific question and attaching code examples in posts will lead to the successful discovery of answers. It is better to be over specific in the initial search, working towards a more general question. One should not copy the code from these websites, but rather use them as resources for conceptual understanding and debugging.

8. Equity

When writing code involving humans, it is important to consider the social aspects of the problem being solved. In terms of facial recognition, the solution to creating a working algorithm involves the correct identification of specific humans. It is a problem that directly correlates with identity. While this was not directly applicable to my thesis, it is a question that should be considered by students engaging with computation. How can I write code that is inclusive? If one were to have a training set with the majority of images being of white males, would the algorithm be able to correctly identify the difference between two black women? Large corporations such as Google and Microsoft have created examples of products that are not inclusive, including a racist image classifier [**11**]. It is important to have representation during creation. One must consider the identities they are engaging in order to ensure the creation of successful and equitable computational solutions.

9. Additional Considerations

One extension that was not implemented in this thesis was the usage of Neural Networks. Convolutional Neural Networks (CNN) are the industry standard for facial recognition in 2018, so it would be prevalent to explore the mathematics behind them. Perhaps it would be best to use SVD for compression and CNN's for the actual recognition that takes place. This would allow one to ignore some of the problems discovered while building the recognition algorithm. The confidence of matches made were affected greatly by elements such as growth in facial hair, aging, and variation in lighting. These elements inhibited matches from being made when using the Math Camp training set. Separately, one could focus on algorithm efficiency, making sure that the script would work with higher resolution images and at a larger scale. Finally, researching the aforementioned methods of interpolation for image scaling would be a worthwhile venture. This would allow one to figure out which method is most suited for recognition within a selected margin of error and run time.

10. Appendix

1. Master Script

```
% Master Facial Recognition Script
% Michael Somkuti

% Find and set up our training sets
% Would improve by also returning array of original image sizes
training_sets = setup();
dims = [73 58];   % Default image dimensions
K = 20;           % Number of sing values

% Create face spaces and find each respective one's mean face
% Compression also occurs
% Would improve by making preprocessing only happen once
[face_spaces, mean_faces, max_dists, least_likes] =
    space_creator(training_sets, K, dims);

% Facial recognition
new = 'zzzz.JPG';                                % Set input image
new = preProcessing(new, 1, dims, 0);            % Preprocess
input_image = double(svdPartialSum(new, K));     % Compress
MSE = mean_squared_error(new, input_image, 0); % Gauge error


% Perform recognition
[min_info, max_info, s_index] = recognition(face_spaces, mean_faces,
    input_image, K);

% Calculate confidence of match
% Ratio of distance(input to match) / distance(match to furthest in set)
distance_ratio = min_info(1) / max_dists{s_index}(min_info(2));
confidence = (1 - distance_ratio) * 100;

% Find file name of match and image least like / furthest from the match
[~,name,~] = fileparts(training_sets{s_index, min_info(2)});
least_like = least_likes{s_index}(min_info(2));   % Index of furthest image
[~,furthest_name,~] = fileparts(training_sets{s_index, least_like});

% Ouput results
fprintf(['The input was matched with -> ', name, ' <- \nThe match has a
    confidence of ', num2str(confidence), '%%.\n'])
fprintf(['The input is least similar to -> ', furthest_name, ' <- \n'])
```

2. Face Space Creator

```
function [face_spaces, mean_faces, max_dists, least_likes] =
    space_creator(training_sets, K, dims)
% Function that built off of setup and some of Lije Cao's code to make a
% new function that is dynamic.

face_spaces = cell(size(training_sets, 1), 1); % Cell array to hold sub spaces
mean_faces = cell(size(training_sets, 1), 1); % Hold mean faces
num_images = zeros(size(training_sets, 1), 1); % Hold num ims in each
    training set
mean_images = cell(size(training_sets, 1), 1); % Hold mean face images

plot_count = 1; % Index subplots

num_ts = size(training_sets, 1); % Num of training sets

for i = 1:num_ts
    len = length(training_sets);
    mean_face_image = zeros(dims);

    im_count = 0; % Count num training images in each set
    for j = 1:len
        if training_sets(i,j) ~= ""
            im_count = im_count + 1;
        else
            continue
        end
    end
    num_images(i) = im_count;
    S = zeros(dims(1) * dims(2), im_count);
                                        % Preallocate array to be have
                                        % num rows = dims of images when
                                        % transformed to col vectors and,
                                        % num cols = num of images

    s_plot = subplot(size(training_sets, 1), 2, plot_count);

    % Create facespace
    for j = 1:im_count
        fileName = char(training_sets(i,j)); % Read our new image
        tempIm = preProcessing(fileName, 1, dims, 0);
        tempIm = double(svdPartialSum(tempIm, K)); % Compress our images

        mean_face_image = mean_face_image + tempIm; % Add to mean face
            image

        columnf = tempIm(:); % Turn image into column vector
```

```
        S(:,j) = columnf; % Append our column

        % Plot face space
        hold on
        scatter(s_plot,1:length(columnf),columnf,'.');
        axis([0 length(columnf) 0 255])
        hold off
    end

    plot_count = plot_count + 1; % Adv plot indx

    %
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    % This section of code builds on Lije Cao's code on facial recognition
        from her
    % paper "Singular value decomposition applied to digital image processing."

    fBar = sum(S,2) / im_count; % Calculate the mean face
    mean_faces{i} = fBar;       % Append mean face
    s = size(S,2);

    fBarMat = repmat(fBar,[1,s]); % Create matrix of mean face col vectors

    A = S - fBarMat;   % Shift face space by mean face
                       % Vectorized version of (19) in cao paper

    face_spaces{i} = A; % Append our face space to set of training images
    %
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~



    % Scale m_f_i by num images in training set, append to set
    mean_images{i} = mean_face_image ./ im_count;

    % Plot mean face for each training set
    s_plot = subplot(size(training_sets,1),2,plot_count);
    scatter(s_plot,1:length(fBar),fBar,'.');
    axis([0 length(columnf) 0 255])
    plot_count = plot_count + 1; % Adv plot indx
end

% Display images of each face spaces' mean face
for i = 1:num_ts
    mean_face_test = uint8(mean_images{i});
    figure
    imshow(mean_face_test)
    %TITLE THIS HERE
end
```

```
%
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% For any image, the distance between it and the one least like it is the
% furthest an unknown can be from it while still being a match

max_dists = cell(size(training_sets, 1), 1);     % Max distances
least_likes = cell(size(training_sets, 1), 1);   % Least likes

% For each image in training set, find image least like it and its distance
for i = 1:num_ts
    % Hold distances(input image to least similar in training set)
    maxDistFromOther = zeros(1, num_images(i));

    % For each image, store index of image least like it in set
    % I.E. if image 5 is least like image 1, leastLike(1) = 5
    leastLike = zeros(1, num_images(i));

    for j = 1:num_images(i)
        disp(training_sets)
        fileName = char(training_sets(i,j));
        tempIm = preProcessing(fileName, 1, dims, 0);
        tempImComp = double(svdPartialSum(tempIm, K));

        % Use recognition to find least like and its distance
        [~, max_info] = simpleRecognition(face_spaces{i}, K, mean_faces{i},
            tempImComp);
        max_dist = max_info(1) + mean_squared_error(tempIm, tempImComp,
            0);
        maxDistFromOther(j) = max_dist;
        leastLike(j) = max_info(2); % Store index
    end
    max_dists{i} = maxDistFromOther;
    least_likes{i} = leastLike;
end
end
```

3. Mean Squared Error

```
function error = mean_squared_error(originalImage,
    compressedImage, num)
% Original function
% This function calculates Mean Squared Error through 2
    different approaches
mseSize = size(originalImage) ; % Size of image

% Vectorized version of vanilla Mean Squared Error
if (num == 0)
    error = 1/(mseSize(1)*mseSize(2))*sum(sum((originalImage -
        compressedImage).^2));
end

% Vectorized version of Mean Squared Error utilizing the
    1-Norm
if (num == 1)
    error = 1/(mseSize(1)*mseSize(2)) * norm((originalImage -
        compressedImage), 1);
end
end
```

4. Preprocessing

```
function procIm = preProcessing(Image, gray, resize, contrast)
% Preprocess input image
    procIm = imread(Image); % Read image data

    % Convert to grayscale
    if gray == 1
         procIm = double(rgb2gray(procIm));
    end

    % Equalize contrast
    if contrast == 1
            procIm = histeq(procIm);
    end

    procIm = double(imresize(procIm, resize)); % Scale the
        image
end
```

4. Recognition

```
function [min_info, max_info, s_index] =
    recognition(face_spaces, mean_faces, input_im, K)
% Recognition function

dist_space = zeros(1, size(face_spaces, 1));
x_mats = cell(size(face_spaces, 1), 1);
X_mats = cell(size(face_spaces, 1), 1);

%
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This section of code builds on Lije Cao's code on facial
    recognition from her
% paper "Singular value decomposition applied to digital
    image processing."

for i = 1:size(face_spaces, 1)

    colTemp = input_im(:);      % Vectorize input image
    [U,~,~] = svd(face_spaces{i}); % SVD our matrix of faces

    Ur = U(:, 1:K);          % Compressed basis of face space
    X = Ur' * face_spaces{i}; % Coordinate matrix of face space
    size(X)
    f0 = double((colTemp - mean_faces{i})); %Shift input by
        mean face
    x = Ur' * f0;       % Find coordinate vector of f0
                        % projected onto basis of face space

    fp = Ur * x;        % Find projection of input onto face
        space
    ef = norm(f0 - fp); % Calculate max distance between face
        and new ones

    dist_space(i) = ef; % Append distance from input to face
        space
    x_mats{i} = x;      %       coordinate vector of input
    X_mats{i} = X;      %       coordinate maxtix of face space
end
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

[~, s_index] = min(dist_space); % Find closest face space
```

```
% for i = 1:size(face_spaces, 1) % Uncomment and replace
    s_index with i
                                % and add an end, to check for
                                    all
                                % face spaces

% Calculate magnitude of distance between input and faces in
    face space.
% Take norm of difference between coordinate matrices
D = X_mats{s_index} - x_mats{s_index}*ones(1,
    size(X_mats{s_index},2));
d = sqrt(diag(D' * D));

%
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Plot distances between input image and faces in training set
figure
for i=1:length(d)
    hold on
    plot([0 i],[0 d(i)], '--o') % Plot points and dashed line
    hold off
end
%
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

[dmin, indx] = min(d); % dmin is actual distance from face
min_info = [dmin, indx]; % Return the info about the closest
    match

[dmax, indx] = max(d); % Find the maximum between input
    images, and all
                    % other images in the training set

max_info = [dmax, indx]; % Which face in the facespace is
    most unlike the
                      % match to the input image
% end
end
```

5. Setup

```
function training_sets = setup()
% Original function
%Find all images in subfolders with .jpg extension.
image_files = dir('**/*.jpg')
num_files = length(image_files);

% Find num of folders by locating where images are in the MATLAB directory

all_folders = strings(0,0); % Pre-allocate string array
folder_old = '';
j = 1;

for i = 1:num_files % Check folder that each file is in
    folder = string(image_files(i).folder);
    if folder ~= folder_old
        all_folders(j,1) = folder;
        folder_old = folder;
        j = j+1; % Advance folders
    end
end

num_folders = length(all_folders);

% Preallocate string array holds paths of all images, each row is new
% training set. Images must be sorted beforehand
training_sets = strings(num_folders, num_files);

folder_old = image_files(1).folder; % Get path of our first training set
j = 1; % Which training set paths belong in
k = 1; % Which column image paths should be placed in

for i = 1:num_files
    folder = string(image_files(i).folder);
    if folder == folder_old
        image = strcat(image_files(i).folder,'\',image_files(i).name); % Get full
            path
        training_sets(j,k) = image; % Append path
        k = k+1;
    else
        j = j+1; % Move to next row / training set
        k = 1; % Reset our index to first column

        image = strcat(image_files(i).folder,'\',image_files(i).name);
        training_sets(j,k) = image; % Append path
        folder_old = folder; % Update training set folder
        k = k+1;
    end
end
end
```

6. Simple Recognition

```
function [min_info, max_info] = simpleRecognition(A, K, fBar,
    input_im)
% Would improve by implementing e0 --> max distance a face
    can be

colTemp = input_im(:); % Vectorize input image

%
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This section of code builds on Lije Cao's code on facial
    recognition from her
% paper "Singular value decomposition applied to digital
    image processing."

[U,~,~] = svd(A); % SVD our matrix of faces

Ur = U(:, 1:K);
X = Ur' * A;
f0 = double((colTemp - fBar));
x = Ur' * f0;      % Equation 23 in Cao
fp = Ur * x;       % Projection of f - fbar onto face space
% ef = norm(f0 - fp); % Calculate max distance between face
    and new ones

D = X - x*ones(1, size(X,2));
d = sqrt(diag(D' * D));
[dmin, indx] = min(d); % dmin is actual distance from face
min_info = [dmin, indx]; % Return the info about the closest
    match
%
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

[dmax, indx] = max(d); % Find the maximum between input
    images, and all
                       % other images in the training set

max_info = [dmax, indx]; % Which face in the facespace is
    most unlike the
                         % match to the input image
end
```

7. SVD Partial Sum

```
function AK = svdPartialSum(A,K)

%
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This section of code builds on Lije Cao's code on facial
    recognition from her
% paper "Singular value decomposition applied to digital
    image processing."

A = double(A);                    % Convert image data
[u,s,v]=svd(A);                   % Perform SVD

AK = u(:,1:K)*s(1:K,1:K)*v(:,1:K)'; % Compression takes place
                                  % Cutting off r - k singular
                                      values
                                  % U and V' scaled as well
%
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

% Histogram the image's singular values
s1= s(1:K,1:K);
s2 = s1(:);
n = size(s2,1);
singValues = []; % Array of singular values
for i = 1:n
    if s2(i) ~= 0
        singValues = [singValues; s2(i)];
    end
end
numBins = size(singValues,1);

% figure; hist(singValues,numBins); % Display histogram

AK = uint8(AK);   % Convert back to the image data
% figure; imshow(AK) % Display the converted image
end
```

Bibliography

[1] Lay, David C. Linear Algebra and Its Applications. Addison-Wesley, 2012.

[2] Cao, Lijie. "Singular Value Decomposition Applied to Digital Image Processing." Division of Computing Studies, Arizona State University Polytechnic Campus, Mesa, Arizona State University polytechnic Campus (2006): 1-15.

[3] "rgb2gray." MathWorks, www.mathworks.com/help/matlab/ref/rgb2gray.html.

[4] "Resize." MathWorks, www.mathworks.com/help/images/ref/imresize.html.

[5] Edelman, Alan. "The Singular Value Decomposition (SVD)." Mathematics Lecture Series, IAP 2016, MIT Mathematics. https://math.mit.edu/classes/18.095/2016IAP/lec2/SVD˙Notes.pdf.

[6] "Mean Squared Error." Wikipedia, Wikimedia Foundation, 2 Sept. 2018, https:// en.wikipedia.org/wiki/Mean˙squared˙error.

[7] "min." MathWorks, https://www.mathworks.com/help/matlab/ref/min.html.

[8] "max." MathWorks, https://www.mathworks.com/help/matlab/ref/max.html.

[9] Guoliang Zeng, "Face Recognition with Singular Value Decomposition.", CISSE Proceeding, 2006.

[10] "MATLAB" Wikipedia, Wikimedia Foundation, 3 Dec. 2018, https://en.wikipedia.org/wiki/MATLAB.

[11] "Google Apologises for Photos App's Racist Blunder." BBC News, BBC, 1 July 2015, www.bbc.com/news/technology-33347866.