a)

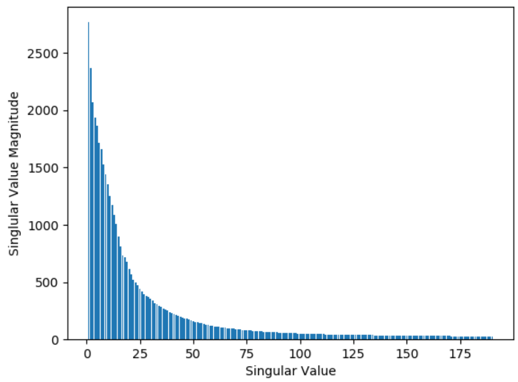

Figure 1: Plot of singular values for corresponding principal components of 190/200 neutral faces

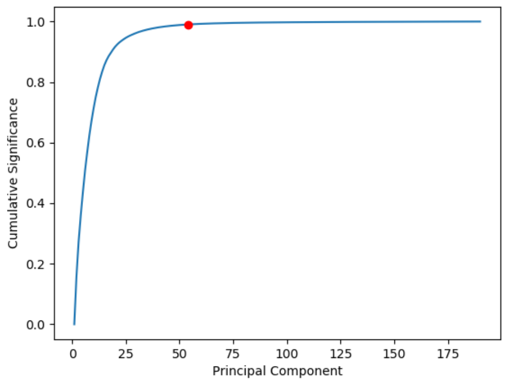

Figure 2: Plot of cumulative significance as additional principal components are used

I decided to use the number of principal components that would encompass 99% of the total sum of the magnitudes of all the singular values. Since the significance of a principal component is determined by the magnitude of its singular value (larger singular values represent larger variance of the data in the direction of the corresponding eigenvector), it is reasonable to expect that if the sum of corresponding eigenvalues for the first 54 principal components equates to 99% of the total sum of all the eigenvalues – then these 54 principal components are able to encapsulate 99% of the data required to reconstruct an image. This allows me to decrease the number of principal components by 72% while only losing 1% of the data (assuming this heuristic approach equates to this conclusion). I have no proof of the validity of this approach – it is just an intuitive, heuristic approach obtained by observing Figures 1 and 2.

b)

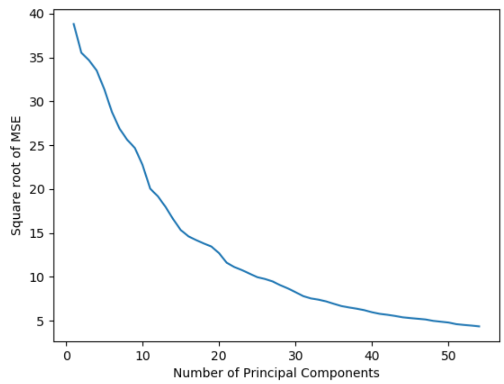

Figure 3: Square root of MSE between neutral face 4a and its reconstruction vs number of principal components used for the reconstruction

The square root of the mean squared error was plotted to aid in the visualization of the plot. As the number of principal components used is increased the MSE decreases exponentially and reaches a minimum value of approximately 25 (5^2). This equates to approximately 1.95% of error*, which is not far off from the 1% goal.

* If the average difference between each pixel is 5 for pixels that range from 0 to 255, then the average percentage of difference between the images is calculated by $\frac{5}{256} * 100 = 1.953125\%$.

c)

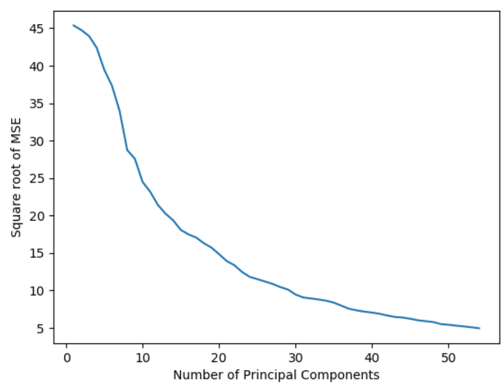

Figure 4: Square root of MSE between smiling face 4b and its reconstruction vs number of principal components used for the reconstruction

The square root of the mean squared error was plotted to aid in the visualization of the plot. As the number of principal components used is increased the MSE decreases exponentially and, just as Figure 3 does, reaches a minimum value of approximately 25 (5^2). This equates to approximately 1.95% of error*, which is not far off from the 1% goal. One noticeable difference is that the MSE hits a maximum value of approximately 45 in Figure 4, whereas the maximum value in Figure 3 is approximately 39. Also, the decrease in Figure 4 has a greater initial slope – indicating the greater impact of additional principal components on the MSE when principal component levels are below ~10. Since the principal components were created from neutral faces, it makes sense that the principal components would perform better at recreating a neutral face as opposed to a smiling one. This seems to be true when the number of principal components is less than or equal to approximately 10. After that, however, the principal components are just as successful in recreating a smiling face as they are in recreating a neutral face. However, this is only one example and it would be interesting to see if this trend holds for all pairs of neutral and smiling faces.

* If the average difference between each pixel is 5 for pixels that range from 0 to 255, then the average percentage of difference between the images is calculated by $\frac{5}{256} * 100 = 1.953125\%$.

d)

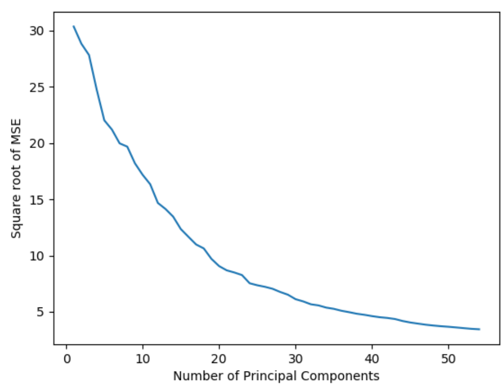

Figure 5: Square root of MSE between neutral face 195a and its reconstruction vs number of principal components used for the reconstruction

The square root of the mean squared error was plotted to aid in the visualization of the plot. As the number of principal components used is increased the MSE decreases exponentially and reaches a minimum value of approximately 16 (4^2). This equates to approximately 1.56% of error*, which is even closer to the 1% goal than previous attempts. One noticeable difference is that the MSE hits a maximum value of approximately 31 in Figure 5, whereas the maximum values were approximately 39 and 45 in Figures 3 and 4 respectively. It's interesting that the minimum MSE values are approximately the same, if not smaller, when the image to reconstruct is not used in the initial construction of the principal components. However, it may just have been that the faces selected for recreation in Figures 4 and 5 were easy to reconstruct for the set of principal components.

* If the average difference between each pixel is 4 for pixels that range from 0 to 255, then the average percentage of difference between the images is calculated by $\frac{4}{256} * 100 = 1.5625\%$.

e)

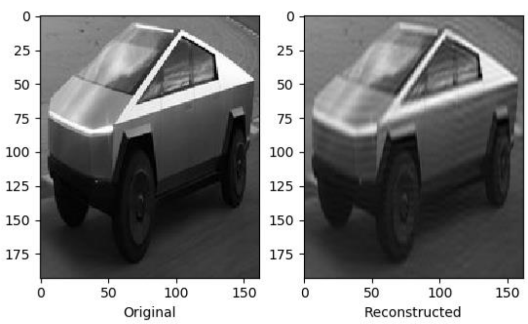

Figure 6: Reconstruction of the Tesla Cybertruck using all principal components

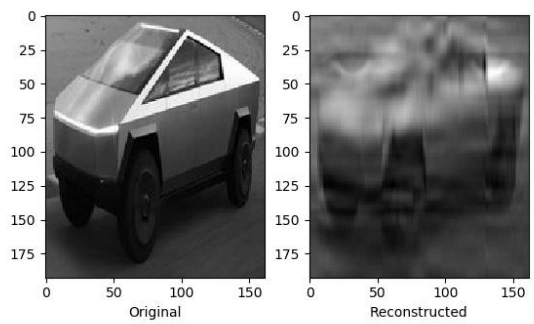

Figure 7: Reconstruction of the Tesla Cybertruck using 15 principal components

I tested the ability for the principal components created by using neutral faces to reconstruct a Tesla Cybertruck. The reconstruction in Figure 6 is quite good. Although some of the sharper detail is lost, one can clearly tell that it is a car. Just to see what would happen, I attempted a reconstruction of the truck using only 15 principal components. You can see a blurry car in the background accompanied by a pair of creepy eyes. I'm surprised at how good the reconstruction is when using all the principal components – I was expecting the results in Figure 6 to be about as good as they are in Figure 7.

f)

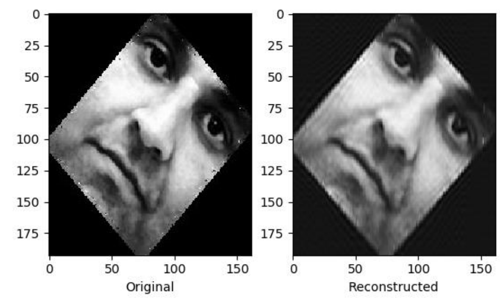

Figure 8: Reconstruction of neutral face 4a rotated 45 degrees using all principal components

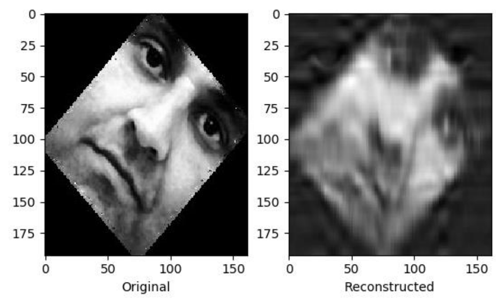

Figure 9: Reconstruction of neutral face 4a rotated 45 degrees using 15 principal components

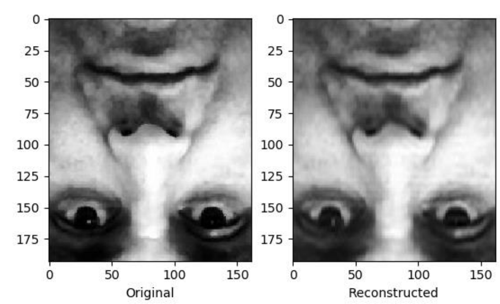

Figure 10: Reconstruction of neutral face 4a rotated 180 degrees using all principal components

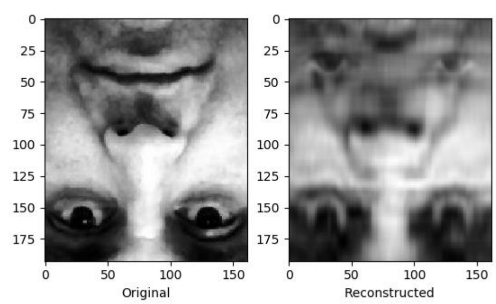

Figure 11: Reconstruction of neutral face 4a rotated 180 degrees using 15 principal components

I tested the ability for the principal components created from neutral faces to reconstruct image 4a (an image in the set of faces used to create the principal components) after being rotated. In Figures 8 and 10 the recreation is very good – there is just a bit of detail lost in the finer features of the face. When only using 15 principal components the creepy eyes come back, but one can begin to see the eyes and part of the nose of the original image (in the correct orientation). I'm also impressed with the reconstruction results of Figure 8, more so than the reconstruction of Figure 10. I had assumed the greatest variation would be in the vertical direction and the second greatest variation to be in the horizontal direction (simply from observing the organization of features on the faces). Since Figure 10 keeps a vertical and horizontal organization of features I expected the reconstruction to be fine. In Figure 8, however, the orientation of features is now in a diagonal direction and I expected this to hinder the ability for the principal components to reconstruct the image to the point that Figure 8 would look like Figure 9.

Appendix

Code:

```
###################################
# Joseph Bell                     #
# ECE 269 PCA Eigenfaces Project #
# 12/1/2019                       #
###################################

import matplotlib.pyplot as plt
import matplotlib.image as mpltimg
import os
import numpy as np
import math


#########################################################
# input: absolute path to folder containing face images#
# returns: m * L * N array, where m = number of face   #
#          images, L rows in image, N cols in image    #
#########################################################
def vectorize_face_folder(face_folder_path): #converts face images to matrix
    face_images = os.listdir(face_folder_path)
    first_face_path = os.path.join(face_folder_path,face_images[0]) # Used to initialize np.array to store all faces - assumes all faces are same size as first image
    num_of_faces = len(face_images)                                    # which is true in this case as they've all been cropped
    first_face = mpltimg.imread(first_face_path)
    faces_vector = np.zeros((num_of_faces, first_face.shape[0], first_face.shape[1]))

    for i in range(num_of_faces):
        face_path = os.path.join(face_folder_path, face_images[i])
        face_read = mpltimg.imread(face_path)
        faces_vector[i,:,:] = face_read

    return faces_vector

#########################################################
# input: m*L*N array, where m = number of face         #
#          images, L rows in image, N cols in image    #
# returns: L*N array which is the mean of m face       #
```

```
#          images                                       #
#######################################################
def calculate_mean_face(faces_vector): #calculates mean of all matrices that represent faces
    num_of_faces = faces_vector.shape[0]
    face_summation = np.zeros(faces_vector[0].shape)
    for i in range(num_of_faces):
        face_summation = face_summation + faces_vector[i,:,:]

    mean_face = face_summation/num_of_faces

    return mean_face

#################################################################
# input (faces_vector): m*L*N array, where m = number of face    #
#                       images, L rows in image, N cols in image #
# input (mean_face): L*N array of mean face of m face images     #
# returns: m*L*n array of faces_vector[i,:,:] - mean_face        #
#                                                                #
#################################################################
def calculate_mean_adjusted_faces(faces_vector, mean_face):
    mean_adjusted_faces = np.zeros(faces_vector.shape)
    num_of_faces = faces_vector.shape[0]
    for i in range(num_of_faces):
        mean_adjusted_faces[i,:,:] = faces_vector[i,:,:] - mean_face

    return mean_adjusted_faces

#################################################################
# input: m*L*N array, where m = number of face                   #
#        images, L rows in image, N cols in image                #
# returns: L*L array which is the covariance matrix              #
#################################################################
def calculate_covariance_matrix(mean_adjusted_faces):
    num_of_faces = mean_adjusted_faces.shape[0]
    first_face = mean_adjusted_faces[0] #used for initializing AAT_face_summation shape
    AAT = np.zeros((first_face.shape[0], first_face.shape[0])) # where A is a mean adjusted face matrix

    # Used 3D arrays as the computation time is almost identical this way, however I did read through the quicker method in the paper they used
    # (they used 2D arrays) and understand it (i.e. why it's quicker and how it results in the same eigenvalues and scaled eigenvectors.
    # Also, had already implemented 3D array method in all my code and it would have been a hassle to change everything
    # to work with 2D dimensions that also would've been of minor benefit to computation time (and less intuitive for me - 3D arrays make more
    # sense to me when dealing with images.

    for i in range(num_of_faces):
        face = mean_adjusted_faces[i,:,:]
        AAT = AAT + np.matmul(face, face.T) #if face is 193*162 then AAT_face_summation is 193*193

    return AAT

```

```
##################################################################
# input: L*L array which is the covariance matrix                #
# returns: L, L*L array of eigenvalues and eigenvectors          #
##################################################################
def calculate_eigenpairs(covariance_matrix):
    eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)

    return eigenvalues, eigenvectors

########################################################################
# input: L array of eigenvalues                                        #
# input: L*L array of eigenvectors                                     #
# input: num_of_faces is number of PCs for face space                  #
# returns: L*L, L array of eigenvectors and eigenvalues for face space #
########################################################################
def select_face_space(eigenvalues, eigenvectors, num_of_faces):
    sorted_eigenvalues = sorted(eigenvalues, reverse=True)
    face_space_eigenvectors = np.zeros((eigenvectors.shape[0], num_of_faces)) #each column is an eigenvector
    face_space_eigenvalues = np.zeros(num_of_faces)
    for i in range(num_of_faces):
        eigenvalue = sorted_eigenvalues[i]
        corresponding_eigenvector_index = np.where(eigenvalues == eigenvalue)[0][0]

        corresponding_eigenvector = eigenvectors[:, corresponding_eigenvector_index]

        face_space_eigenvectors[:,i] = corresponding_eigenvector
        face_space_eigenvalues[i] = eigenvalue

    return face_space_eigenvectors, face_space_eigenvalues

########################################################################
# input: L array of eigenvalues for face space                         #
# return: no return, just creates bar plot for singular values         #
########################################################################
def create_principal_component_bar_plot(face_space_eigenvalues):
    number_of_eigenvalues = face_space_eigenvalues.shape[0]
    singular_values = np.sqrt(face_space_eigenvalues)
    eigenvalue_index_array = np.linspace(1, number_of_eigenvalues, number_of_eigenvalues)
    plt.bar(eigenvalue_index_array, singular_values)
    plt.xlabel('Singular Value')
    plt.ylabel('Singlular Value Magnitude')
    plt.show()

#########################################################################
# input (face_to_reconstruct): L*N image to reconstruct using face space#
# input (face_space_eigenvectors): L*L array of face space eigenvectors #
# input (mean_face): L*N mean face (constructed from 190 neutral faces) #
# returns: L*N reconstructed image                                      #
#########################################################################
def principal_component_image_reconstruction(face_to_reconstruct, face_space_eigenvectors, mean_face): #takes in a single image and reconstructs it by using principal components
    number_of_faces = face_space_eigenvectors.shape[1]
```

```
    reconstructed_image = np.zeros(face_to_reconstruct.shape)
    mean_adjusted_face = face_to_reconstruct - mean_face
    for i in range(number_of_faces):
        eigenvector = face_space_eigenvectors[:,i].reshape((face_space_eigenvectors[:,i].shape[0],1))
        w = np.matmul(eigenvector.T, mean_adjusted_face)
        w_u = np.matmul(eigenvector, w)
        reconstructed_image = reconstructed_image + w_u

    reconstructed_image = reconstructed_image + mean_face
    return reconstructed_image

########################################################################
# input (image_reconstruction): L*N reconstructed image                #
# input (face_to_reconstruct): original image used for reconstruction  #
# returns: Mean Squared Error of inputs                                #
########################################################################
def calculate_MSE(image_reconstruction, face_to_reconstruct):
    L = image_reconstruction.shape[0]
    N = image_reconstruction.shape[1]

    MSE = np.sum(np.square(image_reconstruction - face_to_reconstruct))/(L*N)

    return MSE

########################################################################
# input: L*N rgb image                                                 #
# returns: L*N gray image                                              #
########################################################################
def rgb2gray(image):
    gray_image = np.zeros((image.shape[0], image.shape[1]))
    gray_image[:,:] = image[:,:,0]*0.2989 + image[:,:,1]*0.5870 + image[:,:,2]*0.1140

    return gray_image

########################################################################
# input: L*L array of eigenvectors                                     #
# returns: L array of cumulative significance used for plotting        #
########################################################################
def calculate_significance_of_eigenvalues(eigenvalues):
    total_sum = sum(eigenvalues)
    cumulative_significance = np.zeros(eigenvalues.shape[0])
    for i in range(eigenvalues.shape[0]):
        cumulative_significance[i] = sum(eigenvalues[:i])/total_sum
    return cumulative_significance

# Folder used to create Principal Components
face_folder = 'neutral_faces'

# Folder used to access images to reconstruct
face_folder_reconstruct = 'neutral_faces'
#face_folder_reconstruct = 'smiling_faces'
```

```
# Creating absolute paths
face_folder_path = os.path.join(os.getcwd(), face_folder)
face_folder_path_reconstruct = os.path.join(os.getcwd(), face_folder_reconstruct)

#### Below are all used one at a time for different reconstructions ###
#face_to_reconstruct = mpltimg.imread(os.path.join(face_folder_path_reconstruct, '4b.jpg'))
#face_to_reconstruct = mpltimg.imread(os.path.join(face_folder_path_reconstruct, '195a.jpg')) #to be used later to test MSE of image reconstruction
#face_to_reconstruct = rgb2gray(mpltimg.imread(os.path.join(os.getcwd(),'tesla_cybertruck.jpg')))
face_to_reconstruct = mpltimg.imread(os.path.join(os.getcwd(),'neutral_rotate_45.jpg'))

###### Code to calculate Principal Components ######
faces_vector = vectorize_face_folder(face_folder_path=face_folder_path)
mean_face = calculate_mean_face(faces_vector=faces_vector)

mean_adjusted_faces = calculate_mean_adjusted_faces(faces_vector=faces_vector, mean_face=mean_face)
covariance_matrix = calculate_covariance_matrix(mean_adjusted_faces=mean_adjusted_faces)
eigenvalues, eigenvectors = calculate_eigenpairs(covariance_matrix=covariance_matrix) #eigenvectors have euclidean norm of 1

#99% significance occurs at 54
num_of_PC = 54
face_space_eigenvectors, face_space_eigenvalues = select_face_space(eigenvalues=eigenvalues, eigenvectors=eigenvectors, num_of_faces=num_of_PC) #max value for num_of_faces is total number of eigenvectors, number is chosen heuristically
create_principal_component_bar_plot(face_space_eigenvalues=face_space_eigenvalues)
cumulative_significance = calculate_significance_of_eigenvalues(eigenvalues=face_space_eigenvalues)
##################################################################################################################3

# Uncomment code to plot cumulative significance chart - only required once to determine how many principal components to used
# for the project. 99% of the sum of magnitudes of all eigenvalues is obtained at eigenvalue 54. Therefore, 54 principal components
# are used.

indices = np.linspace(1, cumulative_significance.shape[0], cumulative_significance.shape[0])
plt.plot(indices, cumulative_significance)
plt.plot(54,0.99,'or')
plt.xlabel('Principal Component')
plt.ylabel('Cumulative Significance')
plt.show()

```

```python
# Reconstruct image
image_reconstruction = principal_component_image_reconstruction(face_to_reconstruct=face_to_reconstruct, face_space_eigenvectors=face_space_eigenvectors, mean_face=mean_face)

# Plotting original and reconstructed image next to each other
f = plt.figure()
f.add_subplot(1,2,1)
plt.imshow(face_to_reconstruct, cmap='gray')
plt.xlabel('Original')
f.add_subplot(1,2,2)
plt.imshow(image_reconstruction, cmap='gray')
plt.xlabel('Reconstructed')
plt.show()

###### Uncomment code to calculate MSE and create plots of MSE vs number of principal components #######

MSE = calculate_MSE(image_reconstruction=image_reconstruction, face_to_reconstruct=face_to_reconstruct)
PC_indices = np.linspace(1,num_of_PC,num_of_PC)
MSE_array = np.zeros(num_of_PC)

for i in range(num_of_PC):
    face_space_eigenvectors, face_space_eigenvalues = select_face_space(eigenvalues=eigenvalues, eigenvectors=eigenvectors, num_of_faces=i)
    image_reconstruction = principal_component_image_reconstruction(face_to_reconstruct=face_to_reconstruct, face_space_eigenvectors=face_space_eigenvectors, mean_face=mean_face)
    MSE = calculate_MSE(image_reconstruction=image_reconstruction, face_to_reconstruct=face_to_reconstruct)
    MSE_array[i] = MSE

plt.plot(PC_indices, np.sqrt(MSE_array))
plt.xlabel('Number of Principal Components')
plt.ylabel('Square root of MSE')
plt.show()
```