Problem 2 (20 Points): SVD: Image Compression

Problem 2 SVD: Image Compression

```
# I am gonna put all the libraries i use here
import numpy as np # to average the rgb values and save it as grayscale
import matplotlib.pyplot as plt # to plot the given mandrill image in grayscale
from PIL import Image # to load the image from our folder
import pandas as pd # I am using this to show the final results in df
from sklearn.preprocessing import StandardScaler # for question3 part 6 [finding normalized z score]
```

```
Part 1:
```

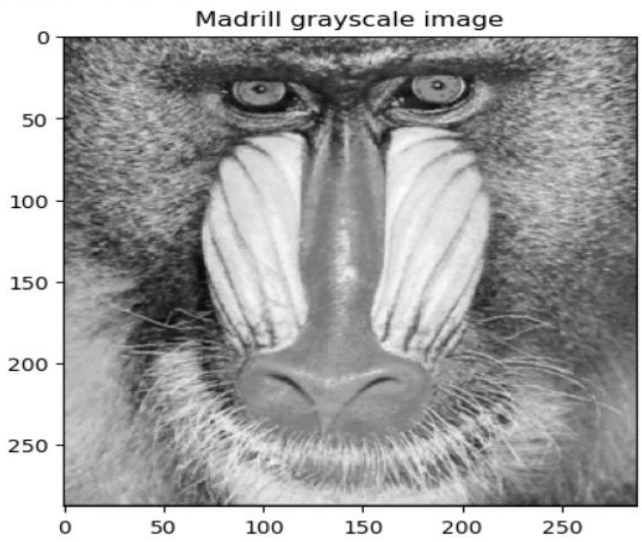
```
# Loading the mandrill image
img = Image.open('D:/Projects/assignments/intro_ml/supp_HW4/supp/mandrill_color.png')
img_array = np.array(img)
print(img_array.shape)
```

```
(512, 512, 3)
```

The given image is in (512,512, 3) matrix, as asked in the question i will resize it to 288,288.

```
img_resized = img.resize((288, 288))
img_arr = np.array(img_resized)
# To convert to grayscale, We average the R, G, B values
X = np.mean(image_array, axis= 2) #we use axis=2 to take the mean across the third dimension
print("Shape of X is: ", X.shape)
# Lets display the grayscale image now
plt.imshow(X, cmap='gray')
plt.title('Madrill grayscale image')
plt.show()
```

```
Shape of X is:  (288, 288)
```



Part 2

```
# using default SVD function from numpy
U, Sigma, Vt = np.linalg.svd(X, full_matrices=False)
# Print both the unitary matrices:
print("U matrix:\n", U)
print("-"*80)
print("V matrix: \n", Vt)
print("-"*80)
print("Sigma matrix:\n", Sigma)
```

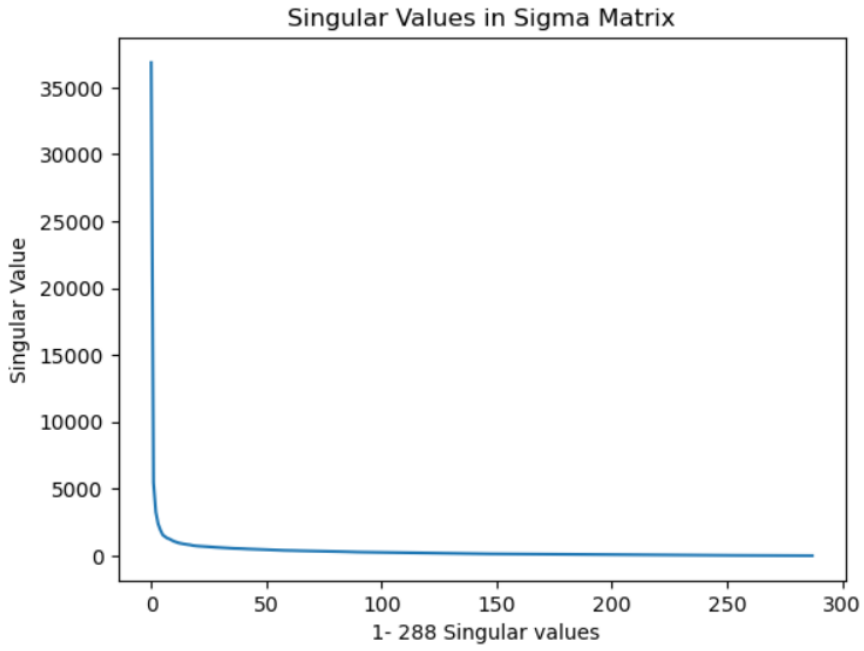
```
U matrix:
 [[-0.05428517  0.04254675 -0.04249892 ...  0.01862328 -0.00870827
   0.01057283]
 [-0.05288364  0.04485292 -0.04152562 ... -0.02671966  0.01042844
  -0.00273105]
 [-0.05215826  0.06385132 -0.03482105 ...  0.00832023 -0.00766202
  -0.04709541]
 ...
 [-0.0638747  -0.04232971  0.07336338 ... -0.15384523  0.09385182
  -0.08671226]
 [-0.06472307 -0.04559619  0.08479452 ... -0.07040719 -0.0296205
  -0.05694269]
 [-0.03393585 -0.01548418  0.03694751 ...  0.0025049   0.18146724
   0.09494565]]
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
V matrix:
 [[-0.05543511 -0.05563651 -0.05576337 ... -0.04723973 -0.04695631
  -0.04717478]
 [ 0.00714523  0.01283016  0.01054174 ...  0.05476757  0.0512919
   0.05256895]
 [ 0.14250342  0.14829659  0.15285403 ...  0.02039841  0.02332461
   0.02258692]
 ...
 [-0.0388507   0.00209704 -0.02154807 ... -0.12910313  0.03873677
   0.00560337]
 [-0.04105655  0.04675514 -0.03729384 ...  0.02981681 -0.06960599
   0.08747385]
 [-0.05256062  0.00705469 -0.03416634 ... -0.20744595  0.08159744
   0.03109849]]
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
Sigma matrix:
 [3.68518555e+04 5.46654135e+03 3.26948457e+03 2.38574665e+03
 1.91821715e+03 1.55420595e+03 1.42047286e+03 1.30527239e+03
 1.24318104e+03 1.14595478e+03 1.06756797e+03 1.00970673e+03
 9.48050109e+02 9.10344480e+02 8.81562534e+02 8.51370974e+02
 8.27620582e+02 8.05021052e+02 7.60294668e+02 7.39723529e+02
 7.21911828e+02 7.12433447e+02 6.96906255e+02 6.92871413e+02
 6.76056026e+02 6.64764492e+02 6.53236629e+02 6.35591617e+02
 6.26408333e+02 6.14672019e+02 6.10612579e+02 5.94132830e+02
```

```
print(Sigma.shape)
```

```
(288,)
```

Sigma is a diagonal matrix and the 288 values displayed above are the diagonal elements. These elements are in order of magnitude, when we use SVD to compress the image, singular values plays the important role, the values in the start carry most of the information and thats how we choose k to get most information from image even after choosing K much lesser than 288(actual size of image).

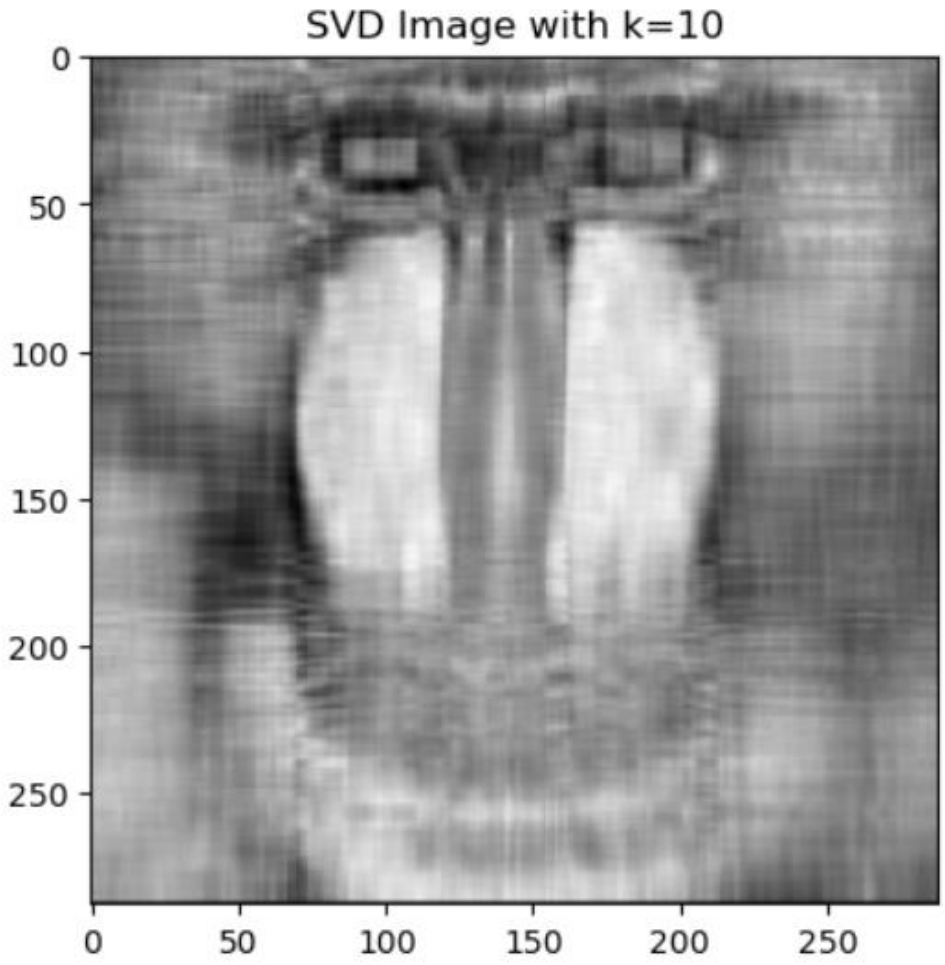
```
#plotting sigma values in the order of their magnitude:
plt.plot(Sigma)
plt.title('Singular Values in Sigma Matrix')
plt.xlabel(' 1- 288 Singular values')
plt.ylabel('Singular Value')
plt.show()
```



Part 3

```
#Both part 3 and 4 are about reconstructing the image by changing values of K with 10, 20,
#40 and 60. So I will define a function to reconstruct the image and call it for all Ks.



def SVD_image(U, Sigma, Vt, k):
    U_k = U[:, :k] # choosing first K columns
    Sigma_k = np.diag(Sigma[:k]) # choosing first K diagonal elements[singular values]
    Vt_k = Vt[:k, :] # choosing first K rows [its a transpose, so we are considering
                        #the first k eigen vectors]
    X_k = np.dot(U_k, np.dot(Sigma_k, Vt_k)) #[ Uk *(sigma_k * Vt_k)]
    # Compression ratio
    compressed_size = U_k.size + Sigma_k.size + Vt_k.size # [numbers needed to store
                                                # the compressed image]
    compression_ratio = compressed_size / original_size #[formula as mentioned in question]
    return X_k, compression_ratio, compressed_size

original_size = X.size
print("The original size is 288 * 288 = \n", original_size)
# Reconstructing the image using the top 10 singular values
k = 10
X_svd, compression_ratio, compressed_size = SVD_image(U, Sigma, Vt, k)

# Display the reconstructed image
plt.imshow(X_svd, cmap='gray')
plt.title(f'SVD Image with k={k}')
plt.show()

print(f'Compressed size with k = {k}: {compressed_size}')
print(f'Compression ratio with k={k}: {compression_ratio}')
```

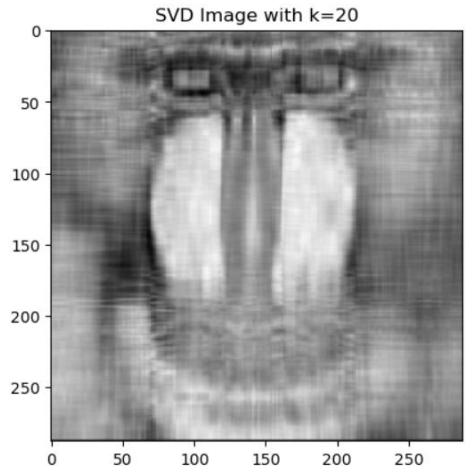
```
The original size is 288 * 288 =
 82944
```

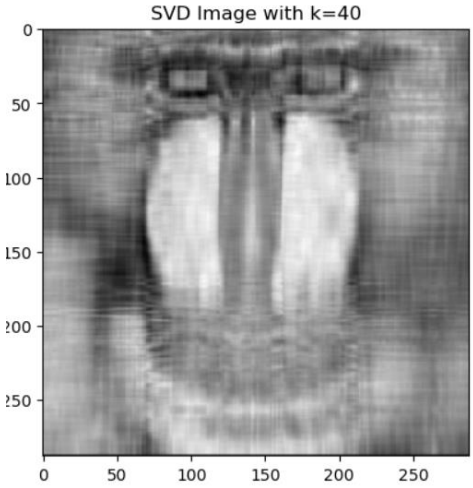



```
Compressed size with k = 10: 5860
Compression ratio with k=10: 0.07065007716049383
```

Part 4

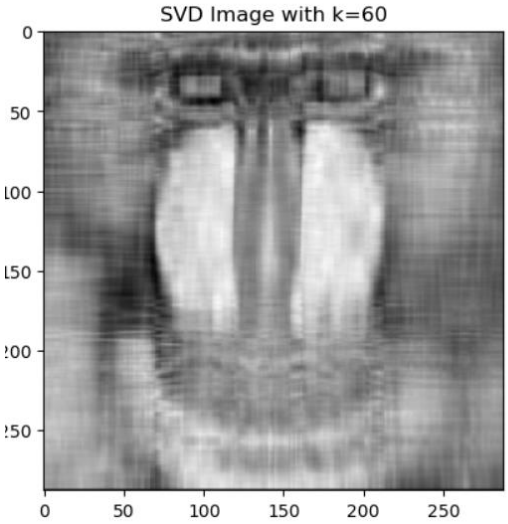
```
# Reconstruction using SVD and store results for each k
K = [20, 40, 60]
results = []
for i in K:
    X_svd, compression_ratio, compressed_size = SVD_image(U, Sigma, Vt, i)
    plt.imshow(X_k, cmap='gray')
    plt.title(f'SVD Image with k={i}')
    plt.show()
    # saving the results in a list for now
    results.append((i, compressed_size, compression_ratio))

# Creating dataframe to show the results with 'k', 'Compressed Size', 'Compression Ratio'
# as columns

df_results = pd.DataFrame(results, columns=['k', 'Compressed Size', 'Compression Ratio'])
print(df_results)
```





	k	Compressed Size	Compression Ratio
0	20	11920	0.143711
1	40	24640	0.297068
2	60	38160	0.460069

With increase in K, the numbers required to store the image increases, which is expected since with k we increase the size of U, sigma, Vt matrices. So In my observation I can see a significant difference in compressed images and original images in terms of clarity, but when working with large networks and deep learning models, it is advisable to work with compressed images, as storing such high clarity images costs hugely in large scale datasets. Also, I do not observe a significant difference in compressed images for k = 20, k= 60. But the compressed ratio has difference of approx 32%. So between k=20,40,60 there isnt much benefit with increasing in k.



Problem 3 (20 Points): PCA: Best Places to Live

Problem 3 [PCA: Best Places to Live]

```
import pandas as pd
import matplotlib.pyplot as plt
```

Part 1

```
# Given 9 columns:
column_names = ["City", "Climate", "Housing", "Healthcare", "Crime", "Transportation", "Education", "Arts", "Recreation", "Economic_Welfare"]
#reading the given data
data = pd.read_csv("D:/Projects/assignments/intro_ml/supp_HW4/supp/places.txt", names=column_names,  header=None)
print(data.head())
```

```
                    City                                       Climate  \
0                Abilene  TX                                   521...
1                  Akron  OH\t\t\t\t\t   575 8138 1656 886 4883 2438 556...
2                 Albany  GA\t\t\t\t\t   468 7339 618 970 2531 2560 237 ...
3  Albany-Schenectady-Troy  NY\t\t\t   476 7908 1431 610 6883 3399 4655 16...
4            Albuquerque  NM\t\t\t\t\t   659 8393 1853 1483 6558 3026 44...

  Housing  Healthcare  Crime  Transportation  Education  Arts  Recreation  \
0     NaN         NaN    NaN             NaN        NaN   NaN         NaN
1     NaN         NaN    NaN             NaN        NaN   NaN         NaN
2     NaN         NaN    NaN             NaN        NaN   NaN         NaN
3     NaN         NaN    NaN             NaN        NaN   NaN         NaN
4     NaN         NaN    NaN             NaN        NaN   NaN         NaN

   Economic_Welfare
0               NaN
1               NaN
2               NaN
3               NaN
4               NaN
```

Clearly data can be seen with large number of NaN, which indicates that there is issue with loading the data. I can see in the text file these columns are seperated by uneven number of spaces. To counter this issue I will use delimiter with " ", indicated the columns are seperated by spaces and as mentioned in the question i am using only first 10 columns and leaving the rest. Also to counter the issue of uneven number of spaces, I am using skipinitalspace to be True as it skips extra spaces after a delimiter, which helps to ignore unnecessary spaces between values that end up giving NaN to the end columns.

```
# Given 9 columns:
column_names = ["City", "Climate", "Housing", "Healthcare", "Crime", "Transportation", "Education", "Arts", "Recreation", "Economic_Welfare"]
# reading the given text file into
data = pd.read_csv("D:/Projects/assignments/intro_ml/supp_HW4/supp/places.txt", delimiter=' ', names=column_names, usecols=range(10), header=None, skipin
# Display the table
print(data.head())
```

```
                          City  Climate  Housing  Healthcare  Crime  \
0                   Abilene,TX      521     6200         237    923
1               Akron,OH\t\t\t\t\t      575     8138        1656    886
2              Albany,GA\t\t\t\t\t      468     7339         618    970
3  Albany-Schenectady-Troy,NY\t\t\t      476     7908        1431    610
4         Albuquerque,NM\t\t\t\t\t      659     8393        1853   1483

   Transportation  Education  Arts  Recreation  Economic_Welfare
0            4031       2757   996        1405              7633
1            4883       2438  5564        2632              4350
2            2531       2560   237         859              5250
3            6883       3399  4655        1617              5864
4            6558       3026  4496        2612              5727
```

Coming to city column, clear we can say there is need of extra cleaning. Here we only need the city and state is not required. Every row has state after ','. So, I am gonna redefine city column with only first string in each row before "," and ignore the rest.

```
# Process the "City" column to consider only the first word before a comma
data["City"] = data["City"].str.split(',').str[0]
print(data.head())
```

```
                    City  Climate  Housing  Healthcare  Crime  \
0                Abilene      521     6200         237    923
1                  Akron      575     8138        1656    886
2                 Albany      468     7339         618    970
3  Albany-Schenectady-Troy      476     7908        1431    610
4             Albuquerque      659     8393        1853   1483

   Transportation  Education  Arts  Recreation  Economic_Welfare
0            4031       2757   996        1405              7633
1            4883       2438  5564        2632              4350
2            2531       2560   237         859              5250
3            6883       3399  4655        1617              5864
4            6558       3026  4496        2612              5727
```

```
print(data.isna().sum())
```

```
City                0
Climate             0
Housing             0
Healthcare          0
Crime               0
Transportation      0
Education           0
Arts                0
Recreation          0
Economic_Welfare    0
dtype: int64
```

We can see the data is cleaned with 0 null values. Now lets delve deeper into the problem.

Part 2:

```
X_matrix = data.drop('City', axis=1) # we are only considering the ratings(numerical values)
```

```
X = X_matrix.apply(np.log10) # performing log base 10 on data matrix X.[This is one method of standardization.
print(X.head())
print(len(X))
```

```
    Climate   Housing  Healthcare     Crime  Transportation  Education  \
0  2.716838  3.792392    2.374748  2.965202        3.605413   3.440437
1  2.759668  3.910518    3.219060  2.947434        3.688687   3.387034
2  2.670246  3.865637    2.790988  2.986772        3.403292   3.408240
3  2.677607  3.898067    3.155640  2.785330        3.837778   3.531351
4  2.818885  3.923917    3.267875  3.171141        3.816771   3.480869

       Arts  Recreation  Economic_Welfare
0  2.998259    3.147676          3.882695
1  3.745387    3.420286          3.638489
2  2.374748    2.933993          3.720159
3  3.667920    3.208710          3.768194
4  3.652826    3.416973          3.757927
329
```

Data matrix X whose rows are 9-dimensional vectors representing the different cities with 329 rows in total.

Part 3: .

```
# center the data points first by computing the mean data vector μ and subtracting it from every point.[from the question]
rating_mean = X.mean(axis=0) # axis = 0 this run through each element in the column
centered_data = X - rating_mean
print(centered_data)
```

```
      Climate   Housing  Healthcare     Crime  Transportation  Education  \
0   -0.001656 -0.115099   -0.580738  0.013575        0.008211  -0.006170
1    0.041174  0.003027    0.263574 -0.004193        0.091485  -0.059573
2   -0.048248 -0.041854   -0.164498  0.035145       -0.193910  -0.038367
3   -0.040887 -0.009424    0.200153 -0.166297        0.240576   0.084745
4    0.100391  0.016426    0.312389  0.219514        0.219570   0.034262
..        ...       ...         ...       ...             ...        ...
324  0.031242  0.032776    0.300991 -0.119118       -0.035743   0.071776
325  0.009860 -0.098605   -0.454427  0.092128       -0.025376  -0.050233
326  0.013900  0.015286   -0.102397 -0.308174       -0.241750   0.016240
327  0.037381 -0.061092    0.084720  0.020576       -0.069057   0.018776
328  0.065409 -0.011241   -0.629151  0.119887       -0.155036  -0.068754

         Arts  Recreation  Economic_Welfare
0   -0.207950   -0.078891          0.148513
1    0.539178    0.193719         -0.095693
2   -0.831461   -0.292574         -0.014023
3    0.461710   -0.017857          0.034012
4    0.446617    0.190406          0.023745
..        ...         ...               ...
324  0.045185   -0.267526         -0.031752
325 -0.207950    0.103846         -0.036430
326 -0.196759   -0.301255         -0.039928
327  0.240483   -0.103696         -0.143786
328 -1.119850   -0.263725         -0.062639

[329 rows x 9 columns]
```

```
# performing SVD on centered data:
U, Sigma, Vt = np.linalg.svd(centered_data, full_matrices=False) # i am setting the full_matrices = false so that matrices will be restricted from fillin
                                                                 # with zero values.

# The principal components are the rows of Vt
principal_components = Vt.T # In the 2nd problem we mentioned matrix v has the eigen vectors which are the principal components of the
                            # given data matrix, hence we define principal components as Vt.T[rows of Vt are eigen vectors, hence we transform it]

# Display the principal components
print("Principal components (Vt):\n", principal_components)
print("Shape of Principal components:\n", principal_components.shape)
```

```
Principal components (Vt):
 [[ 0.03507288  0.0088782  -0.14087477  0.15274476 -0.39751159  0.83129501
  -0.0559096  -0.31490125 -0.06448925]
 [ 0.09335159  0.00923057 -0.12884967 -0.17838233 -0.1753133   0.20905725
   0.6958923   0.61361583  0.08687702]
 [ 0.40776448 -0.85853187 -0.27605769 -0.03516139 -0.05032469 -0.08967085
  -0.06245284 -0.0210358  -0.06550333]
 [ 0.10044536  0.22042372 -0.5926882   0.72366303  0.01345714 -0.16401885
  -0.05553037  0.1823479   0.05421223]
 [ 0.15009714  0.05920111 -0.22089816 -0.12620531  0.86996951  0.37244964
   0.0724604  -0.05714199 -0.07183942]
 [ 0.03215319 -0.06058858 -0.0081447  -0.00519693  0.04779772  0.02362804
   0.05738567 -0.20447312  0.97327107]
 [ 0.87434057  0.30380632  0.36328732  0.08111571 -0.05506994 -0.02812147
  -0.0232698  -0.01673991 -0.00525656]
 [ 0.15899622  0.33399255 -0.58362605 -0.62822609 -0.21328989 -0.14179906
  -0.23451524 -0.08353911  0.01749472]
 [ 0.01949418  0.0561011  -0.12085337  0.05216997 -0.02965242 -0.26481279
   0.66448592 -0.66203179 -0.16826376]]
Shape of Principal components:
 (9, 9)
```

These principal components are arranged in their respective eigen value magnitude wise. Principal components in the begining capture majority of the variance which can be later chosen to reduce the dimension of data matrix by projection with selected k principal components.

Part 4:

```
v1 = principal_components[:, 0] #v1
v2 = principal_components[:, 1] # v2

print("First principal component (v1): \n", v1)
print("Second principal component (v2): \n", v2)
```

```
First principal component (v1):
 [0.03507288 0.09335159 0.40776448 0.10044536 0.15009714 0.03215319
 0.87434057 0.15899622 0.01949418]
Second principal component (v2):
 [ 0.0088782   0.00923057 -0.85853187  0.22042372  0.05920111 -0.06058858
  0.30380632  0.33399255  0.0561011 ]
```

Column order: ["Climate", "Housing", "Healthcare", "Crime", "Transportation", "Education", "Arts", "Recreation", "Economic_Welfare"] for reference. PCA1: Maximum magnitude values in V1: 0.874 --> Art 0.407 --> Healthcare 0.158 --> Recreation Art (0.874) has the largest positive weight, which means this component is most strongly influenced by the "Arts" factor. Cities that have higher scores in "Arts" tend to be aligned with this principal component. And Factors such as "Climate" (0.035) and "Economic Welfare" (0.019) have very low contributions to PC1, indicating that they are less influential in defining this principal component

PCA2: Maximum magnitude values in V2: -0.858 --> Healthcare 0.333 --> Recreation 0.303 --> Arts

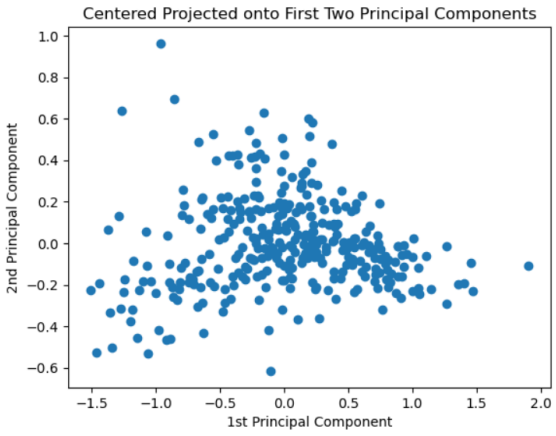
Healthcare has highest magnitude but with negative sign. This suggests that cities with better healthcare have lower scores along this component. Arts and recreation are also important in explaining the variance along this component, but with a positive influence. Cities with better recreation and arts ratings tend to score higher on this component, while those with better healthcare systems score lower.

Part 5:

```
projected_data = centered_data.dot(np.vstack((v1, v2)).T) # Similar to problem 1, I am constructing new data with reduced dimensions
                                                          # by matrix multiplication of centered data with pca matrix

print(projected_data.head())
```

```
          0         1
0 -0.436677  0.420163
1  0.620958  0.005346
2 -0.873256 -0.212104
3  0.502948 -0.063621
4  0.609775 -0.007233
```

```
# These values act as coordinate points of data points and used in scatter plot by calling their index.
# Plotting the projected data
plt.scatter(projected_data.iloc[:, 0], projected_data.iloc[:, 1])
plt.title('log transformed data Projected onto First Two Principal Components')
plt.xlabel('1st Principal Component')
plt.ylabel('2nd Principal Component')
plt.show()
```



```
# Setting up the distance to call a city as outlier is key here. I can observe very few points are lying far from other data points
# and these can be observed after a distance of 1.5 magnitude from origin. So, i will set the distance as 1.5
# Identify outliers in the scatter plot
outliers = data.iloc[projected_data[(np.abs(projected_data) > 1.5).any(axis=1)].index]
print("Outliers observed from data projection on 2 PCs :", outliers['City'])
```

```
Outliers: 212     New-York
298    Texarkana
Name: City, dtype: object
```

So I observe New-york and Texarkana are the 2 cities identified as outliers.

Part 6:

Most of the code is repeatable from part 2-5, so I will just reuse it and comment on the modifications below in the code.

```
# Given 9 columns:
column_names = ["City", "Climate", "Housing", "Healthcare", "Crime", "Transportation", "Education", "Arts", "Recreation", "Economic_Welfare"]
# reading the given text file into
data = pd.read_csv("D:/Projects/assignments/intro_ml/supp_HW4/supp/places.txt", delimiter=' ', names=column_names, usecols=range(10), header=None, skipin
data["City"] = data["City"].str.split(',').str[0]
# printing given data
print("Given data of places:\n", data.head())
X_matrix = data.drop('City', axis=1) # we are only considering the ratings(numerical values)
print("-"*70)
print(" The numerical data matrix will be: \n", X_matrix)
print("-"*70)
```

```
Given data of places:
                      City  Climate  Housing  Healthcare  Crime  \
0                  Abilene      521     6200         237    923
1                    Akron      575     8138        1656    886
2                   Albany      468     7339         618    970
3  Albany-Schenectady-Troy      476     7908        1431    610
4              Albuquerque      659     8393        1853   1483

   Transportation  Education  Arts  Recreation  Economic_Welfare
0            4031       2757   996        1405              7633
1            4883       2438  5564        2632              4350
2            2531       2560   237         859              5250
3            6883       3399  4655        1617              5864
4            6558       3026  4496        2612              5727
----------------------------------------------------------------------
 The numerical data matrix will be:
      Climate  Housing  Healthcare  Crime  Transportation  Education  Arts  \
0         521     6200         237    923            4031       2757   996
1         575     8138        1656    886            4883       2438  5564
2         468     7339         618    970            2531       2560   237
3         476     7908        1431    610            6883       3399  4655
4         659     8393        1853   1483            6558       3026  4496
..        ...      ...         ...    ...             ...        ...   ...
324       562     8715        1805    680            3643       3299  1784
325       535     6440         317   1106            3731       2491   996
326       540     8371         713    440            2267       2903  1022
327       570     7021        1097    938            3374       2920  2797
328       608     7875         212   1179            2768       2387   122

     Recreation  Economic_Welfare
0          1405              7633
1          2632              4350
2           859              5250
3          1617              5864
4          2612              5727
..          ...               ...
324         910              5040
325        2140              4986
326         842              4946
327        1327              3894
328         918              4694
```

part2 of part 6

```
scaler = StandardScaler()
X_normalized = scaler.fit_transform(X_matrix)
```

```
# Step 2: Center the data (mean vector μ and subtract from the data)
mean_X = np.mean(X_normalized, axis=0)
print(mean_X)
```

```
[ 2.26768958e-16  3.88746786e-16 -3.23955655e-17  1.29582262e-16
 -2.91560089e-16  2.64563785e-16  1.07985218e-17 -1.61977827e-17
 -3.72549003e-16]
```

We know normalized z-score will have mean of 0 which can be observed in the above results. All the values have 10^-17. So, we do not have to center the normalized z-score matrix anymore. Still I will calculate to prove my statement.

part3 of part 6

```
centered_data = X_normalized - mean_X  # Center the data
print("normalized X:\n", X_normalized)
print("-"*70)
print(" Centered data :\n", centered_data)
```

```
normalized X:
 [[-0.14700595 -0.90129655 -0.9473398  ... -0.46489336 -0.54664636
   1.94643332]
 [ 0.30066422 -0.08756979  0.46956805 ...  0.52060395  0.97444164
  -1.08546728]
 [-0.58638594 -0.42305363 -0.56690154 ... -0.62863952 -1.22351192
  -0.25430354]
 ...
 [ 0.01050762  0.01026216 -0.47204161 ... -0.45928414 -1.24458649
  -0.53505218]
 [ 0.25921328 -0.55657536 -0.08860777 ... -0.07634681 -0.64334144
  -1.50659023]
 [ 0.57424044 -0.19799814 -0.97230294 ... -0.65344954 -1.15037077
  -0.76777803]]
----------------------------------------------------------------------
 Centered data :
 [[-0.14700595 -0.90129655 -0.9473398  ... -0.46489336 -0.54664636
   1.94643332]
 [ 0.30066422 -0.08756979  0.46956805 ...  0.52060395  0.97444164
  -1.08546728]
 [-0.58638594 -0.42305363 -0.56690154 ... -0.62863952 -1.22351192
  -0.25430354]
 ...
 [ 0.01050762  0.01026216 -0.47204161 ... -0.45928414 -1.24458649
  -0.53505218]
 [ 0.25921328 -0.55657536 -0.08860777 ... -0.07634681 -0.64334144
  -1.50659023]
 [ 0.57424044 -0.19799814 -0.97230294 ... -0.65344954 -1.15037077
  -0.76777803]]
```

We can clearly see there is no difference in the values for normalized x and centered x.

```
# performing SVD on centered data:
U, Sigma, Vt = np.linalg.svd(centered_data, full_matrices=False) # i am setting the full_matrices = false so that matrices will be restricted from fillin
                                                                 # with zero values.

# The principal components are the rows of Vt
principal_components = Vt.T # In the 2nd problem we mentioned matrix v has the eigen vectors which are the principal components of the
                            # given data matrix, hence we define principal components as Vt.T[rows of Vt are eigen vectors, hence we transform it]

# Display the principal components
print("Principal components (Vt):\n", principal_components)
print("Shape of Principal components:\n", principal_components.shape)
```

```
Principal components (Vt):
 [[ 2.06413954e-01  2.17835308e-01 -6.89955982e-01  1.37321246e-01
   -3.69149929e-01  3.74604694e-01 -8.47057741e-02 -3.62308330e-01
    1.39135150e-03]
 [ 3.56521608e-01  2.50624000e-01 -2.08172230e-01  5.11828708e-01
    2.33487781e-01 -1.41639825e-01 -2.30638624e-01  6.13855131e-01
    1.36003402e-02]
 [ 4.60214647e-01 -2.99465282e-01 -7.32492550e-03  1.47018320e-02
   -1.03240518e-01 -3.73848037e-01  1.38676115e-02 -1.85676120e-01
   -7.16354893e-01]
 [ 2.81298380e-01  3.55342273e-01  1.85104981e-01 -5.39050473e-01
   -5.23939687e-01  8.09232850e-02  1.86064572e-02  4.30024765e-01
   -5.86084614e-02]
 [ 3.51150781e-01 -1.79604477e-01  1.46376283e-01 -3.02903705e-01
    4.04348475e-01  4.67591803e-01 -5.83390970e-01 -9.35986618e-02
    3.62945266e-03]
 [ 2.75292636e-01 -4.83382093e-01  2.29702548e-01  3.35411034e-01
   -2.08819059e-01  5.02169811e-01  4.26181860e-01  1.88667565e-01
    1.10840191e-01]
 [ 4.63054489e-01 -1.94789920e-01 -2.64842979e-02 -1.01080391e-01
   -1.05097637e-01 -4.61880719e-01 -2.15251529e-02 -2.03989691e-01
    6.85758213e-01]
 [ 3.27887907e-01  3.84474638e-01 -5.08526400e-02 -1.89800816e-01
    5.29540576e-01  8.99157817e-02  6.27877892e-01 -1.50595968e-01
   -2.55062915e-02]
 [ 1.35412251e-01  4.71283277e-01  6.07314475e-01  4.21769940e-01
   -1.59620056e-01  3.26081342e-02 -1.49740658e-01 -4.04809255e-01
    4.37794248e-04]]
Shape of Principal components:
 (9, 9)
```

part4 of part 6

```
v1 = principal_components[:, 0] #v1
v2 = principal_components[:, 1] # v2

print("First principal component (v1): \n", v1)
print("Second principal component (v2): \n", v2)
```

```
First principal component (v1):
 [0.20641395 0.35652161 0.46021465 0.28129838 0.35115078 0.27529264
 0.46305449 0.32788791 0.13541225]
Second principal component (v2):
 [ 0.21783531  0.250624   -0.29946528  0.35534227 -0.17960448 -0.48338209
 -0.19478992  0.38447464  0.47128328]
```

Column order: ["Climate", "Housing", "Healthcare", "Crime", "Transportation", "Education", "Arts", "Recreation", "Economic_Welfare"] for reference. PCA1: Maximum magnitude values in V1: 0.463 --> Art 0.460 --> Healthcare 0.356 --> Housing Art (0.463) has the largest positive weight, which means this component is most strongly influenced by the "Arts" factor. Cities that have higher scores in "Arts" tend to be aligned with this principal component. And Factors such as "Economic welfare" (0.135) and "Climate" (0.206) have very low contributions to PC1, indicating that they are less influential in defining this principal component

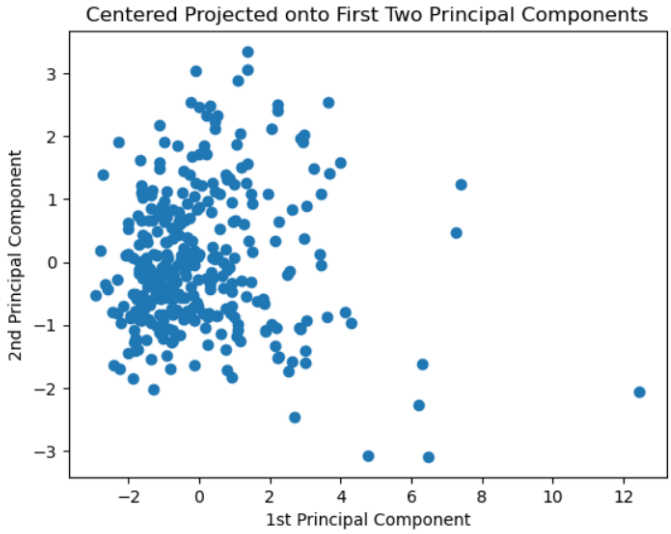
PCA2: Maximum magnitude values in V2: -0.481 --> Education 0.471 --> Economic welfare 0.384 --> Recreation

Education has highest magnitude but with negative sign. This suggests that cities with better education have lower scores along this component. economic welfare and recreation are also important in explaining the variance along this component, but with a positive influence. Cities with better recreation and economic welfare ratings tend to score higher on this component, while those with better education score lower.

```
# part 5 of part 6
projected_data = centered_data.dot(np.vstack((v1, v2)).T) # Similar to problem 1, I am constructing new data with reduced dimensions
                                                          # by matrix multiplication of centered data with pca matrix

print(projected_data[:5,:])
```

```
[[-1.04176435  0.89513038]
 [ 0.44048353  0.07518052]
 [-1.87839614  0.069898  ]
 [ 0.91212869 -1.82035075]
 [ 2.15252133  0.32935901]]
```

```
# These values act as coordinate points of data points and used in scatter plot by calling their index.
# Plotting the projected data
plt.scatter(projected_data[:, 0], projected_data[:, 1])
plt.title('Centered Projected onto First Two Principal Components')
plt.xlabel('1st Principal Component')
plt.ylabel('2nd Principal Component')
plt.show()
```




```
# Setting up the distance to call a city as outlier is key here. I can observe few points are lying far from other origin points
# and these can be observed after a distance of 4 magnitude from origin. So, i will set the distance as 4

indices = np.where((np.abs(projected_data) > 4).any(axis=1))[0]  # Get indices of outliers
outliers = data.iloc[indices]  # Extract outlier rows from the original data
print("Outliers observed from data projection on 2 PCs :\n", outliers['City'])
```

```
Outliers observed from data projection on 2 PCs :
 25         Baltimore
42            Boston
64           Chicago
178      Los-Angeles
212         New-York
213           Newark
233     Philadelphia
269    San-Francisco
313       Washington
Name: City, dtype: object
```

Conclusion: When I performed PCA using SVD on log transformed data and normalized z-score data, I have observed a difference between the principal components, projected data points and scatter plot. I can conclude that the choice of transformation plays a vital role in PCA, and should be chosen based on our requirement. In case of treating all the features equally, normalized z-score is preferable while if you want to reduce the number of outliers, log based is preferable.

Problem 4 (20 Points): Manifold Learning: Order the Faces:

Given face.mat dataset containing 33 faces of the same person. Y dataset have dimensions of 112x92x33.

Lets understand and visualize the given dataset.

Verifying the size of dataset with the dimensions mentioned in the question:

```
%Question 4:
% Loading the given face.mat file
load('face.mat');

% Given Y dataset is of size 112 x 92 x 33 (33 face images, each 112x92 pixels)
[rows, cols, faces] = size(Y);
disp('The dimensions of the given dataset is:');
disp(size(Y));
```

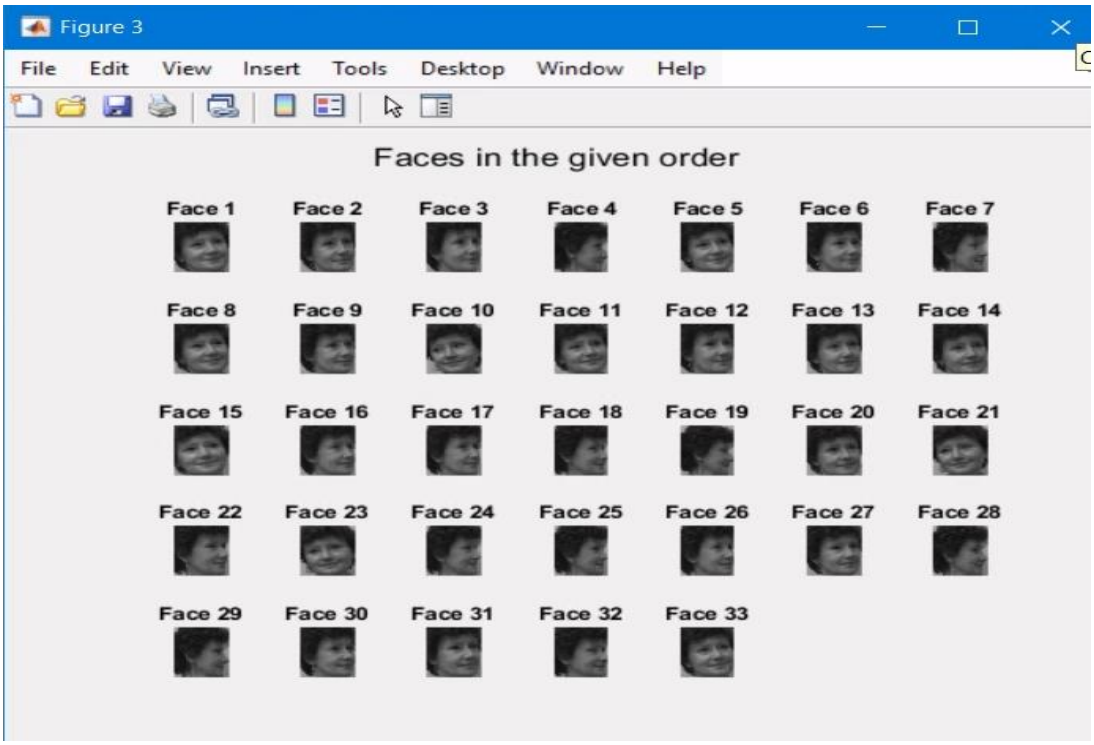
Output:

```
Warning: Function le has the same name as a MATLAB built-in.
>> akashhw4_q4
The dimensions of the given dataset is:
   112    92    33
```

Lets visualize the 33 faces in the given order:

```
% Visualize the faces in the given order
figure;
for i = 1:faces
    faceImage = Y(:,:,i); % selecting each face of ith index
    subplot(5, 7, i);  % displays 33 images in 5 rows, 7 columns
    imshow(faceImage);
    title(['Face ' num2str(i)]);
    sgtitle('Faces in the given order');
end
```

Output:



Creating a Data Matrix X with dimensions nxp (where n = 33, p = 112 × 92 = 10304)

```
% Creating a Data Matrix X with dimensions nxp (where n = 33, p = 112 × 92 = 10304)
X = reshape(Y, [10304, 33])';  % X is now 33x10304
disp(X(1:5, 1:10)); %displaying first 5 rows and 10 columns of data matrix
```

```
 42    43    42    37    38    34    29    29    30    29
 45    46    45    37    38    38    36    36    43    35
 38    36    35    33    28    28    27    34    33    30
164   163   164   164   164   159   161   156   151   140
 39    38    44    44    44    41    39    36    35    34
```

Okay we explored the data and understood how the values are in the given dataset. Now lets perform different embedding techniques and compare the results.

1. **MDS – embedding:**

   The structure of MDS – embedding is as follows:

   Distance matrix → Centering the Distance matrix → Eigen value decomposition → Dimension reduction.

```
% Part 1: MDS embedding

% Step 1: Lets measure the distance between each face
D = pdist2(X, X, 'euclidean').^2;  % distance matrix [33x33]

% Step 2: Center the distance matrix using double centering method
n = size(D, 1);  % No. of faces = 33
H = eye(n) - (1/n) * ones(n, n);  % Centering matrix
B = -0.5 * H * D * H;  % Centered matrix (Gram matrix)

% Step 3: Eigenvalue decomposition on the centered matrix
[V, Lambda] = eig(B);  % Eigenvalue decomposition of the centered matrix
eig_vals = diag(Lambda);  % eigen values are diagonal elements of lambda

% Sorting eigenvalues in desceding order as we want the highest values
% first because the higher the eigen value, the more the corresponding
% eigen vector is able to explain the variance.

[eig_vals_sorted, idx] = sort(eig_vals, 'descend');
V_sorted = V(:, idx);  % Sorting the eigenvectors based on eigen values
```

```
% Step 4: As mentioned in question we are using only the top two eigen
% vectors and performing dot product with eigen values for Y_mds
Y_mds = V_sorted(:, 1:2) .* sqrt(eig_vals_sorted(1:2)');  % Reduced datamatrix into 2D.

% Step 5: Sort the faces based on the first eigenvector and visualize the results
[v1_values, face_order] = sort(V_sorted(:,1), 'descend');  % Sort based on the first eigenvector
disp(face_order'); % displays the indices of sorted first
disp(v1_values');
```

Output:

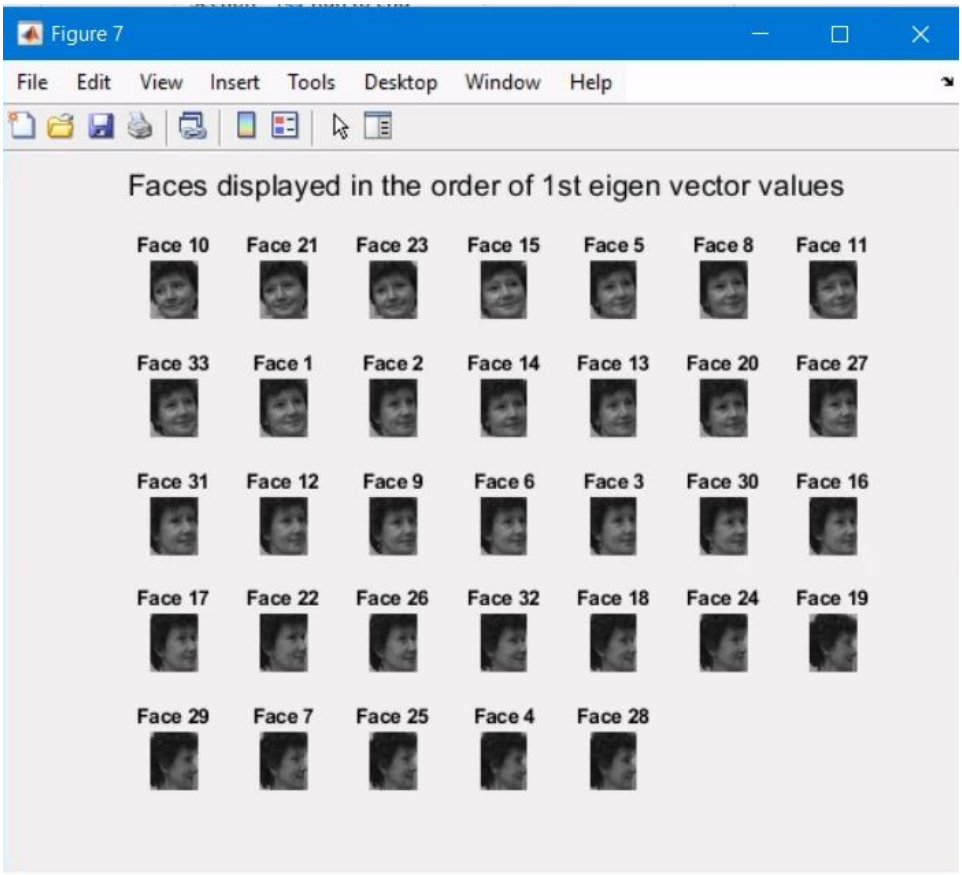
Order of values in 1st eigen vector:

```
Columns 1 through 25

  10   21   23   15    5    8   11   33    1    2   14   13   20   27   31   12    9    6    3   30   16   17   22   26   32

Columns 26 through 33

  18   24   19   29    7   25    4   28
```

Sorted 1st eigen vector in descending order:

```
Columns 1 through 15

  0.2460   0.2410   0.2191   0.1999   0.1783   0.1767   0.1727   0.1654   0.1651   0.1149   0.1122   0.1107   0.0893   0.0865   0.0861

Columns 16 through 30

  0.0781   0.0484   0.0326   0.0294  -0.0109  -0.0494  -0.0821  -0.0950  -0.1199  -0.1830  -0.2194  -0.2282  -0.2522  -0.2554  -0.2595

Columns 31 through 33

 -0.2623  -0.2625  -0.2728
```

```
figure;
for i = 1:33
    subplot(5, 7, i);
    imshow(reshape(X(face_order(i),:), [112, 92]));
    title(['Face ' num2str(face_order(i))]);
    sgtitle(' Faces displayed in the order of 1st eigen vector values');
end
```



Observation:
In the image of faces displayed in the order of 1st eigen vector [descending order], we can see the initial faces do not have any rotation and minimal distortion. This distortion has increased as we go down the order. This perfectly supports our statement of how the eigen vectors tend to explain the variance of the data and in 1st eigen vector, the faces which explain high variance are have high values in the vector.

2. **ISOMAP-embedding**

The structure of ISOMAP – embedding is as follows:
Distance matrix → Construct KNN graph(k=5) → Compute Geodesic distance → Classical MDS

```
% Step 1: Lets measure the distance between each face
D = pdist2(X, X, 'euclidean').^2;  % distance matrix [33x33]
disp(D);
```

```
Columns 1 through 15

         0    0.5092    0.7840    1.6546    0.1697    0.7487    1.4729    0.0815    0.7037    1.0233
    0.5092         0    0.4631    1.4310    0.4713    0.3586    1.3057    0.4645    0.2705    1.1234
    0.7840    0.4631         0    1.3199    0.8742    0.1207    1.1464    0.8378    0.2101    1.4399
    1.6546    1.4310    1.3199         0    1.6809    1.2288    0.2992    1.7055    1.2455    2.2216
    0.1697    0.4713    0.8742    1.6809         0    0.8254    1.5105    0.0923    0.7355    0.9155
    0.7487    0.3586    0.1207    1.2288    0.8254         0    1.0909    0.8072    0.0739    1.3568
    1.4729    1.3057    1.1464    0.2992    1.5105    1.0909         0    1.5130    1.1172    2.2395
    0.0815    0.4645    0.8378    1.7055    0.0923    0.8072    1.5130         0    0.7319    0.9614
    0.7037    0.2705    0.2101    1.2455    0.7355    0.0739    1.1172    0.7319         0    1.2742
    1.0233    1.1234    1.4399    2.2216    0.9155    1.3568    2.2395    0.9614    1.2742         0
```

```
% Step 2: Constructing k-NN graph with k=5
k = 5;
neighbors = knnsearch(X, X, 'K', k+1);  % first neighbor is the point itself, so we use k+1
                                        % as no. of neighbors for each
                                        % points
W = inf(size(D));  % Initializing graph distance matrix with infinity
                   % ensures that only the k nearest neighbors will have finite distances.
for i = 1:33
    W(i, neighbors(i,2:end)) = D(i, neighbors(i,2:end));
end
% A could have B as its nearest neighbor but that doesnt have to vice
% versa. So when calculating geodesic distance, we first modify the W.
W = min(W, W');  % I am taking the minimum between W and its transpose, since
% Taking the minimum ensures we get the closest distance in the case of asymmetric distances.
```

```
% Step 3: Compute geodesic distances using shortest paths
disp(W(1:10, 1:10)); % displaying some part of the knn graph
G = graph(W);
D_geo = distances(G); % Geodesic distances
disp(D_geo(1:10, 1:10)); % display first 10 rows and columns of geodesic matrix
```

Graph distance matrix:

```
   Inf       Inf       Inf       Inf    1.6968       Inf       Inf    0.8147       Inf       Inf
   Inf       Inf       Inf       Inf       Inf       Inf       Inf    4.6446    2.7046       Inf
   Inf       Inf       Inf       Inf       Inf    1.2069       Inf       Inf    2.1008       Inf
   Inf       Inf       Inf       Inf       Inf       Inf    2.9919       Inf       Inf       Inf
1.6968       Inf       Inf       Inf       Inf       Inf       Inf    0.9228       Inf    9.1550
   Inf       Inf    1.2069       Inf       Inf       Inf       Inf       Inf    0.7394       Inf
   Inf       Inf       Inf    2.9919       Inf       Inf       Inf       Inf       Inf       Inf
0.8147    4.6446       Inf       Inf    0.9228       Inf       Inf       Inf       Inf       Inf
   Inf    2.7046    2.1008       Inf       Inf    0.7394       Inf       Inf       Inf       Inf
   Inf       Inf       Inf       Inf    9.1550       Inf       Inf       Inf       Inf       Inf
```

Geodesic matrix:

```
         0    0.5459    0.9992    2.0932    0.1697    0.8785    1.8085    0.0815    0.8046    1.0467
    0.5459         0    0.4533    1.5473    0.5567    0.3326    1.2626    0.4645    0.2586    1.4443
    0.9992    0.4533         0    1.2557    0.9717    0.1207    0.9710    0.9177    0.1946    1.8872
    2.0932    1.5473    1.2557         0    2.0658    1.2288    0.2847    2.0118    1.2887    2.9813
    0.1697    0.5567    0.9717    2.0658         0    0.8510    1.7811    0.0923    0.7771    0.9155
    0.8785    0.3326    0.1207    1.2288    0.8510         0    0.9441    0.7970    0.0739    1.7665
    1.8085    1.2626    0.9710    0.2847    1.7811    0.9441         0    1.7271    1.0040    2.6966
    0.0815    0.4645    0.9177    2.0118    0.0923    0.7970    1.7271         0    0.7231    0.9798
    0.8046    0.2586    0.1946    1.2887    0.7771    0.0739    1.0040    0.7231         0    1.6926
    1.0467    1.4443    1.8872    2.9813    0.9155    1.7665    2.6966    0.9798    1.6926         0
```

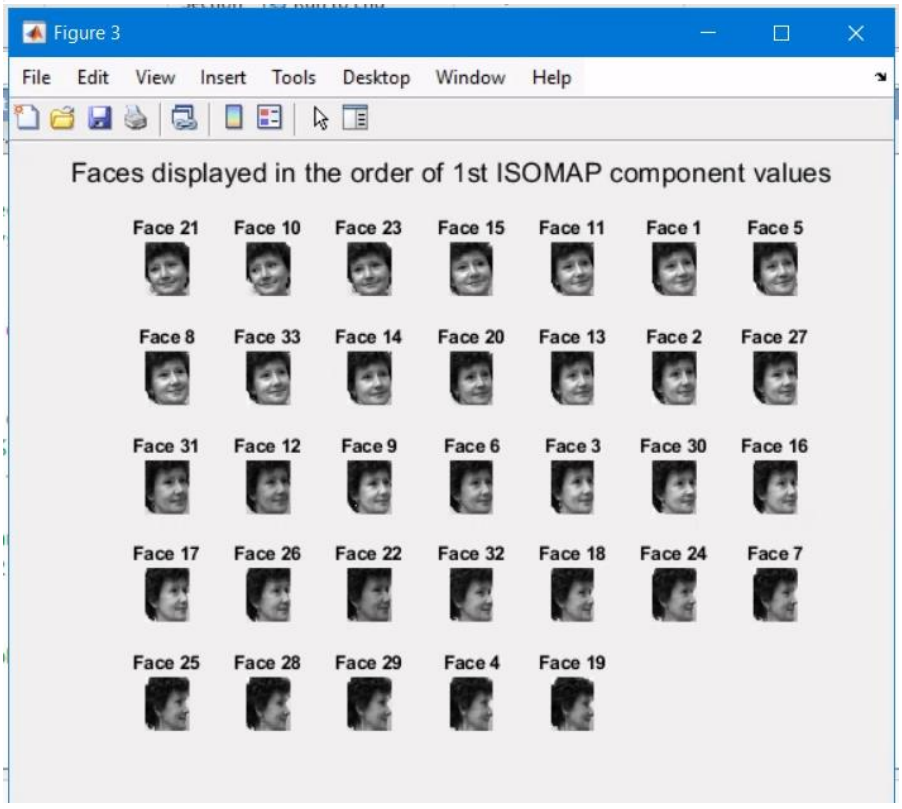
```
% Step 4: I am using builtin cmdscale function for performing classical MDS
[Y_isomap, ~] = cmdscale(D_geo);

% Step 5: Compare with MDS by sorting faces according to first ISOMAP component
[isomap_comp1, face_order] = sort(Y_isomap(:,1), 'descend');  % Sort based on the first ISOMAP component
disp(face_order)
disp(isomap_comp1)
figure;

for i = 1:33
    subplot(5, 7, i);
    imshow(reshape(X(face_order(i),:), [112, 92]), []);
    title(['Face ' num2str(face_order(i))]);
    sgtitle("Faces displayed in the order of 1st ISOMAP component values")
end
```

Face indexes in order of magnitude of first ISOMAP component:

```
Columns 1 through 25

   21    10    23    15    11     1     5     8    33    14    20    13     2    27    31    12     9     6     3    30    16    17    26    22    32

Columns 26 through 33

   18    24     7    25    28    29     4    19
```

First ISOMAP component in descending order:

```
Columns 1 through 15

   1.6675    1.6624    1.5690    1.4702    0.7994    0.7747    0.7713    0.7196    0.7011    0.5072    0.4313    0.3657    0.2523    0.1518    0.0996

Columns 16 through 30

   0.0649   -0.0052   -0.0736   -0.1341   -0.2040   -0.2853   -0.3557   -0.4553   -0.5537   -0.7330   -0.8304   -0.9429   -1.0166   -1.1171   -1.1756

Columns 31 through 33

  -1.2350   -1.3011   -1.5895
```



Observation:

There are few changes in Face indexes in order of magnitude of first ISOMAP component. This change in order of index could be because ISOMAP preserves geodesic distances on a manifold using a k-nearest neighbor graph. This technique captures nonlinear relationships between the faces, representing variations in angles or poses while MDS captures linear

relationships only. Even though there is slight change in order, when we consider the faces in overall, there are same set of faces in the corners of the manifold.

3. Locality Linear Embedding (LLE)-embedding

In this case the structure of LLE embedding is:

Distance matrix → KNN matrix → Reconstruction weight matrix → Embedding matrix → Eigen decomposition → Display the faces in the order of first lle components.

```
% Step 1: Measuring the distance between each face
D = pdist2(X, X, 'euclidean').^2;  % Distance matrix [33x33]
disp('Displaying first 10 rows and columns of the distance matrix:');
disp(D(1:10, 1:10)); % Displaying first 10 rows and columns of distance matrix
[N, d] = size(X);  % N = number of data points (33), d = dimensionality (10304)
[sorted_dist_values, neighbors] = sort(D, 2);  % Sorting distances to find nearest neighbors
k = 5;  % Given in the question

% Keeping only the k nearest neighbors for each point
neighbors_knn = neighbors(:, 2:k+1);  % Ignoring self-distance (first one)
disp('k nearest neighbors for each face:');
disp(neighbors_knn);
```

```
Displaying first 10 rows and columns of the distance matrix:
   1.0e+07 *

         0    0.5092    0.7840    1.6546    0.1697    0.7487    1.4729    0.0815    0.7037    1.0233
    0.5092         0    0.4631    1.4310    0.4713    0.3586    1.3057    0.4645    0.2705    1.1234
    0.7840    0.4631         0    1.3199    0.8742    0.1207    1.1464    0.8378    0.2101    1.4399
    1.6546    1.4310    1.3199         0    1.6809    1.2288    0.2992    1.7055    1.2455    2.2216
    0.1697    0.4713    0.8742    1.6809         0    0.8254    1.5105    0.0923    0.7355    0.9155
    0.7487    0.3586    0.1207    1.2288    0.8254         0    1.0909    0.8072    0.0739    1.3568
    1.4729    1.3057    1.1464    0.2992    1.5105    1.0909         0    1.5130    1.1172    2.2395
    0.0815    0.4645    0.8378    1.7055    0.0923    0.8072    1.5130         0    0.7319    0.9614
    0.7037    0.2705    0.2101    1.2455    0.7355    0.0739    1.1172    0.7319         0    1.2742
    1.0233    1.1234    1.4399    2.2216    0.9155    1.3568    2.2395    0.9614    1.2742         0

k nearest neighbors for each face:
     8    11    33     5    14
    27    13    12     9    31
     6    30     9    12    16
    29    28    25     7    19
    11    33     8     1    14
     9     3    30    12    27
    24    25    28    18    29
    11     1     5    33     2
     6    27    30     3    12
    21    23    11    15     5
     5     8    33     1    14
    31     6     9     3    27
    20     2    14    27    33
    20    13    33     5    11
     1    23    11    21     8
    17    30    26     6     3
    16    26    30    22     3
    32    24     7    25    28
     4    28    29    25     7
    14    13    27     2    33
    10    23    15    11     5
```

I have pasted the output of knn for first 22 faces.

```
% Step 2: Computing the reconstruction weights
W = zeros(N, N);  % Weight matrix
for i = 1:N
    % Centering the neighbors
    Z = X(neighbors_knn(i,:), :) - X(i, :);  % Z is the difference between neighbors and point i
    Z = double(Z);  % Ensuring that Z is a double-precision matrix
    C = Z * Z';  % Local covariance matrix

    % Solving for reconstruction weights
    W(i, neighbors_knn(i,:)) = C \ ones(k, 1);  % Solving Cw = 1 for weights
    W(i, neighbors_knn(i,:)) = W(i, neighbors_knn(i,:)) / sum(W(i, neighbors_knn(i,:)));  % Normalizing weights
end
disp('Reconstruction weight matrix is :\n');
disp(W(1:10, 1:10)); %choosing first 1 to 10 rows and columns for better display
%we can see, the weight matrix will only have weights for nearest neighbors
%to each data point and 0 otherwise.
```

```
Reconstruction weight matrix is :\n
         0         0         0         0   -0.3947         0         0    0.9701         0         0
         0         0         0         0         0         0         0         0   -0.0746         0
         0         0         0         0         0    0.6789         0         0   -0.1123         0
         0         0         0         0         0         0   -0.1231         0         0         0
   -0.0794         0         0         0         0         0         0   -0.0178         0         0
         0         0    0.1420         0         0         0         0         0    0.6432         0
         0         0         0         0         0         0         0         0         0         0
    0.7476   -0.0368         0         0   -0.0131         0         0         0         0         0
         0         0   -0.1187         0         0    0.9874         0         0         0         0
         0         0         0         0    0.0291         0         0         0         0         0
```

```
% Step 3: Creating the embedding matrix: (I - W)' * (I - W)
M = (eye(N) - W)' * (eye(N) - W);
[eigenvectors, eigenvalues] = eig(M);  % Eigen decomposition on M

% Sorting the eigenvalues in ascending order
[sorted_eigenvalues, idx] = sort(diag(eigenvalues));

disp('sorted eigen values are:\n');
disp(sorted_eigenvalues');

% since the first eigen vector is trivial(all zeros) we take the 2nd eigen
% vector as first component of lle
Y_lle = eigenvectors(:, 2);
disp('First lle component values are:\n');
disp(Y_lle');
```

```
sorted eigen values are:\n
  Columns 1 through 16

    0.0000    0.0000    0.0001    0.0010    0.0047    0.0059    0.0253    0.0511    0.1018    0.1151    0.1995    0.3953    0.4991    0.5540    0.7953    0.8621

  Columns 17 through 32

    0.9278    1.3133    1.4012    1.4591    1.6619    2.1349    2.2242    2.8963    3.4201    3.8104    3.8152    4.4819    4.5906    4.7049    5.2022    5.7542

  Column 33

    6.5664

First lle component values are:\n
  Columns 1 through 16

   -0.0615   -0.0616   -0.0607   -0.0630   -0.0607   -0.0606   -0.0623   -0.0612   -0.0607    0.5523   -0.0607   -0.0609   -0.0623   -0.0625    0.1887   -0.0609

  Columns 17 through 32

   -0.0610   -0.0615   -0.0631   -0.0627    0.5635   -0.0611    0.4814   -0.0619   -0.0626   -0.0611   -0.0613   -0.0629   -0.0629   -0.0608   -0.0608   -0.0613

  Column 33

   -0.0612
```

The first column of Y_lle (i.e., Y_lle(: , 1)) represents the first principal direction/component in the lower-dimensional space, which captures the most significant variation among the face images based on LLE's projection.

```
% Step 4: Sorting the faces based on the first LLE component and visualizing the results
[V_values, face_order] = sort(Y_lle, 'descend');  % Sorting based on the first LLE component
disp('face order with respect to 1st lle component:\n');
disp(face_order')
figure;
for i = 1:33
    subplot(5, 7, i);
    imshow(reshape(X(face_order(i),:), [112, 92]), []);  % Reshaping and displaying face images
    title(['Face ' num2str(face_order(i))]);
    sgtitle('Faces displayed in the order of 1st LLE component values')
end
```

```
face order with respect to 1st lle component:\n
  Columns 1 through 28

   21   10   23   15    6   11    3    9    5   31   30   12   16   17   26   22   33    8   32   27    1   18    2   24   13    7   14   25

  Columns 29 through 33

   20   28   29    4   19
```

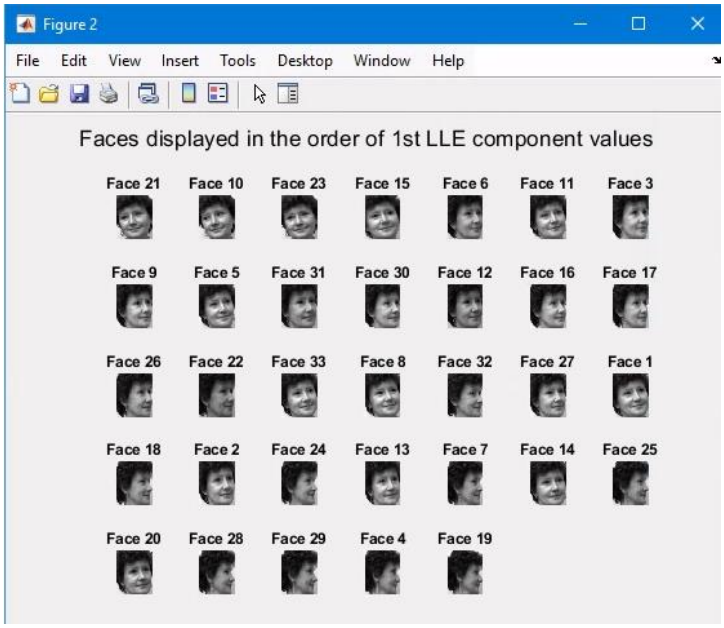
Face indexes in order of magnitude of first ISOMAP component:

```
  Columns 1 through 25

   21   10   23   15   11    1    5    8   33   14   20   13    2   27   31   12    9    6    3   30   16   17   26   22   32

  Columns 26 through 33

   18   24    7   25   28   29    4   19
```

There is a difference in the face order between ISOMAP embeddings and LLE embeddings

In LLE embeddings the points that were close to each other in the high-dimensional space will also be close in the low-dimensional space. This embedding captures the local geometric structure of the data while reducing the dimensionality.

In contrast to ISOMAP, which preserves global distances, LLE preserves local relationships between points.



4. **Laplacian Eigenmap (LE)-embedding**
   Structure of LE embeddings:

   Distance matrix → KNN matrix → Adjacency matrix(A) → Degree matrix(D) → Laplacian matrix(L=D-A) → Eigen decomposition → Display the faces in the order of first le components

```
% Step 1: Measuring the distance between each face
D = pdist2(X, X, 'euclidean').^2;  % Distance matrix [33x33]
disp('Displaying first 10 rows and columns of the distance matrix:');
disp(D(1:10, 1:10)); % Displaying first 10 rows and columns of distance matrix
[N, d] = size(X);  % N = number of data points (33), d = dimensionality (10304)
[sorted_D, neighbors] = sort(D, 2);  % Sorting distances to find nearest neighbors
k = 5;  % Given in the question

% Step 2: Keeping only the k nearest neighbors for each point
neighbors_knn = neighbors(:, 2:k+1);  % Ignoring self-distance (first one)
disp('k nearest neighbors for each face:');
disp(neighbors_knn(1:10,:));%displaying Knn of first 10 faces
```

```
Displaying first 10 rows and columns of the distance matrix:
   1.0e+07 *

         0    0.5092    0.7840    1.6546    0.1697    0.7487    1.4729    0.0815    0.7037    1.0233
    0.5092         0    0.4631    1.4310    0.4713    0.3586    1.3057    0.4645    0.2705    1.1234
    0.7840    0.4631         0    1.3199    0.8742    0.1207    1.1464    0.8378    0.2101    1.4399
    1.6546    1.4310    1.3199         0    1.6809    1.2288    0.2992    1.7055    1.2455    2.2216
    0.1697    0.4713    0.8742    1.6809         0    0.8254    1.5105    0.0923    0.7355    0.9155
    0.7487    0.3586    0.1207    1.2288    0.8254         0    1.0909    0.8072    0.0739    1.3568
    1.4729    1.3057    1.1464    0.2992    1.5105    1.0909         0    1.5130    1.1172    2.2395
    0.0815    0.4645    0.8378    1.7055    0.0923    0.8072    1.5130         0    0.7319    0.9614
    0.7037    0.2705    0.2101    1.2455    0.7355    0.0739    1.1172    0.7319         0    1.2742
    1.0233    1.1234    1.4399    2.2216    0.9155    1.3568    2.2395    0.9614    1.2742         0

k nearest neighbors for each face:
     8    11    33     5    14
    27    13    12     9    31
     6    30     9    12    16
    29    28    25     7    19
    11    33     8     1    14
     9     3    30    12    27
    24    25    28    18    29
    11     1     5    33     2
     6    27    30     3    12
    21    23    11    15     5
```

```
% Step 3: Adjacency matrix W (Gaussian similarity measure)
sigma = mean(sorted_D(:, 2));  % Use mean distance of 2nd nearest neighbor as sigma
W = zeros(size(D));
for i = 1:faces
    for j = 1:k
        neighbor = neighbors_knn(i, j);
        W(i, neighbor) = exp(-D(i, neighbor) / (2 * sigma^2));  % Gaussian kernel
        W(neighbor, i) = W(i, neighbor);  % Symmetrize
    end
end
```

```
% Step 4: Degree matrix (D)
D_m = diag(sum(W, 2));

% Step 5:  Laplacian L = D - W
L = D_m - W;

% Step 6: Solve the generalized eigenvalue problem L * v = lambda * D * v
[eigenvectors, eigenvalues] = eig(L, D_m);

% Sort eigenvalues in ascending order and get the corresponding indices
[sorted_eigenvalues, idx] = sort(diag(eigenvalues));
disp('the sorted eigenvalues are:\n');
disp(sorted_eigenvalues');
```

```
the sorted eigenvalues are:\n
  Columns 1 through 16

   0.0000    0.0177    0.0688    0.2537    0.4414    0.6385    0.7245    0.8345    0.9432    0.9674    1.0053    1.0525    1.0640    1.0847    1.1375    1.1429

  Columns 17 through 32

   1.1564    1.1667    1.1713    1.2000    1.2000    1.2000    1.2000    1.2000    1.2349    1.2646    1.2883    1.3072    1.3160    1.3497    1.4187    1.4495

  Column 33

   1.5001
```

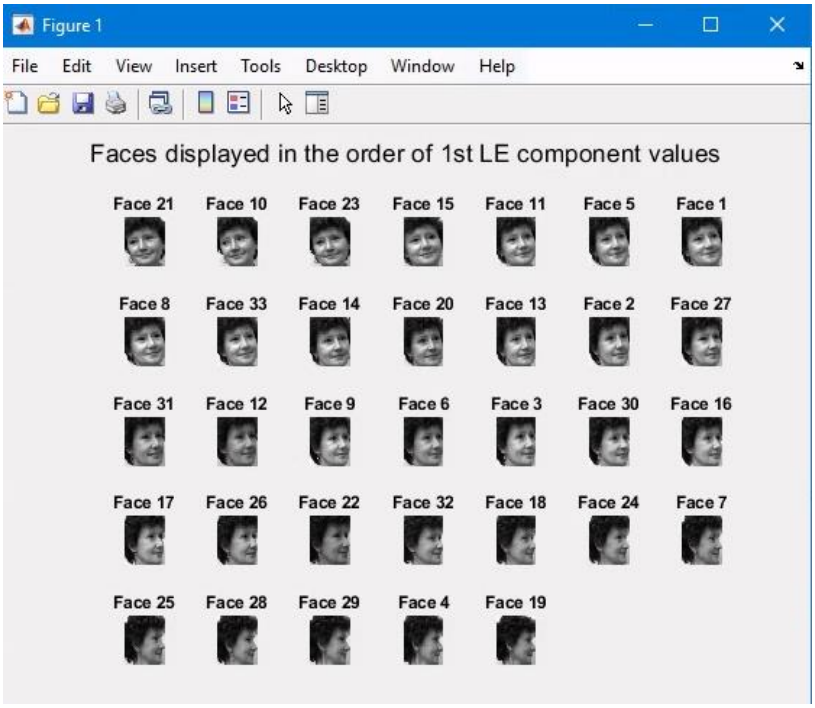
```
%we can the see the first eigen value is 0, so we choose the next smallest
%corresponding eigen vector
Y_le = eigenvectors(:, 2);

% Step 7: Sort the faces based on the first LE component and visualize
[V_values, face_order] = sort(Y_le, 'descend');  % Sorting based on the first LE component
disp('face order with respect to 1st le component:\n');
disp(face_order')
for i = 1:33
    subplot(5, 7, i);
    imshow(reshape(X(face_order(i), :), [rows, cols]), []);
    title(['Face ' num2str(face_order(i))]);
    sgtitle('Faces displayed in the order of 1st LE component values')
end
```

```
face order with respect to 1st le component:\n
 Columns 1 through 28

   21   10   23   15   11    5    1    8   33   14   20   13    2   27   31   12    9    6    3   30   16   17   26   22   32   18   24    7

 Columns 29 through 33

   25   28   29    4   19
```

Face order in with respect to 1st lle component:

```
face order with respect to 1st lle component:\n
 Columns 1 through 28

   21   10   23   15    6   11    3    9    5   31   30   12   16   17   26   22   33    8   32   27    1   18    2   24   13    7   14   25

 Columns 29 through 33

   20   28   29    4   19
```

We can observe slight difference in the face order between LLE and LE embeddings. LE and LLE both attempt to uncover the underlying low-dimensional manifold but focus on different aspects. LE uses a graph-based approach, while LLE uses local linearity to maintain the data's structure.



LE and LLE both attempt to uncover the underlying low-dimensional manifold but focus on different aspects. LE uses a graph-based approach, while LLE uses local linearity to maintain the data's structure.

Problem 5 (25 Points): Random Projections:

```
Part 1
```

```
# Given parameter values:

n = 10  # Number of points
d = 5000  # Original dimensionality
# Constructing a random data matrix X from given n, d dimensions
X = np.random.randn(n, d)

print(' The generated X matrix is:', X[:,1:10]) # printing first 10 columns of all 10 points
epsilon = 0.1  # Allowed distortion to find suitable m

# embedding dimension based on Johnson-Lindenstrauss Lemma
m = int(4 * np.log(n) / (epsilon ** 2))


# Output embedding dimension
print("number of dimensions in lower dimensional space (m): ", m)
```

```
 The generated X matrix is: [[-4.06394836e-01  2.91151790e-01 -1.61749500e+00 -1.87915355e-01
  -1.01672058e+00 -3.21258228e-01 -3.45867181e-01  1.49561243e+00
   8.06422178e-01]
 [-3.22712146e-01 -1.12927215e+00  1.61279821e+00 -1.33027216e-01
  -1.11960246e+00 -6.21870613e-02  2.37810003e-01  9.93234563e-01
   5.96067583e-01]
 [-1.05288387e+00  2.77856257e-01  3.24124677e-01  2.43931445e-01
  -1.59334666e+00 -1.19555245e-01  1.77796891e-01  2.03352098e-03
   5.03147952e-01]
 [ 5.72519322e-01  6.92560771e-02  1.32471230e-01  2.05778365e+00
  -1.59062255e+00  1.36342392e+00 -3.39153094e-02 -8.60755091e-01
  -6.46004649e-01]
 [ 3.43947682e-01  2.87357888e-01  6.64702886e-01 -1.15106939e+00
   1.75421745e+00  2.53305646e-01 -6.20175866e-01 -3.21055203e-01
   1.52442766e-01]
 [ 1.20397882e+00  6.75659383e-02  7.22100871e-01 -2.15831643e+00
   1.87044603e+00  2.82795654e-01  4.98683626e-01  1.07263153e+00
  -2.30666297e+00]
 [-1.24615080e+00 -4.61697216e-01  1.66342779e+00  1.26944003e+00
  -1.24332307e-01  1.11826608e+00  1.26748438e+00 -9.22848816e-01
   9.23064449e-01]
```

So we can observe that:

number of dimensions in higher space = 5000

number of dimensions in lower space calculated using embedding formula = 921.

Part 2:

```
# Given condition for A matrix:
# A can be constructed by choosing m × d i.i.d. entries from a
# zero mean Gaussian with variance 1/m.
A = np.random.normal(0, np.sqrt(1/m), size=(m, d)) # 0 mean, 1/m variance, mxd dimensions

# Projecting the data points in X with A to lower dimension space.
X_projected = np.dot(X, A.T)

# to find the pair wise distance for X and XT, lets define a function to
# calculate pairwise distance and call it later.
def pairwise_distances(matrix):
    n = matrix.shape[0]
    dist_squared = np.zeros((n, n))  # Initialize distance matrix with 0s
    #since distance matrix is symmertric, We compute only the upper
    #triangular part (symmetry)
    for i in range(n):
        for j in range(i, n):
            # Computing the squared distance between point u, v
            dist_squared[i, j] = np.sum((matrix[i] - matrix[j]) ** 2)
            dist_squared[j, i] = dist_squared[i, j]  # Symmetric property

    return dist_squared
# Scalling the function to calculate the pairwise distances of X, X*At
original_distances = pairwise_distances(X)
projected_distances = pairwise_distances(X_projected)

# defining lower_bound, upper_bound for every pairwise distance in original distance
lower_bound = (1 - epsilon) * original_distances
upper_bound = (1 + epsilon) * original_distances

# now we will verify if the projected distances are in this [lower_bound, Upper_bound] range
#In the question, its mentioned "Does the Lemma hold (i.e.,
#for every pair of data points,", so even if one datapoint does not satisfy this, it
# means lemma did not hold.
jl_holds = np.all((projected_distances >= lower_bound) & (projected_distances <= upper_bound))

if jl_holds:
    print("lemma holds for every pair of data points")
else:
    print("lemma did not hold for all pair of data points")
```

```
lemma did not hold for all pair of data points
```

Part 3:

```
# Parameters
n = 10  # Number of points
d_1 = 5000  # Initial dimensionality
epsilon = 0.1  # Allowed distortion
m = int(4 * np.log(n) / (epsilon ** 2))  # Embedding dimension based on Johnson-Lindenstrauss Lemma

#since i need to verify if Johnson-Lindenstrauss Lemma holds or not for every instance of
#d, i will define a function for that and call it
def verify_jllemma(X, A, epsilon):
    X_projected = np.dot(X, A.T)  # Project data to lower dimensions
    original_distances = pairwise_distances(X)  # Original pairwise distances
    projected_distances = pairwise_distances(X_projected)  # Projected pairwise distances
    # Lower and upper bounds for the projected distances
    lower_bound = (1 - epsilon) * original_distances
    upper_bound = (1 + epsilon) * original_distances
    # Check if all projected distances fall within the bounds
    jl_holds = np.all((projected_distances >= lower_bound) & (projected_distances <= upper_bound))
    return jl_holds

# increasing dimensionality d by a factor of 2 until memory is exhausted
d = d_1
while True:
    try:
        # Constructing a random data matrix X of size (n, d)
        X = np.random.randn(n, d)
        print("Shape of X:", np.shape(X)) # dimensions of X
        m = int(4 * np.log(n) / (epsilon ** 2))
        # Constructing a random projection matrix A of size (m, d) with variance 1/m
        A = np.random.normal(0, np.sqrt(1 / m), size=(m, d))

        # Verifying if the Johnson-Lindenstrauss Lemma holds
        if verify_jl_lemma(X, A, epsilon):
            print(f"Johnson-Lindenstrauss Lemma holds for d = {d}")
        else:
            print(f"Johnson-Lindenstrauss Lemma does not hold for d = {d}")

        # Doubling the dimensionality for the next iteration
        d *= 2

    except MemoryError:
        print("Memory limit reached, stopping execution.")
        break
```

```
Shape of X: (10, 5000)
Johnson-Lindenstrauss Lemma does not hold for d = 5000
Shape of X: (10, 10000)
Johnson-Lindenstrauss Lemma does not hold for d = 10000
Shape of X: (10, 20000)
Johnson-Lindenstrauss Lemma holds for d = 20000
Shape of X: (10, 40000)
Johnson-Lindenstrauss Lemma does not hold for d = 40000
Shape of X: (10, 80000)
Johnson-Lindenstrauss Lemma holds for d = 80000
Shape of X: (10, 160000)
Johnson-Lindenstrauss Lemma does not hold for d = 160000
Shape of X: (10, 320000)
Johnson-Lindenstrauss Lemma does not hold for d = 320000
Shape of X: (10, 640000)
Johnson-Lindenstrauss Lemma does not hold for d = 640000
Shape of X: (10, 1280000)
Johnson-Lindenstrauss Lemma does not hold for d = 1280000
Shape of X: (10, 2560000)
Johnson-Lindenstrauss Lemma does not hold for d = 2560000
Shape of X: (10, 5120000)
Memory limit reached, stopping execution.
```

Part 4

```python
# I have defined the functions of finding pairwise distance and veifying jl_lemma
# in the previous questions, So i am going to just call them increasing dimensionality n by a factor of 2 until memory is exhausted
d = 5000# resetting the value of d as 5000, keeping it constant
n=10 # initial n = 10
while True:
    try:
        # Constructing a random data matrix X of size (n, d)
        X = np.random.randn(n, d)
        print("Shape of X:", np.shape(X)) # dimensions of X
        m = int(4 * np.log(n) / (epsilon ** 2))
        # Constructing a random projection matrix A of size (m, d) with variance 1/m
        A = np.random.normal(0, np.sqrt(1 / m), size=(m, d))

        # Verifying if the Johnson-Lindenstrauss Lemma holds
        if verify_jl_lemma(X, A, epsilon):
            print(f"Johnson-Lindenstrauss Lemma holds for n = {n}")
        else:
            print(f"Johnson-Lindenstrauss Lemma does not hold for n = {n}")

        # Increasing n with factor 2 keeping d fixed for the next iteration
        n *= 2

    except MemoryError:
        print("Memory limit reached, stopping execution.")
        break
```

```
Shape of X: (10, 5000)
Johnson-Lindenstrauss Lemma does not hold for n = 10
Shape of X: (20, 5000)
Johnson-Lindenstrauss Lemma does not hold for n = 20
Shape of X: (40, 5000)
Johnson-Lindenstrauss Lemma does not hold for n = 40
Shape of X: (80, 5000)
Johnson-Lindenstrauss Lemma does not hold for n = 80
Shape of X: (160, 5000)
Johnson-Lindenstrauss Lemma does not hold for n = 160
Shape of X: (320, 5000)
Johnson-Lindenstrauss Lemma does not hold for n = 320
Shape of X: (640, 5000)
Johnson-Lindenstrauss Lemma does not hold for n = 640
Shape of X: (1280, 5000)
Johnson-Lindenstrauss Lemma does not hold for n = 1280
Shape of X: (2560, 5000)
Johnson-Lindenstrauss Lemma does not hold for n = 2560
Shape of X: (5120, 5000)
Johnson-Lindenstrauss Lemma does not hold for n = 5120
Shape of X: (10240, 5000)
Johnson-Lindenstrauss Lemma does not hold for n = 10240
Shape of X: (20480, 5000)
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
```

The main purpose of the 5.3 and 5.4 is to verify if Johnson-lindestrauss lemma holds or not.

We know that Johnson-Lindenstrauss Lemma says:
The number of dimensions m neeeded to approximately preserve distance between points in projected data matrix(in low dimensional space) with respect to pairwise distance in original matrix depends only on n & epsilon(e). Which can be observed from the formula of m:

$m = 4 \log n/\epsilon 2.$

Coming to our results in 5.3:
We can see JL_lemma is not holding even when we are increasing d in most cases.
But, surprisingly for d = 80000, & 20000 the lemma is holding.
And in 5.4:
I have run the code for almost 3hours, but it is still not reaching the next iteration. Hence, I have interrupted it but did not observe any case of Lemma holding till n = 10240.
But, based on the embedding formula, it is highly likely that lemma will hold for some value of n before reaching my memory limit.