Graph Laplacian and its Eigenvalues [5 pts]

Definition

Given a graph $G = (V, E)$ of $|V| = n$ nodes with adjacency matrix A, and let $\deg(v)$ be the degree of a vertex v. The degree matrix D for G is a $n \times n$ diagonal matrix where $D_{i,j} = \deg(v_i)$ if $i = j$, and 0 otherwise. The Laplacian matrix of G is defined as $L = D - A$, commonly used in spectral graph theory. We will explore one of its interesting properties in this section.

Networkx

Given a networkx graph object, the library has functions to get the Laplacian matrix or the eigenvalues of the matrix. See the example below.

In [1]:
```
import networkx as nx
import numpy as np

G_example = nx.Graph()
# Create a triangle graph
G_example.add_edges_from([[1,2],[2,3],[1,3]])

# laplacian_matrix(G) returns the laplacian of G in sparse matrix forma
nx.laplacian_matrix(G_example).todense()
```

Out[1]:
```
matrix([[ 2, -1, -1],
        [-1,  2, -1],
        [-1, -1,  2]], dtype=int64)
```

In [2]:
```
# laplacian_spectrum(G) returns the eigenvalues of the laplacian.
nx.laplacian_spectrum(G_example)
```

Out[2]:
```
array([-1.11022302e-16,  3.00000000e+00,  3.00000000e+00])
```

In the next cell, do the following:

1. Initialize a graph that contains one triangle.
2. Then add a disjoint triangle into the graph and report the eigenvalues of the new graph laplacian matrix. As you may noticed, the output array from the previous cell may contains small float value very close to 0, but not exactly 0 (which one of the eigenvalues should be). This is because Python has issue with representing floating-point numbers, see here (https://docs.python.org/3/tutorial/floatingpoint.html). Therefore, Round all eigenvalues to 2 decimals using np.around function.
3. Repeat step 2 for 4 times.

Report your observation and hypothesis.

In [3]:
```
# Initialize a graph with one triangle
G = nx.Graph()
G.add_edges_from([(1, 2), (2, 3), (3, 1)])
eigenvalues = nx.laplacian_spectrum(G)
print(np.round(eigenvalues,2))

# add a disjoint triangle into the graph
# report the eigenvalues of the new graph laplacian matrix
G.add_edges_from([[4,5],[5,6],[6,4]])
nx.laplacian_matrix(G).todense()
eigenvalues = nx.laplacian_spectrum(G)
print(np.round(eigenvalues,2))
G.add_edges_from([[7,8],[8,9],[9,7]])
nx.laplacian_matrix(G).todense()
eigenvalues = nx.laplacian_spectrum(G)
print(np.round(eigenvalues,2))
G.add_edges_from([[10,11],[11,12],[12,10]])
nx.laplacian_matrix(G).todense()
eigenvalues = nx.laplacian_spectrum(G)
print(np.round(eigenvalues,2))
G.add_edges_from([[13,14],[14,15],[15,13]])
nx.laplacian_matrix(G).todense()
eigenvalues = nx.laplacian_spectrum(G)
print(np.round(eigenvalues,2))
```

```
[-0.  3.  3.]
[-0. -0.  3.  3.  3.  3.]
[-0. -0. -0.  3.  3.  3.  3.  3.  3.]
[-0. -0. -0. -0.  3.  3.  3.  3.  3.  3.  3.  3.]
[-0. -0. -0. -0. -0.  3.  3.  3.  3.  3.  3.  3.  3.  3.  3.]
```

Your answer:

By observation, the eigenvalues of the Laplacian matrix are always the same set of eigenvalues as 0 and 3. Also, the number of eigenvalues increases by 3 each time I add a triangle, which is the number of node added to the graph. The number of zero eigenvalue increase by one each time I add a distinct triangle, which lead to my hypothesis that the number of zero eigenvalue is the number of distinct triangle in the graph, and the nonzero eigenvalue represents the number of edges in the connected subgraph.

PA8 - Eigenfaces [25 pts]

Problem Statement

In this exercise you will be using SVD to create a dictionary of eigenfaces from a training set that will be used to reconstruct faces from a testing set.

This assignment is broken down into the following three categories and each of their sub-categories:

Please make sure that all images are displayed in grayscale, an example of how to do this has been given in the import cell

1. Data visualization
    A. Display single image that contains a face from all 36 training people in the training set
    B. Display a single image that contains all faces from the 10th person in the training set
    C. Display and return the average training face
2. Compute SVD of training data
    A. Mean center your training data (subtract the average face from all training faces)
    B. Take SVD of mean-centered training data
3. Reconstruction experiments
    A. Reconstruct a face from the training set using the first p rows of your SVD matrix, U
    B. Experiment on different values of p

Data set description:

The data set you will be using contains images of 38 different people's faces. Each person has 64 or less images taken of their face. Each image is taken under unique lightning conditions. The "nfaces" variable loaded in and described below details how many images are associated with each participant. There are a total of 2410 images in this data set.

All images of the same participant are grouped to be in order adjacent columns in the data matrix. For instance, the first participant has 64 associated images (given in the variable n_faces), and their images are found in the first 64 columns (index 0 to 63) of the matrix. The second participant has 62 associated images, and their images are found in the next 62 columns of the matrix (index 64 to 125) and so forth.

Your training set will comprise of all images related to the first 36 people, and your testing set will be all images of the last 2 people

The data is stored in a matlab data file (.mat). You can think of the .mat file as a large dictionary where each key in the .mat dictionary points to some relevant information about the data. I have provided code that loads the .mat file (scipy.io.loadmat function) and have stored the following information you will need to complete this assignment:

1. m_prime = int - number of pixel rows per image
2. n_prime = int - number of pixel columns per image
3. nfaces = List - each index, i, represents the number of photos provided for participant i
4. faces = 2D numpy array ((m*n) x 2410) in shape. each column is the "flattened" image of a participants face (all 38 people)
5. trainingFaces = 2D numpy array which represents all images of the first 36 participants
6. testingFaces = 2D numpy array which represents all images of the last 2 participants

Each column in the matrices faces, trainingFaces, and testingFaces, is a "flattened" image of one of the participants. See below for the description of "falttened", and how you can reshape the image if needed.

Flattening and reshaping an image

When dealing with images, it is common practice to "flatten" each image from a 2D array of (m' x n') dimensions, to a 1D array of dimensions $m' * n' \times 1$ column vector.

To flatten a 2D image to a 1D vector, simply call the function: "flattened = nd_array.flatten()"

To reshape a flatten image to its original shape ($m' \times n'$), call the following function: "original_shape_image = np.reshape(flattened, (m', n'))" where "flattened" is the 1D flattened image

Single Value Decomposition (SVD)

Please review the following slides regarding SVD linked here (https://drive.google.com/file/d/1mVzWOKpFq1qKKNNKsAzbUtSrp6QrQESz/view). If you are looking to gather a more intuitive sense of SVD, take a look at the "Intuitive interpretations" section of the SVD wikipedia (https://en.wikipedia.org/wiki/Singular_value_decomposition#Intuitive_interpretations), and in particular the animated gif of SVD on the wikipedia page.

Recall that SVD factors can factor a real valued matrix, A, into the form $A = U\Lambda V^T$ where

1. A is a real matrix of dimensions $n \times m$
2. U is a real matrix of dimensions $n \times r$
3. Λ is a real matrix of dimensions $r \times r$
4. V is a real matrix of dimensions $m \times r$

Key Notes for Using SVD in this assignemnt:

1. n is the number of pixels per image, and m is the number of images
2. you can use the built in np function "np.linalg.svd"; its documentation can be found here (https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html).
3. Before taking the SVD, mean center your training data: subtract the average face from each image of your training data.
4. When calling np.linalg.svd, set the parameter "full_matrices" to 0
5. We will only be interested in one of the resulting matrices, U, of the decomposition.

Eigenfaces (U)

Notice that if my original matrix, A, is a $n \times m$ matrix, then one of my decomposition matrices from SVD, U, has a dimension $n \times r$. Recall that we can reshape a n dimensional column of my original matrix into an image of a face. As it tursn out, we can also reshape a n dimensional column of U into an image of a face as well.

To be more precise, we call each column of U an **eigenface**. Our collection of eigenfaces or a subset of our collection, organized as a matrix, can be used to reconstruct images of faces as a linear combination of our set set of eigenfaces. Of particular interest are images that did not contribute to the SVD (images from our testing set).

Using Eigenfaces to reconstruct new faces

Let us define U_p to be an $n \times p$ matrix that is the first p columns of U from the SVD of our mean-centered training set, and let us define x to be an n dimensional vector that is the mean centered flattened image of an image from our testing set.

Use the average face of the training data to also mean-center your testing image.

Consider the following matrix multiplication

$$\alpha = (U_p^T)x$$

Our resulting vector, α will be of dimensions $p \times 1$. Each index of α, i, holds a value that represents the amount of eigenface i that is needed to reconstruct x. In particular

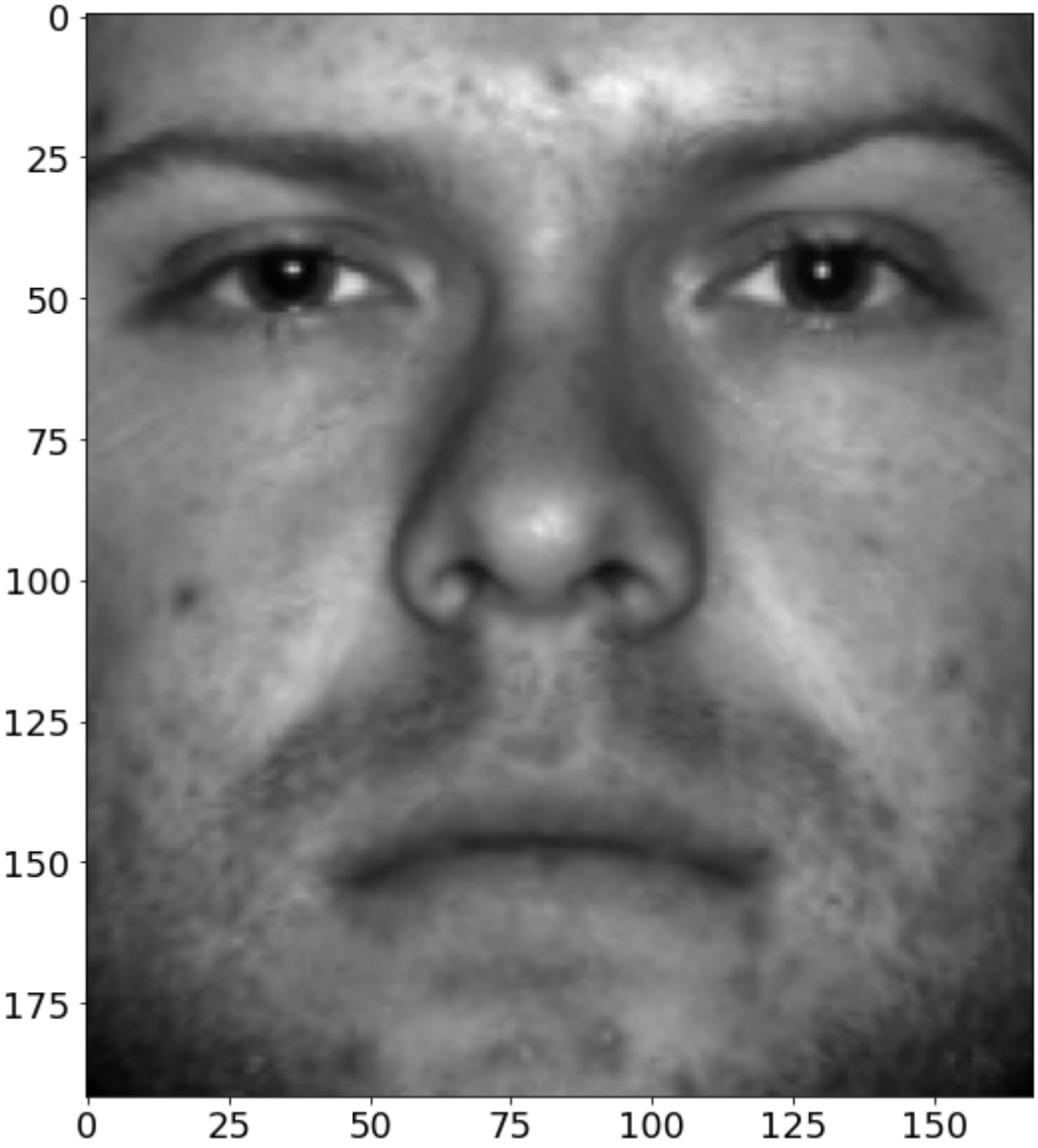
$$\hat{x} = \sum_{i=1}^{p} \alpha_i * (U_p)_i$$

In [4]:
```
#import cell and data loading
import matplotlib.pyplot as plt
import numpy as np
import os
import scipy.io
plt.rcParams['figure.figsize'] = [10, 10]
plt.rcParams.update({'font.size': 18})

#set as global variables that can balled in any function
mat_contents = scipy.io.loadmat(os.path.join('.','DATA','allFaces.mat')

#common variables
faces = mat_contents['faces'] #data, where each column is a flattened i
m_prime = int(mat_contents['m']) #number of pixel rows in each image
n_prime = int(mat_contents['n']) #number of pixel cols in each image
nfaces = np.ndarray.flatten(mat_contents['nfaces']) #list of how many i
trainingFaces = faces[:,:np.sum(nfaces[:36])]
testingFaces = faces[:,:np.sum(nfaces[36:])]

#example of how to display an image in gray scale
img = plt.imshow(np.reshape(testingFaces[:,0],(m_prime,n_prime)).T)
img.set_cmap('gray')
```

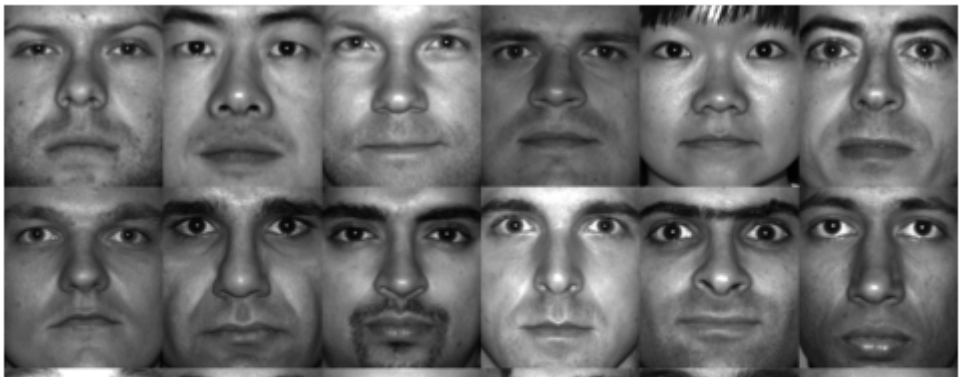

0
25
50
75
100
125
150
175
0
25
50
75
100
125
150

In [5]:
```
def display_all_participants():
    '''
    Input: None (remember that you have access to the global variables i
    Output: None

    Create a single image that contains one photo from each of the 36 ti
    organized as a 6 x 6 matrix of images
    '''
#     Write your code here
    fig, axes = plt.subplots(6,6, figsize=(10, 10))
    axes = axes.flatten()
    pics = 0
    m = len(trainingFaces)
    n = len(trainingFaces[0])
    for i in range(36):
        pics += nfaces[i]
        flattened = trainingFaces[:,pics%n]
        original_shape_image = np.reshape(flattened,(m_prime,n_prime)).T
        axes[i].imshow(original_shape_image, cmap='gray')
        axes[i].axis('off')
    plt.subplots_adjust(wspace = 0, hspace =0)
    plt.show()

display_all_participants()
```

Here is an example output for "display_all_participants()"



In [6]:
```
def display_one_participant(p_id):
    '''
    Input:
        p_id = index of the person to be displayed
        (remember that you have access to the global variables from the
    Output: None

    create and display a single image of all images of the participant a
    This single image should be organized as an 8x8 image matrix, and i
    unused image matrix cells can be left as all 0's

    '''
#     write your code here
    fig, axes = plt.subplots(8,8, figsize=(10,10))
    axes = axes.flatten()
    pics = 0
    m = len(trainingFaces)
    n = len(trainingFaces[0])
    num_faces = nfaces[p_id]
    picspassed = 0
    for passed in range(p_id):
        picspassed += nfaces[passed]
    for i in range(num_faces):
        flattened = trainingFaces[:,picspassed%n]
        original_shape_image = np.reshape(flattened,(m_prime,n_prime)).T
        axes[i].imshow(original_shape_image, cmap='gray')
        axes[i].axis('off')
        picspassed +=1
    for i in range(64-num_faces, 64):
        axes[i].axis('off')
    plt.subplots_adjust(wspace = 0, hspace =0)
    plt.show()


display_one_participant(0)
```

Here is an example of an output of "display_one_participant" for person_id (p_id) = 8

In [7]:
```
def average_face():
    '''
    Input: None (remember that you have access to the global variables 
    Output: np-array - (n'*m'x 1) flattened image of the average face f

    Take the average of the training set to find the average face. Displ
    '''
#     write your code here
    avg_face = np.mean(trainingFaces, axis=1)
    avg_face_T = avg_face.reshape(-1, 1)
    avg_face_img = np.reshape(avg_face, (m_prime, n_prime)).T
    plt.figure(figsize=(5,5))
    plt.imshow(avg_face_img, cmap='gray')
    plt.axis('off')
    plt.title('Average Face')
    plt.show()
    return avg_face_T

avg_face = average_face()#once this cell is run, you can access "avg_fa
```

In [8]:
```
def mean_center_SVD(avg_face):
    '''
    Input:
        avg_face = np array (n*m by 1) which is the result from "average
        (remember that you have access to the global variables from the

    Output:
        np-array: U from the SVD, which is a (n x r) matrix

    1. Take the SVD of the mean-centered training data
    2. Display the eigenface at index 0
    3. Return the matrix U.
    '''
    #     write your code here
    # mean center
    centered_training_faces = trainingFaces - avg_face
    # SVD
    U, s, Vt = np.linalg.svd(centered_training_faces, full_matrices=0)

    # Display the eigenface at index 0
    eigenface = U[:, 0]
    eigenface_image = np.reshape(eigenface,(m_prime, n_prime)).T
    plt.figure(figsize=(5,5))
    plt.imshow(eigenface_image, cmap='gray')
    plt.title('Eigenface at Index 0')
    plt.axis('off')
    plt.show()

    return U

U = mean_center_SVD(avg_face) #once this cell is run, you can call "U"
                              #Be sure to run "average_face()" first to
```

In [9]:
```
def reconstruct(U, p, x, avg_face):
    '''
    Input:
        U = np array (n x r) from "mean_centered_SVD()"
        p = int, representing the first p eigenfaces to use in the recons
        x = np array (n x 1) represents an original image
        avg_face = np array from "average_face()"
        (remember that you have access to the global variables from the i

    output:
        x_hat = np array (n x 1) reconstruction of x using the eigenfaces

    Reconstruct x, x_hat, using the first p columns of U.

    A few notes to remember:
        1. x_hat = Up(Up.T)x, to speed up computation, we recommend first
            Up(alpha).
        2. Mean center x before reconstruction
        3. Because U and x will both be mean centered, your final step in
    '''
    #     write your code here
    x_mean_centered = x - avg_face
    alpha = U[:, :p].T @ x_mean_centered
    x_hat = U[:, :p] @ alpha
    x_hat += avg_face

    return x_hat
```

In [10]:
```
def reconstruct_experiments(photo_index, p_list = [25, 50, 100, 200, 40(
    '''
    Input:
        photo_index: int between 0 <= photo_index <= cols(testingFaces)
        p_list: List, represents the values of p to be used in the recor

    Output:
        None

    Make sure this function does the following:
        1. Displays the original image to be reconstructed
        2. Displays the reconstruction of the original image for each p
        3. All reconstructions are labelled clearly as to the value of p
    '''
    #     write your code here
    # Display the original image
    original = testingFaces[:, photo_index]
    original_image = np.reshape(original,(m_prime, n_prime)).T
    plt.figure(figsize=(5,5))
    plt.imshow(original_image, cmap='gray')
    plt.title('Original Image')
    plt.axis('off')
    i = 0
    fig, axes = plt.subplots(1,5)
    for p in p_list:
        x = original.reshape((-1, 1))
        x_hat = reconstruct(U, p, x, avg_face)
        reconstructed_image = np.reshape(x_hat,(m_prime, n_prime)).T

        # Display the reconstructed image
        axes[i].axis('off')
        axes[i].imshow(reconstructed_image, cmap='gray')
        i+=1

    plt.show()
    print(f'Reconstructed Image (p = 25, 50, 100, 200, 400)')
```

In [11]:
```
#  MAKE SURE YOU RUN THIS CELL AND HAVE OUTPUT AVAILABLE IN THE FINAL Pi
reconstruct_experiments(0)
```

Reconstructed Image (p = 25, 50, 100, 200, 400)

In [ ]: