

```
In [ ]: import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import numpy.random as rnd
```

```
In [ ]:
```

Distinct element estimation using k-th min (6 pts)

In the lecture, we studied the algorithm named Idealized F_0 estimation (slide 19). The algorithm uses a random hash function to map elements from the stream to float values between 0 and 1. Ultimately, it maintains the smallest hash value V and outputs $\frac{1}{V} - 1$ as the estimate \tilde{F}_0 for the number of distinct elements.

This algorithm uses the idea that the expected value of the smallest hash value is $\frac{1}{F_0+1}$, where F_0 is the number of distinct elements. In fact, we can generally use the k -th smallest hash value V_k for $k = 1, 2, \dots$. We will use the results from exercise 4 to conduct experiments to see how different k values affect the accuracy of your estimate.

[Optional]: Let m be the length of the stream. You can maintaining the k -th smallest element in an unsorted list in time $O(m \log k)$ using min heap, see <https://docs.python.org/3/library/heapq.html>.

```
In [ ]: # Import packages needed.
import random, math
import numpy as np
import matplotlib.pyplot as plt
```

To test the effect of k , we must first implement a function that takes a data sequence, hash each element to a value between 0 and 1, and returns the k -th smallest hash value. Python has a built-in hash function `hash()` that takes any hashable object and returns an integer hash. To convert a hash value to a float, use modular the hash with a large int and divide by it, for instance, $MAXINT = 2^{63} - 1$.

```
In [ ]: import sys
MAXINT = sys.maxsize
```

```
In [ ]: def kth_smallest_hash_value(input_list, k):
#     Write your code here
    hash_table = []
    for i in range(len(input_list)):
        hash_value = (hash(input_list[i]) % MAXINT) / MAXINT
        hash_table.append(hash_value)
    hash_table.sort()
```

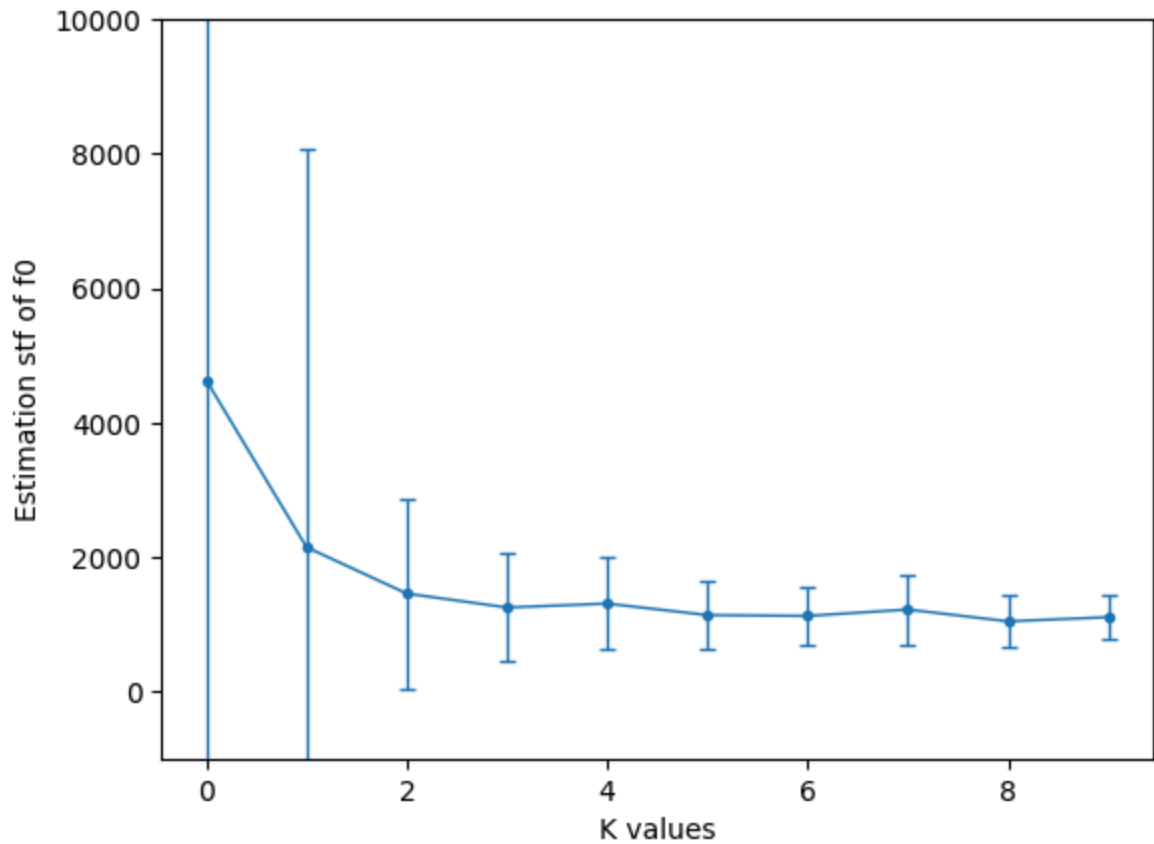
```
return hash_table[k-1]
```

Now let us test k values between 1 to 10. For each k , we will generate a list of 1000 random **strings** using `str(random.uniform(0,100))`, and estimate its cardinality via the returned value from the function `kth_smallest_hash_value` you implemented. For each k , repeat this process 100 times and record the average and std of the estimates. Finally, generate a plot with error bars to show the relation between estimates and k values. Note that the std for small k can be very large, so you may need to set `plt.ylim(-1000, 10000)` to cap the y-axis for better visualization.

```
In [ ]: # Write your code here
average = np.zeros(10)
std = np.zeros(10)
result = np.zeros(100)

for k in range(10):
    for r in range(100):
        input_list = [str(random.uniform(0, 100)) for i in range(1000)]
        kth_value = kth_smallest_hash_value(input_list, k+1)
        f0 = (k+1)/(kth_value) - 1
        result[r] = f0
    # print(result)
    average[k] = np.mean(result)
    std[k] = np.std(result)

plt.errorbar(range(10), average, yerr=std, linestyle='-', marker='o', marker
plt.xlabel('K values')
plt.ylabel('Estimation stf of f0')
plt.ylim(-1000, 10000)
plt.show()
```



In []:

The median trick useful technique (slide 13) (6 pts)

Please implement the function `median_trick` below.

```
In [ ]: def median_trick(generator, expectation, var, eps, delta):
    """
    Input:
        generator - a function that generates one sample from a distribution
        expectation - Expectation of the distribution
        var - Variance of the distribution
        eps - epsilon (accuracy parameter) as defined in slide 13
        delta - delta (confidence parameter) as defined in slide 13
    Output:
        estimated value Q
    """
    # Write your code here
    t = math.ceil(math.log(1/delta))
    k = math.ceil(var/(eps**2 * expectation**2))
    array = []
    for i in range(t):
        tmp = []
        for j in range(k):
            tmp.append(generator())
        array.append(np.average(tmp))
```

```
Q = np.median(array)
return Q
```

Now we want to test the function with the following idea. Assume $Q=2$. The unbiased estimator, X of Q , generates estimates that follow a normal distribution with variance equal to 1. The generator for X is already given below as `normal_generator`. Please generate two plots below.

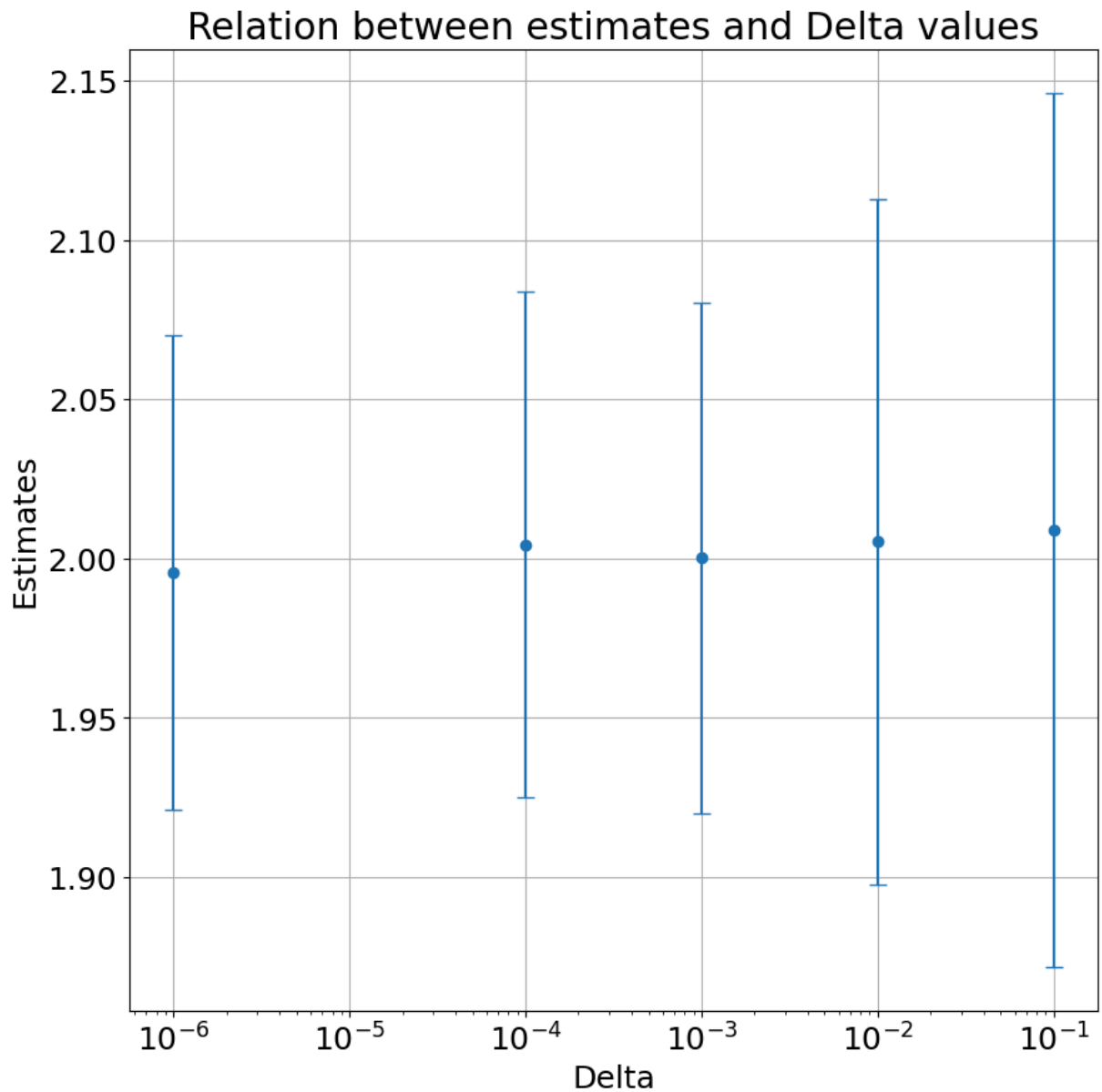
- Set $\text{eps}=0.1$, and test how the delta affects the estimates. Range delta in $[1e-6, 1e-4, 1e-3, 0.01, 0.1]$; repeat the estimation 100 times for each delta value. Generate a plot with std as error bars to show how the average estimates change as the delta changes.
- Set $\text{delta}=0.1$, and test how the epsilon affects the estimates. Range epsilon in $[0.01, 0.02, 0.05, 0.1, 0.2]$; repeat the estimation 100 times for each epsilon value. Generate a plot with std as error bars to show how the average estimates change as the epsilon changes.

```
In [ ]: # Don't change
def normal_generator():
    return np.random.normal(2,1)
```

```
In [ ]: # Write your code here
eps = 0.1
delta = [1e-6, 1e-4, 1e-3, 0.01, 0.1]
average = np.zeros(5)
std = np.zeros(5)
result = np.zeros(100)

for k in range(len(delta)):
    for r in range(100):
        q_value = median_trick(normal_generator, 2, 1, eps, delta[k])
        result[r] = q_value
        # f0 = (k+1)/(kth_value) - 1
        # result[r] = f0
    # print(result)
    average[k] = np.mean(result)
    std[k] = np.std(result)

plt.errorbar(delta, average, yerr=std, fmt='o', capsize=5)
plt.xscale('log')
plt.xlabel('Delta')
plt.ylabel('Estimates')
plt.title('Relation between estimates and Delta values')
plt.grid(True)
plt.show()
```



In []:

In []: "----- Delta = 0.1, Epsilon = [0.01, 0.02, 0.05, 0.1

```

# Write your code here
eps = [0.01, 0.02, 0.05, 0.1, 0.2]
delta = 0.1
average = np.zeros(5)
std = np.zeros(5)
result = np.zeros(100)

for k in range(len(eps)):
    for r in range(100):
        q_value = median_trick(normal_generator, 2, 1, eps[k], delta)
        result[r] = q_value
        # f0 = (k+1)/(kth_value) - 1
        # result[r] = f0
    # print(result)
    average[k] = np.mean(result)

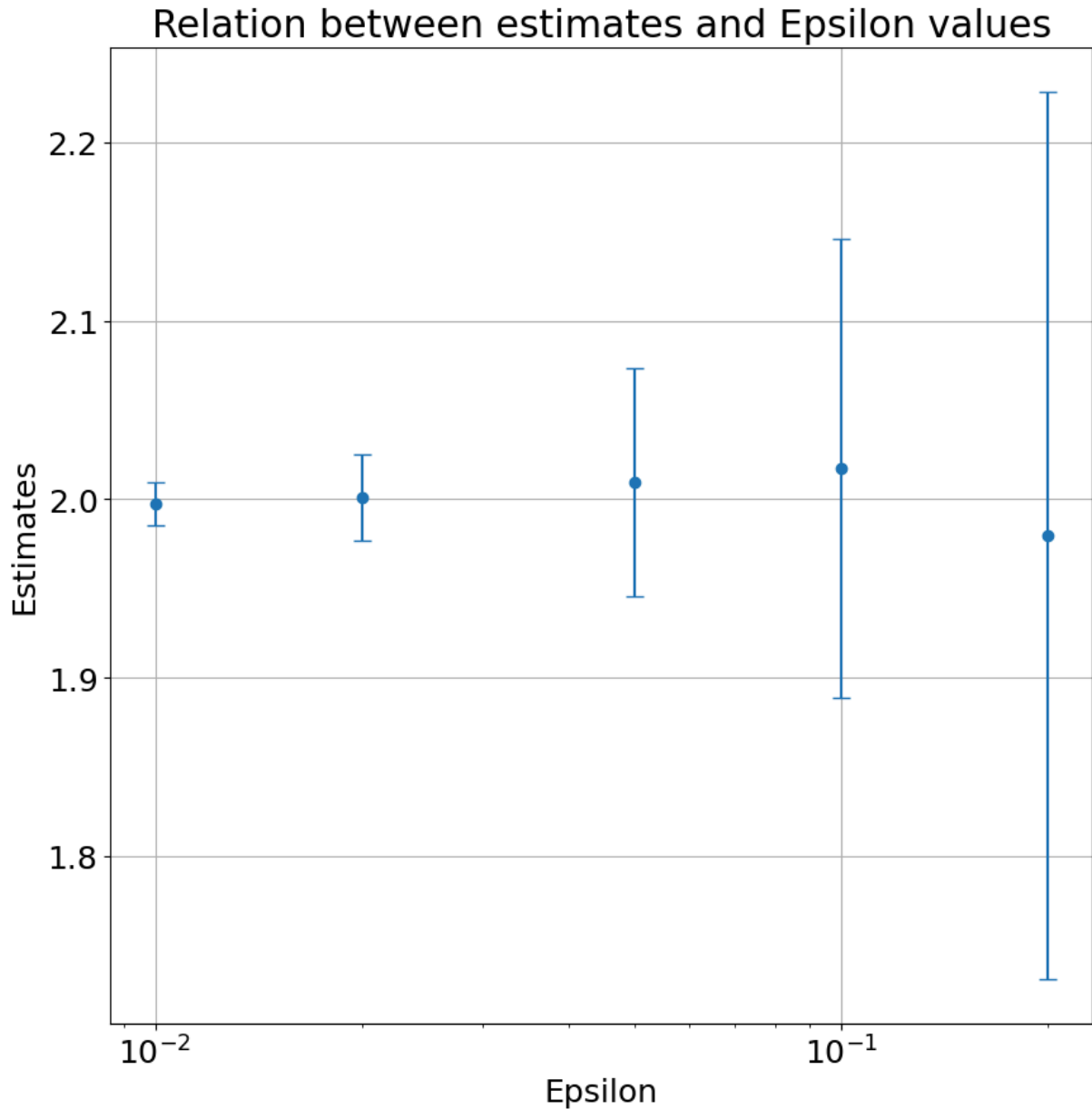
```

```

std[k] = np.std(result)

plt.errorbar(eps, average, yerr=std, fmt='o', capsize=5)
plt.xscale('log')
plt.xlabel('Epsilon')
plt.ylabel('Estimates')
plt.title('Relation between estimates and Epsilon values')
plt.grid(True)
plt.show()

```



Morris Algorithm (slide 45) (6 pts)

Morris algorithm maintains a counter c that, for every element in the stream, itself increments by 1 with probability $\frac{1}{2^c}$. In the end, it outputs an estimate as $2^c - 1$.

In this section, we will change the base of this counter (slide 51). Instead of using 2 only, we use any base $1 + \alpha$. We now increase the counter c with probability $\frac{1}{(1+\alpha)^c}$. First, let us implement the function `morris_update_base_alpha` below. **This function is called whenever we see an element from the stream to update the counter.**

```
In [ ]: def morris_update_base_alpha(counter, alpha):
        ...
        Input:
            counter - current value of counter c
            alpha - as defined in slide 51 alpha
        Output:
            updated value of counter c
        ...
        # print(1 / ((1+alpha)**counter))
        probability = (1 / ((1+alpha)**counter))
        flip = np.random.binomial(1, probability)
        # print(flip)
        if flip == 1:
            return counter + 1
        else:
            return counter
```

Now let us test the function with the edge list file "soc-hamsterster.edges" in the same folder. Reading the file line by line in python can generate a stream of strings. Counting the number of strings/lines in this file tells us the number of edges of this "soc-hamsterster" graph. Let us try different alpha values ranging from 2 to 9. Again, for each alpha, estimate the number of lines in the edge list file using the morris algorithm (the key component of which is `morris_update_base_alpha`), and repeat this 100 times. Besides, check how many bits are needed to maintain the counter via `math.ceil(math.log(counter, 2))` at the end of each estimation. Finally, generate two plots with std as error bars to show

- How the average estimate changes as the alpha value increases.
- How the space usage (in bits) changes as the alpha value increases.

```
In [ ]: alpha = [2, 3, 4, 5, 6, 7, 8, 9]
        average = np.zeros(len(alpha))
        std = np.zeros(len(alpha))
        bit_average = np.zeros(len(alpha))
        bit_std = np.zeros(len(alpha))

        result = np.zeros(100)
        bit_result = np.zeros(100)

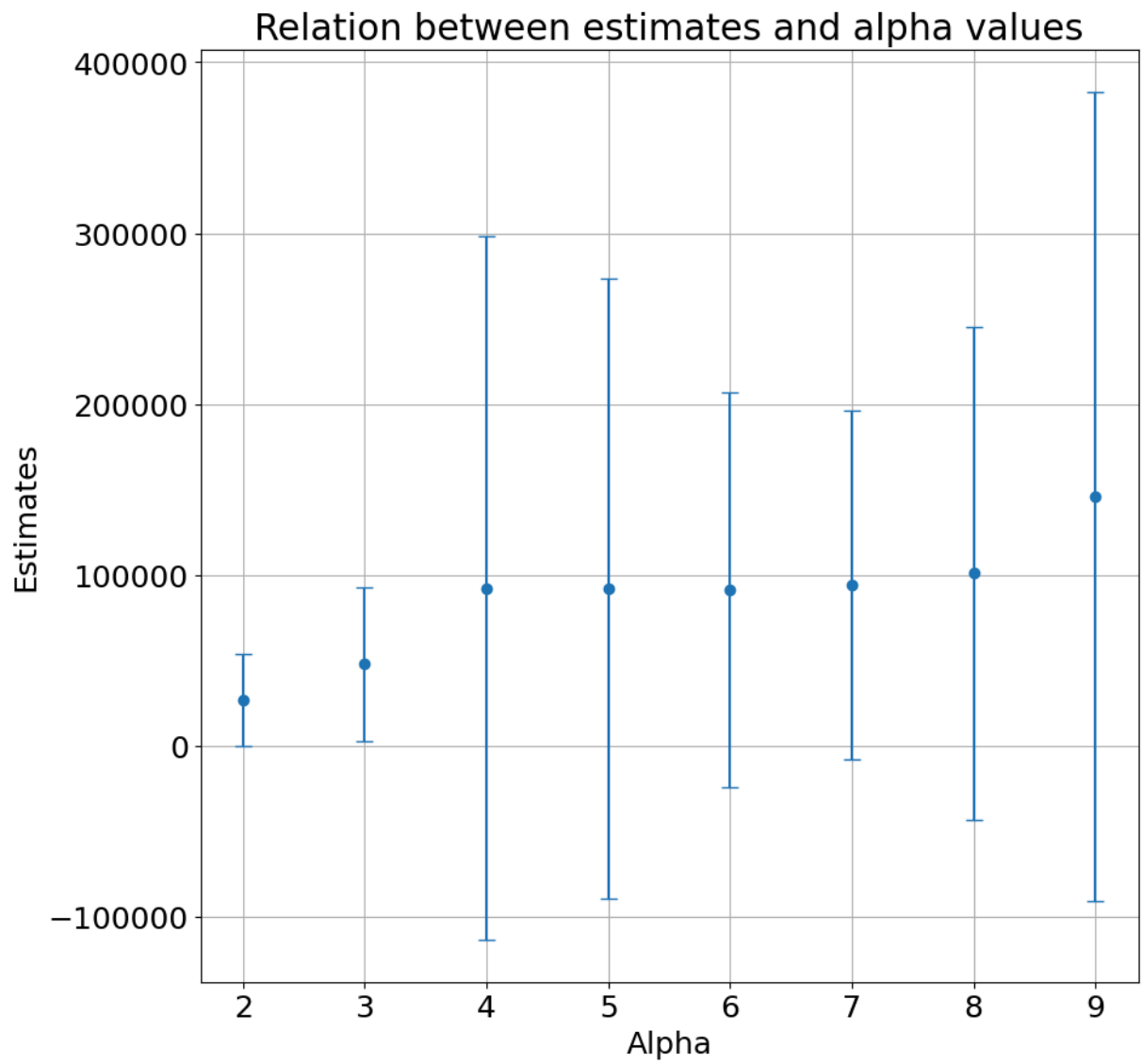
        for i in range(len(alpha)):
            for j in range(100):
                counter = 0
                f = open("./soc-hamsterster.edges", "r")
                for line in f:
```

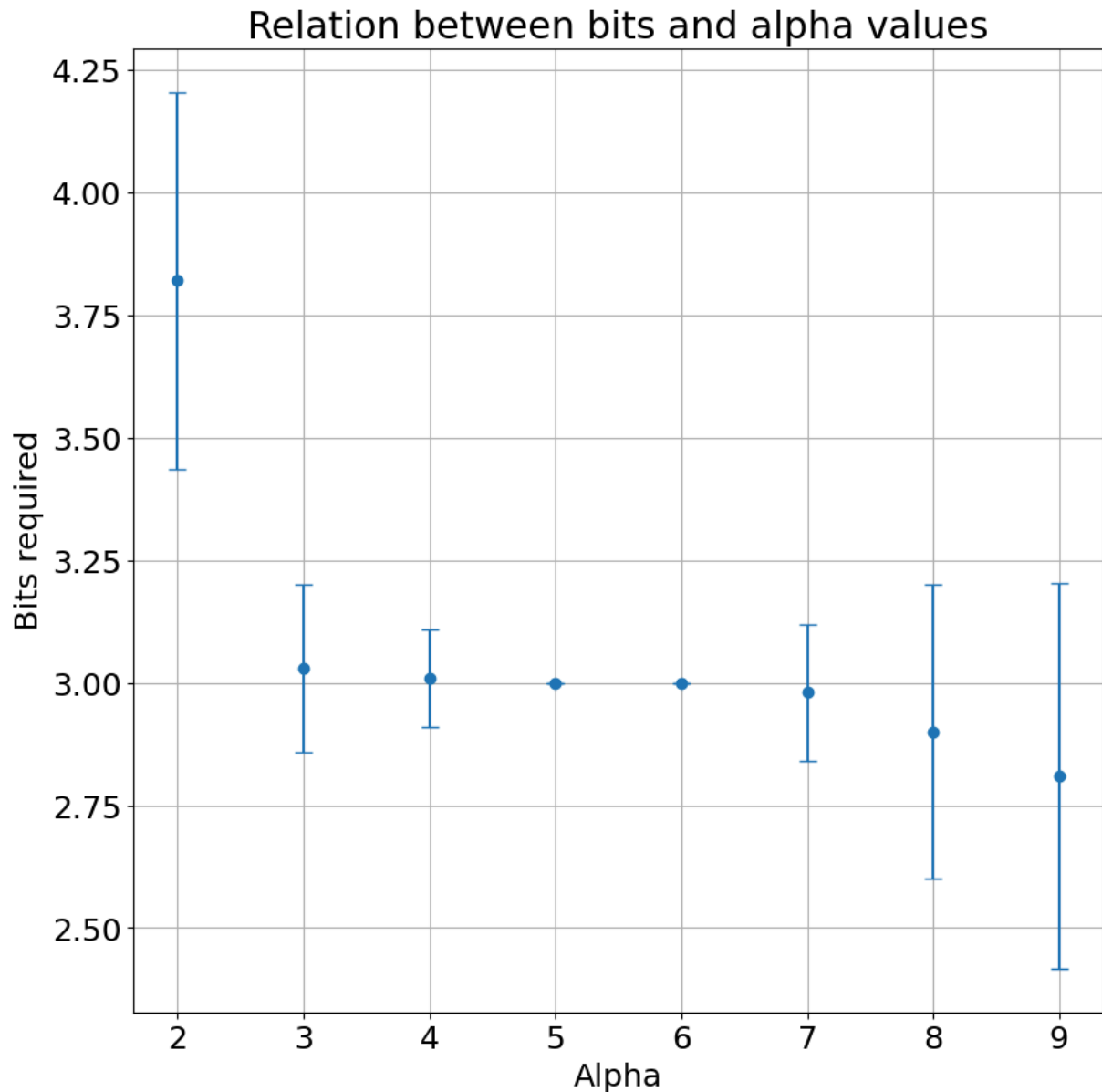
```
        counter = morris_update_base_alpha(counter, alpha[i])
        output_estimate = ((1 + alpha[i]) ** counter) - 1
        result[j] = output_estimate
        bit_result[j] = math.ceil(math.log(counter, 2))
        f.close()
    average[i] = np.mean(result)
    std[i] = np.std(result)

    bit_average[i] = np.mean(bit_result)
    bit_std[i] = np.std(bit_result)

plt.errorbar(alpha, average, yerr=std, fmt='o', capsize=5)
plt.xlabel('Alpha')
plt.ylabel('Estimates')
plt.title('Relation between estimates and alpha values')
plt.grid(True)
plt.show()

plt.errorbar(alpha, bit_average, yerr=bit_std, fmt='o', capsize=5)
plt.xlabel('Alpha')
plt.ylabel('Bits required')
plt.title('Relation between bits and alpha values')
plt.grid(True)
plt.show()
```



Eigenfaces [40 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 $((m*n) \times 2410)$ 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 "flattened", 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 ($m' \times n'$) 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](#) . If you are looking to gather a more intuitive sense of SVD, take a look at the "Intuitive interpretations" section of the [SVD wikipedia](#) , 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](#) .
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 turns 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 vector form:

$$\hat{x} = U_p \alpha$$

Where \hat{x} is our reconstruction of the original image x . We clearly see that \hat{x} is simply a linear combination of columns from our matrix, U .

```
In [ ]: #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})

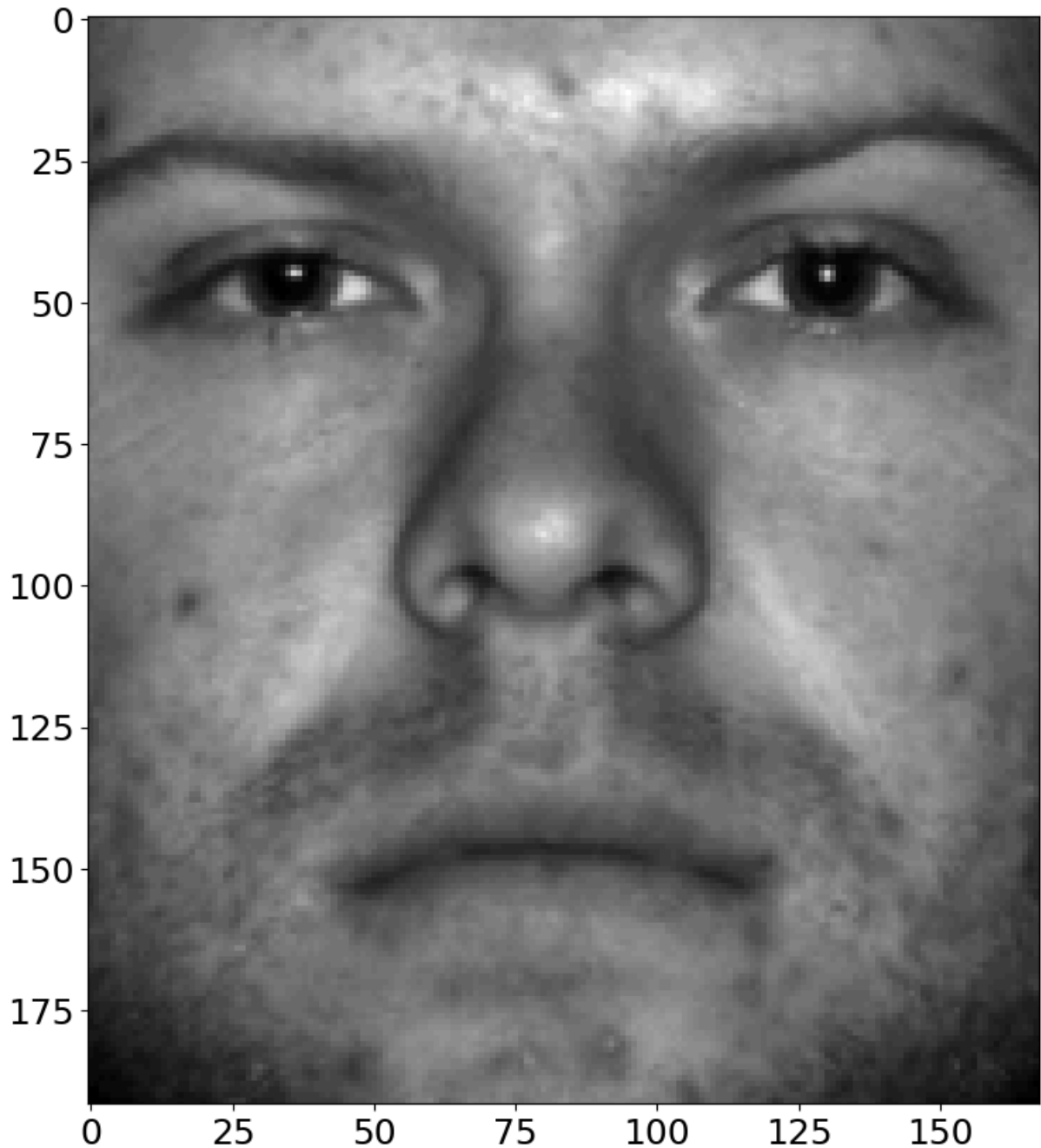
#set as global variables that can be called in any function
mat_contents = scipy.io.loadmat(os.path.join('.', 'DATA', 'allFaces.mat'))

#common variables
faces = mat_contents['faces'] #data, where each column is a flattened image
```

```
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 images
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')
```

```
/var/folders/xt/y_17jz6n3p77v5p6cbm1tstr0000gn/T/ipykernel_77275/1690186382.
py:14: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar
is deprecated, and will error in future. Ensure you extract a single element
from your array before performing this operation. (Deprecated NumPy 1.25.)
    m_prime = int(mat_contents['m']) #number of pixel rows in each image
/var/folders/xt/y_17jz6n3p77v5p6cbm1tstr0000gn/T/ipykernel_77275/1690186382.
py:15: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar
is deprecated, and will error in future. Ensure you extract a single element
from your array before performing this operation. (Deprecated NumPy 1.25.)
    n_prime = int(mat_contents['n']) #number of pixel cols in each image
```



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

        Create a single image that contains one photo from each of the 36 training
        images organized as a 6 x 6 matrix of images
        """
        # Write your code here
        index = 0
        index_array = []

        for i in range(36):
            index_array.append(index)
            index += nfaces[i]
```

```

fig, axs = plt.subplots(6, 6)
for i in range(6):
    for j in range(6):
        img = axs[i, j].imshow(np.reshape(trainingFaces[:, index_array[i]], (48, 48)))
        img.set_cmap('gray')
        axs[i, j].axis('off')

display_all_participants()

```



Here is an example output for "display_all_participants()"

 example all participants image

```

In [ ]: 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 impo
Output: None

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

```

'''
# write your code here
index = 0
for i in range(p_id):
    index += nfaces[i]


fig, axs = plt.subplots(8, 8)

for i in range(8):
    for j in range(8):
        if (i*8+j) < nfaces[p_id]:
            img = axs[i, j].imshow(np.reshape(trainingFaces[:, index+i*8+
            img.set_cmap('gray')
        else:
            img = axs[i, j].imshow(np.zeros((m_prime, n_prime)).T)
            img.set_cmap('gray')
            axs[i, j].axis('off')
display_one_participant(0)

```



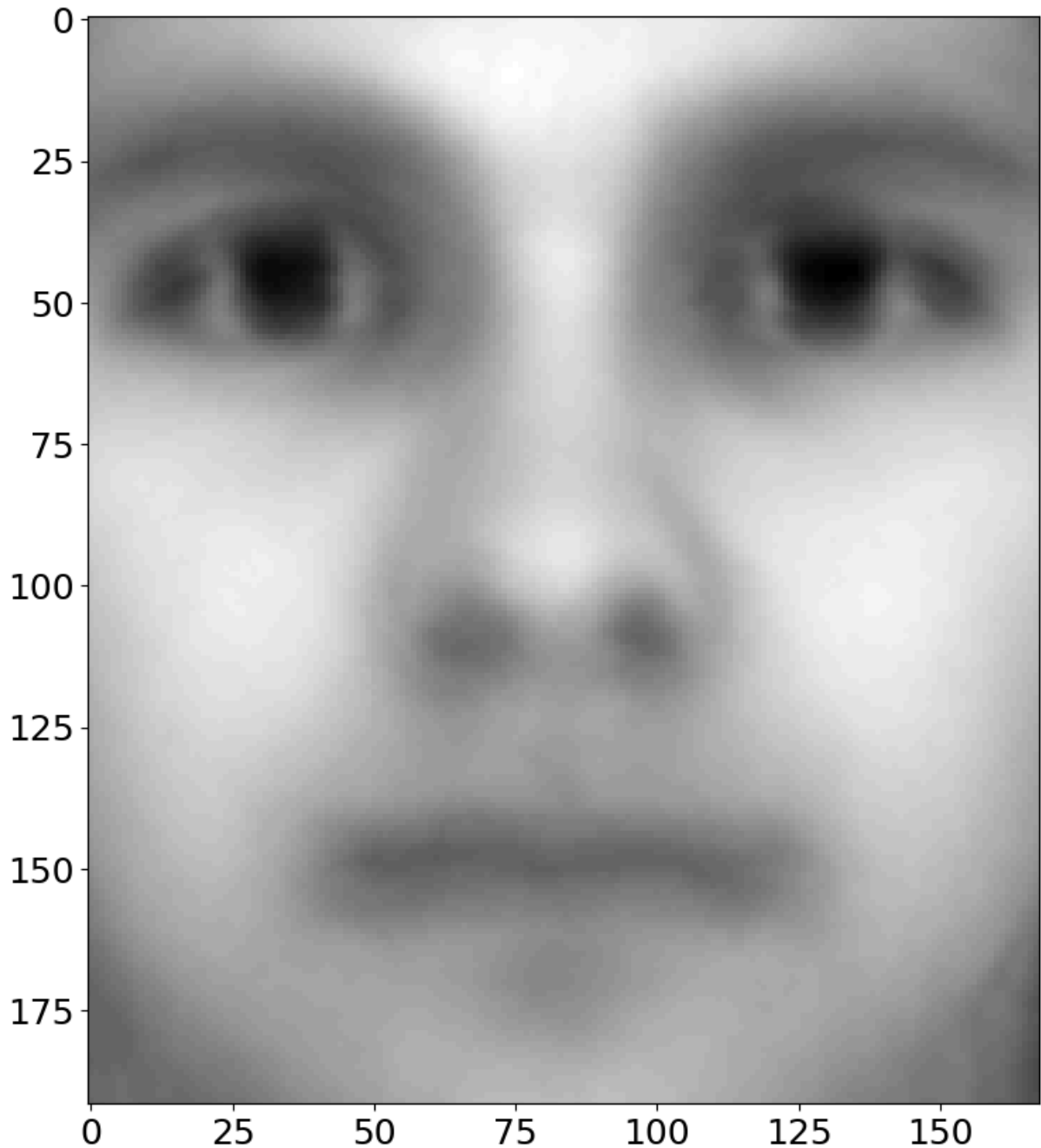
Here is an example of an output of "display_one_participant" for person_id (p_id) = 8

 example one participant image

```
In [ ]: def average_face():
    """
    Input: None (remember that you have access to the global variables from
    Output: np-array - (n*m'x 1) flattened image of the average face from t

    Take the average of the training set to find the average face. Display t
    """
    # write your code here
    avg_face = np.mean(trainingFaces, axis=1)
    img = plt.imshow(np.reshape(avg_face, (m_prime, n_prime)).T)
    img.set_cmap('gray')
    return avg_face

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



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

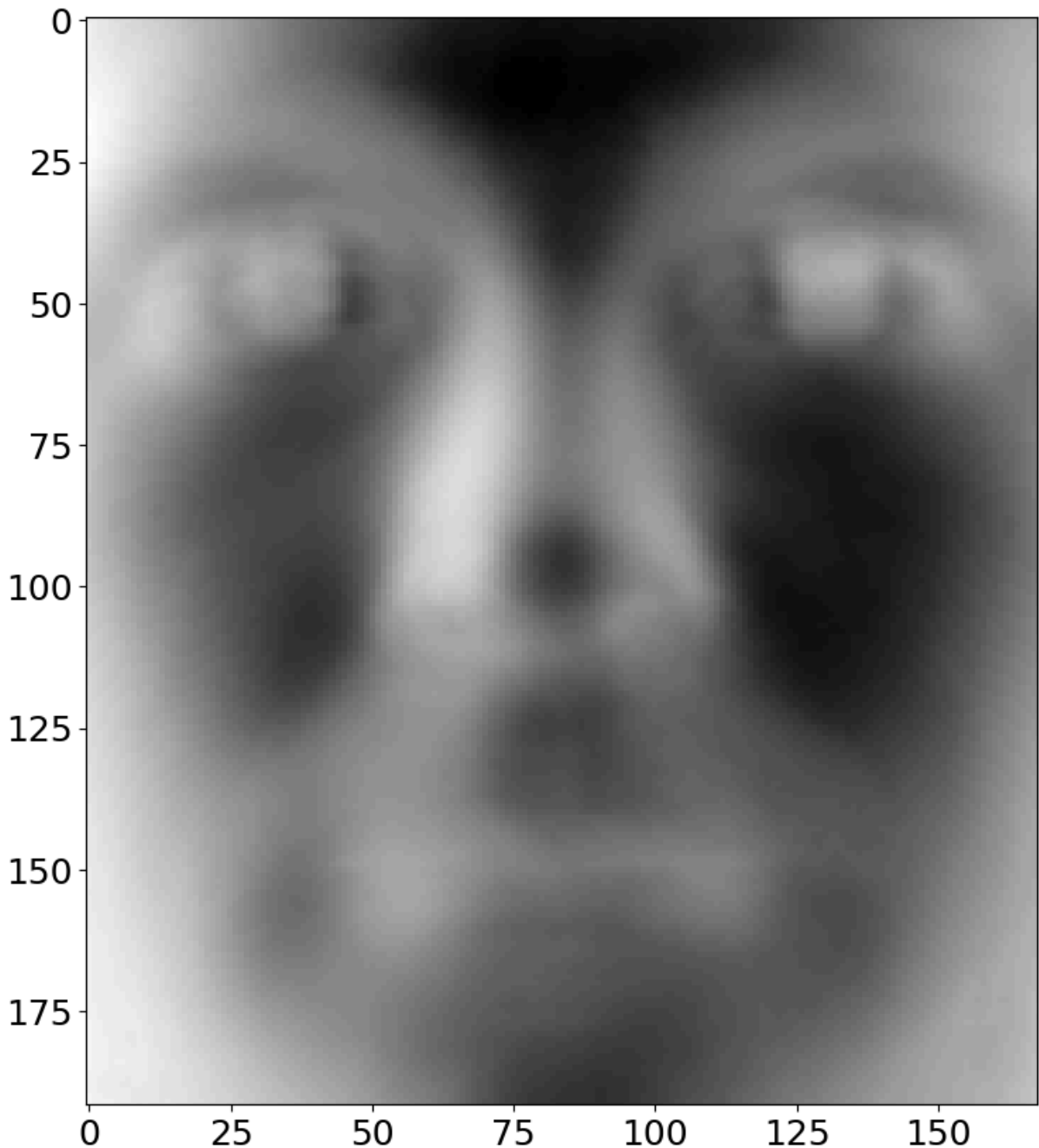
        Output:
            np-array: U from the SVD, which is a (n x r) matrix

        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
        U, S, VT = np.linalg.svd(trainingFaces - avg_face[:, np.newaxis], full_matrices=True)
```

```
img = plt.imshow(np.reshape(U[:,0],(m_prime,n_prime)).T)
img.set_cmap('gray')
return U
```

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

```
[61.45223488 62.01139351 62.26029798 ... 41.20464505 41.97589833
42.38825592]
```



```
In [ ]: 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 reconstru
            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 impo

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

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 co
        Up(alpha).
    2. Mean center x before reconstruction
    3. Because U and x will both be mean centered, your final step in th
    ...

#     write your code here
alpha = np.dot(U[:, :p].T, x - avg_face)
x_hat = np.dot(U[:, :p], alpha)
x_hat += avg_face
return x_hat

```

```

In [ ]: def reconstruct_experiments(photo_index, p_list = [25, 50, 100, 200, 400]):
    ...
    Input:
        photo_index: int between 0 <= photo_index <= cols(testingFaces). rep
        p_list: List, represents the values of p to be used in the reconstru

    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 in p
        3. All reconstructions are labelled clearly as to the value of p tha
    ...

#     write your code here
' Display the original image '
img = plt.imshow(np.reshape(testingFaces[:,photo_index],(m_prime,n_prime)
img.set_cmap('gray')
plt.title('Original Image to be reconstructed')
plt.show()

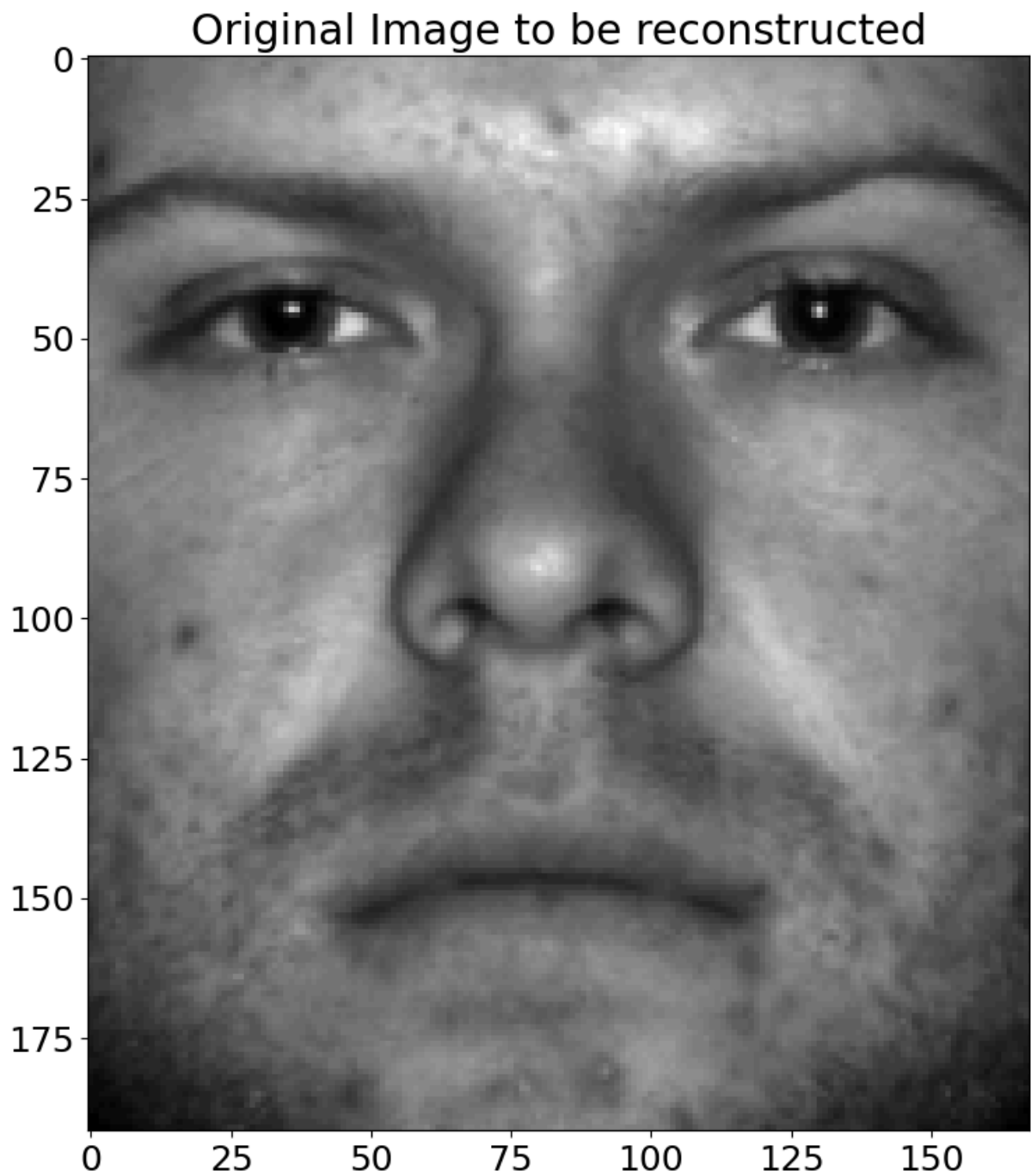
' Display the reconstruction of the original image for each p in p_list
for p in p_list:
    x_hat = reconstruct(U, p, testingFaces[:,photo_index], avg_face)
    img = plt.imshow(np.reshape(x_hat,(m_prime,n_prime)).T)
    img.set_cmap('gray')
    plt.title('p = ' + str(p))
    plt.show()

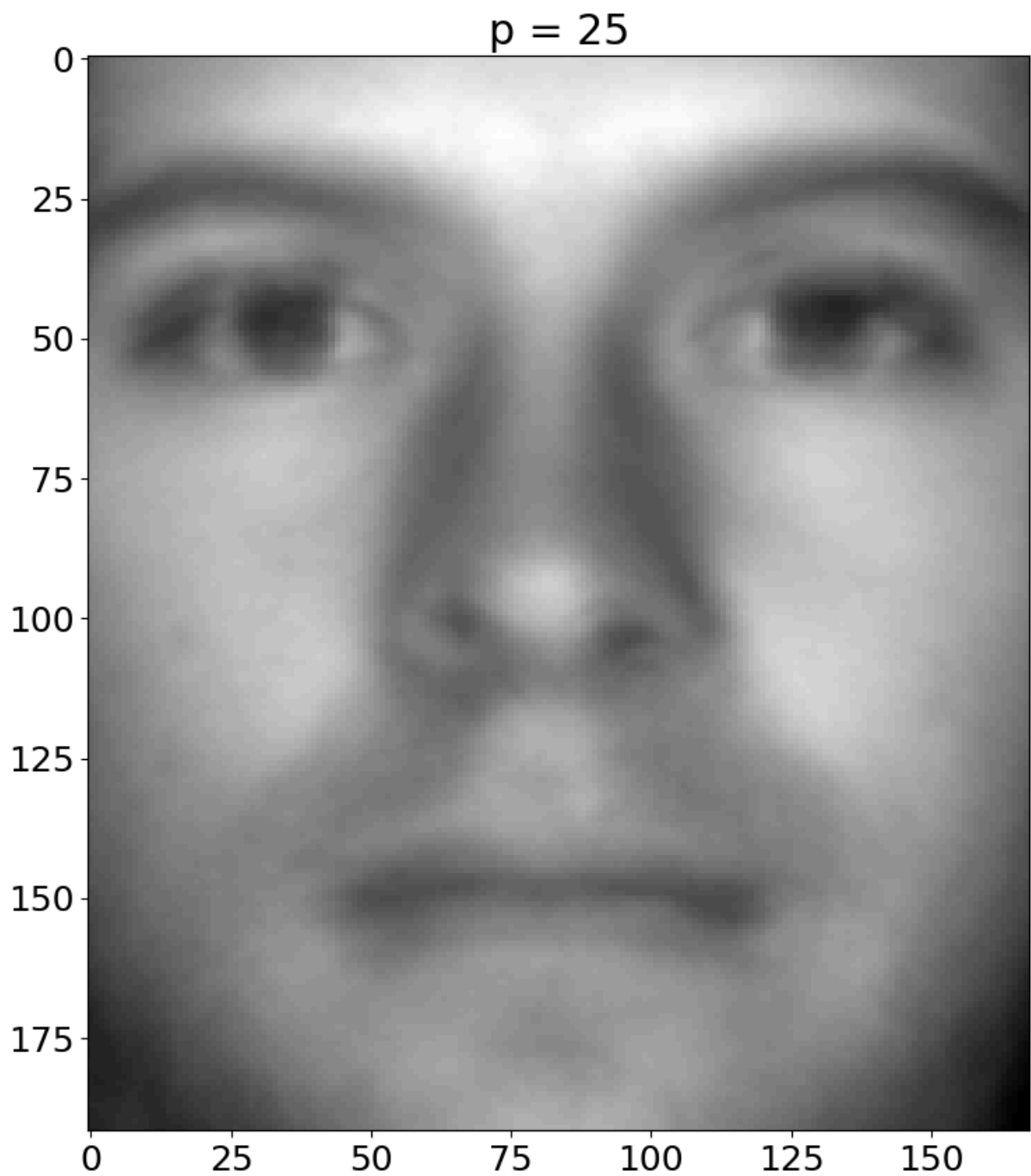
```

```

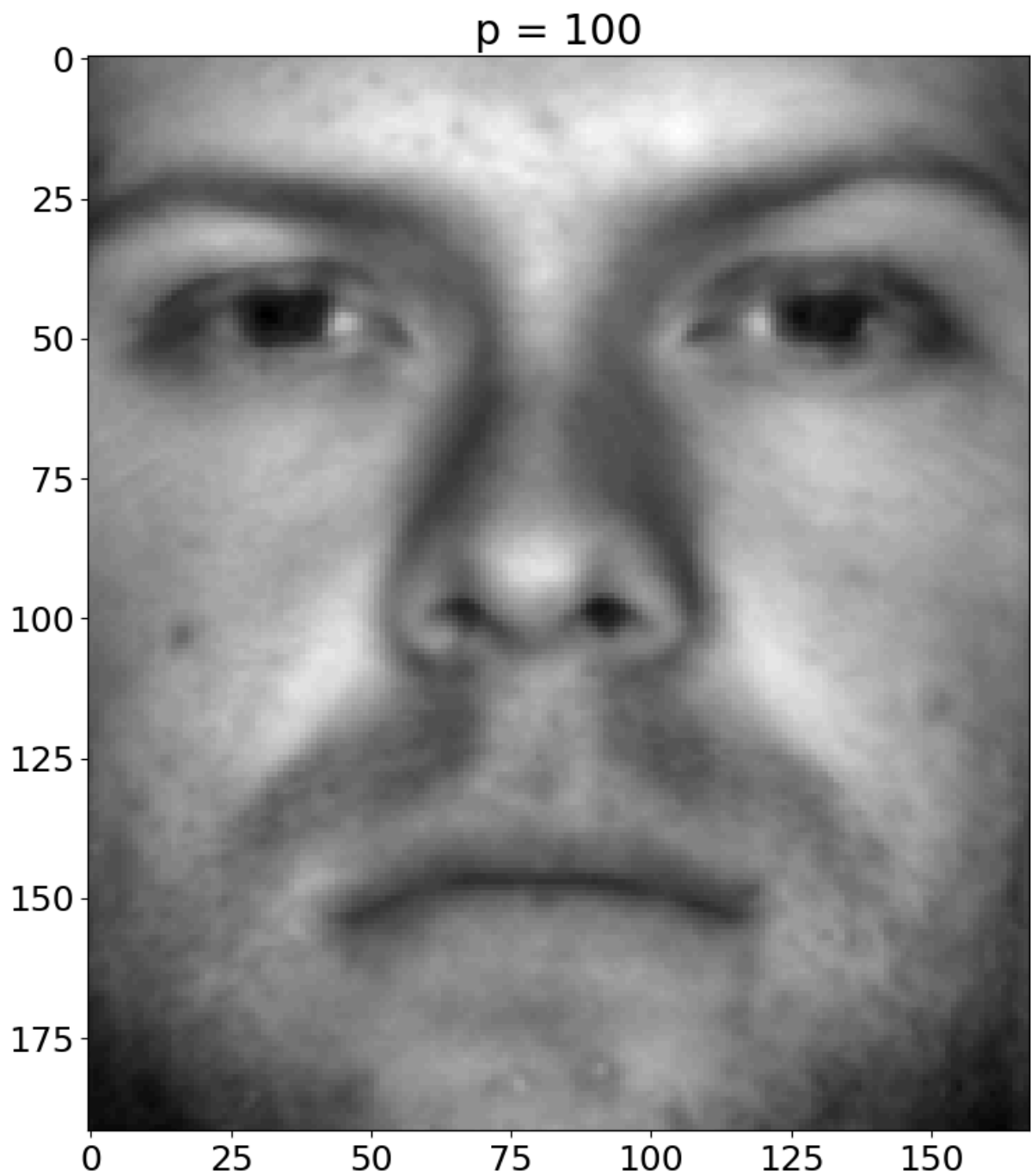
In [ ]: # MAKE SURE YOU RUN THIS CELL AND HAVE OUTPUT AVAILABLE IN THE FINAL PDF!!
reconstruct_experiments(0)

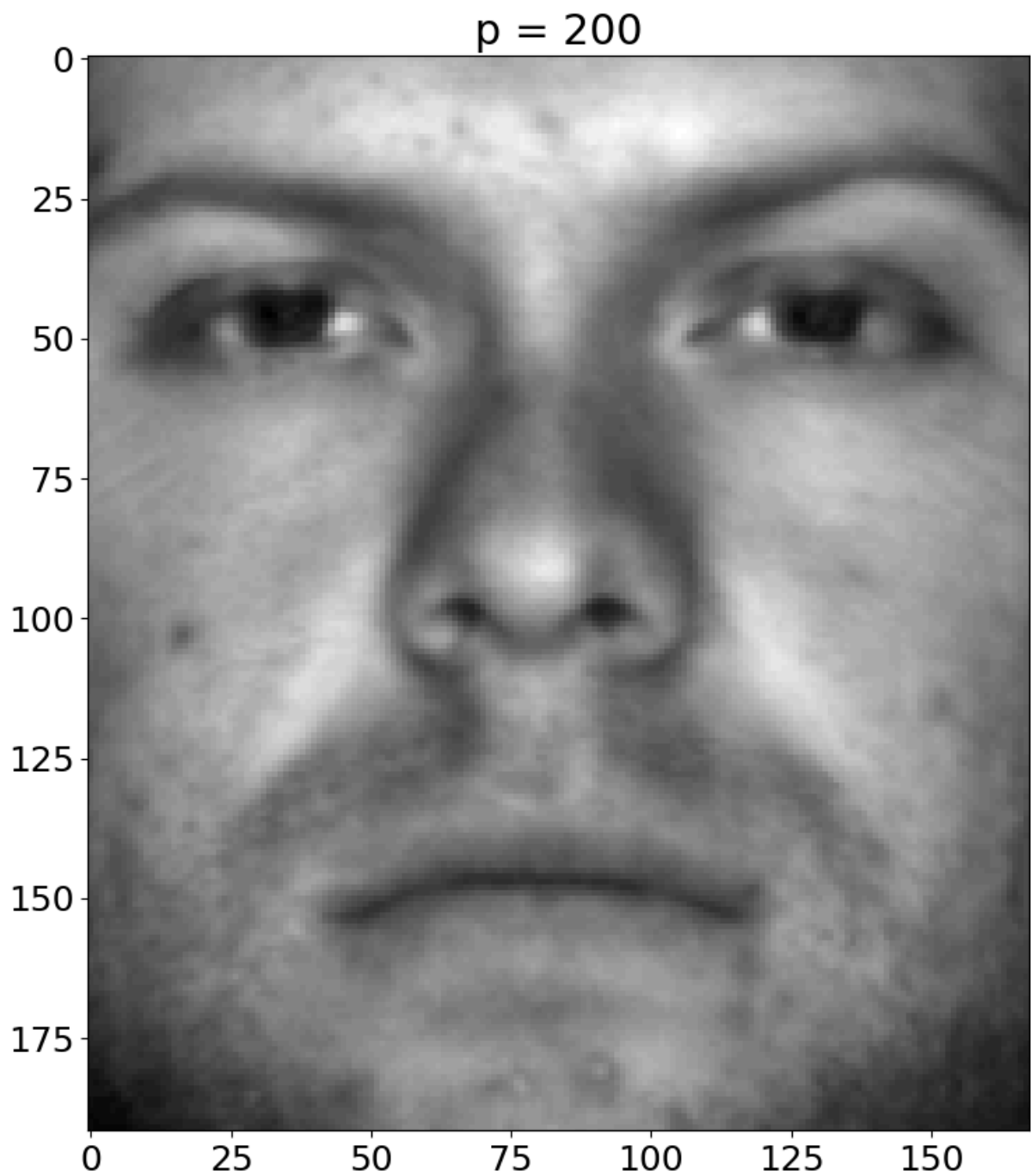
```

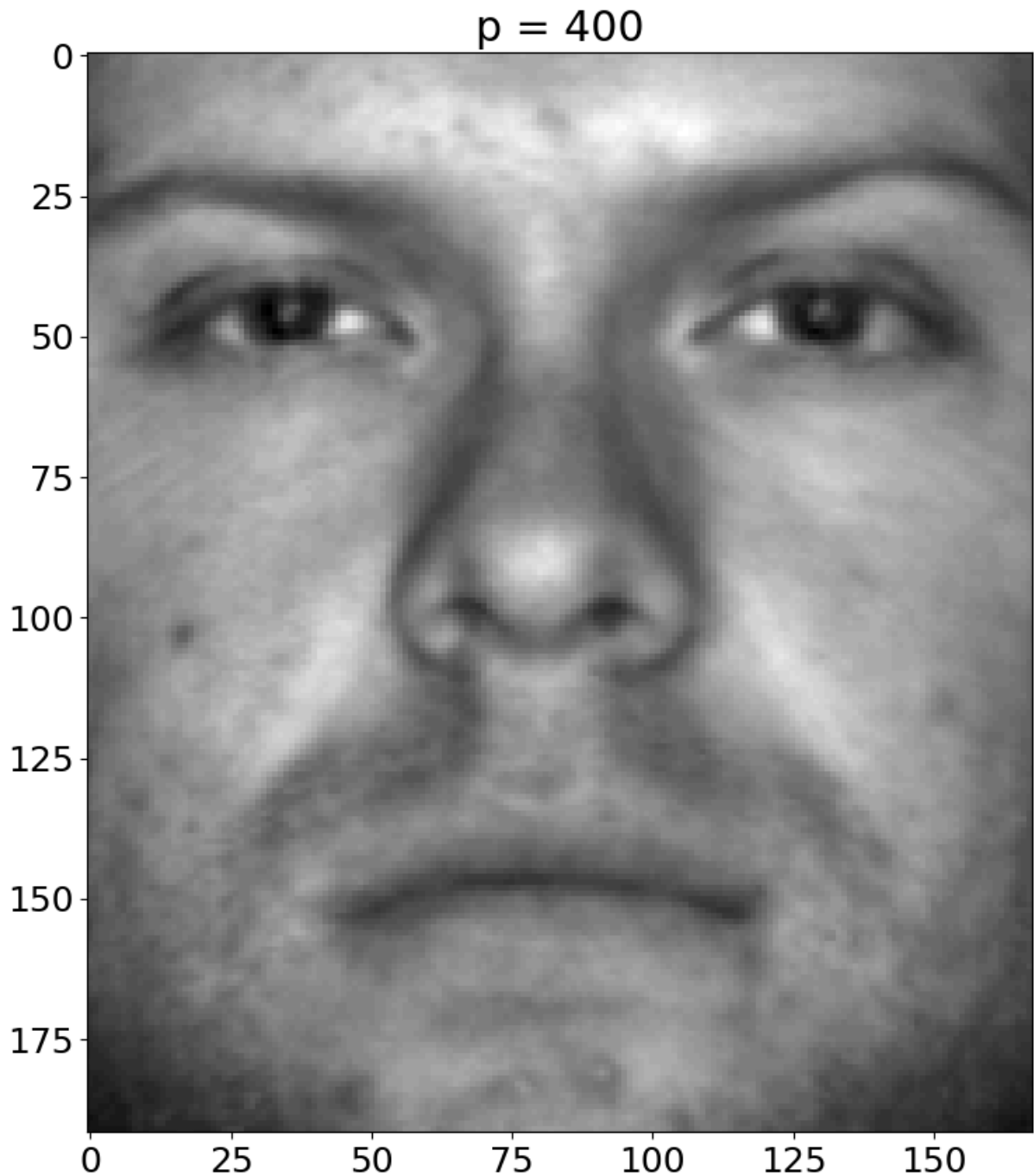












In []:

PCA (7 pts)

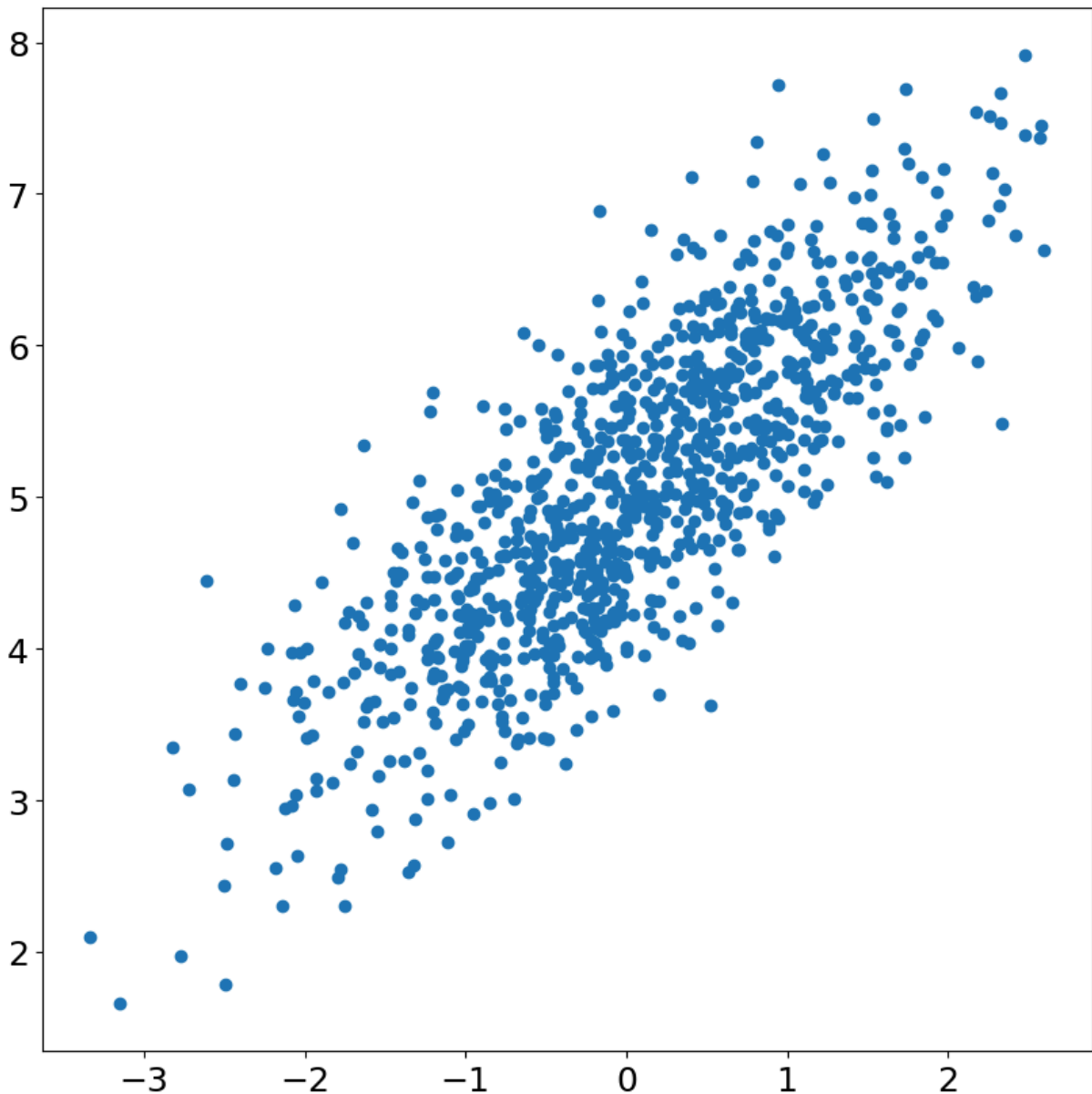
Implement PCA using eigendecomposition.

Let's first generate random data from a multivariate Gaussian distribution, and provide a scatter plot of the original data.

```
In [ ]: mu = np.array([0,5])
        sigma = np.array([[1, 0.8], [0.8,1]])
```

```
org_data = rnd.multivariate_normal(mu, sigma, size=(1000))
```

```
In [ ]: plt.scatter(org_data[:,0], org_data[:,1])
plt.show()
```



Compute the covariance matrix using `np.cov`. Data needs to be centered at zero in advance, so we should reduce both dimensions by mean.

```
In [ ]: # x = org_data[:,0]
# y = org_data[:,1]
# print(org_data)
# print(org_data.shape)
# print(np.average(org_data, axis=0))
# centered_x = x - np.mean(x)
# centered_y = y - np.mean(y)
centered_data = org_data - np.mean(org_data, axis=0)
# print(np.mean(centered_data, axis=0))
```

```
cov_matrix = np.cov(centered_data.T)
print(cov_matrix)
```

```
[[0.99537366 0.8026643 ]
 [0.8026643  1.00164892]]
```

Compute the eigenvalues and eigenvectors of the covariance matrix, and sort by the eigenvalues.

```
In [ ]: eig_vals, eig_vecs = np.linalg.eig(cov_matrix)
        idx = eig_vals.argsort()[::-1]
        eigenValues = eig_vals[idx]
        eigenVectors = eig_vecs[:,idx]
        print(eigenValues)
        print(eigenVectors)
```

```
[1.80118173 0.19584086]
[[-0.70572339 -0.70848747]
 [-0.70848747  0.70572339]]
```

Print the eigenvectors. What is the interpretation of them?

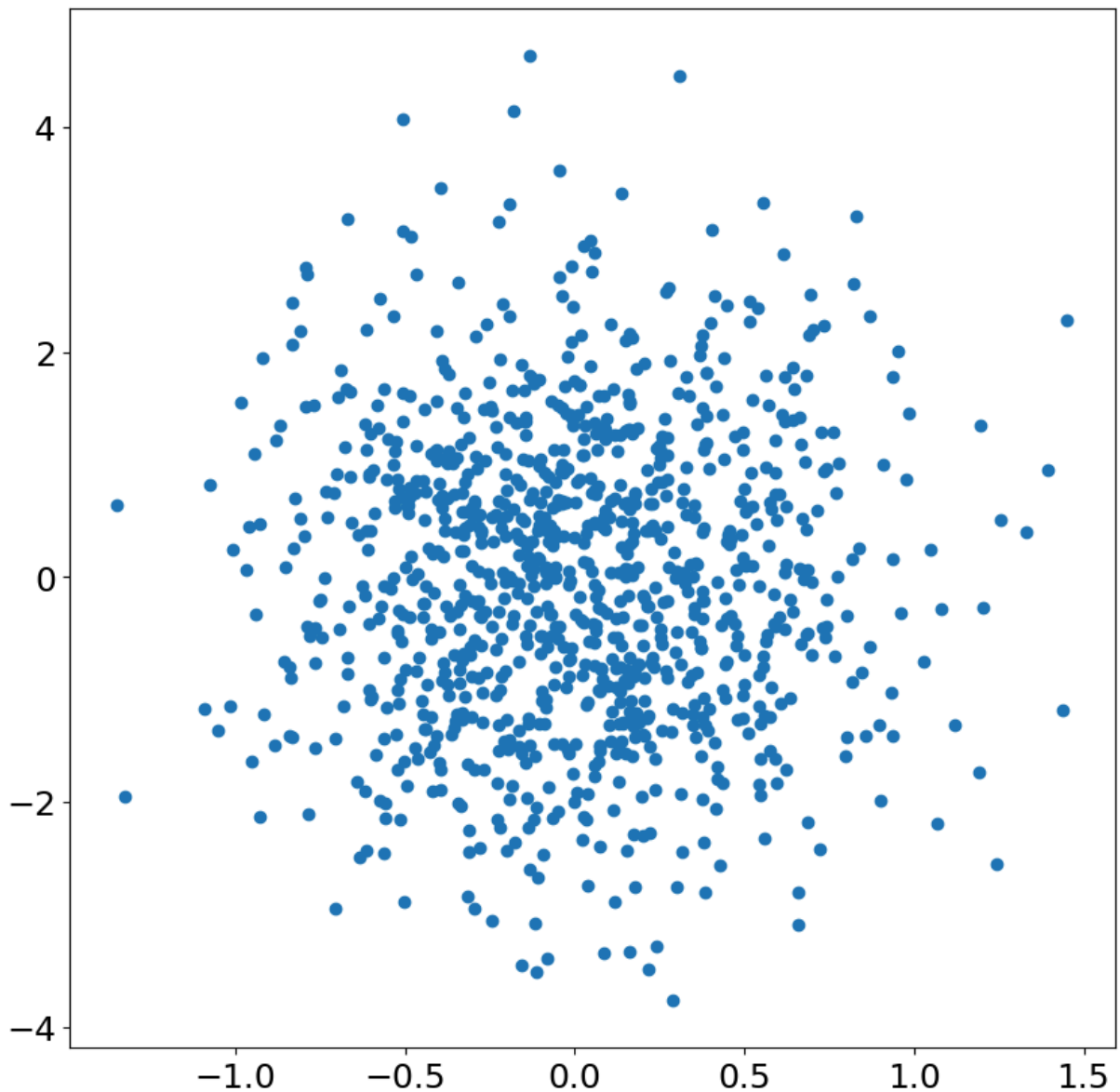
```
In [ ]: print(eigenVectors)
```

```
[[-0.70572339 -0.70848747]
 [-0.70848747  0.70572339]]
```

Transform the data by multiplying the eigenvectors, generate a scatter plot afterwards.

```
In [ ]: transformed_data = np.dot(centered_data, eig_vecs)
        plt.scatter(transformed_data[:,0], transformed_data[:,1])
```

```
Out[ ]: <matplotlib.collections.PathCollection at 0x14ac083e0>
```



Scikit-learn also provide a PCA function. See <https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html>.

Use this function to transform the random data we generate. You can standardize the data with `StandardScaler`.

```
In [ ]: from sklearn.decomposition import PCA
        from sklearn.preprocessing import StandardScaler

        pca = PCA(n_components=2)
        std_scaler = StandardScaler()
        # Write your code below

        scaled_data = std_scaler.fit_transform(org_data)
        pca_result = pca.fit(scaled_data)

        variance = pca_result.explained_variance_ratio_
        print(variance)
```

```
[0.90193249 0.09806751]
```

print `pca.components_` which are the "principle" directions found by PCA, and plot how the transformed data look like. Compare with the previous results we had.

```
In [ ]: print(pca.components_)
```

```
[[-0.70710678 -0.70710678]  
 [-0.70710678  0.70710678]]
```

```
In [ ]: res = np.dot(scaled_data, pca.components_.T)  
  
plt.scatter(res[:,0], res[:,1])  
plt.show()
```

