Batu Arda Düzgün 21802633
EEE-482 HW-4

Question 1:

In this question, we are given Stimuli consisting of face images that have been used in a study on visual perceptions. The face images are down sampled to a 32×32 square grid and there are 1000 of them in the given data. The images are stored in a matrix, with 1000 rows (number of different images) and 1024 columns (number of image pixels).

a) Use The experimenter would like to fit encoding models between the stimuli (i.e., face images) and the measured neural responses. However, one first needs to define explanatory variables (i.e., regressors) that capture important variations in stimulus properties during the experiments. To accomplish this goal, perform PCA on the 1000 faces images. Plot the proportion of variance explained by each individual PC, for the first 100 PCs. Display the first 25 PCs using the function dispImArray.m

In this problem we are asked to apply PCA on the given data. PCA would work better if we subtracted the mean from the data but in this problem the given data already has 0 mean so this step is skipped. Subtracting the mean is skipped in the further parts of this question too. PC of the data set are actually just the eigenvectors of the square of the data matrix. The first PCs are the eigenvectors with the largest eigenvalues. The equation for this is given below.

$$X_{1000\times1024} = data, \qquad v_i = eigenvector, \qquad \lambda_i = eigenvalue$$
$$(X^T X - I\lambda_i)v_i = 0 \tag{1}$$

Then we simply order the eigenvalues from large to small with their eigenvectors and pick the first 25 to find the 25 PCs. I did this computation and all the other computations in this homework using Python 3.8 Python works almost fully on library's and without them it almost doesn't have any functions for the scientific calculations we what we want to do here. Because of this I imported the related packages to do the tasks we are given. I paid attention to not use any library that doesn't already come with the Anaconda install. So, all the libraries in this code already should come with python and they shouldn't require an extra download. I wrote the import part of my code down below. If your python doesn't have the used packages and you have pip, you can install the missing packages with command such as "python -m pip install numpy, python -m pip install scipy or python -m pip install matplotlib."

```
import sys # the primer already has it

import numpy as np # this brings simple matrix operations to python
import matplotlib.pyplot as plt # this brings nice loking plots to python

import scipy.io as sio # this is used to load MATLAB files


from sklearn.decomposition import PCA # PCA function
from sklearn.decomposition import FastICA # function given in the manual
from sklearn.decomposition import NMF # function given in the manual
```

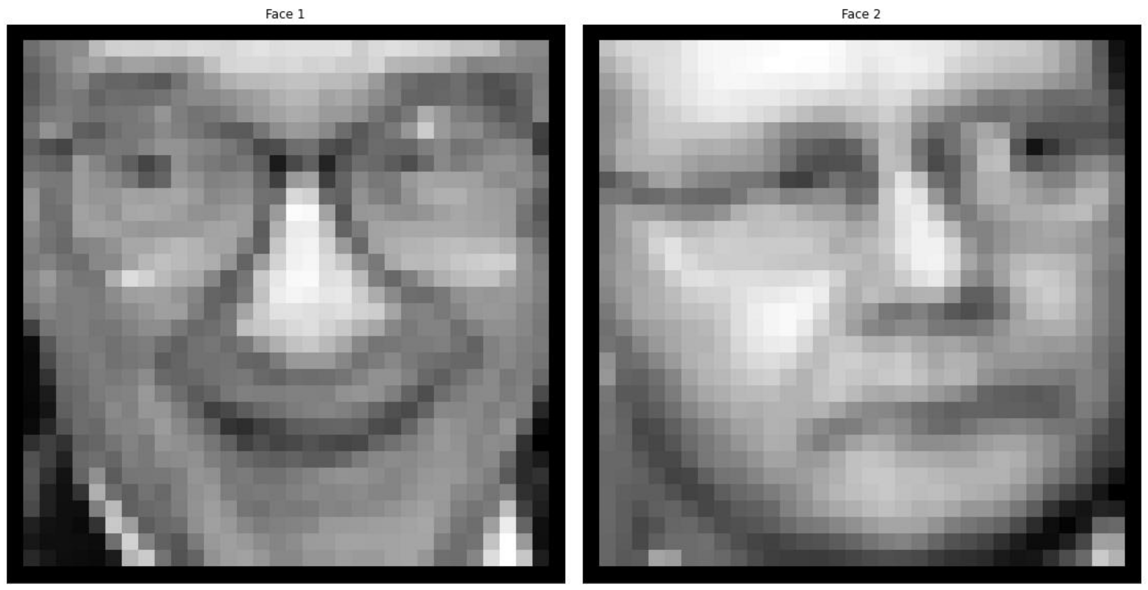
To do any of these I first need to import the data given to us into Python. The data for this homework is given to as MATLAB file. The file, has a too new format for the Pythons functions, which come with a standard Python install. To fix this, I opened the files on my MATLAB and re-saved them as a file format compatible with Python. Then I imported them into Python using the loadmat function. I used the same method and code for all parts of the homework to import the relevant data. The code for this is given below.

```
    data = sio.loadmat('matlab_hw4_data1') # loading the data given to us
    faces = data["faces"]
```

Then I to cheek if the images are loaded correctly, I displayed the first 2 images. The code for this is given below.

```
    my_dispImArray(faces[0:1,:], "Face 1") # to test if the images are loaded
#correctly
    my_dispImArray(faces[1:2,:], "Face 2")
```

This gave the output given below.



These images looked correct. To solve the question Python has a PCA function. I used it to find the first 100 PCs. This function simply does the computation in equation 1 then gives all the relevant results. Then the equation for the calculation of the proportion of explained variance is given below.

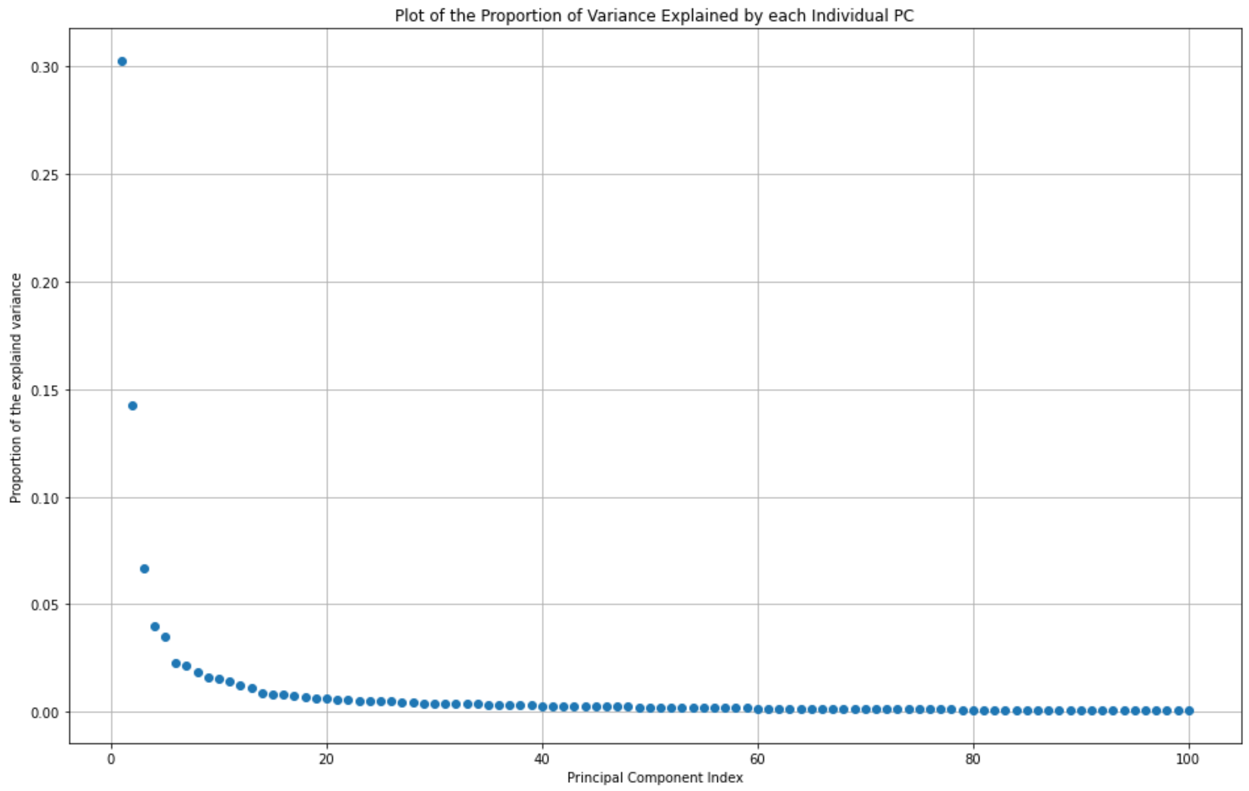
(2)

$$Proportion\ of\ the\ explained\ variance_i = \frac{\lambda_i}{\sum_{n=1}^{1000} \lambda_n}$$

I didn’t do the computation in equation 2 because the PCA function already outputs the proportion of the explained variance. I took does values and plotted them. The code for this is given below.

```
    pca = PCA(100)
    PCA_of_faces = pca.fit(faces) # calculating PCA

    pca_explaind_proprtion_of_var = PCA_of_faces.explained_variance_ratio_ # the
#function already computs this for us

    plt.figure()
    plt.plot(range(1, 101), pca_explaind_proprtion_of_var)
    plt.xlabel('Principal Component Index')
    plt.ylabel('Proportion of the explaind variance')
    plt.title('Plot of the Proportion of Variance Explained by each Individual PC
')
    plt.grid()
```

This gives this plot as its output.



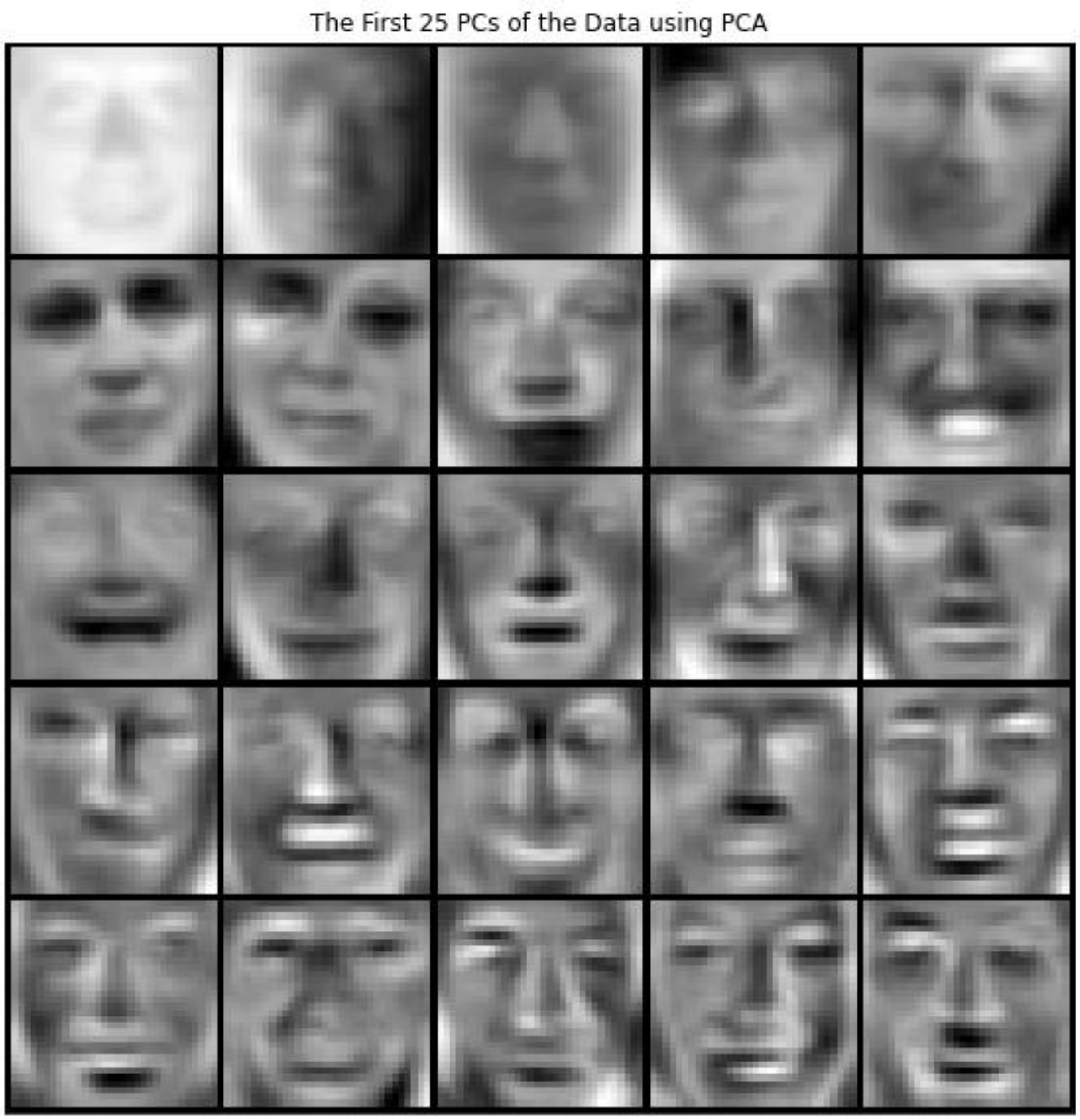
From this plot we can see the PC with the highest explained variance explains around 31% of the variance, the PC with the second highest explained variance explains around 14% and so on. Then we are asked to draw first 25 PCs using the function dispImArray.m, however this is a MATLAB function. So, I created my own dispImArray function in Python by implementing the same function in Python. My functions only differences from the given function are, it allows the displayed image to have title and it gives the paddings 0 instead of -1 while displaying non-negative matrices. This makes the MFs in part d to look clearer. The code for my dispImArray is given below.

```
def my_dispImArray(images, title): # I am not explaing this in detail because it
#is the function which is given with the instructions.

    width = round((np.shape(images)[1]) ** (0.5) )

    mn = np.shape(images) # Compute rows, cols
    m = mn[0]
    n = mn[1]
    height = int(n / width)

    display_rows = int(np.floor(m ** (0.5))) # Compute number of items to display
    display_cols = int(np.ceil(m / display_rows))

    pad = 1 # padding that will be used

    if np.min(images) < 0: #this is to make the non negative matricies look nice.

        display_array = -
np.ones((pad + display_rows * (height + pad), pad + display_cols * (width + pad))
); # we will fill this in with the images

    else:

        display_array = np.zeros((pad + display_rows * (height + pad), pad + disp
lay_cols * (width + pad))); # we will fill this in with the images

    curr_ex = 0

    for j in range(0, display_rows):
        for i in range(0, display_cols):

            if curr_ex == m:

                break
            max_val = max(abs(images[curr_ex, :]))

            display_array[pad + j * (height + pad) : pad + j * (height + pad) + h
eight, pad + i * (width + pad) : pad + i * (width + pad) + width] = images[curr_e
x, :].reshape(height, width).T / max_val

            curr_ex += 1

        if curr_ex == m:

            break

    # Display Image
    plt.figure(figsize=(10, 10))
    plt.imshow(display_array, cmap=plt.cm.gray)
    plt.xticks([])
    plt.yticks([])
    plt.title(title)
```

Then using this function, I display the first 25 PCs. The PCs are already computed and ordered by the PCA function so I only need to display the first 25 of them. The code which does this is given below.

```
    pca_componets = PCA_of_faces.components_ # taking the principal components
#from the function

    my_dispImArray(pca_componets[0:25 , :], "The First 25 PCs of the Data using P
CA") # displaying the first 25 PCs
```

This code gives the output given below.



b) The experimenter would like to know how many PCs are sufficient to obtain a reasonable representation of the stimuli. Reconstruct each image in the matrix f aces using their PC projections (i.e., reconstructed images are a linear weighted combination of stimulus PCs). Obtain separate reconstructions based on first 10, 25, and 50 PCs. Display the original images and the reconstructions using dispImArray.m, for the first 36 images. Find the mean-squared error (MSE) between the original and reconstructed images, and report the mean and standard deviation of MSE across 1000 images. Interpret the results.

For PCA generating a low dimensional representation of the data and then reconstructing images are simple once the PCs are chosen. The computation required for them is given below.

$$X_{1000\times1024} = data, \qquad n = number\ of\ PCs, \qquad P_{1024\times n} = PCs\ of\ the\ data,$$

$$S_{1000\times n} = low\ dimensional\ reprisentation, \qquad \hat{X}_{1000\times1024} = reconstructed\ data$$

$$S_{1000\times n} = X_{1000\times1024} \cdot P_{1024\times n} \tag{3}$$

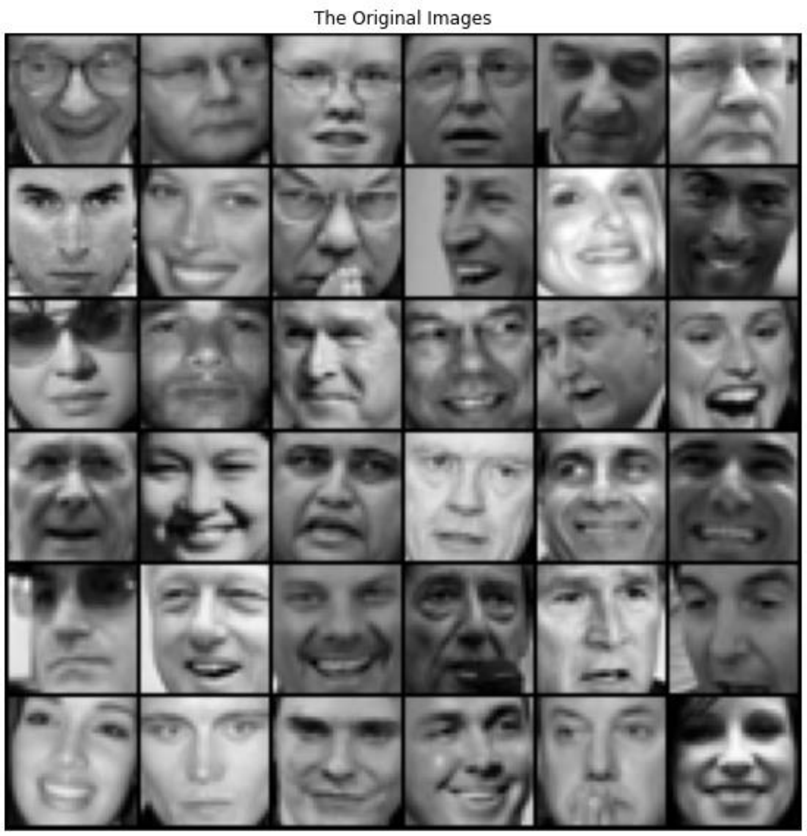
$$\hat{X}_{1000\times1024} = S_{1000\times n} \cdot P^{T}_{n\times1024} \tag{4}$$

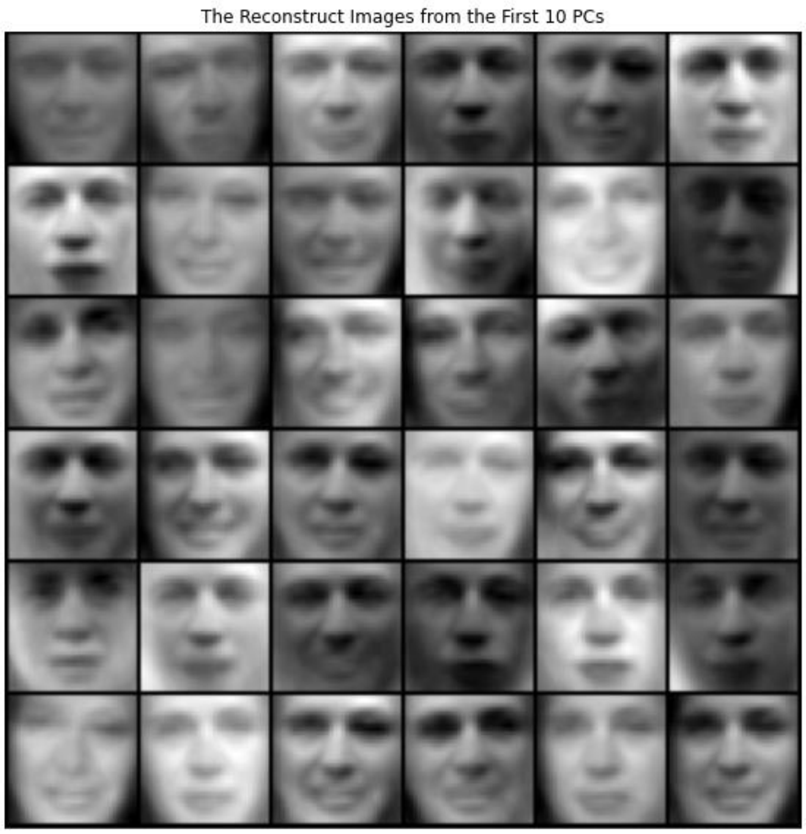
I want every part of my code to be functional without running them in order. Because of this I repeat the PCA function from part a in this part. I display the original faces for the first 36 images. Then I do the computation in equation 3 and 4 for n= 10, 25, 50 and then display the results. The code for this is given below.

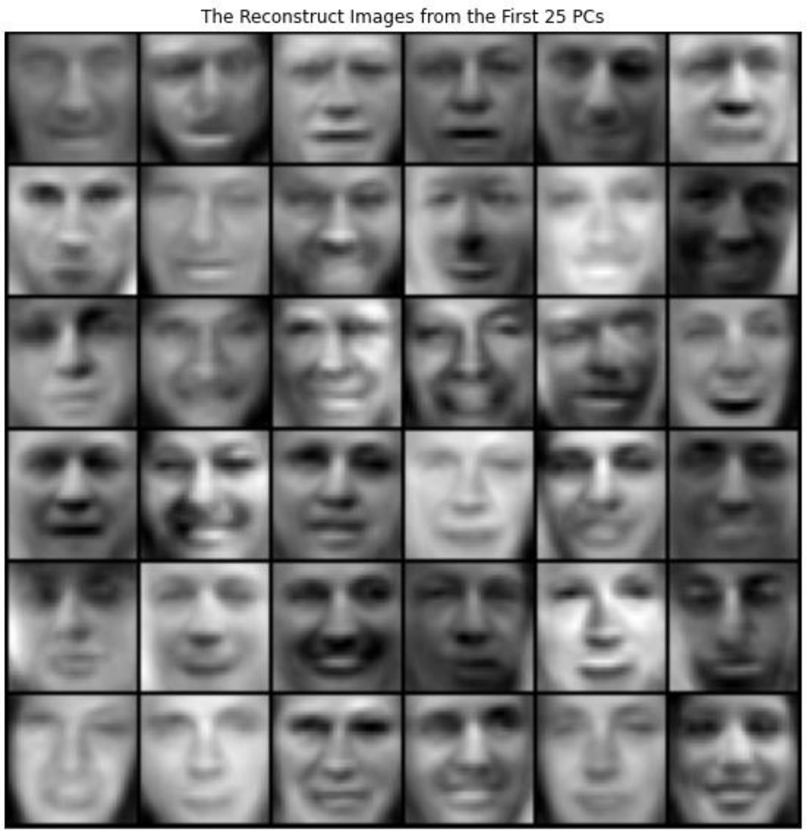
```
    data = sio.loadmat('matlab_hw4_data1') # loading the data given to us
    faces = data["faces"]

    pca = PCA(100)
    PCA_of_faces = pca.fit(faces) # calculating PCA

    pca_componets = PCA_of_faces.components_  # taking the principlal components.
# this part is repeated



    pca_projection_10 = faces.dot(pca_componets[0:10].T) # lower dimensionlar
#reprisentation for 10 PCs

    Reconstructed_faces_10 = pca_projection_10.dot(pca_componets[0:10])
# Reconstructed faces for 10 PCs

    my_dispImArray(faces[0:36], "The Original Images")

    my_dispImArray(Reconstructed_faces_10[0: 36], "The Reconstruct Images from th
e First 10 PCs")


    pca_projection_25 = faces.dot(pca_componets[0:25].T) # lower dimensionlar
#reprisentation for 25 PCs

    Reconstructed_faces_25 = pca_projection_25.dot(pca_componets[0:25])
# Reconstructed faces for 25 PCs
```

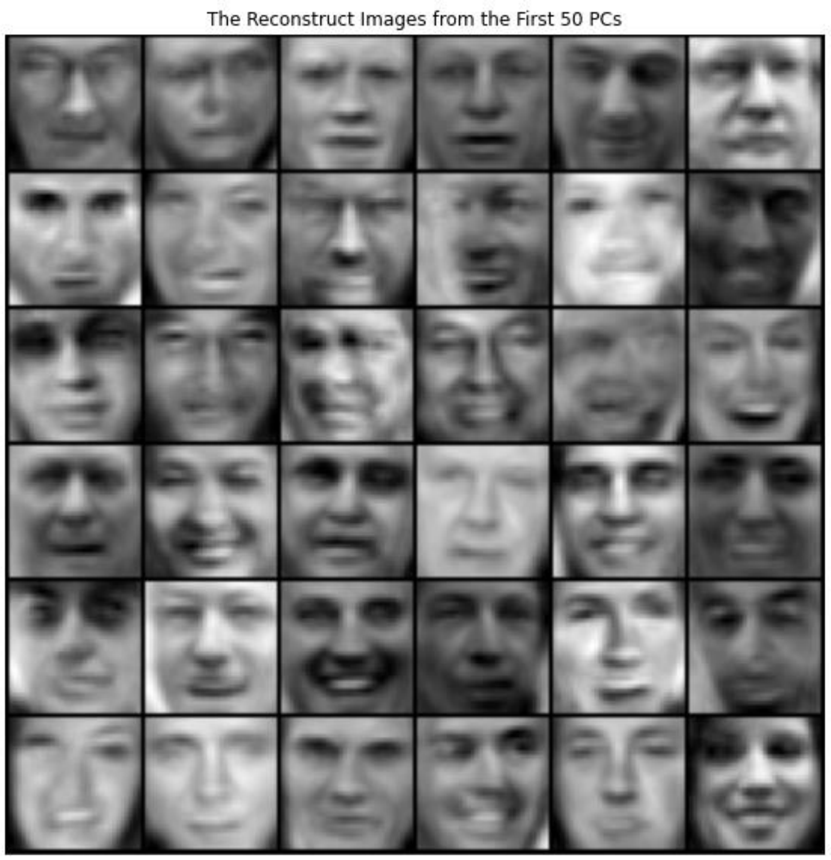
```
    my_dispImArray(Reconstructed_faces_25[0: 36], "The Reconstruct Images from th
e First 25 PCs")


    pca_projection_50 = faces.dot(pca_componets[0:50].T) # lower dimensionlar
#reprisentation for 50 PCs

    Reconstructed_faces_50 = pca_projection_50.dot(pca_componets[0:50])
# Reconstructed faces for 50 PCs

    my_dispImArray(Reconstructed_faces_50[0: 36], "The Reconstruct Images from th
e First 50 PCs")
```

This gives the output images given below.




The Reconstruct Images from the First 10 PCs


The Reconstruct Images from the First 25 PCs



We can see that, as the number of PCs used to reconstruct the images increases, the images look more like the originals. This is not surprising as increasing number of PCs means storing more detailed information in the low dimensional representation and images which are created with more information are detailed and look better. Something which is important to note, this system cannot properly store and reconstruct images which are not directly facing the camera. A few examples to this are the 17th image (third row from top and fifth column) and his faces is not centered on the image and 10th image (second row from top and forth column). They are facing left, PCA tries to reconstruct them while facing the camera which significantly deforms their faces. This happens because most images in the data set given to us is directly facing the camera so the PCs are specialized for faces of this nature and a differently oriented face can not be properly processed. However, this is not a big problem for us because as I said there are very few images which do not face the camera so the problem is only on a select few images.

There is another interesting thing to note. For the lowest number of PCs all the faces look like human faces. Even though generally for 50 PCs the images look better, some of the faces look very deformed. The reason for this is, for 10 PCs there are so few features to work with the all the faces look like a generic human face. This means, even though all the faces look like human faces they do not look like the original given faces.

Then, we are asked to calculate mean squared error for each reconstruction then take the MSE's means and standard deviation. The computation done to compute MSE of an original image and reconstructed image is given below in equation 5.

$$X_{i,\ 1\times 1024} = data,\ \ \hat{X}_{i,\ 1\times 1024} = reconstructed\ data$$
$$MSE_i = \frac{1}{1024}\sum_{j=1}^{1024}(\hat{X}_{ij} - X_{ij})^2 \tag{5}$$

I applied equation 5 to every original, reconstructed image pair for 10 PCs, 25 PCs and 50 PCs. Then computed the mean and STDs of the MSE's. The code which does this is given below.

```
    MSE_10 = (Reconstructed_faces_10 - faces) ** 2 # calculating MSE for 10 PCs

    mean_10 = np.mean( np.mean(MSE_10 , 1) )# calculating the mean of MSE for 10 
#PCs
    STD_10 = np.std( np.mean(MSE_10 , 1) )# calculating the std of MSE for 10 PCs

    print("Mean of MSE for the first 10 PCs: " + str(mean_10)) #displaying the 
#results
    print("STD of MSE for the first 10 PCs: " + str(STD_10) + "\n")


    MSE_25 = (Reconstructed_faces_25 - faces) ** 2 # calculating MSE for 25 PCs

    mean_25 = np.mean( np.mean(MSE_25 , 1) )# calculating the mean of MSE for 25 
#PCs
    STD_25 = np.std( np.mean(MSE_25 , 1) )# calculating the std of MSE for 25 PCs

    print("Mean of MSE for the first 25 PCs: " + str(mean_25)) #displaying the 
#results
    print("STD of MSE for the first 25 PCs: " + str(STD_25) + "\n")


    MSE_50 = (Reconstructed_faces_50 - faces) ** 2 # calculating MSE for 50 PCs

    mean_50 = np.mean( np.mean(MSE_50 , 1) )# calculating the mean of MSE for 50 
#PCs
    STD_50 = np.std( np.mean(MSE_50 , 1) )# calculating the std of MSE for 50 PCs

    print("Mean of MSE for the first 50 PCs: " + str(mean_50)) #displaying the 
#results
    print("STD of MSE for the first 50 PCs: " + str(STD_50) + "\n")
```

This gives the output of:

Mean of MSE for the first 10 PCs: 552.3352072837199

STD of MSE for the first 10 PCs: 264.2382972670229

Mean of MSE for the first 25 PCs: 345.5693403426197

STD of MSE for the first 25 PCs: 156.3617196136948

Mean of MSE for the first 50 PCs: 203.62448227318143

STD of MSE for the first 50 PCs: 84.9506963340033

This output is something which is expected. From directly looking at the reconstructed images we could see images reconstructed from the more PCs looked a lot closer to the original images. And as expected the both the mean and STD of the mean squared errors are smaller for images which look more like their original.

c) Instead of PCA, find explanatory variables to capture stimulus properties using independent component analysis (ICA). Use the FastICA package available on Moodle, and use PCA-based reduction to 50 dimensions during the calls (i.e., by setting lastEig). Note that unlike PCA, ICA results are not deterministic. Recall fastica.m to return 10 ICs, 25 ICs, and 50 ICs. Display the obtained ICs using dispImArray.m. Reconstruct face images based on their IC projections. Display the original and reconstructed images based on 10, 25, and 50 ICs. Report the mean and std of MSE in each case. Compare your results with part b.

As I said, I did this homework on Python, because of this I used the FastICA function of Python. The functions are very similar in nature.

ICA is easier to explain with PCA. PCA simply tries to find the directions with maximum variance, while ICA finds the ICs which are independent while best describing the data given. Finding ICs is equivalent to finding direction of maximum non-Gaussianity. Gaussianity can be defined in many ways to give some examples. Kurtosis is the classical measure of non-Gaussianity or the entropy of a variable can be seen as its Gaussianity. These are all very hard to compute. Luckily Python has an ICA function and I used it to find the ICs. The result of the ICA function will give us the A and S matrix defined below. The result will change every time the function is run because ICA results are not deterministic.

$$\hat{X}_{1000\times1024} = Reconstructed\ data, \quad S_{n\times1024} = Sources\text{: } ICs\ of\ the\ data,$$

$$A = Mixing\ matrix\text{: } low\ dimensional\ reprisentation, \quad n = number\ of\ ICs$$

$$\hat{X}_{1000\times1024} = A_{1000\times n} \cdot S_{n\times1024} \tag{6}$$

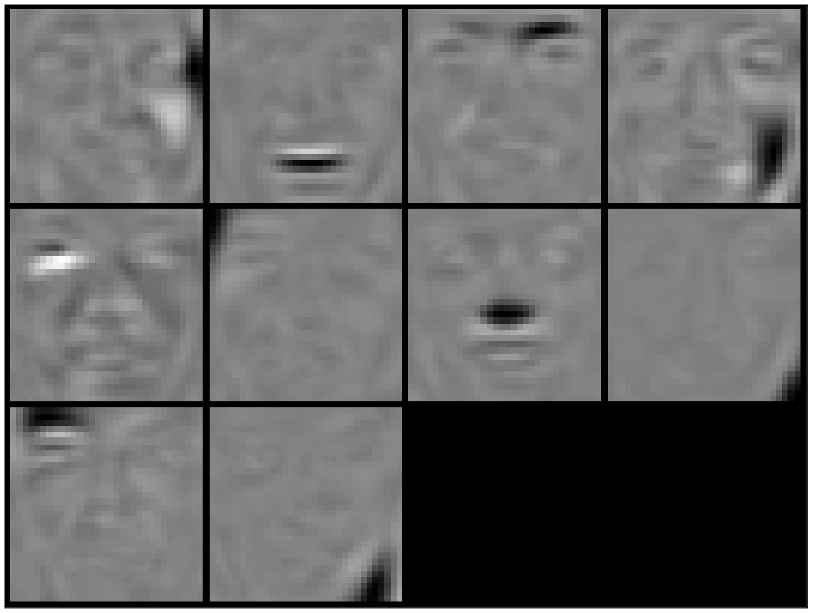
Something important to note is the independent components are independent in the probabilistic sense not it the sense that their dot product is 0.

The code which finds the ICs using the FastICA function for lastEig equal to 50 for all 3 cases is given below.

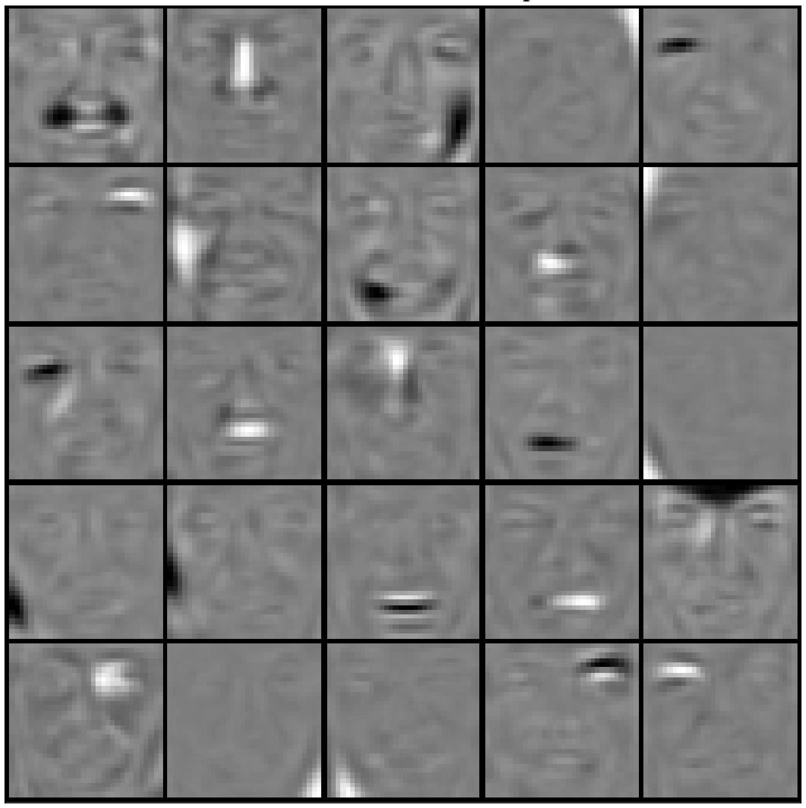
```
    data = sio.loadmat('matlab_hw4_data1') # loading the data given to us
    faces = data["faces"]


    ICA_model_10 = FastICA(50, max_iter=10000) # data used for the transformation
#, it seemslike the only number which we enter is the lasteig
    ICA_of_faces_10 = ICA_model_10.fit(faces.T) # data used for the
#transformation fit to our face data


    components_10 = ICA_model_10.fit(faces.T).transform(faces.T)[:,0:10]
# indipendent components of the model. we pick the first 10

    my_dispImArray(components_10.T, "The First 10 ICs of the Data using ICA")  #
displaying the ICs


    ICA_model_25 = FastICA(50, max_iter=10000) # data used for the transformation
    ICA_of_faces_25 = ICA_model_25.fit(faces.T) # data used for the
#transformation fit to our face data


    components_25 = ICA_model_25.fit(faces.T).transform(faces.T)[:,0:25]
# indipendent components of the model.

    my_dispImArray(components_25.T, "The First 25 ICs of the Data using ICA")
# displaying the ICs


    ICA_model_50 = FastICA(50, max_iter=10000) # data used for the transformation
    ICA_of_faces_50 = ICA_model_50.fit(faces.T) # data used for the
#transformation fit to our face data

    components_50 = ICA_model_50.fit(faces.T).transform(faces.T)[:,0:50]
# indipendent components of the model.

    my_dispImArray(components_50.T, "The First 50 ICs of the Data using ICA")
# displaying the ICs
```

The code is quite simple once the mode is fit for our face data, then the transform function gives us the sources matrix which is the ICs. Then for different number of IC cases we pick that many first ICs. Doing this we find the ICs for 10 ICs, 25 ICs and 50 ICs cases. Then I just display them using my_dispImArray function. The outputs of this code are given below.

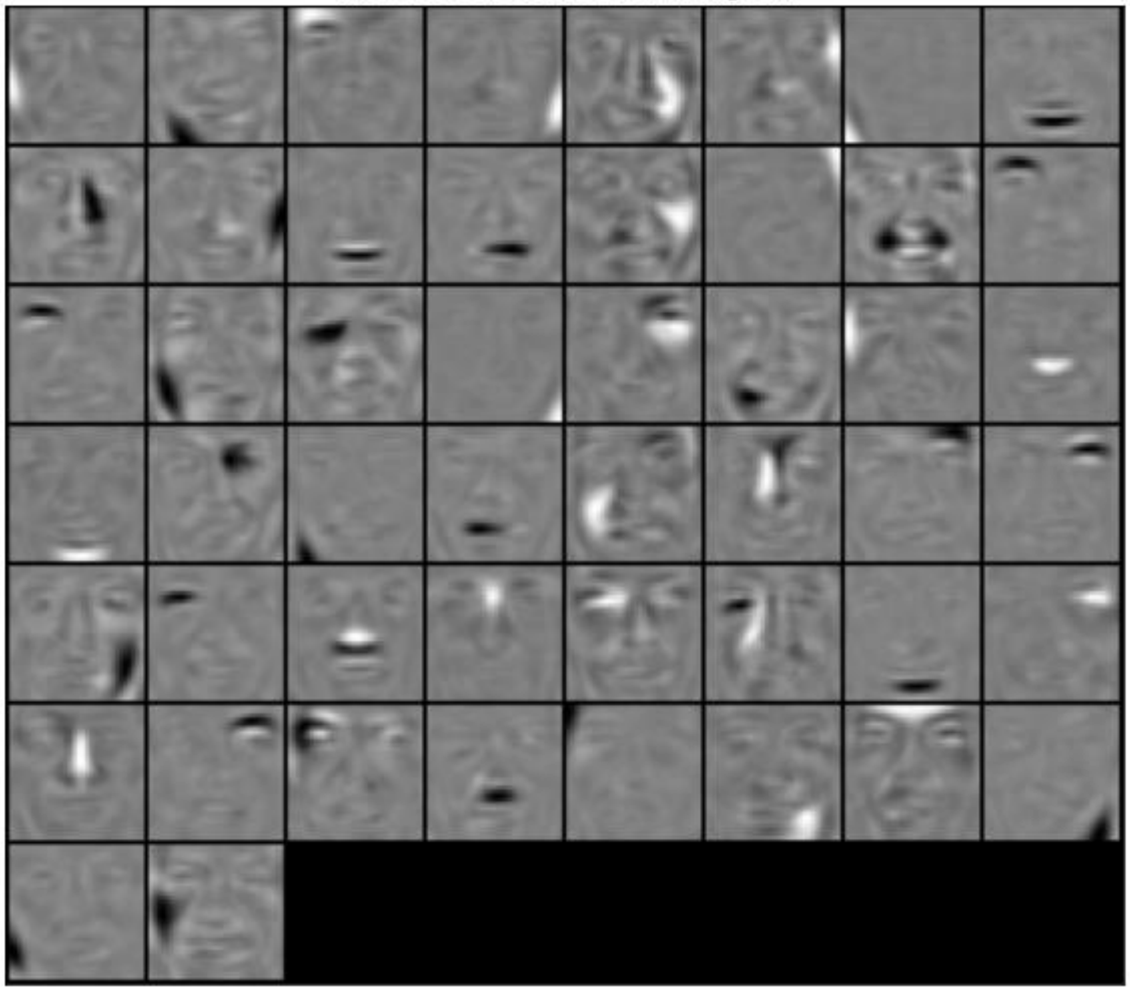
The First 10 ICs of the Data using ICA



The First 25 ICs of the Data using ICA



The First 50 ICs of the Data using ICA



Something to note from these images is, ICs have sign ambiguity and it is easy to see it from these ICs as they are mostly composed of a black or white patch. They are randomly white or black because of the sign ambiguity.

After this I need to reconstruct the faces based on their IC projections. The mixing_ function on a model which is fit to our data gives us the mixing matrix which is equal to IC projections. So, it is easy to find the projections. However, we cannot simply use equation 6 to reconstruct the faces. The FastICA function de-means all the samples separately and we need to add the subtracted mean back to the samples. The equation used for the reconstruction is given below.

$$\hat{X}_{1000\times1024} = Reconstructed\ data, \quad S_{n\times1024} = Sources\text{: } ICs\ of\ the\ data, \quad M_{1000} = means$$

$$A = Mixing\ matrix\text{: } low\ dimensional\ reprisentation, \quad n = number\ of\ ICs$$

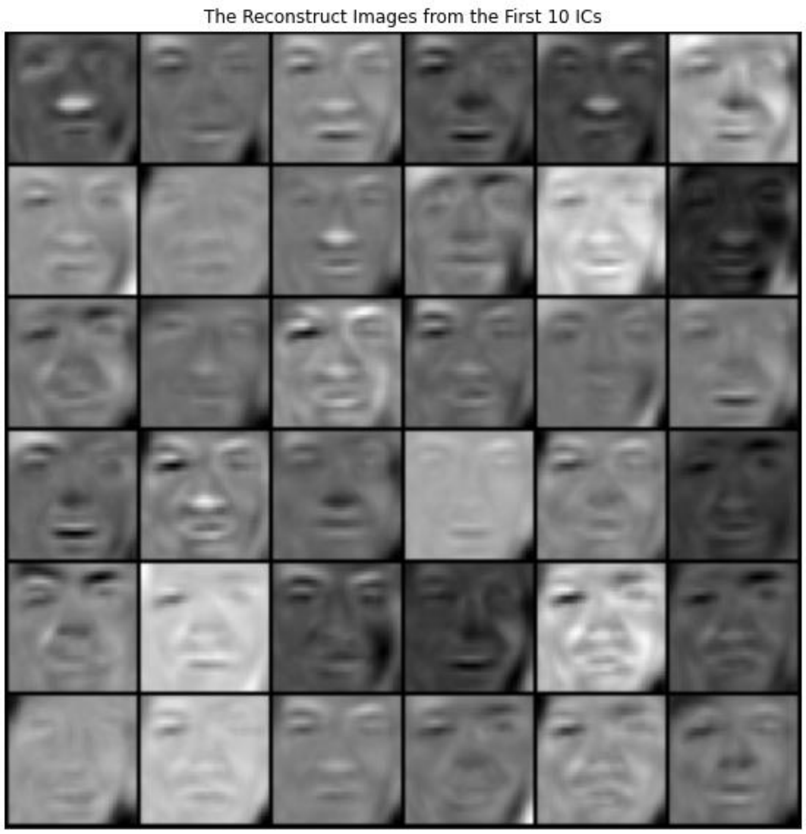
$$\hat{X}_{1000\times1024} = A_{1000\times n} \cdot S_{n\times1024} + M \tag{7}$$

Now, I display the original faces for the first 36 images. Then I do the computation in equation 7 for n= 10, 25, 50 and then display the results. The means are accessed from the function once again but they can be easily computed with a mean function on the faces data. The code for this is given below.

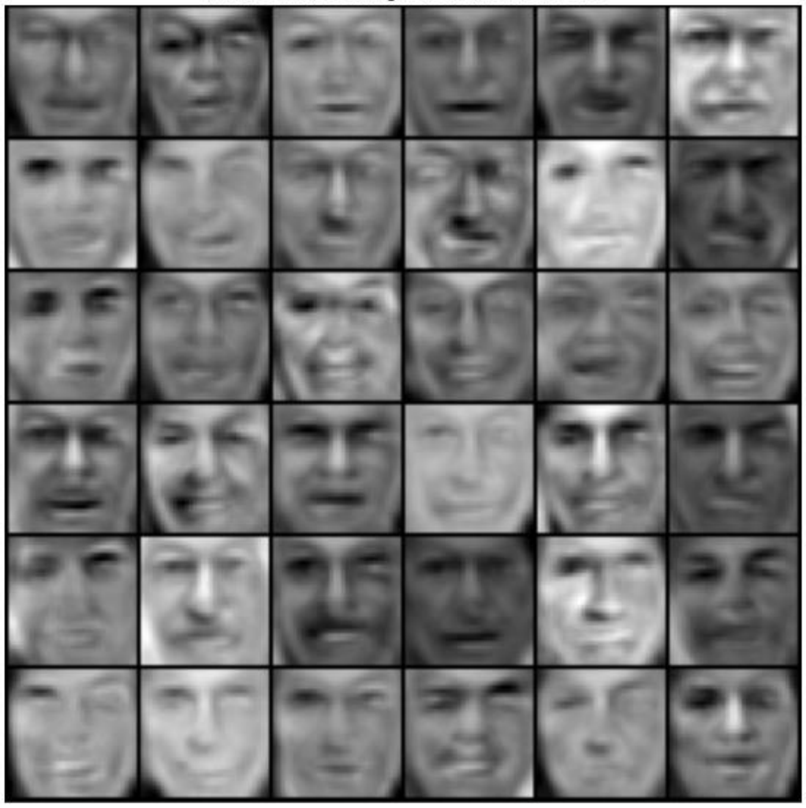
```
    data = sio.loadmat('matlab_hw4_data1') # loading the data given to us
    faces = data["faces"]


    ICA_model_10 = FastICA(50, max_iter=10000) # data used for the transformation
#, it seemslike the only number which we enter is the lasteig
    ICA_of_faces_10 = ICA_model_10.fit(faces.T) # data used for the
#transformation fit to our face data


    components_10 = ICA_model_10.fit(faces.T).transform(faces.T)[:,0:10]
# indipendent components of the model. we pick the first 10

    my_dispImArray(components_10.T, "The First 10 ICs of the Data using ICA")
# displaying the ICs


    ICA_model_25 = FastICA(50, max_iter=10000) # data used for the transformation
    ICA_of_faces_25 = ICA_model_25.fit(faces.T) # data used for the
#transformation fit to our face data


    components_25 = ICA_model_25.fit(faces.T).transform(faces.T)[:,0:25]
# indipendent components of the model.

    my_dispImArray(components_25.T, "The First 25 ICs of the Data using ICA")
# displaying the ICs


    ICA_model_50 = FastICA(50, max_iter=10000) # data used for the transformation
    ICA_of_faces_50 = ICA_model_50.fit(faces.T) # data used for the
#transformation fit to our face data

    components_50 = ICA_model_50.fit(faces.T).transform(faces.T)[:,0:50]
# indipendent components of the model.

    my_dispImArray(components_50.T, "The First 50 ICs of the Data using ICA")
# displaying the ICs
```

The output of this code is given below.

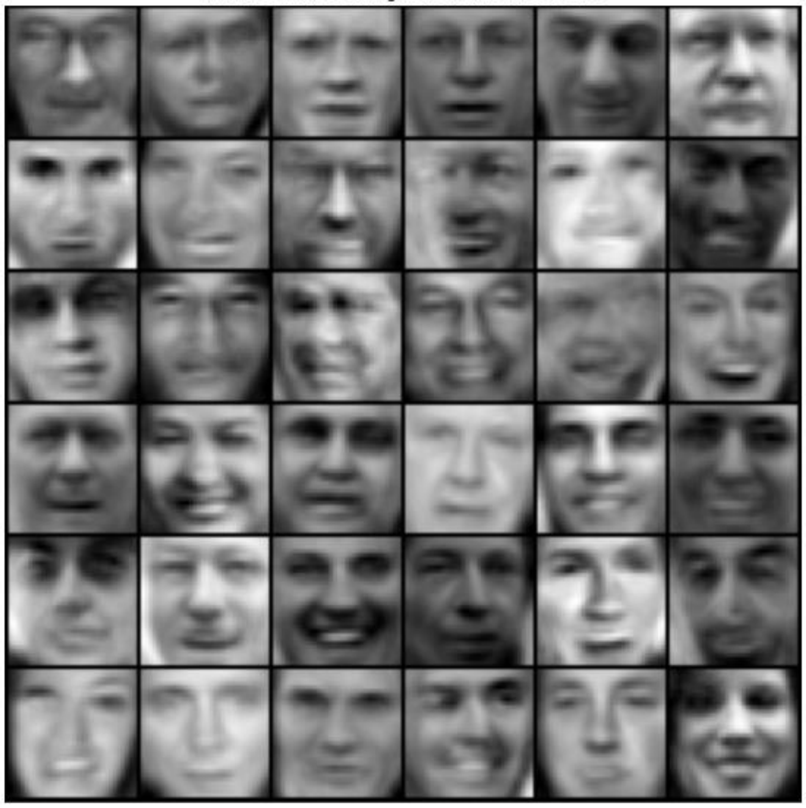
The Original Images


The Reconstruct Images from the First 10 ICs

The Reconstruct Images from the First 25 ICs



The Reconstruct Images from the First 50 ICs



Again, we can see that, as the number of ICs used to reconstruct the images increases, the images look more like the originals. This is not surprising as increasing number of ICs means storing more detailed information in the low dimensional representation and images which are created with more information are detailed and look better. However, for this one the 10 ICs case and the 25 ICs case looks significantly worse then the PCA case. The reason for this is we are setting the lastEig to 50 in those cases while we are only using the first 10 and 25 ICs. I tested it and the reconstructions look like the PCA case if the models are fit for lastEig equals to 10 and 25 respectively. I didn't add these tests to not make the report too long.

Something which is important to note, this system cannot properly store and reconstruct images which are not directly facing the camera. We are seeing this problem again because it is an issue for all dimensionality reductions. A few examples to this problem are again the 17th image (third row from top and fifth column) and his faces is not centered on the image and 10th image (second row from top and forth column). They are facing left; ICA tries to reconstruct them while facing the camera which significantly deforms their faces. This happens because most images in the data set given to us is directly facing the camera so the ICs are specialized for faces of this nature and a differently oriented face cannot be properly processed. However, this is not a big problem for us because as I said there are very few images which do not face the camera so the problem is only on a select few images.

Then, we are asked to calculate mean squared error for each reconstruction then take the MSE's means and standard deviation. The computation done to compute MSE of an original image and reconstructed image is given below in equation 8.

$$X_{i,\ 1\times1024} = data,\ \ \hat{X}_{i,\ 1\times1024} = reconstructed\ data$$

$$MSE_i = \frac{1}{1024}\sum_{j=1}^{1024}(\hat{X}_{ij} - X_{ij})^2 \tag{8}$$

I applied equation 8 to every original, reconstructed image pair for 10 ICs, 25 ICs and 50 ICs. Then computed the mean and STDs of the MSE's. The code which does this is given below.

```
    MSE_10 = (Reconstructed_faces_10 - faces) ** 2 # calculating MSE

    mean_10 = np.mean( np.mean(MSE_10 , 1) )# calculating the mean
    STD_10 = np.std( np.mean(MSE_10 , 1) )# calculating the std

    print("Mean of MSE for the first 10 ICs: " + str(mean_10)) #displaying the
#results
    print("STD of MSE for the first 10 ICs: " + str(STD_10) + "\n")


    MSE_25 = (Reconstructed_faces_25 - faces) ** 2 # calculating MSE
    mean_25 = np.mean( np.mean(MSE_25 , 1) )# calculating the mean
    STD_25 = np.std( np.mean(MSE_25 , 1) )# calculating the std

    print("Mean of MSE for the first 25 ICs: " + str(mean_25)) #displaying the
#results
    print("STD of MSE for the first 25 ICs: " + str(STD_25) + "\n")


    MSE_50 = (Reconstructed_faces_50 - faces) ** 2 # calculating MSE

    mean_50 = np.mean( np.mean(MSE_50 , 1) )# calculating the mean
    STD_50 = np.std( np.mean(MSE_50 , 1) )# calculating the std

    print("Mean of MSE for the first 50 ICs: " + str(mean_50)) #displaying the
#results
    print("STD of MSE for the first 50 ICs: " + str(STD_50) + "\n")
```

The results of this code are given below.

```
Mean of MSE for the first 10 ICs: 1136.6292411675095

STD of MSE for the first 10 ICs: 580.7977933304011

Mean of MSE for the first 25 ICs: 786.4329421085231

STD of MSE for the first 25 ICs: 400.87015336357774

Mean of MSE for the first 50 ICs: 195.49954475862725

STD of MSE for the first 50 ICs: 82.73754174951138
```

This output is expected. From directly looking at the reconstructed images we could see images reconstructed from 10 ICs look worse than those created with 10 PCs, images reconstructed from 25 ICs look worse than those created with 25 PCs and images reconstructed from 50 ICs look as good as those created with 50 PCs. If we look at the errors closely, we can see the error on the ICA is slightly lower than those PCA for the 50 component cases. As I said before the very large errors compared to the PCA, on the 10 IC and the 25 IC case is because we are setting their lastEig as 50. I tested it and the error on reconstructions are slightly less then PCA case for the 10 ICs and 25 ICs if the models are fit for lastEig equals to 10 and 25 respectively. I didn't add these tests to not make the report too long. Also while comparing the 2 methods we shouldn't forget PCA is computationally cheaper.

d) Finally, find explanatory variables to capture stimulus properties using non-negative matrix factorization (NNMF). NNMF requires its input to be strictly positive, so add a single scalar constant to all entries of the matrix f aces to satisfy this constraint (but only add the minimum amount required). Recall nnmf.m to return 10, 25, and 50 MFs. Display the obtained MFs using dispImArray.m. Reconstruct face images based on their MF projections. Display the original and reconstructed images based on 10, 25, and 50 MFs. Report the mean and std of MSE in each case. Compare your results with parts b and c

As I said, I did this homework on Python, because of this I used the NMF function of Python. The function is very similar to its MATLAB version.

NNMF is easy to explain it factors the input matrix as it is shown in equation 9 and keeps both W and H matrices non-negative. This might seem like unnecessary work but this decomposition into non-negative matrices makes the faces sum of other pieces. Think of it as picking an eye shape, a nose shape, and a mouth shape from a list of face component and adding them up to define a face. This definition of sum of component is more similar to the actual process which is going inside the brain.

$$\hat{X}_{1000\times1024} = Reconstructed\ data,\quad H_{n\times1024} = MFs\ of\ the\ data,$$
$$W_{1000\times n} = low\ dimensional\ reprisentation,\quad n = number\ of\ MFs$$
$$\hat{X}_{1000\times1024} = W_{1000\times n} \cdot H_{n\times1024} \tag{9}$$

To find the W and H matrices, NNMF tries to minimize the mean squared error function while keep W, H non-negative. This is defined more mathematically below.

$$\underset{W,H}{\operatorname{argmin}} MSE = \|X - WH\|^2, such\ that\ W, H \geq 0 \tag{10}$$

To solve equation 10 one possible method is an update function which we repeat until convergence. The update function is given below.

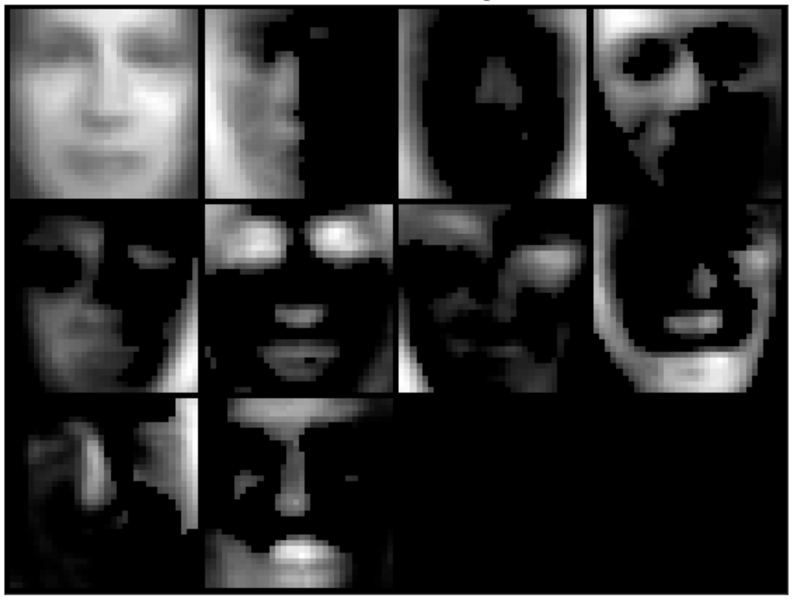
$$H_{au} \leftarrow H_{au} \sum_i W_{ia} \frac{X_{iu}}{(WH)_{iu}},\qquad W_{ia} \leftarrow W_{ia} \sum_u \frac{X_{iu}}{(WH)_{iu}} W_{au} \tag{11}$$

Luckily, I do not need to use equation 11 by hand because Python already has a NNMF function. I found decomposition for 10 MFs, 25 MFs and 50 MFs cases separately. I need to do them separately because due to the nature of NNMF decomposition for different numbers of MFs the decomposition needs to be redone. Also, if any new data is introduced the Decomposition needs to be re-done again. ICA was like this too; it needed to be re-run for new data. This aspect of these methods didn't affect us in this homework but they still exist. Also, as the questions say NNMF requires a non-negative matrix. I added the smallest possible value that makes the data matrix non-negative to make the data matrix non-negative. The code which uses NMF function to find MFs for the 3 cases and then displays them is given below.

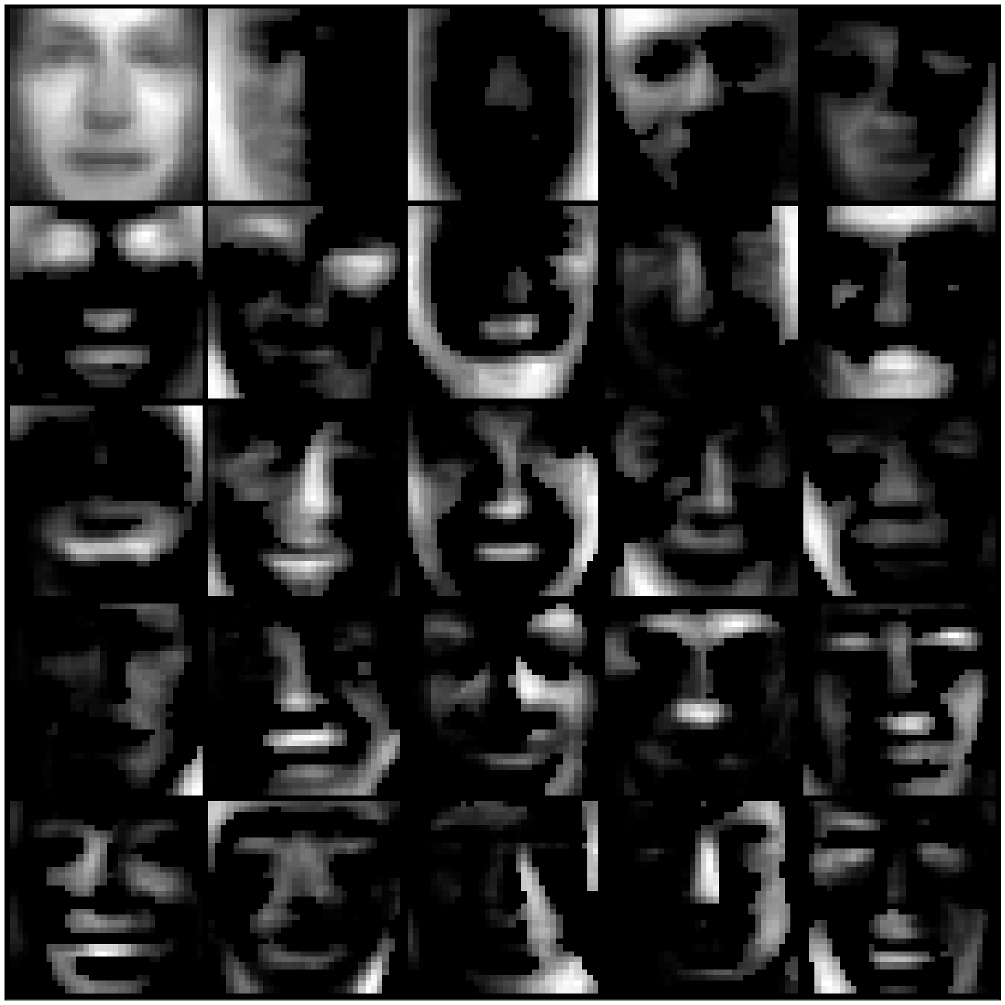
```
    data = sio.loadmat('matlab_hw4_data1') # loading the data given to us
    faces = data["faces"]

    min_number_to_add = np.abs(np.min(faces)) # the most negative number in the
#faces matrix
    positive_faces = faces + min_number_to_add # prepairing for the NNMF

    nnfm_model_10 = NMF(n_components=10,solver="mu", max_iter=1000) # model
#creation

    W_10 = nnfm_model_10.fit(positive_faces).transform(positive_faces) # these
#are the down samples faces.
    H_10 = nnfm_model_10.components_ # these are the MFs

    my_dispImArray(H_10, "The 10 MFs of the Data using NNMF") # displaying the
#10 MFs


    nnfm_model_25 = NMF(n_components=25,solver="mu", max_iter=1000) # model
#creation

    W_25 = nnfm_model_25.fit(positive_faces).transform(positive_faces) # these
#are the down samples faces.
    H_25 = nnfm_model_25.components_ # these are the MFs

    my_dispImArray(H_25, "The 25 MFs of the Data using NNMF") # displaying the
#25 MFs


    nnfm_model_50 = NMF(n_components=50, solver="mu", max_iter=1000) # model
#creation

    W_50 = nnfm_model_50.fit(positive_faces).transform(positive_faces) # these
#are the down samples faces.
    H_50 = nnfm_model_50.components_ # these are the MFs

    my_dispImArray(H_50, "The 50 MFs of the Data using NNMF") # displaying the
#50 MFs
```

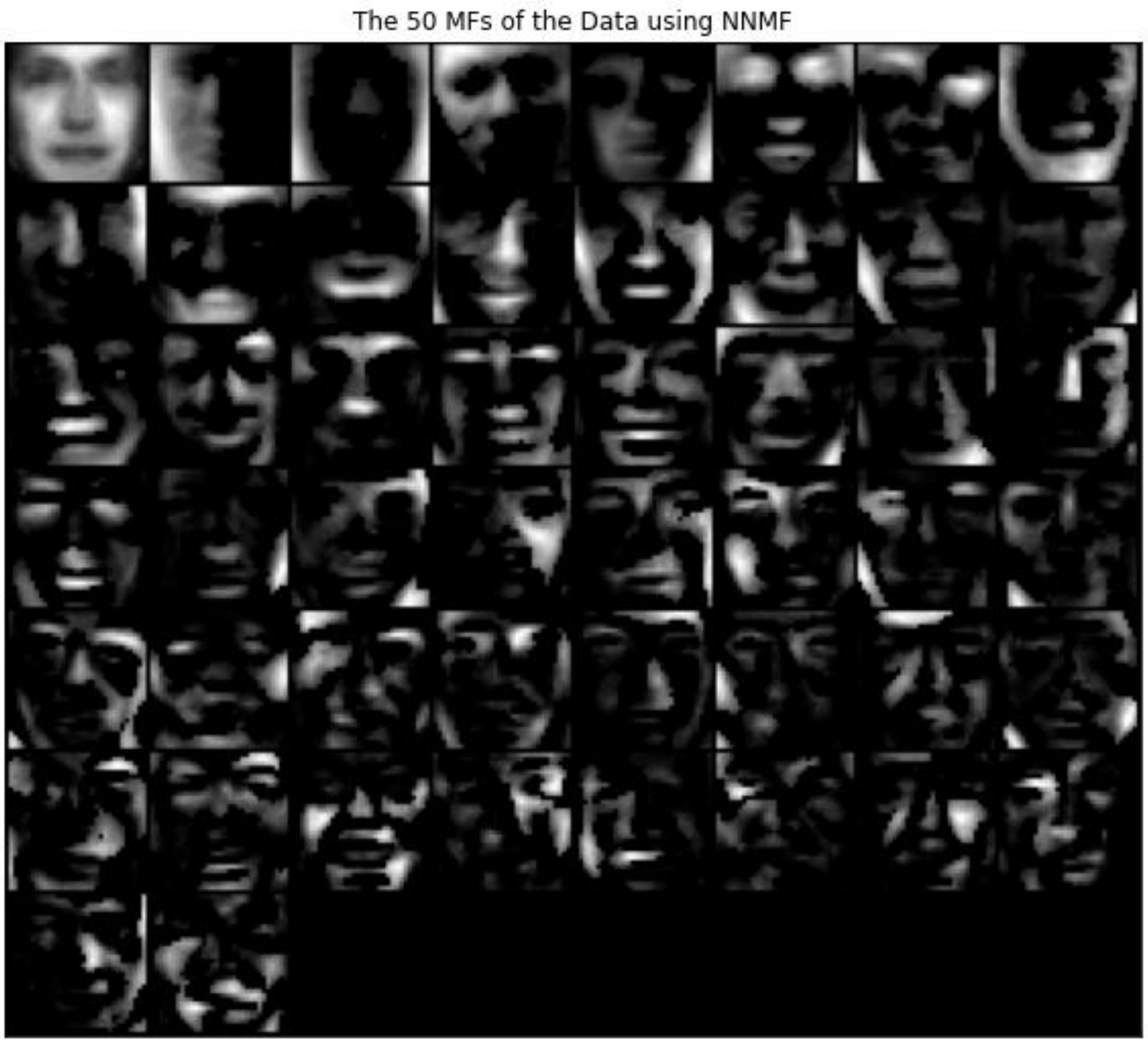
The Output of this code is given below. The output will be slightly different every time this code is run because the NMF function is not deterministic. My output should look a bit different than anyone who used the given dispImArray function because of my modification.

The 10 MFs of the Data using NNMF



The 25 MFs of the Data using NNMF





The some of different face part effect can be seen in the 50 MFs case. Same MFs in them are mostly a nose (12. face) or a chin (8. face) or eyes (6. face). This effect will also be visible in the reconstruction.

After this I need to reconstruct the faces based on their MF projections. The NMF function already gives W and H matrix as its output. So, it is easy to find the projections, I will just use equation 9. Then I subtracted the number I added in the begging to re-shift the faces.

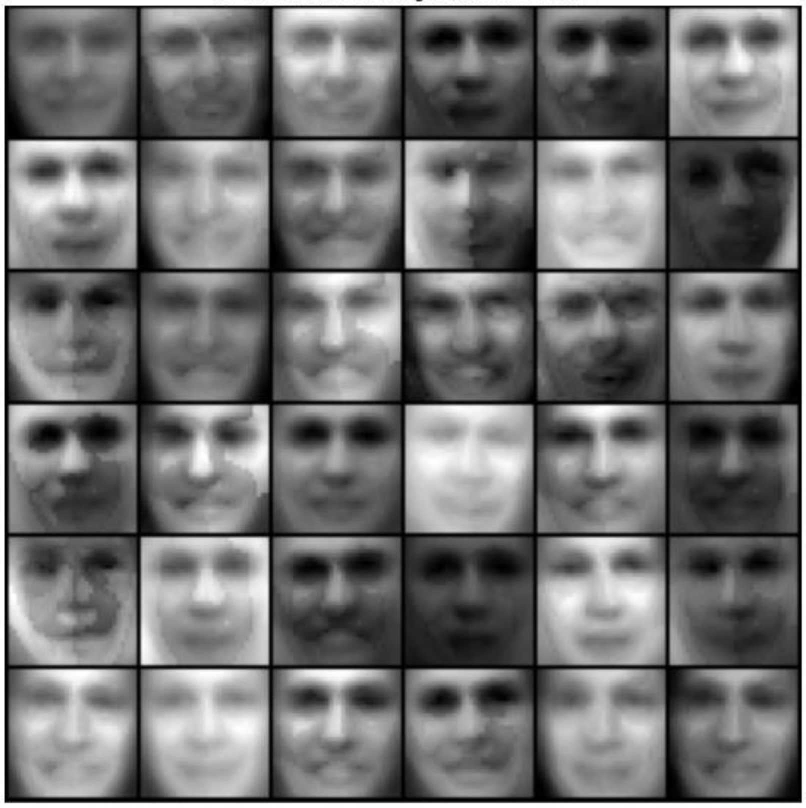
Now, I displayed the original faces for the first 36 images. Then I did the computation in equation 9 for n= 10, 25, 50 then subtracted the number I added in the begging and displayed the results. The code for this is given below.

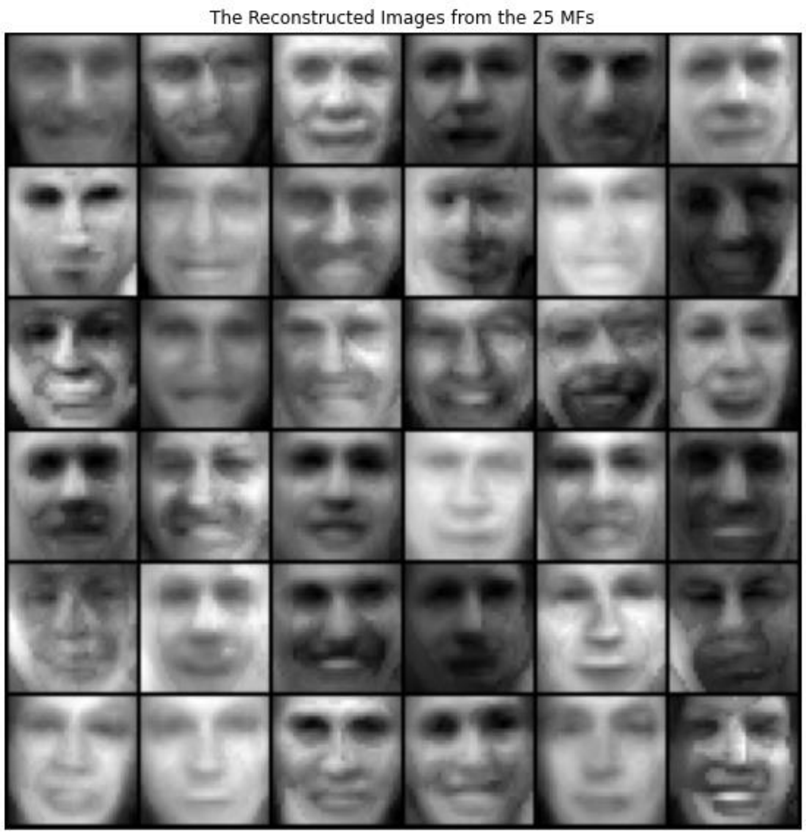
```
    my_dispImArray(faces[0:36], "The Original Images") # displaying the original 
#faces


    Reconstructed_faces_10 = W_10.dot(H_10) - min_number_to_add# reconstructing 
#the original faces

    my_dispImArray(Reconstructed_faces_10[0:36, :], "The Reconstructed Images fro
m the 10 MFs") # displaying the reconstranced images.



    Reconstructed_faces_25 = W_25.dot(H_25) - min_number_to_add# reconstructing 
#the original faces

    my_dispImArray(Reconstructed_faces_25[0:36, :], "The Reconstructed Images fro
m the 25 MFs") # displaying the reconstranced images.



    Reconstructed_faces_50 = W_50.dot(H_50) - min_number_to_add# reconstructing 
#the original faces

    my_dispImArray(Reconstructed_faces_50[0:36, :], "The Reconstructed Images fro
m the 50 MFs") # displaying the reconstranced images.
```

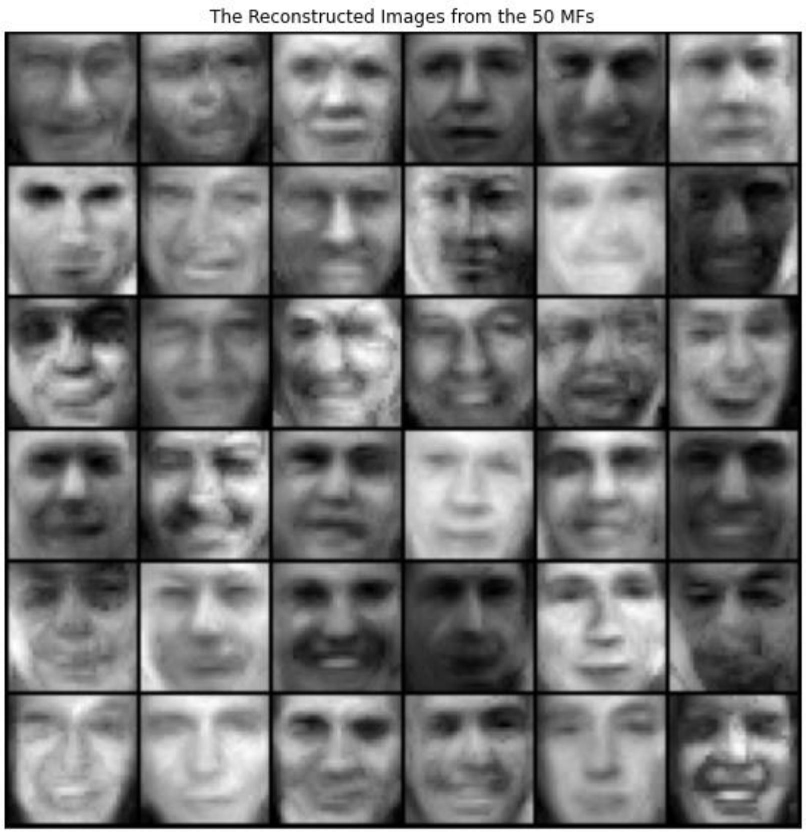
the output of this code is given below.

The Original Images

The Reconstructed Images from the 10 MFs




The Reconstructed Images from the 25 MFs


The Reconstructed Images from the 50 MFs

Again, we can see that, as the number of MFs used to reconstruct the images increases, the images look more like the originals. This is not surprising as increasing number of MFs means giving more options for the face's reconstruction. A face made from more option is obviously better. However, all the recontractions look worse then the PCA case. This is expected as this decomposition has an additional constraint of only using non-negative matrices. The 10 and 25 component cases look better then the ICA ones but the reason for that is the lastEig of the ICA cases and the ICA is better in the 50 component case.

Again, the problem with faces which are not oriented towards the camera can be seen. A few examples to this problem are again the 17th image (third row from top and fifth column) and his faces is not centered on the image and 10th image (second row from top and forth column).

The composing faces of pieces of faces effect is easy to see on the 50 MFs faces. Some faces look like they are made out of pieces from other faces, like the Frankenstein's monster. This doesn't look great in this example but as I said this is closer to the brains inner working.

Then, we are asked to calculate mean squared error for each reconstruction then take the MSE's means and standard deviation. The computation done to compute MSE of an original image and reconstructed image is given below in equation 12.

$$X_{i,\ 1\times1024} = data,\ \ \hat{X}_{i,\ 1\times1024} = reconstructed\ data$$

$$MSE_i = \frac{1}{1024}\sum_{j=1}^{1024}(\hat{X}_{ij} - X_{ij})^2 \tag{12}$$

I applied equation 12 to every original, reconstructed image pair for 10 MFs, 25 MFs and 50 MFs. Then computed the mean and STDs of the MSE's. The code which does this is given below.

```
    MSE_10 = (Reconstructed_faces_10 - faces) ** 2 # calculating MSE

    mean_10 = np.mean( np.mean(MSE_10 , 1) )# calculating the mean
    STD_10 = np.std( np.mean(MSE_10 , 1) )# calculating the std

    print("Mean of MSE for the first 10 PCs: " + str(mean_10)) #displaying the re
sults
    print("STD of MSE for the first 10 PCs: " + str(STD_10) + "\n")


    MSE_25 = (Reconstructed_faces_25 - faces) ** 2 # calculating MSE

    mean_25 = np.mean( np.mean(MSE_25 , 1) )# calculating the mean
    STD_25 = np.std( np.mean(MSE_25 , 1) )# calculating the std
    print("Mean of MSE for the first 25 PCs: " + str(mean_25)) #displaying the re
sults
    print("STD of MSE for the first 25 PCs: " + str(STD_25) + "\n")


    MSE_50 = (Reconstructed_faces_50 - faces) ** 2 # calculating MSE

    mean_50 = np.mean( np.mean(MSE_50 , 1) )# calculating the mean
    STD_50 = np.std( np.mean(MSE_50 , 1) )# calculating the std

    print("Mean of MSE for the first 50 PCs: " + str(mean_50)) #displaying the re
sults
    print("STD of MSE for the first 50 PCs: " + str(STD_50) + "\n")
```

The results of this code are given below.

```
Mean of MSE for the first 10 PCs: 666.0905902820485

STD of MSE for the first 10 PCs: 375.5878619791094

Mean of MSE for the first 25 PCs: 488.68926863610693

STD of MSE for the first 25 PCs: 265.94313176911436

Mean of MSE for the first 50 PCs: 358.9466403958055

STD of MSE for the first 50 PCs: 197.1970364263301
```

Once again, the error correlate with the interpretation of the images. we saw images reconstructed from 10 MFs look worse than those created with 10 PCs and looks better than those created with 10 ICs. Images reconstructed from 25 MFs look worse than those created with 25 PCs and looks better than those created with 25 ICs. Images reconstructed from 50 MFs worse than the images created from 50 PCs and images created from 50 ICs. All the error values directly correlate with these results. However even though the error on the NNMF is worse it is closer to the process in the brain and it has inherent clustering property. So, one method is not strictly better than the others, and the best method to use can change application to application. Also the correct number of components to use changes from application to application depending on how much information is desired to be preserved.

Question 2:

There are 21 independent neurons with Gaussian-shaped tuning curves. The equation for these curves is given below.

$$f_i(x) = A \cdot e^{-\frac{(x-\mu_i)^2}{2\sigma_i^2}} \qquad (13)$$

The tuning curves have an amplitude of 1, so A will be 1, and a standard deviation of $\sigma_i = 1$, with centers μ_i evenly spaced between -10 and 10 along the x-axis.

a) Plot all tuning curves in the population on the same axis. Simulate the population response to the stimulus x = −1, and plot the population response as a function each neuron's preferred stimulus value.

First, I created a function to generate the response of a neuron according to the equation 13 for a given σ_i, μ_i. The function is given below.

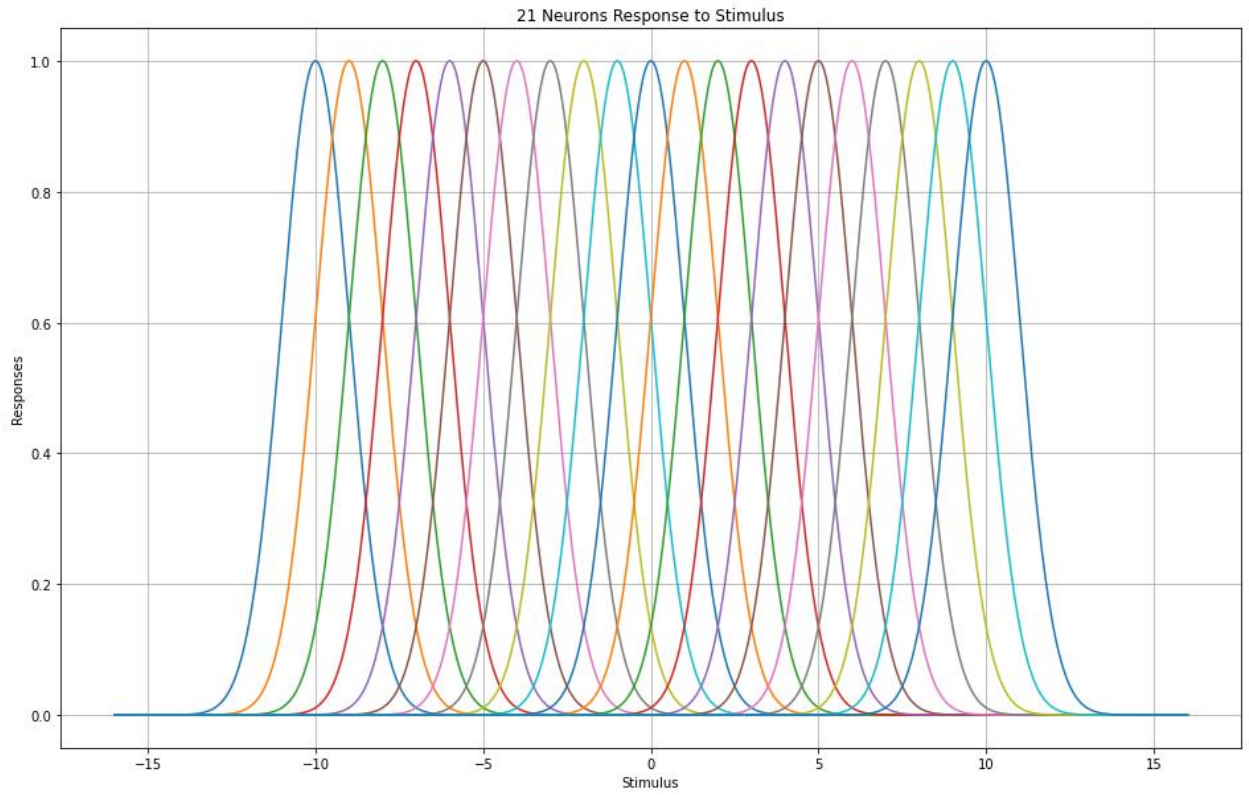
```
def neuron_response(x, mu_i, sigma = 1): # a single neuron

    result = 1 * np.exp(-((x - mu_i) ** 2) / (2 * (sigma ** 2)))

    return result
```

This function will be used for all five parts of this question. I made it take different sigma values if desired so it can be used in part e without any changes.

I then generated an array of input between -16 and 16 and looked at the response of each neuron for all the different stimulus values. The plotted the responses of the 21 neurons on the same graph. The code which does this is given below.

```
    x = np.linspace(-16, 16, 3200) # different inputs to test the neurons
    neurons = np.arange(-10, 11, 1) # the 21 neurons


    plt.figure(figsize=(16,10)) # the plot of neurons tuning curves

    for i in neurons:

        plt.plot(x, neuron_response(x, i))

    plt.grid()
    plt.xlabel('Stimulus')
    plt.ylabel('Responses')
    plt.title('21 Neurons Response to Stimulus')
```

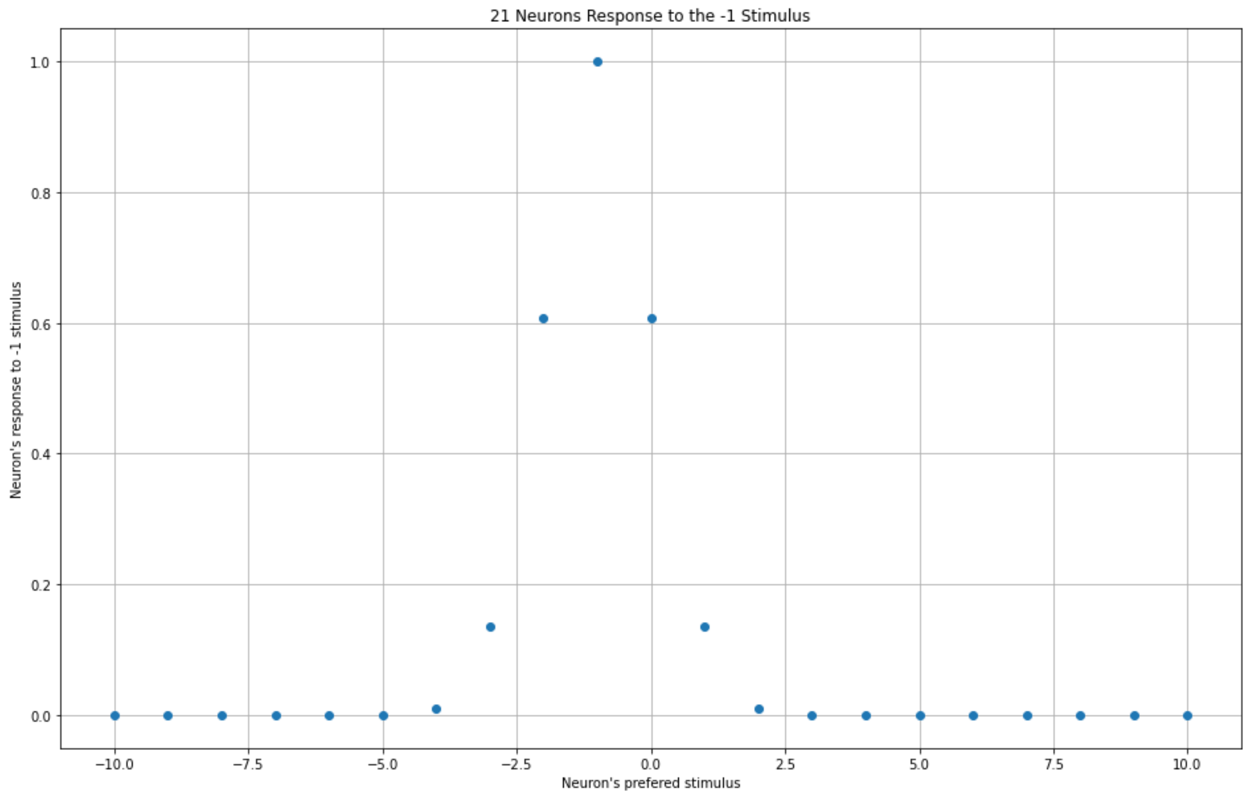
And the output plot of this code is given below.



Then for the second part of part a, I sampled each neuron's response to the stimulus of -1 and plotted the results versus the neurons preferred stimulus value. The code for this is given below.

```
plt.figure(figsize=(16,10)) # the plot of neurons response to the input -1

plt.plot(neurons, neuron_response(-1, neurons), "o")
plt.grid()
plt.xlabel("Neuron's prefered stimulus")
plt.ylabel("Neuron's response to -1 stimulus")
plt.title('21 Neurons Response to the -1 Stimulus')
```

the output plot of this code is given below.



As we have expected the neuron which has a preferred response of -1 gives the best response and the neurons next to it give a decreasing response.

b) Sample a stimulus intensity uniformly from the interval [−5 5], simulate the 21-long vector of population response $\vec{r}$ Assume that each neuron's response is corrupted by an additive Gaussian noise with zero mean and $\sigma/20$ standard deviation. Implement a winner-take-all decoder, and calculate the stimulus estimate x_{WTA} for each trial. Plot the actual and estimated stimulus on the same graph. Compute the mean and standard deviation of error in stimulus estimation across 200 trials.

For this question I first need a winner takes all decoder. This decoder is very simple it just finds the neuron with the higest response and chooses its prefered stimulus value as the estimet stimulus x_{WTA}. Even though it is simple it needs its own function because it will be called multiple times. The code for it is given below.

```
def wta_decoder(neurons, responses):

    result = neurons[np.argmax(responses)] # finding the neuron which gives the
#highest response.

    return result
```

Then we are asked to make 200 triales which all have a samples x value between -5 and 5. The report is ambiguis in how these trials will be chosen. I decided to do uniform sampling between -5 and 5 in the probabilistic sense. The error shouldn't change much between what I did and picking 200 equaly spaced points. I created a function which generates the needed number of triales for the given nerurons. It uses the neuron_response function defiend in part a, and it returns the choosen sample value and the response of all the neurons to that value with the additive noise for each triale. I made a function for this because in part c, d and e we will be doing a computation simular to this again so it will be used a lot. The code for this function is given below.

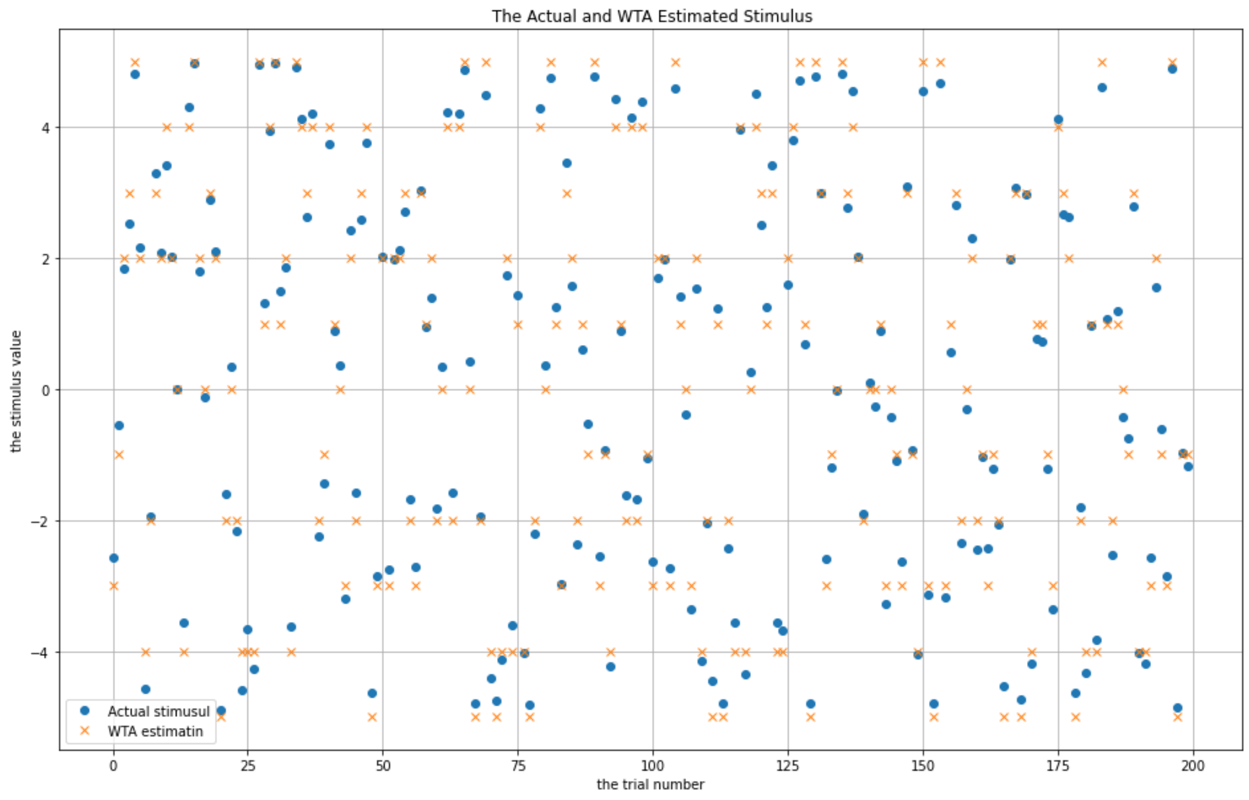
```
def trials(neurons, sigma, trial_count ):


    stimulus = [] # these will be filed in
    responses = []

    for i in range(trial_count): # doing each trial

        noise = np.random.normal(0, 1/20, len(neurons)) # generating random
#gausian noise

        stimulus.append((np.random.random(size = 1) - 0.5) * 10) # generating the
# random stimulus

        response_without_noise = neuron_response(stimulus[i], neurons, sigma)
# generating the  responses

        response = response_without_noise + noise # generating the noisey
#responses

        responses.append(response)

    return stimulus, responses
```

the sampleing and the random gausian noise are all randomly generated using the random package of Python, so every time this code is run it will genearte a different set of trials. This means the results off part b, c, d and e will be slityly different ever time the code is run.

Then finally to plot the actual and estimated stimulus on the same plot I generated 200 trials using the given function. Then estimated the x_{WTA} for each of the trials using the WTA decoder I created, also I calcuated the error of each trial in this step. Finaly I ploted all of these points on the same graph. The code which does this is given below.

```
    neurons = np.arange(-10, 11, 1) # the 21 neurons

    stimulus, responses = trials(neurons, 1, 200) # the generated trials

    wta_estimate = [] # thses will be fild in
    wta_error = []

    for i in range(200): # looping throgh the triales

        wta_estimate.append(wta_decoder(neurons, responses[i])) # using the
#decoder

        wta_error.append(np.abs(wta_estimate[i] - stimulus[i])) # calculating the
 #error of each trial


    plt.figure(figsize=(16,10)) # ploting the actual and estimated stimulus on
#the same graph
    plt.plot(range(200), stimulus, "o")
    plt.plot(range(200), wta_estimate, "x")
    plt.legend(['Actual stimusul', 'WTA estimatin',])
    plt.ylabel('the stimulus value')
    plt.xlabel('the trial number')
    plt.title(' The Actual and WTA Estimated Stimulus')
    plt.grid()
```

The output of the code is given below.



As expected all the estimated points are on intiger values. Because the aditive noise is so small basicly this function pulls the stimulus value to the nearest intiger. If the additive noise was larger then this, there would be realistic probability for a stimulus to go to a intiger which it is not close to.

Then the calcualtion of the mean and STD of the error is very simple. I already found the absulute error of all each trial, so I just used the mean and STD function of Python to calculate the mean and STD of the error. The question doesn't spesify the error function, so I choose absulute error as it is very commen and it explains the stuiation nicely. The code for the error calculation is given below.

```
    mean = np.mean( wta_error )# calculating the mean
    STD = np.std( wta_error )# calculating the std

    print("Mean of error for WTA estimation: " + str(mean)) #displaying the
#results
    print("STD of error for WTA estimation: " + str(STD) + "\n")
```

This code does found the mean and STD of error as:

Mean of error for WTA estimation: 0.25998

STD of error for WTA estimation: 0.15606

We haven't seen any other method yet but WTA decoder is the worst option and as we will see it gives the higgest error.

c) For the experimental trials simulated in part b, implement a maximum-likelihood decoder, and calculate the stimulus estimate x_{ML} for each trial. Plot the actual and estimated stimulus on the same graph. Compute the mean and standard deviation of error in stimulus estimation across 200 trials.

For this question I need to first create a maximum-likelihood decoder. Simply put this decoder will find the stimulus value which makes the response we got the most likely. Below I explain this in more mathematical terms.

$$x = the\ original\ stimulus,\ \ r = the\ noisy\ responses\ of\ the\ neurons$$

$$x_{ML} = \underset{x}{\operatorname{argmax}} f(r \mid x) \tag{14}$$

Solving equation 14 on its own is difficult and we know the x which will maximize this will also maximize the logarithm of the same equation. So as the problem suggest I will use log-likelihood for this question. Then the equation to solve becomes.

$$x_{ML} = \underset{x}{\operatorname{argmax}} \ln\left(f(r \mid x)\right) \tag{15}$$

Now we need to find $f(r|x)$. We already know it so I will just express it. The noise in this problem is never specified as independent but the correlation between the noise terms is never told as well. Because of this I assumed the noises to be independent because it is more logical then making up a correlation relation between different noises.

$$noise\ for\ the\ ith\ neuron = n_i = N\left(0, \left(\frac{1}{20}\right)^2\right), this\ is\ a\ RV$$

$$response\ of\ the\ ith\ neuron = k_i = e^{-\frac{(x-\mu_i)^2}{2\sigma_i^2}}, this\ is\ a\ constant\ for\ know\ x$$

$$noisy\ response\ of\ the\ ith\ neuron = r_i = k_i + n_i = N\left(k_i, \left(\frac{1}{20}\right)^2\right), this\ is\ a\ RV \tag{16}$$

Then $f(r \mid x)$ is very easy to compute as it is multiplication of 21 gaussians.

$$f(r \mid x) = \prod_{i=1}^{21} f(r_i \mid x)$$

$$f(r \mid x) = \prod_{i=1}^{21} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(k_i-\mu_i)^2}{2\sigma^2}}$$

$\sigma = \frac{1}{20}$ so, it is same for all r_i as we found in equation 16.

$$f(r \mid x) = \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^{21} \prod_{i=1}^{21} e^{-\frac{(k_i-\mu_i)^2}{2\sigma^2}}$$

Then we take its logarithm to find the log-likelihood.

$$\ln\left(f(r \mid x)\right) = 21 \cdot \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \sum_{i=1}^{21} ln\left(e^{-\frac{(k_i-\mu_i)^2}{2\sigma^2}}\right)$$

$$\ln\left(f(r \mid x)\right) = 21 \cdot \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \sum_{i=1}^{21} -\frac{(k_i-\mu_i)^2}{2\sigma^2} ln(e)$$

$$\ln\left(f(r \mid x)\right) = 21 \cdot \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \sum_{i=1}^{21} -\frac{(k_i-\mu_i)^2}{2\sigma^2}$$

So now we can express the ML estimate as:

$$x_{ML} = \underset{x}{\operatorname{argmax}}\, 21 \cdot \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \sum_{i=1}^{21} -\frac{(k_i-\mu_i)^2}{2\sigma^2} \tag{17}$$

$\ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)$ is a constant so equation 17 is same as equation 18.

$$x_{ML} = \underset{x}{\operatorname{argmin}} \sum_{i=1}^{21} \frac{(k_i-\mu_i)^2}{2\sigma^2} \tag{18}$$

And because $\sigma = \frac{1}{20}$, for all k_i we can further simplify equation 15 into:

$$x_{ML} = \underset{x}{\operatorname{argmin}} \sum_{i=1}^{21} (k_i-\mu_i)^2 \tag{19}$$

Normally I would take equation 19's derivative with respect to x but here each k_i has its own non-linear equation. So, solving this lineally is not possible or I can't see it. if we put the equation 13 in equation 19 it is easier to see this problem.

$$x_{ML} = \underset{x}{\operatorname{argmin}} \sum_{i=1}^{21} \left(e^{-\frac{(x-\mu_i)^2}{2\sigma_i^2}} - \mu_i \right)^2 \tag{20}$$

To solve equation 20, I test 2400 equally spaced x values between -6 and 6 then pick the one which gives the smallest result. I do not test for only values between -5 and 5 because that would be cheating for the algorithm as it would be using the additional knowledge about the prior. I can do this instead of using a minimization algorithm because x is only one dimensional and can be only in a relatively small interval. Also finding it this way saves me from falling into a local-minima which might be a problem for a minimizing algorithm. I created a decoder function which uses the above explained logic to find the maximum-likelihood estimate response for a given set of responses. The code for this function is given below.

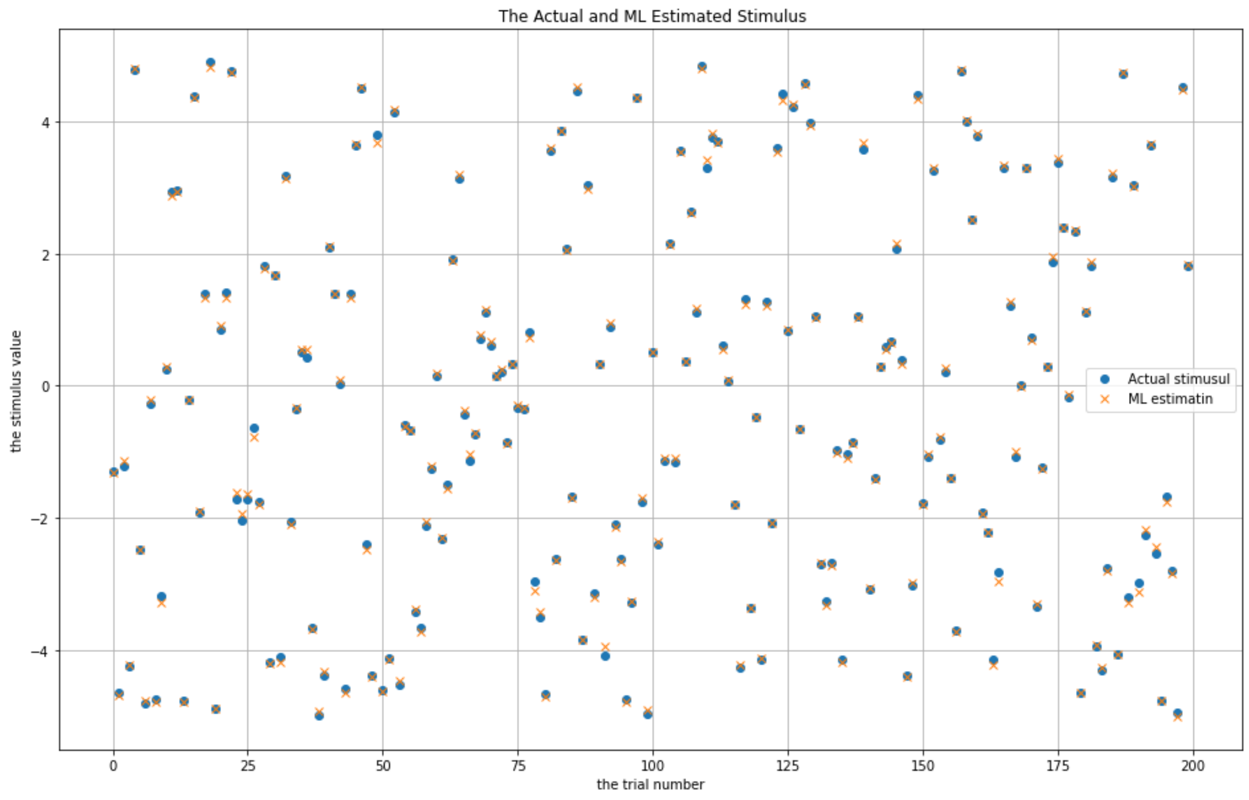
```
def ml_decoder(neurons, responses, sigma_of_neurons):

    posible_x = np.arange(-6, 6, 1/200) # the function for sum of f(x_i) can not
#be solved linearly so we will test difrent x values

    log_likelyhoods = [] # these are not actualy the diferences from the perfect
#x but if x is perfrect this will give 0. because of this it is named like that

    for x in posible_x: # we find how well each x value minimizes the logarithmic
# log-likelihood function.

        log_likelyhood = sum((responses - neuron_response(x, neurons,
sigma_of_neurons) ) ** 2)

        log_likelyhoods.append(log_likelyhood)

    index_of_best_x = np.argmin(log_likelyhoods) # finding the index of best x
    best_x = posible_x[index_of_best_x] # finding the best x

    return best_x
```

This function will be re-used in part e. Because of this it can different sigma values.

Then to plot the actual and estimated stimulus on the same plot I generated 200 trials using the trial generation function. Then estimated the x_{ML} for each of the trials using the ML decoder I created, also I calcuated the error of each trial in this step. Finaly I ploted all of these points on the same graph. The code which does this is given below.

```
    neurons = np.arange(-10, 11, 1) # the 21 neurons

    stimulus, responses = trials(neurons, 1, 200)

    ml_estimate = []
    ml_error = []

    for i in range(200):

        ml_estimate.append(ml_decoder(neurons, responses[i], 1)) # using the 
#decoder

        ml_error.append(np.abs(ml_estimate[i] - stimulus[i])) # calculating the 
#error of each trial



    plt.figure(figsize=(16,10)) # ploting the actual and estimated stimulus on 
#the same graph
    plt.plot(range(200), stimulus, "o")
    plt.plot(range(200), ml_estimate, "x")
    plt.legend(['Actual stimusul', 'ML estimatin',])
    plt.ylabel('the stimulus value')
    plt.xlabel('the trial number')
    plt.title(' The Actual and ML Estimated Stimulus')
    plt.grid()
```

The output plot is given below.



As expected all the estimations mostly on actaul stimulus values.

Then the calcualtion of the mean and STD of the error is very simple. I already found the absulute error of all each trial, so I just used the mean and STD function of Python to calculate the mean and STD of the error. The code for the error calculation is given below.

```
    mean = np.mean( ml_error )# calculating the mean
    STD = np.std( ml_error )# calculating the std

    print("Mean of error for ML estimation: " + str(mean)) #displaying the 
#results
    print("STD of error for ML estimation: " + str(STD) + "\n")
```

This code does found the mean and STD of error as:

Mean of error for ML estimation: 0.041865

STD of error for ML estimation: 0.031421

This error is a lot better than the error from WTA decoder. This is expected as this utilizes the response of all the decoders at the same time, so it is making a more informed decision on the estimation. Also, its algorithm allows is to pick estimations which are not integers which allows it to make a more prices choice.

d) For the experimental trials simulated in part b, implement a maximum-a-posteriori decoder, and calculate the stimulus estimate x_{MAP} for each trial. Assume that the prior of the stimulus value x follows a Gaussian distribution with a mean of 0 and a standard deviation of 2.5. Plot the actual and estimated stimulus on the same graph. Compute the mean and standard deviation of error in stimulus estimation across 200 trials. Interpret your results.

For this question I need to first create a maximum-likelihood decoder. Simply put this decoder will find the most likely stimulus value for the given response. Below I explain this in more mathematical term. By nature, this is very similar to ML and we will see the small difference between these two cases.

$$x = the\ original\ stimulus,\ \ r = the\ noisy\ responses\ of\ the\ neurons$$

$$x_{MAP} = \underset{x}{\operatorname{argmax}} f(x \mid r) \tag{21}$$

Again, solving equation 21 on its own is difficult and we know the x which will maximize this will also maximize the logarithm of the same equation. So as the problem suggest I will use log-likelihood for this question. Then the equation to solve becomes.

$$x_{MAP} = \underset{x}{\operatorname{argmax}} \ln\left(f(x \mid r)\right) \tag{22}$$

Now we need to find $f(x \mid r)$. Using Bayer' rule we can write $f(x \mid r)$ as:

$$f(x \mid r) = \frac{f(r \mid x) f(x)}{f(r)} \tag{23}$$

Placing equation 23 in equation 22 we get:

$$x_{MAP} = \underset{x}{\operatorname{argmax}} \ln\left(\frac{f(r \mid x) f(x)}{f(r)}\right) \tag{24}$$

We already know $f(r \mid x)\ and\ f(x)$ so I will just express them. The noise in this problem is never specified as independent but the correlation between the noise terms is never told as well. Because of this I assumed the noises to be independent because it is more logical then making up a correlation relation between different noises.

$$noise\ for\ the\ ith\ neuron = n_i = N\left(0, \left(\frac{1}{20}\right)^2\right), this\ is\ a\ RV \tag{25}$$

$$response\ of\ the\ ith\ neuron = k_i = e^{-\frac{(x-\mu_i)^2}{2\sigma_i^2}}, this\ is\ a\ constant\ for\ know\ x$$

$$noisy\ response\ of\ the\ ith\ neuron = r_i = k_i + n_i = N\left(k_i, \left(\frac{1}{20}\right)^2\right), this\ is\ a\ RV$$

Then $f(r \mid x)$ is very easy to compute as it is multiplication of 21 gaussians.

$$f(r \mid x) = \prod_{i=1}^{21} f(r_i \mid x)$$

$$f(r \mid x) = \prod_{i=1}^{21} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(k_i-\mu_i)^2}{2\sigma^2}}$$

And $f(x)$ is given in the question

$$f(x) = N(0, 2.5^2) = \frac{1}{\sqrt{2\pi 2.5^2}} e^{-\frac{(x)^2}{2\cdot 2.5^2}}$$

Finally, $f(r)$ is independent from x so it will not affect the final equation. So $f(x \mid r)$ is:

$$f(x \mid r) = \frac{\frac{1}{\sqrt{2\pi 2.5^2}} e^{-\frac{(x)^2}{2\cdot 2.5^2}} \cdot \prod_{i=1}^{21} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(k_i-\mu_i)^2}{2\sigma^2}}}{f(r)}$$

$\sigma = \frac{1}{20}$ so, it is same for all r_i as we found in equation 25.

$$(x \mid r) = \frac{\frac{1}{\sqrt{2\pi 2.5^2}} e^{-\frac{(x)^2}{2\cdot 2.5^2}} \cdot \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^{21} \prod_{i=1}^{21} e^{-\frac{(k_i-\mu_i)^2}{2\sigma^2}}}{f(r)}$$

Then we take its logarithm to find the log-likelihood.

$$\ln\left(f(x \mid r)\right) = 21 \cdot \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \sum_{i=1}^{21} ln\left(e^{-\frac{(k_i-\mu_i)^2}{2\sigma^2}}\right) + \ln\left(\frac{1}{\sqrt{2\pi 2.5^2}}\right) + ln\left(e^{-\frac{(x)^2}{2\cdot 2.5^2}}\right)$$

$$\ln\left(f(x \mid r)\right) = 21 \cdot \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \ln\left(\frac{1}{\sqrt{2\pi 2.5^2}}\right) + \sum_{i=1}^{21} -\frac{(k_i-\mu_i)^2}{2\sigma^2} ln(e) - \frac{(x)^2}{2\cdot 2.5^2} \ln(e)$$

$$\ln\left(f(x \mid r)\right) = 21 \cdot \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \ln\left(\frac{1}{\sqrt{2\pi 2.5^2}}\right) + \left(\sum_{i=1}^{21} -\frac{(k_i-\mu_i)^2}{2\sigma^2}\right) - \frac{(x)^2}{2\cdot 2.5^2}$$

So now we can express the MAP estimate as:

$$x_{MAP} = \underset{x}{\operatorname{argmax}}\, 21 \cdot \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \ln\left(\frac{1}{\sqrt{2\pi 2.5^2}}\right) - \frac{(x)^2}{2\cdot 2.5^2} + \sum_{i=1}^{21} -\frac{(k_i-\mu_i)^2}{2\sigma^2} \tag{26}$$

$\ln\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)$ and $\ln\left(\frac{1}{\sqrt{2\pi 2.5^2}}\right)$ are constants so equation 26 is same as equation 27.

$$x_{MAP} = \underset{x}{\operatorname{argmin}} \frac{(x)^2}{2\cdot 2.5^2} + \sum_{i=1}^{21} \frac{(k_i - \mu_i)^2}{2\sigma^2} \tag{27}$$

Normally I would take equation 27's derivative with respect to x but here each k_i has its own non-linear equation. So, solving this lineally is not possible or I can't see it. if we put the equation 13 in equation 27 it is easier to see this problem.

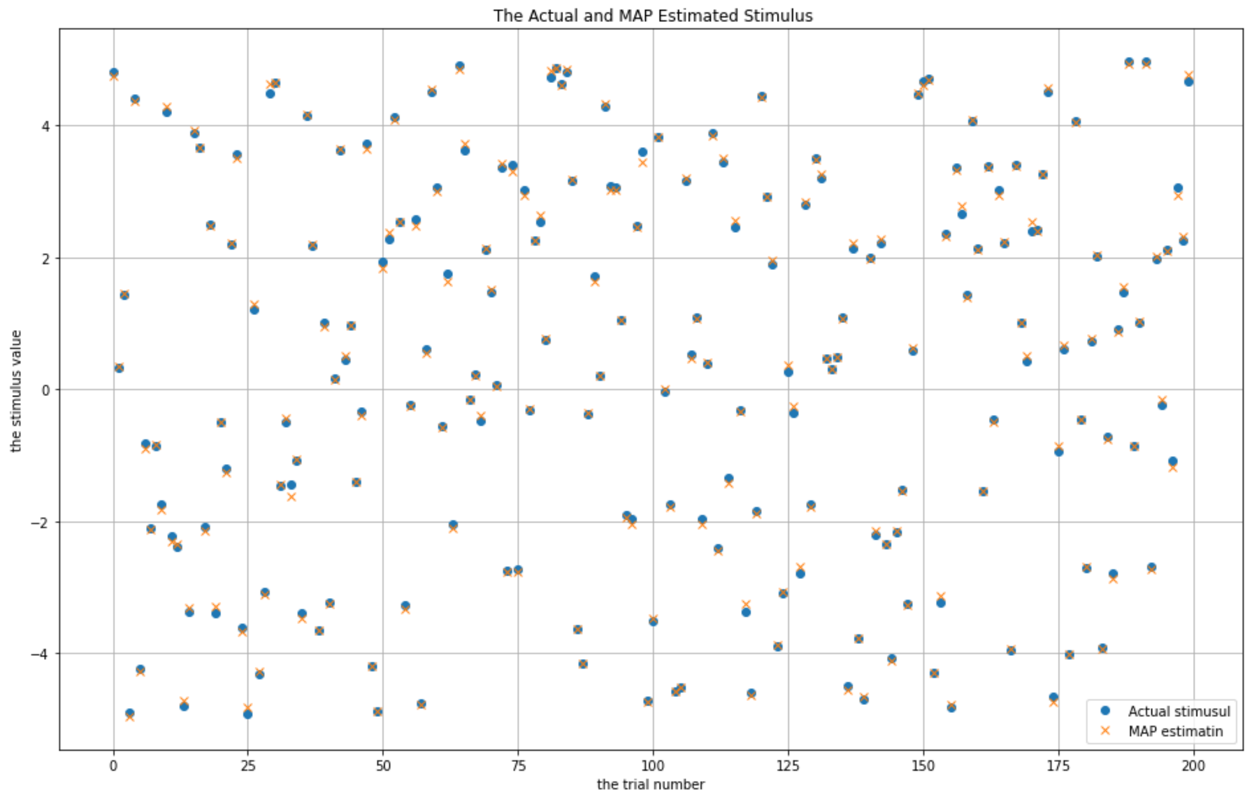
$$x_{MAP} = \underset{x}{\operatorname{argmin}} \frac{(x)^2}{2\cdot 2.5^2} + \sum_{i=1}^{21} \left(e^{-\frac{(x-\mu_i)^2}{2\sigma_i^2}} - \mu_i\right)^2 \tag{28}$$

We can see equation 28 is almost same as the x_{ML} equation. It only has an additional $\frac{(x)^2}{2\cdot 2.5^2}$ term. This is the added prier. It can be seen as a regulating term or implementing our previous knowledge to the problem.

To solve equation 28, I test 2400 equally spaced x values between -6 and 6 then pick the one which gives the smallest result. I can do this instead of using a minimization algorithm because x is only one dimensional and can be only in a relatively small interval. Also finding it this way saves me from falling into a local-minima which might be a problem for a minimizing algorithm. I created a decoder function which uses the above explained logic to find the maximum-a-posteriori estimate response for a given set of responses. The code for this function is given below.

```
def map_decoder(neurons, responses, sigma_of_neurons):

    posible_x = np.arange(-6, 6, 1/200) # the function for sum of f(x_i) can not
be solved linearly so we #will test different x values

    log_likelyhoods = [] # these are not actualy the diferences from the perfect
#x but if x is perfrect this will give 0. because of this it is named like that

    for x in posible_x: # we find how well each x value minimizes the logarithmic
# log-likelihood function.

        log_likelyhood = sum(((responses - neuron_response(x, neurons, sigma_of_n
eurons) ) ** 2) / (2 * ((1/20) ** 2))) + ( x ** 2 ) / (2 * ((2.5) ** 2))
# I found this equation by hand

        log_likelyhoods.append(log_likelyhood)

    index_of_best_x = np.argmin(log_likelyhoods)
    best_x = posible_x[index_of_best_x]
    return best_x
```

The output plot is given below.



As expected all the estimations mostly on actaul stimulus values. MAP estimate addes offset to the resultes of the ML estimation. This offset is the effect of the prior knowlage and it is pulling the estimate of the stimulus to the mean of the stimulus' prior. Here this effect is not very visiable on the plot because the STD of the prior is 50 times larger then the STD of the noise so it has very minimal effect. However we will be able see this effect on the error.

Then the calcualtion of the mean and STD of the error is very simple. I already found the absulute error of all each trial, so I just used the mean and STD function of Python to calculate the mean and STD of the error. The code for the error calculation is given below.

```
    mean = np.mean( map_error )# calculating the mean
    STD = np.std( map_error )# calculating the std

    print("Mean of error for MAP estimation: " + str(mean)) #displaying the 
#results
    print("STD of error for MAP estimation: " + str(STD) + "\n")
```

The output of this code is given below.

Mean of error for MAP estimation: 0.04399277053966217

STD of error for MAP estimation: 0.03169206929168211

The error of this method is very stilly worse than the ML estimation. It is so small in some re-runs there was almost no difference. As I said this small error is because of the small offset the prior introduces to the system. If the STD of the prior was smaller this error would be more significant. This error shouldn't be seen as making the estimation worse. Normally when we are estimating something like this, we wouldn't know the true value of the thing so we won't be able to cheek if the estimation is done well. MAP estimation protects us from making an unlikely estimation and gives us confidence about our estimation correctness. It can also be used to introduce prior research's results to the current one. This example doesn't allow it to show its true usefulness. Also, the error is still a lot better than the error from the WTA decoder. However, this is not a very exiting result as WTA estimation is a very crude method and almost anything can give a better result than it does.

e) Perform an experiment with 200 trials of stimulus intensity. In each trial, sample a stimulus intensity from the interval [−5 5]. For the resulting stimulus vector of length 200, separately simulate the population response vectors ($\vec{r}$) for $\sigma_i = 0.1$, $\sigma_i = 0.2$, $\sigma_i = 0.5$, $\sigma_i = 1$, $\sigma_i = 2$, and $\sigma_i = 5$. In each case, assume additive Gaussian noise with zero mean and 1/20 standard deviation. Calculate MLE estimates of the stimulus x_{ML} based on each population response separately. Compare the mean and standard deviation of error in stimulus estimation for various σ values. Is it better to have narrow or wide tuning curves?

I have been preparing to do this part from the start so all my functions work for different σ_i values. Because of this I do not need write any new functions for this part. I just created 200 trials for each σ_i value then found the mean and STD of error for ML estimation on for each of the σ_i values. The code which does this is given below.

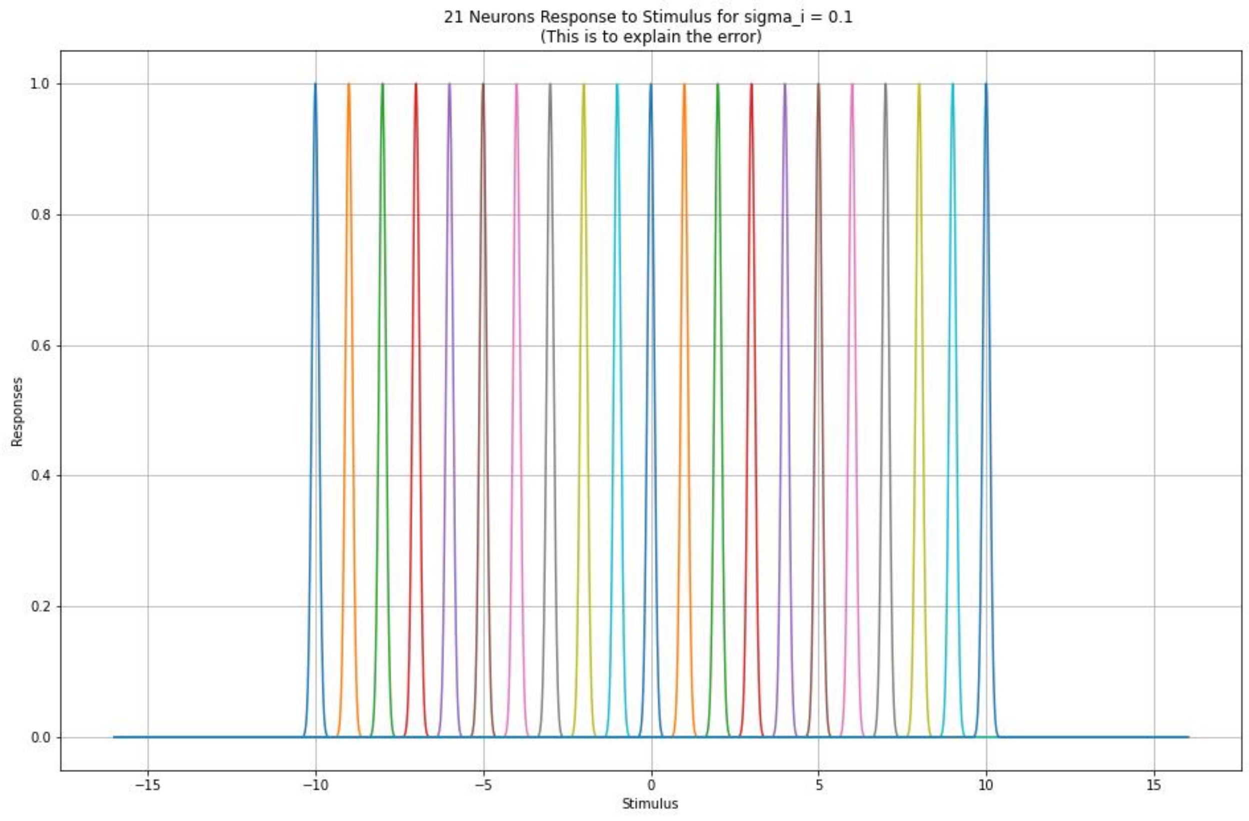
```
test_sigmas = [0.1, 0.2, 0.5, 1, 2, 5] # given in the manual.

neurons = np.arange(-10, 11, 1) # the 21 neurons

for sigma in test_sigmas: # basicly doing part c for each sigma value

    stimulus, responses = trials(neurons, sigma, 200)

    ml_estimate = [] # will fill with new values on each loop
    ml_error = []

    for i in range(200):
            ml_estimate.append(ml_decoder(neurons, responses[i], sigma)) # using
#the ML decoder

            ml_error.append(np.abs(ml_estimate[i] - stimulus[i])) # finding the
#error


        ml_mean = np.mean( ml_error )# calculating the mean
        ml_STD = np.std( ml_error )# calculating the std

        print("Mean of error for ML estimation with sigma_i = " + str(sigma) + "
: " + str(ml_mean)) #displaying the results
        print("STD of error for ML estimation with sigma_i = " + str(sigma) + " :
 " + str(ml_STD) + "\n")
```

The result of this code is given below.

```
Mean of error for ML estimation with sigma_i = 0.1 : 2.2842499829497154

STD of error for ML estimation with sigma_i = 0.1 : 2.7597626561988284


Mean of error for ML estimation with sigma_i = 0.2 : 0.4442685030314978

STD of error for ML estimation with sigma_i = 0.2 : 1.1078809060002117


Mean of error for ML estimation with sigma_i = 0.5 : 0.03529562946390581

STD of error for ML estimation with sigma_i = 0.5 : 0.03195130598010388


Mean of error for ML estimation with sigma_i = 1 : 0.04001665177832476

STD of error for ML estimation with sigma_i = 1 : 0.030016877211216146


Mean of error for ML estimation with sigma_i = 2 : 0.057947879289563825

STD of error for ML estimation with sigma_i = 2 : 0.042466644857612854


Mean of error for ML estimation with sigma_i = 5 : 0.10317168348911018

STD of error for ML estimation with sigma_i = 5 : 0.07295256087069073
```

We can see there is a sweet spot for the width of the tuning curve. Too narrow or wide tuning curves generate more error. The ideal value for our system here is somewhere between 0.5 and 1. At this value every possible point a stimulus can appear is seen by a tuning curve with a high enough rate of change. Because of this we can always get sufficient amount of highly distinguishing information for all stimulus and this allow us to make an accurate decision on the estimate stimuli.

For too wide tuning curves the rate of change of the tuning curves response is too low. Because of this they are not very selective and they are producing more error. And for the too narrow tuning curves there are areas which are not properly seen by any tuning curves. An example to this could be, for $\sigma_i = 0.1$ the point ranges 0.4 to 0.6, 1.4 to 1.6, 2.4 to 2.6 and so are not seen by any tuning curves, so all neurons responses at these intervals can be seen as 0. I placed the response plot of the tuning curves for $\sigma_i = 0.1$ below.



So, for a stimulus in one of these intervals the response is fully composed of the noise. Because the responses do not include any information from the stimulus the estimation is pretty much a completely random guess. This means the error on the $\sigma_i = 0.1$ and $\sigma_i = 0.2$ case depend on the boundaries of the minimization and in theory they can be arbitrarily large for arbitrarily large boundaries. Because having almost infinite error is not desirable, if the tuning curves are not going to have the perfect values it is better for them to be too wide then too narrow.

CODE:

As a formality I am adding all of my code as it is at the end of the report:

```
# -*- coding: utf-8 -*-

import sys # the primer already has it

import numpy as np # this brings simple matrix operations to python
import matplotlib.pyplot as plt # this brings nice loking plots to python

import scipy.io as sio # this is used to load MATLAB files


from sklearn.decomposition import PCA # PCA function

from sklearn.decomposition import FastICA # function given in the manual
from sklearn.decomposition import NMF # function given in the manual


"""
If you want to run this code in a IDE you need to commend out the line below and
write the line
question = '1'
then run the code for question 1 and write the line
question = '2'
then run the code for question 2
"""

question = sys.argv[1]

def Batu_Arda_Düzgün_21802633_hw4(question):
    if question == '1' :
        print("Answer of question 1.) \n")
        question1_part_a()
        question1_part_b()
        question1_part_c()
        question1_part_d()

    elif question == '2' :
        print("Answer of question 2.) \n")
        question2_part_a()
        question2_part_b()
        question2_part_c()
        question2_part_d()
        question2_part_e()


def my_dispImArray(images, title): # I am not explaing this in detail because it
is the function which is given with the instructions.

    width = round((np.shape(images)[1]) ** (0.5) )

    mn = np.shape(images) # Compute rows, cols
    m = mn[0]
    n = mn[1]
    height = int(n / width)

    display_rows = int(np.floor(m ** (0.5))) # Compute number of items to display
    display_cols = int(np.ceil(m / display_rows))

    pad = 1 # padding that will be used

    if np.min(images) < 0: # this is to make the non negative matricies look nice
.

        display_array = -
np.ones((pad + display_rows * (height + pad), pad + display_cols * (width + pad))
); # we will fill this in with the images

    else:

        display_array = np.zeros((pad + display_rows * (height + pad), pad + disp
lay_cols * (width + pad))); # we will fill this in with the images

    curr_ex = 0

    for j in range(0, display_rows):
        for i in range(0, display_cols):

            if curr_ex == m:

                break

            max_val = max(abs(images[curr_ex, :]))

            display_array[pad + j * (height + pad) : pad + j * (height + pad) + h
eight, pad + i * (width + pad) : pad + i * (width + pad) + width] = images[curr_e
x, :].reshape(height, width).T / max_val
```

```
            curr_ex += 1

        if curr_ex == m:

            break

    # Display Image
    plt.figure(figsize=(10, 10))
    plt.imshow(display_array, cmap=plt.cm.gray)
    plt.xticks([])
    plt.yticks([])
    plt.title(title)


def question1_part_a():

    print("\nAnswer of question 1 part a: \n")

    data = sio.loadmat('matlab_hw4_data1') # loading the data given to us
    faces = data["faces"]

    my_dispImArray(faces[0:1,:], "Face 1") # to test if the images are loaded cor
rectly
    my_dispImArray(faces[1:2,:], "Face 2")

    """
    for i in range(0, 2): # to test if the images are loaded correctly
        plt.figure()
        plt.imshow(faces[i,:].reshape(32, 32).T, cmap=plt.cm.gray)
        plt.title('Face ' + str(i+1))
    """

    pca = PCA(100)
    PCA_of_faces = pca.fit(faces) # calculating PCA

    pca_explaind_proprtion_of_var = PCA_of_faces.explained_variance_ratio_ # the
function already computs this for us

    plt.figure(figsize=(16,10)) #ploting Proportion of Variance Explained by each
 Individual PC
    plt.plot(range(1, 101), pca_explaind_proprtion_of_var, "o")
    plt.xlabel('Principal Component Index')
    plt.ylabel('Proportion of the explaind variance')
    plt.title('Plot of the Proportion of Variance Explained by each Individual PC
')
```

```
    plt.grid()

    pca_componets = PCA_of_faces.components_ # taking the principal components fr
om the function

    my_dispImArray(pca_componets[0:25 , :], "The First 25 PCs of the Data using P
CA") # displaying the first 25 PCs

    print("The faces seem to be loaded correctly and the PC's disturbution show t
hese data can PCA decomposed.")

def question1_part_b():

    print("\nAnswer of question 1 part b: \n")

    data = sio.loadmat('matlab_hw4_data1') # loading the data given to us
    faces = data["faces"]

    pca = PCA(100)
    PCA_of_faces = pca.fit(faces) # calculating PCA

    pca_componets = PCA_of_faces.components_  # taking the principlal components.
 this part is repeated



    pca_projection_10 = faces.dot(pca_componets[0:10].T) # lower dimensionlar rep
risentation for 10 PCs

    Reconstructed_faces_10 = pca_projection_10.dot(pca_componets[0:10]) # Reconst
ructed faces for 10 PCs

    my_dispImArray(faces[0:36], "The Original Images")

    my_dispImArray(Reconstructed_faces_10[0: 36], "The Reconstructed Images from 
the First 10 PCs")


    pca_projection_25 = faces.dot(pca_componets[0:25].T) # lower dimensionlar rep
risentation for 25 PCs

    Reconstructed_faces_25 = pca_projection_25.dot(pca_componets[0:25]) # Reconst
ructed faces for 25 PCs
```

```
    my_dispImArray(Reconstructed_faces_25[0: 36], "The Reconstructed Images from
the First 25 PCs")


    pca_projection_50 = faces.dot(pca_componets[0:50].T) # lower dimensionlar rep
risentation for 50 PCs

    Reconstructed_faces_50 = pca_projection_50.dot(pca_componets[0:50]) # Reconst
ructed faces for 50 PCs

    my_dispImArray(Reconstructed_faces_50[0: 36], "The Reconstructed Images from
the First 50 PCs")




    MSE_10 = (Reconstructed_faces_10 - faces) ** 2 # calculating MSE for 10 PCs

    mean_10 = np.mean( np.mean(MSE_10 , 1) )# calculating the mean of MSE for 10
PCs
    STD_10 = np.std( np.mean(MSE_10 , 1) )# calculating the std of MSE for 10 PCs

    print("Mean of MSE for the first 10 PCs: " + str(mean_10)) #displaying the re
sults
    print("STD of MSE for the first 10 PCs: " + str(STD_10) + "\n")


    MSE_25 = (Reconstructed_faces_25 - faces) ** 2 # calculating MSE for 25 PCs

    mean_25 = np.mean( np.mean(MSE_25 , 1) )# calculating the mean of MSE for 25
PCs
    STD_25 = np.std( np.mean(MSE_25 , 1) )# calculating the std of MSE for 25 PCs

    print("Mean of MSE for the first 25 PCs: " + str(mean_25)) #displaying the re
sults
    print("STD of MSE for the first 25 PCs: " + str(STD_25) + "\n")


    MSE_50 = (Reconstructed_faces_50 - faces) ** 2 # calculating MSE for 50 PCs

    mean_50 = np.mean( np.mean(MSE_50 , 1) )# calculating the mean of MSE for 50
PCs
    STD_50 = np.std( np.mean(MSE_50 , 1) )# calculating the std of MSE for 50 PCs
    print("Mean of MSE for the first 50 PCs: " + str(mean_50)) #displaying the re
sults
    print("STD of MSE for the first 50 PCs: " + str(STD_50) + "\n")

    print("We can see that, as the number of PCs used to reconstruct the images i
ncreases, the images look more like the originals. This is not surprising as incr
easing number of PCs means storing more detailed information in the low dimension
al representation and images which are created with more information are detailed
 and look better ")




def question1_part_c(): # the proper one

    print("\nAnswer of question 1 part c: \n")

    print("Every time this code is run the result will be slitly different becaus
e FastICA function is not deterministic.")

    data = sio.loadmat('matlab_hw4_data1') # loading the data given to us
    faces = data["faces"]


    ICA_model_10 = FastICA(50, max_iter=10000) # data used for the transformation
, it seemslike the only number which we enter is the lasteig
    ICA_of_faces_10 = ICA_model_10.fit(faces.T) # data used for the transformatio
n fit to our face data


    components_10 = ICA_model_10.fit(faces.T).transform(faces.T)[:,0:10]# indipen
dent components of the model. we pick the first 10

    my_dispImArray(components_10.T, "The First 10 ICs of the Data using ICA") # d
isplaying the ICs


    ICA_model_25 = FastICA(50, max_iter=10000) # data used for the transformation
    ICA_of_faces_25 = ICA_model_25.fit(faces.T) # data used for the transformatio
n fit to our face data


    components_25 = ICA_model_25.fit(faces.T).transform(faces.T)[:,0:25]# indipen
dent components of the model.
```

```
    my_dispImArray(components_25.T, "The First 25 ICs of the Data using ICA") # d
isplaying the ICs


    ICA_model_50 = FastICA(50, max_iter=10000) # data used for the transformation
    ICA_of_faces_50 = ICA_model_50.fit(faces.T) # data used for the transformatio
n fit to our face data

    components_50 = ICA_model_50.fit(faces.T).transform(faces.T)[:,0:50]# indipen
dent components of the model.

    my_dispImArray(components_50.T, "The First 50 ICs of the Data using ICA") # d
isplaying the ICs



    my_dispImArray(faces[0:36], "The Original Images") # displaying the original
faces



    Reconstructed_faces_10 = components_10.dot((ICA_model_10.mixing_[:,0:10]).T)
+ ICA_model_10.mean_ # The function de-
means each sample seperatly an we need to add it back.

    Reconstructed_faces_10 = Reconstructed_faces_10.T

    my_dispImArray(Reconstructed_faces_10[0:36,:], "The Reconstruct Images from t
he First 10 ICs") #displaying the reconstructed images


    Reconstructed_faces_25 = components_25.dot((ICA_model_25.mixing_[:,0:25]).T)
+ ICA_model_25.mean_ # The function de-
means each sample seperatly an we need to add it back.

    Reconstructed_faces_25 = Reconstructed_faces_25.T

    my_dispImArray(Reconstructed_faces_25[0:36,:], "The Reconstruct Images from t
he First 25 ICs") #displaying the reconstructed images


    Reconstructed_faces_50 = components_50.dot(ICA_model_50.mixing_.T) + ICA_mode
l_50.mean_ # The function de-
means each sample seperatly an we need to add it back.
```

```
    Reconstructed_faces_50 = Reconstructed_faces_50.T

    my_dispImArray(Reconstructed_faces_50[0:36,:], "The Reconstruct Images from t
he First 50 ICs") #displaying the reconstructed images



    MSE_10 = (Reconstructed_faces_10 - faces) ** 2 # calculating MSE

    mean_10 = np.mean( np.mean(MSE_10 , 1) )# calculating the mean
    STD_10 = np.std( np.mean(MSE_10 , 1) )# calculating the std

    print("Mean of MSE for the first 10 ICs: " + str(mean_10)) #displaying the re
sults
    print("STD of MSE for the first 10 ICs: " + str(STD_10) + "\n")


    MSE_25 = (Reconstructed_faces_25 - faces) ** 2 # calculating MSE

    mean_25 = np.mean( np.mean(MSE_25 , 1) )# calculating the mean
    STD_25 = np.std( np.mean(MSE_25 , 1) )# calculating the std

    print("Mean of MSE for the first 25 ICs: " + str(mean_25)) #displaying the re
sults
    print("STD of MSE for the first 25 ICs: " + str(STD_25) + "\n")


    MSE_50 = (Reconstructed_faces_50 - faces) ** 2 # calculating MSE

    mean_50 = np.mean( np.mean(MSE_50 , 1) )# calculating the mean
    STD_50 = np.std( np.mean(MSE_50 , 1) )# calculating the std

    print("Mean of MSE for the first 50 ICs: " + str(mean_50)) #displaying the re
sults
    print("STD of MSE for the first 50 ICs: " + str(STD_50) + "\n")

    print("This method is very slitly better then PCA for the 50 component case.
However, for the 10 and 25 component cases it is worse then the PCA, the reason f
or that is the lastEig value.")


def question1_part_d():

    print("\nAnswer of question 1 part d: \n")
    print("Every time this code is run the result will be slitly different becaus
e NMF function is not deterministic.")

    data = sio.loadmat('matlab_hw4_data1') # loading the data given to us
    faces = data["faces"]

    min_number_to_add = np.abs(np.min(faces)) # the most negative number in the f
aces matrix

    positive_faces = faces + min_number_to_add # prepairing for the NNMF


    nnfm_model_10 = NMF(n_components=10,solver="mu", max_iter=1000) # model creat
ion

    W_10 = nnfm_model_10.fit(positive_faces).transform(positive_faces) # these ar
e the down samples faces.
    H_10 = nnfm_model_10.components_ # these are the MFs

    my_dispImArray(H_10, "The 10 MFs of the Data using NNMF") # displaying the 10
 MFs


    nnfm_model_25 = NMF(n_components=25,solver="mu", max_iter=1000) # model creat
ion

    W_25 = nnfm_model_25.fit(positive_faces).transform(positive_faces) # these ar
e the down samples faces.
    H_25 = nnfm_model_25.components_ # these are the MFs

    my_dispImArray(H_25, "The 25 MFs of the Data using NNMF") # displaying the 25
 MFs


    nnfm_model_50 = NMF(n_components=50, solver="mu", max_iter=1000) # model crea
tion

    W_50 = nnfm_model_50.fit(positive_faces).transform(positive_faces) # these ar
e the down samples faces.
    H_50 = nnfm_model_50.components_ # these are the MFs

    my_dispImArray(H_50, "The 50 MFs of the Data using NNMF") # displaying the 50
 MFs
```

```
    my_dispImArray(faces[0:36], "The Original Images") # displaying the original 
faces


    Reconstructed_faces_10 = W_10.dot(H_10) - min_number_to_add# reconstracting t
he original faces

    my_dispImArray(Reconstructed_faces_10[0:36, :], "The Reconstructed Images fro
m the 10 MFs") # displaying the reconstrancted images.



    Reconstructed_faces_25 = W_25.dot(H_25) - min_number_to_add# reconstracting t
he original faces

    my_dispImArray(Reconstructed_faces_25[0:36, :], "The Reconstructed Images fro
m the 25 MFs") # displaying the reconstrancted images.



    Reconstructed_faces_50 = W_50.dot(H_50) - min_number_to_add# reconstracting t
he original faces

    my_dispImArray(Reconstructed_faces_50[0:36, :], "The Reconstructed Images fro
m the 50 MFs") # displaying the reconstrancted images.




    MSE_10 = (Reconstructed_faces_10 - faces) ** 2 # calculating MSE

    mean_10 = np.mean( np.mean(MSE_10 , 1) )# calculating the mean
    STD_10 = np.std( np.mean(MSE_10 , 1) )# calculating the std

    print("Mean of MSE for the first 10 PCs: " + str(mean_10)) #displaying the re
sults
    print("STD of MSE for the first 10 PCs: " + str(STD_10) + "\n")


    MSE_25 = (Reconstructed_faces_25 - faces) ** 2 # calculating MSE

    mean_25 = np.mean( np.mean(MSE_25 , 1) )# calculating the mean
    STD_25 = np.std( np.mean(MSE_25 , 1) )# calculating the std
    print("Mean of MSE for the first 25 PCs: " + str(mean_25)) #displaying the re
sults
    print("STD of MSE for the first 25 PCs: " + str(STD_25) + "\n")


    MSE_50 = (Reconstructed_faces_50 - faces) ** 2 # calculating MSE

    mean_50 = np.mean( np.mean(MSE_50 , 1) )# calculating the mean
    STD_50 = np.std( np.mean(MSE_50 , 1) )# calculating the std

    print("Mean of MSE for the first 50 PCs: " + str(mean_50)) #displaying the re
sults
    print("STD of MSE for the first 50 PCs: " + str(STD_50) + "\n")

    print("This is generating more error then the PCA because it has an aditional
 constraint of trying to keep every part of it none-
negative. This is more realistic interms of the processes going inside the brain
but it still gives worse results in term of error.")

    plt.show() # this privents the plot from closing themselves at the end





def neuron_response(x, mu_i, sigma = 1): # a single neuron

    result = 1 * np.exp(-((x - mu_i) ** 2) / (2 * (sigma ** 2)))

    return result


def question2_part_a():

    print("\nAnswer of question 2 part a: \n")

    x = np.linspace(-16, 16, 1000) # different inputs to test the neurons
    neurons = np.arange(-10, 11, 1) # the 21 neurons


    plt.figure(figsize=(16,10)) # the plot of neurons tuning curves

    for i in neurons:

        plt.plot(x, neuron_response(x, i))
    plt.grid()
    plt.xlabel('Stimulus')
    plt.ylabel('Responses')
    plt.title('21 Neurons Response to Stimulus')


    plt.figure(figsize=(16,10)) # the plot of neurons response to the input -1

    plt.plot(neurons, neuron_response(-1, neurons), "o")
    plt.grid()
    plt.xlabel("Neuron's prefered stimulus")
    plt.ylabel("Neuron's response to -1 stimulus")
    plt.title('21 Neurons Response to the -1 Stimulus')

    print("The tuning curves looks good, so we can do the rest of the question.")


def wta_decoder(neurons, responses):

    result = neurons[np.argmax(responses)] # finding the neuron which gives the h
ighest response.

    return result

def trials(neurons, sigma, trial_count ):

    stimulus = [] # these will be filed in
    responses = []

    for i in range(trial_count): # doing each trial

        noise = np.random.normal(0, 1/20, len(neurons)) # generating random
#gausian noise

        stimulus.append((np.random.random(size = 1) - 0.5) * 10) # generating the
# random stimulus

        response_without_noise = neuron_response(stimulus[i], neurons, sigma)
# generating the  responses

        response = response_without_noise + noise # generating the noisey
#responses

        responses.append(response)
    return stimulus, responses



def question2_part_b():

    print("\nAnswer of question 2 part b: \n")

    print("Every time this code is run the result will be slitly different becaus
e trials function is random. \n")


    neurons = np.arange(-10, 11, 1) # the 21 neurons

    stimulus, responses = trials(neurons, 1, 200) # the generated trials

    wta_estimate = [] # thses will be fild in
    wta_error = []

    for i in range(200): # looping throgh the triales

        wta_estimate.append(wta_decoder(neurons, responses[i])) # using the decod
er

        wta_error.append(np.abs(wta_estimate[i] - stimulus[i])) # calculating the
 error of each trial


    plt.figure(figsize=(16,10)) # ploting the actual and estimated stimulus on th
e same graph
    plt.plot(range(200), stimulus, "o")
    plt.plot(range(200), wta_estimate, "x")
    plt.legend(['Actual stimusul', 'WTA estimatin',])
    plt.ylabel('the stimulus value')
    plt.xlabel('the trial number')
    plt.title(' The Actual and WTA Estimated Stimulus')
    plt.grid()


    mean = np.mean( wta_error )# calculating the mean
    STD = np.std( wta_error )# calculating the std

    print("Mean of error for WTA estimation: " + str(mean)) #displaying the resul
ts
```

```
    print("STD of error for WTA estimation: " + str(STD) + "\n")

    print("WTA is a very crude method and because of it is creating th largest er
ror")


def ml_decoder(neurons, responses, sigma_of_neurons):

    posible_x = np.arange(-
6, 6, 1/200) # the function for sum of f(x_i) can not be solved linearly so we wi
ll test difrent x values


    log_likelyhoods = [] # these are not actualy the diferences from the perfect
x but if x is perfrect this will give 0. because of this it is named like that


    for x in posible_x: # we find how well each x value minimizes the logarithmic
 log-likelihood function.

        log_likelyhood = sum((responses - neuron_response(x, neurons, sigma_of_ne
urons) ) ** 2) # I found this equation by hand

        log_likelyhoods.append(log_likelyhood)

    index_of_best_x = np.argmin(log_likelyhoods) # finding the index of best x
    best_x = posible_x[index_of_best_x] # finding the best x

    return best_x


def question2_part_c():

    print("\nAnswer of question 2 part c: \n")

    print("Every time this code is run the result will be slitly different becaus
e trials function is random. \n")

    neurons = np.arange(-10, 11, 1) # the 21 neurons

    stimulus, responses = trials(neurons, 1, 200)

    ml_estimate = []
    ml_error = []
    for i in range(200):

        ml_estimate.append(ml_decoder(neurons, responses[i], 1)) # using the deco
der

        ml_error.append(np.abs(ml_estimate[i] - stimulus[i])) # calculating the e
rror of each trial



    plt.figure(figsize=(16,10)) # ploting the actual and estimated stimulus on th
e same graph
    plt.plot(range(200), stimulus, "o")
    plt.plot(range(200), ml_estimate, "x")
    plt.legend(['Actual stimusul', 'ML estimatin',])
    plt.ylabel('the stimulus value')
    plt.xlabel('the trial number')
    plt.title(' The Actual and ML Estimated Stimulus')
    plt.grid()


    mean = np.mean( ml_error )# calculating the mean
    STD = np.std( ml_error )# calculating the std

    print("Mean of error for ML estimation: " + str(mean)) #displaying the result
s
    print("STD of error for ML estimation: " + str(STD) + "\n")

    print("ML gives the over all best resluts, It doesn't use any prior knowlage.
")


def map_decoder(neurons, responses, sigma_of_neurons):

    posible_x = np.arange(-
6, 6, 1/200) # the function for sum of f(x_i) can not be solved linearly so we wi
ll test difrent x values

    log_likelyhoods = [] # these are not actualy the diferences from the perfect
x but if x is perfrect this will give 0. because of this it is named like that

    for x in posible_x: # we find how well each x value minimizes the logarithmic
 log-likelihood function.
```

```
        log_likelyhood = sum(((responses - neuron_response(x, neurons, sigma_of_n
eurons) ) ** 2) / (2 * ((1/20) ** 2))) + ( x ** 2 ) / (2 * ((2.5) ** 2)) # I foun
d this equation by hand

        log_likelyhoods.append(log_likelyhood)

    index_of_best_x = np.argmin(log_likelyhoods)
    best_x = posible_x[index_of_best_x]

    return best_x




def question2_part_d():

    print("\nAnswer of question 2 part d: \n")

    print("Every time this code is run the result will be slitly different becaus
e trials function is random. \n")

    neurons = np.arange(-10, 11, 1) # the 21 neurons

    stimulus, responses = trials(neurons, 1, 200)

    map_estimate = []
    map_error = []



    for i in range(200):

        map_estimate.append(map_decoder(neurons, responses[i], 1))

        map_error.append(np.abs(map_estimate[i] - stimulus[i]))



    plt.figure(figsize=(16,10))
    plt.plot(range(200), stimulus, "o")
    plt.plot(range(200), map_estimate, "x")
    plt.legend(['Actual stimusul', 'MAP estimatin',])
    plt.ylabel('the stimulus value')
    plt.xlabel('the trial number')
    plt.title(' The Actual and MAP Estimated Stimulus')
    plt.grid()


    mean = np.mean( map_error )# calculating the mean
    STD = np.std( map_error )# calculating the std

    print("Mean of error for MAP estimation: " + str(mean)) #displaying the resul
ts
    print("STD of error for MAP estimation: " + str(STD) + "\n")

    print("MAP gives a slitly worse reslut then the ML method, but it does use th
e prior knowlage.")


def question2_part_e():

    print("\nAnswer of question 2 part e: \n")

    print("Every time this code is run the result will be slitly different becaus
e trials function is random. \n")

    test_sigmas = [0.1, 0.2, 0.5, 1, 2, 5] # given in the manual.

    neurons = np.arange(-10, 11, 1) # the 21 neurons

    for sigma in test_sigmas: # basicly doing part c for each sigma value

        stimulus, responses = trials(neurons, sigma, 200)

        ml_estimate = [] # will fill with new values on each loop
        ml_error = []

        for i in range(200):

            ml_estimate.append(ml_decoder(neurons, responses[i], sigma)) # using
the ML decoder

            ml_error.append(np.abs(ml_estimate[i] - stimulus[i])) # finding the e
rror


        ml_mean = np.mean( ml_error )# calculating the mean
        ml_STD = np.std( ml_error )# calculating the std
        print("Mean of error for ML estimation with sigma_i = " + str(sigma) + "
: " + str(ml_mean)) #displaying the results
        print("STD of error for ML estimation with sigma_i = " + str(sigma) + " :
 " + str(ml_STD) + "\n")


    # this is for explanation of the error and not part of the asked plot.

    x = np.linspace(-16, 16, 3200) # different inputs to test the neurons
    neurons = np.arange(-10, 11, 1) # the 21 neurons


    plt.figure(figsize=(16,10)) # the plot of neurons tuning curves

    for i in neurons:

        plt.plot(x, neuron_response(x, i, 0.1))

    plt.grid()
    plt.xlabel('Stimulus')
    plt.ylabel('Responses')
    plt.title('21 Neurons Response to Stimulus for sigma_i = 0.1 \n(This is to ex
plain the error)')

    print("There is a sweet spot for the tuning curve with and for this setup it
seems to be 1. If we can not have the best tuning curve value it is a lot better
to have a too wide tubing curve compared to a too narrow tuning curve.")

    plt.show() # this privents the plot from closing themselves at the end


Batu_Arda_Düzgün_21802633_hw4(question) # this is the only line which runs so it
is very important.
```