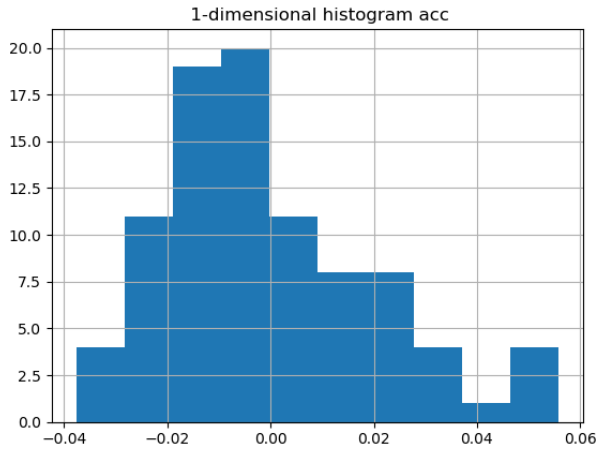
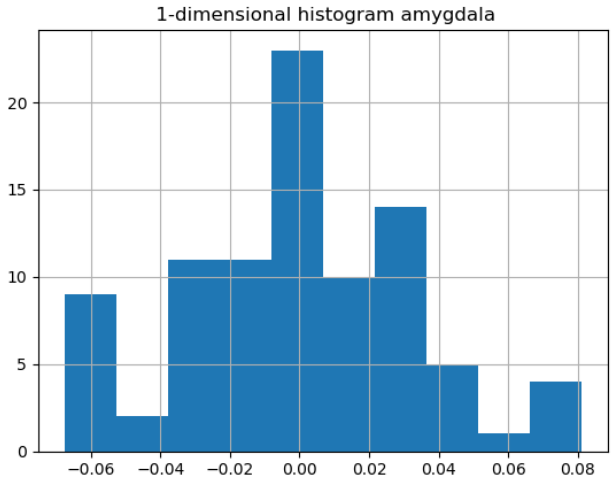
HOMEWORK 3 – COMPUTATIONAL DATA

1. Density estimation : Psychological experiments

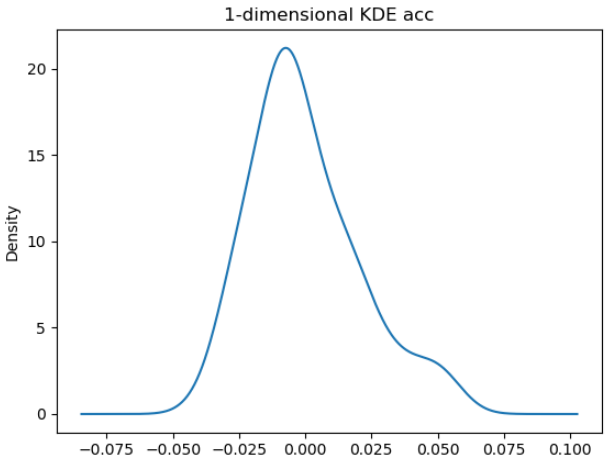
a)

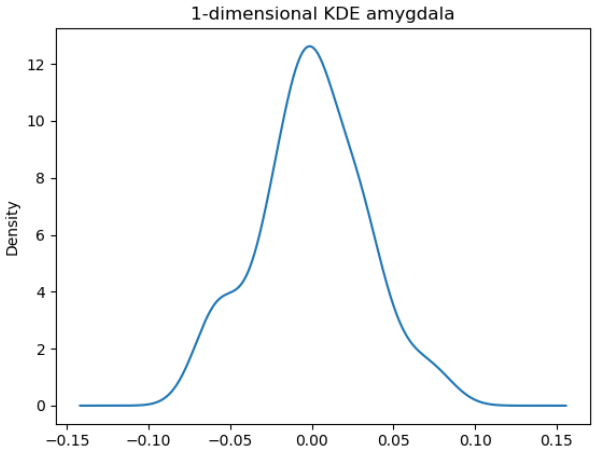
By using the method .hist from Pandas library we obtain for both Amygdala and Acc :





By using the method plot.kde from the library pandas we get:





We see the same trend from both technique of visualization, however the kde has less noise thant the hist method.

b) For the histogram we compute the appropriate number of beans using two techniques :

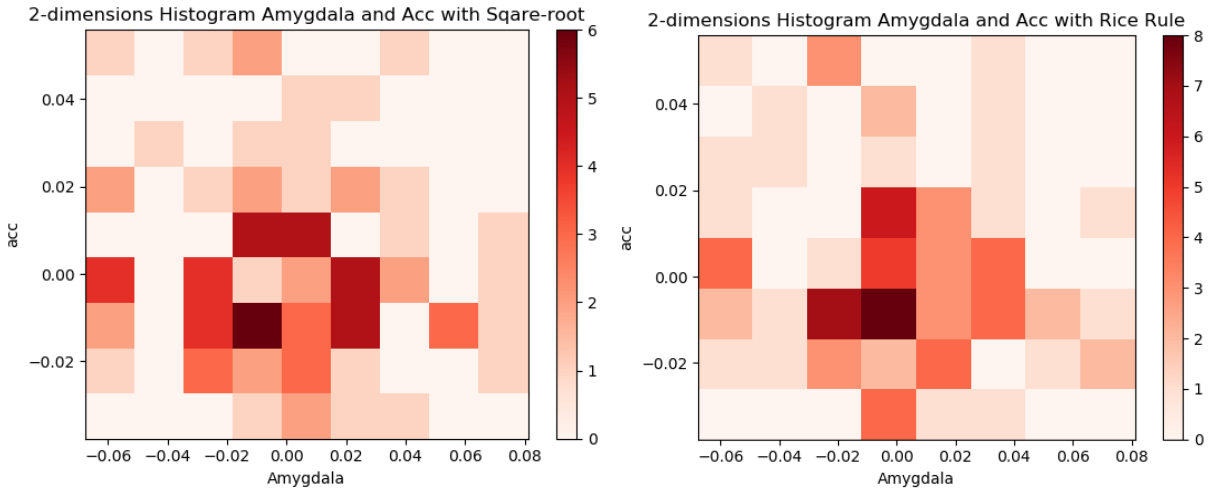
- The squareroot
- The recirule

We get :

```
nb_bins_squareroot=int(m.sqrt(len(n90pol['acc'].values.tolist())))
nb_bins_ricerule=int(2*(len(n90pol['acc'].values.tolist()))**(1/3))
```

Squareroot	Ricerule
9	8

Using the method hist2d we get:



We observe the same trend but with different emphasize this is due to the number of bins that segregate differently the data.

For the kde, we compute the appropriate number of beans using two techniques :

- The silverman method
- The scoot rule

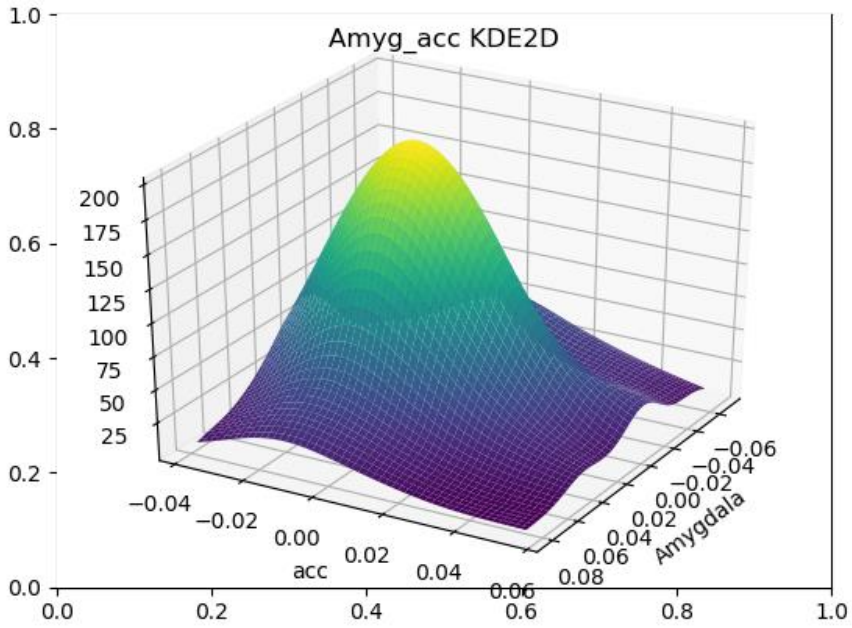
We get :

```
standard_dev=n90pol.iloc[:,:2].std(axis=0)
h_silver=1.06*standard_dev*(90)**(-1/5)
h_scott=3.49*standard_dev/(90**(1/3))
```

Silverman	Scott
amygdala_hscott = 0.0253922, acc_hsilver 0.0159145	amygdala_hsilver = 0.01405227080724824, acc_hsilver= 0.00880724554180054

We obtain for the scott method the KDE for the joint probability :

```
def KDE2D(band_w,Data):
    x=np.asarray(Data.iloc[:,0]).T
    y=np.asarray(Data.iloc[:,1]).T
    xy = np.vstack([x,y])
    fig,ax=plt.subplots()
    kde = KernelDensity(bandwidth=band_w, metric='euclidean',kernel='gaussian', algorithm='ball_tree').fit(xy.T)
    X, Y = np.mgrid[min(x):max(x):1000j, min(y):max(y):1000j]
    pos = np.vstack([X.ravel(), Y.ravel()])
    A = np.reshape(np.exp(kde.score_samples(pos.T)), X.shape)
    ax=fig.gca(projection='3d')
    ax.view_init(elev=30,azim=30)
    ax.plot_surface(X,Y,A,cmap=plt.cm.viridis)

    plt.xlabel('Amygdala')
    plt.ylabel('acc')
    plt.title('Amyg_acc KDE2D')

    return A


amyg_acc2D=KDE2D(0.014,n90pol)
```



c)

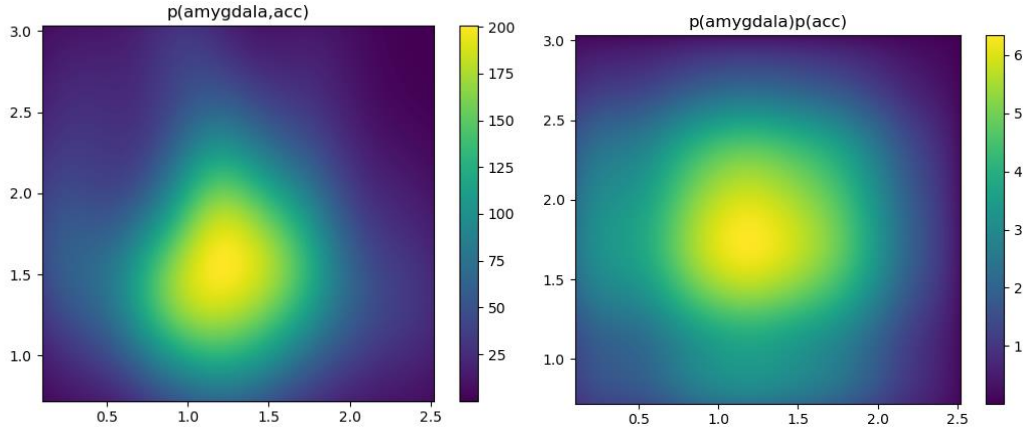
To get verify wether or not ht variables amygdala and acc are independent we use the function independent with :

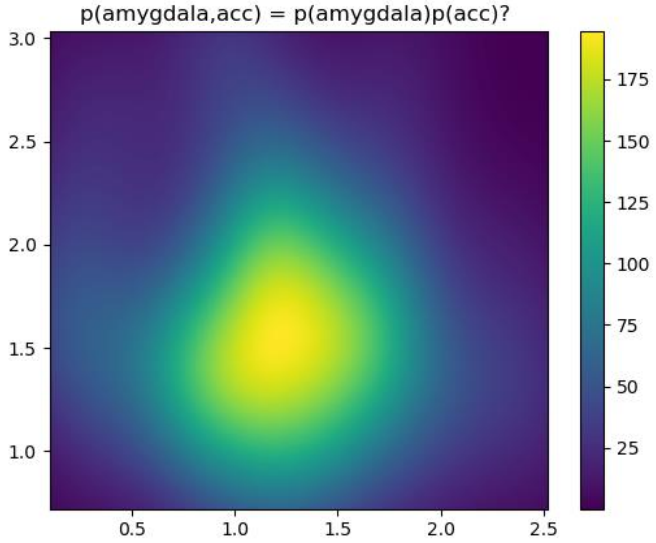
Inputs : p(amygdala) using KDE, p(acc) unsing KDE and the joint probability amyg_acc2D

Outputs: 3 plots as asked in the statement and the matrix Inde (wich relates the differences between the values of p(acc,amyg) and p(acc)p(amyg)

```
def independant(X,Y,XY):
    X=np.asarray(X)
    Y=np.asarray(Y)
    X_Y=np.zeros((len(X),len(X)))
    for i in range (0,len(X)):
        for j in range(0,len(X)):
            X_Y[i][j]=X[i]*X[j]
    Inde=abs(np.asarray(XY)-X_Y)

    plt.figure()
    plt.imshow(np.rot90(X_Y))
    plt.colorbar()
    plt.title('p(amygdala)p(acc)')

    plt.figure()
    plt.imshow(np.rot90(XY))
    plt.colorbar()
    plt.title('p(amygdala,acc)')

    plt.figure()
    fig1,ax_2=plt.subplots()
    plt.imshow(np.rot90(Inde))
    plt.colorbar()
    plt.title('p(amygdala,acc) = p(amygdala)p(acc)? ')
    return Inde

Inde=independant(amyg1D,acc1D,amyg_acc2D)
```

We obtain the plots:





If the variables were independent we would have |p(amygdala,acc)−p(amygdala)p(acc)| =0 (or close to 0) in the entire plot. However, This is not the case here and we can conclude that the two parts of the brain are dependent: our decisions are influenced by our emotions and vice-versa.

d)

We use the function Orient_Condi_prob with:

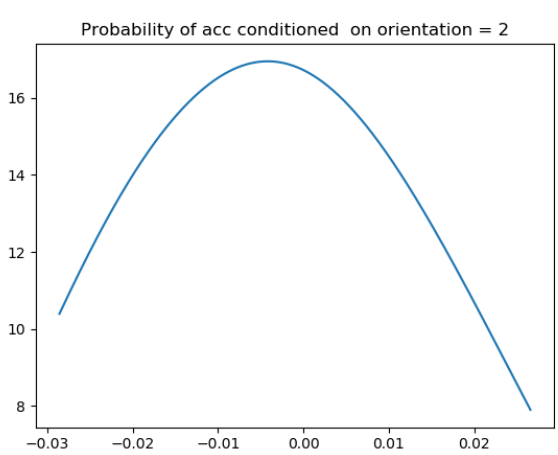
Inputs : the data_set; the orientation we want, and the name of the column we want to be conditioned on.

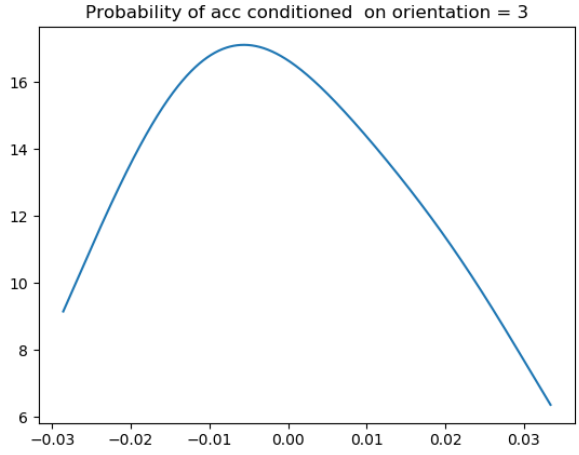
Output : the probability density function of the amyg or acc conditioned on the orientation

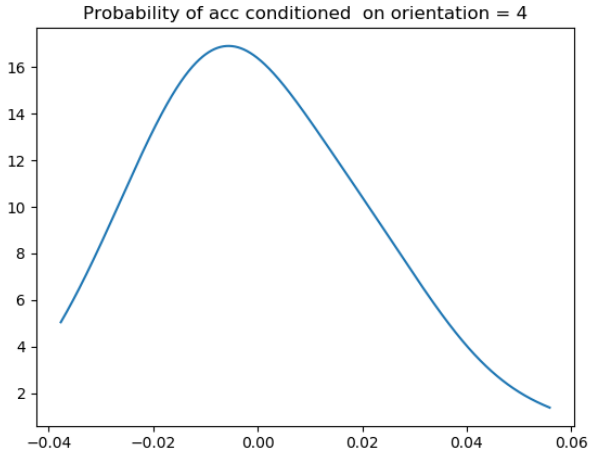
```
def Orient_Condi_prob(Data_set,orientation,column):
    prob=[]
    if column == 'acc':
        Col=np.asarray(Data_set.iloc[:,1])
        count=0
        for orient in np.asarray(Data_set.iloc[:,2]):
            if orient==orientation:
                prob.append(Col[count])
                count+=1

    elif column == 'amyg':
        Col=np.asarray(Data_set.iloc[:,1])
        count=0
        for orient in np.asarray(Data_set.iloc[:,2]):
            if orient==orientation:
                    prob.append(Col[count])
                    count+=1
    a=KDE1D(0.014,prob,'Probability of ' +column +
            ' conditioned  on orientation = {}'.format(orientation))
    return a
```

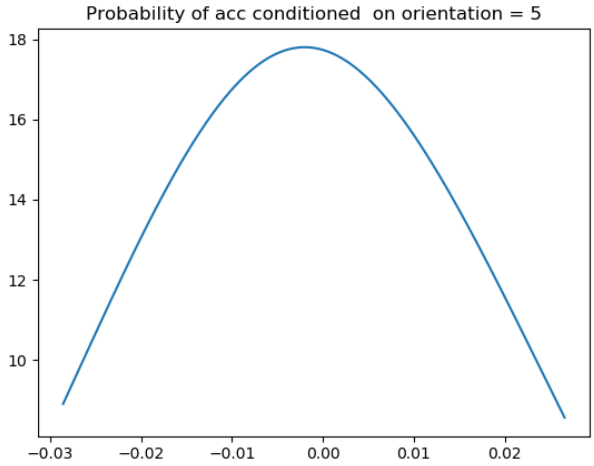
Then we compute :

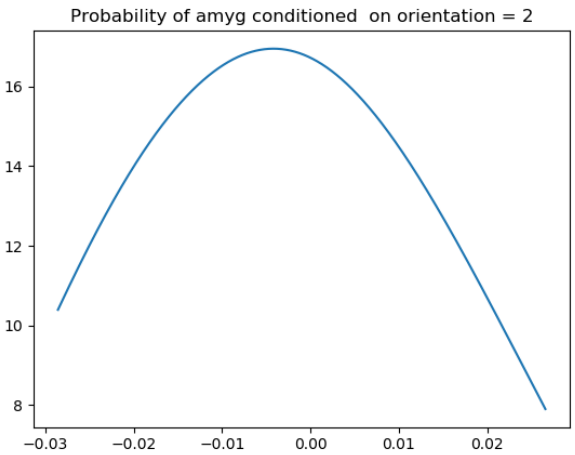
```
Orient_Condi_prob(n90pol,2,'acc')
Orient_Condi_prob(n90pol,3,'acc')
Orient_Condi_prob(n90pol,4,'acc')
Orient_Condi_prob(n90pol,5,'acc')

Orient_Condi_prob(n90pol,2,'amyg')
Orient_Condi_prob(n90pol,3,'amyg')
Orient_Condi_prob(n90pol,4,'amyg')
Orient_Condi_prob(n90pol,5,'amyg')
```

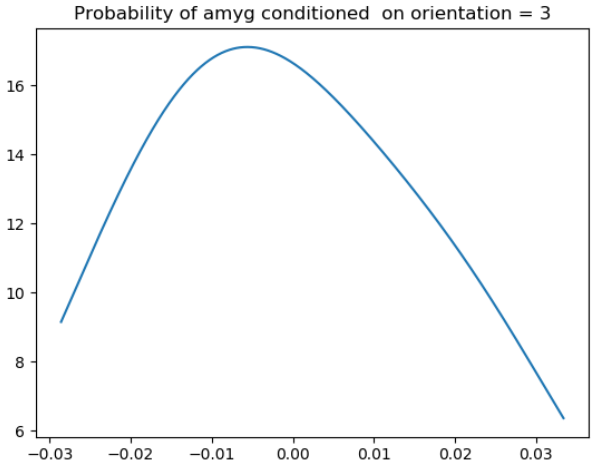

Probability of acc conditioned on orientation = 2
16
14
12
10
8
−0.03
−0.02
−0.01
0.00
0.01
0.02

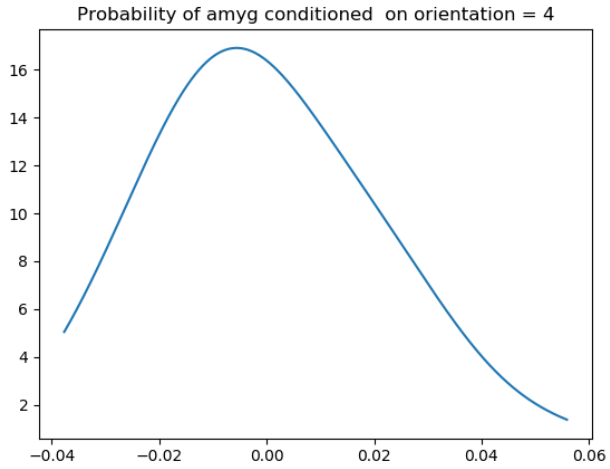

Probability of acc conditioned on orientation = 3
16
14
12
10
8
6
−0.03
−0.02
−0.01
0.00
0.01
0.02
0.03

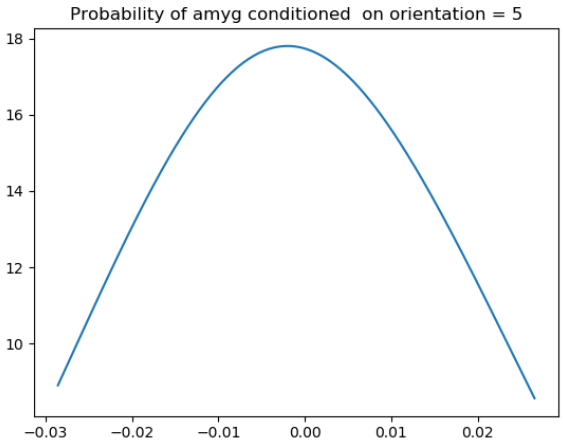

Probability of acc conditioned on orientation = 4
16
14
12
10
8
6
4
2
−0.04
−0.02
0.00
0.02
0.04
0.06


Probability of acc conditioned on orientation = 5
18
16
14
12
10
−0.03
−0.02
−0.01
0.00
0.01
0.02


Probability of amyg conditioned on orientation = 2
16
14
12
10
8
−0.03
−0.02
−0.01
0.00
0.01
0.02


Probability of amyg conditioned on orientation = 3
16
14
12
10
8
6
−0.03
−0.02
−0.01
0.00
0.01
0.02
0.03





e)

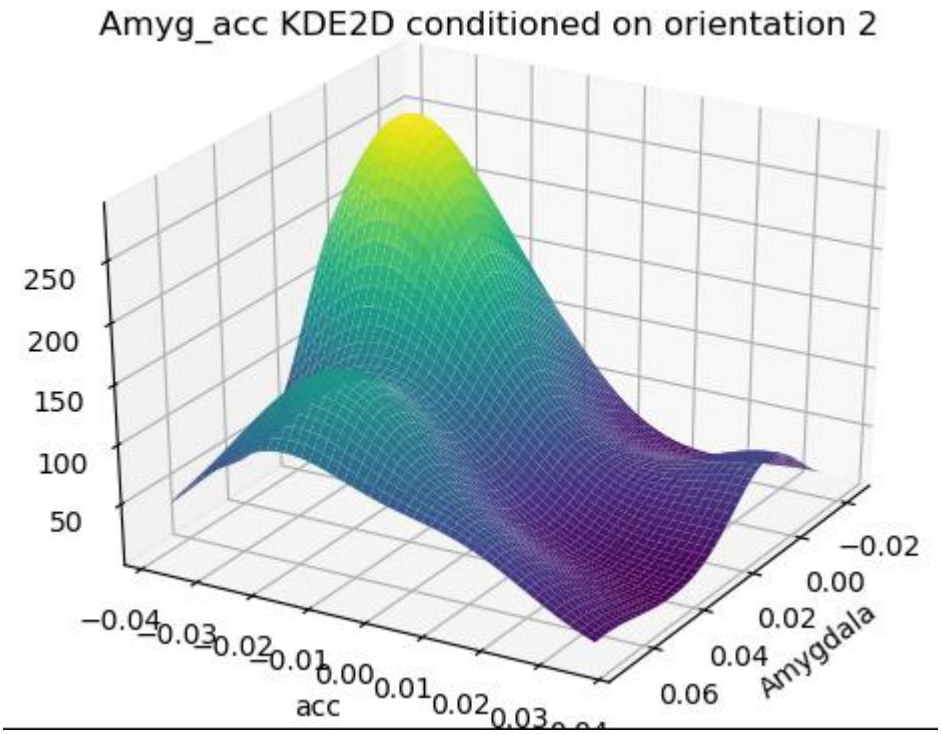
We try to estimate the conditional joint distribution of the volume of the amygdala and acc, conditioning on a function of political orientation using 2D KDE.

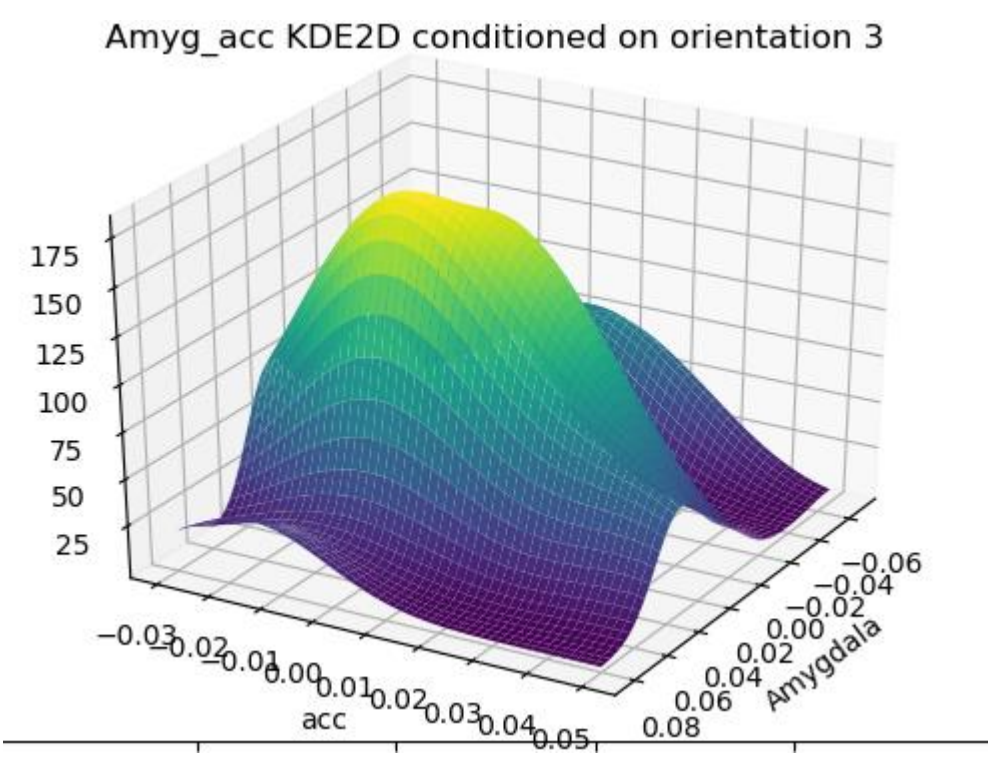
We compute this function:

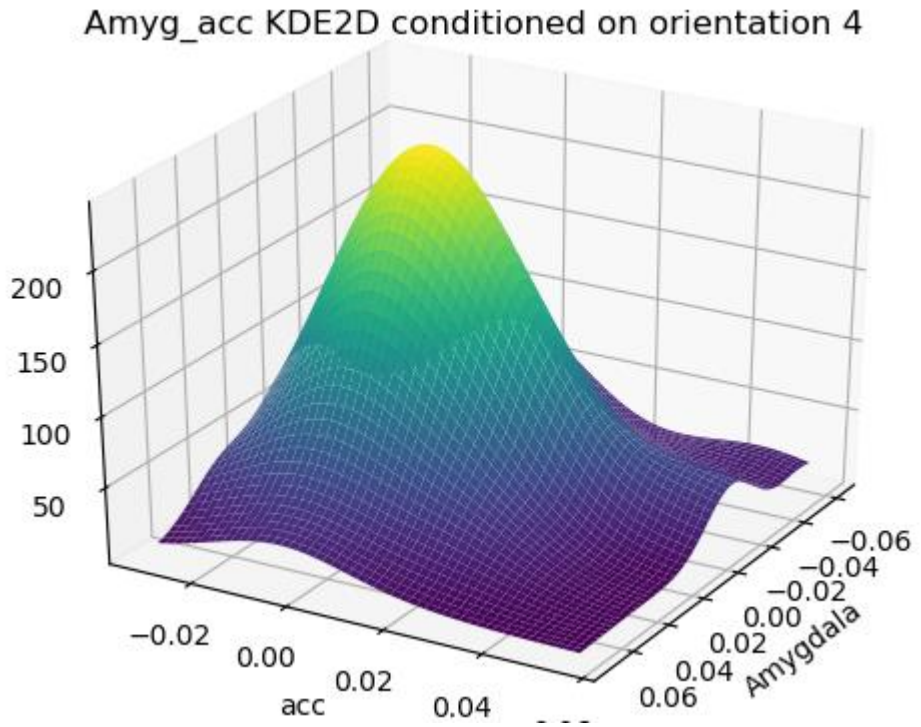
We first try to get the volume of the acc and amygdala that match the desired orientation. And then we fit the data to a 2D KDE.

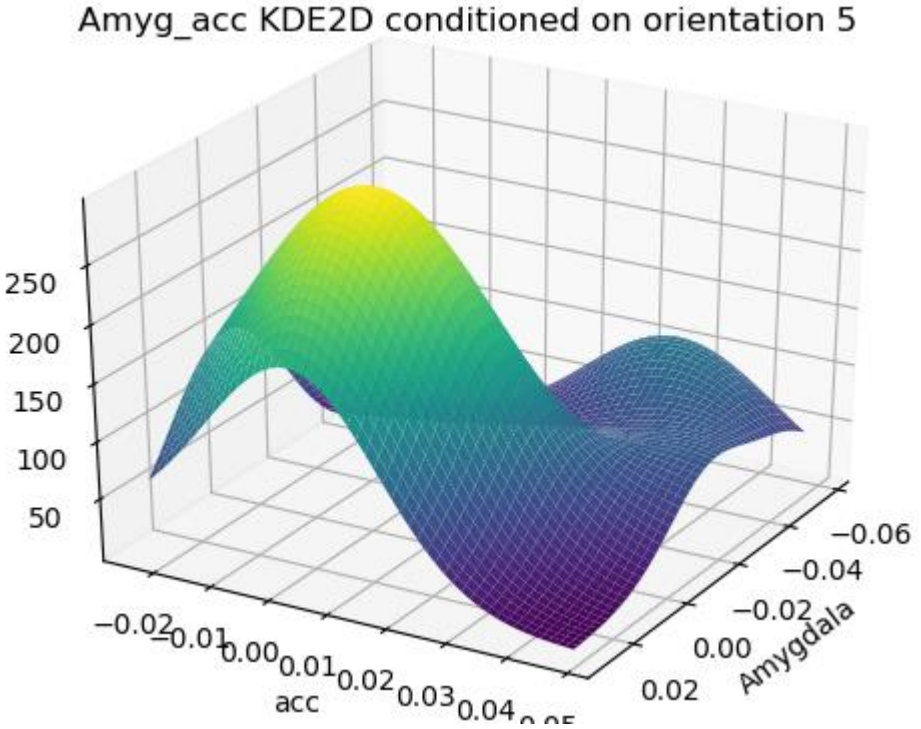
```
def KDE2D_orientation(band_w,Data,orientation):
    count=0
    x=[]
    y=[]
    for orient in Data.iloc[:,2]:
        if orient==orientation:
            x.append(Data.iloc[count,0])
            y.append(Data.iloc[count,1])
        count+=1
    x=np.asarray(x)
    y=np.asarray(y)
    xy = np.vstack([x,y])
    fig,ax=plt.subplots()
    kde = KernelDensity(bandwidth=band_w, metric='euclidean',
                        kernel='gaussian',
                        algorithm='ball_tree').fit(xy.T)
    X, Y = np.mgrid[min(x):max(x):1000j, min(y):max(y):1000j]
    pos = np.vstack([X.ravel(), Y.ravel()])
    A = np.reshape(np.exp(kde.score_samples(pos.T)), X.shape)
    ax=fig.gca(projection='3d')
    ax.view_init(elev=30,azim=30)
    ax.plot_surface(X,Y,A,cmap=plt.cm.viridis)

    plt.xlabel('Amygdala')
    plt.ylabel('acc')
    plt.title('Amyg_acc KDE2D'+ ' conditioned on orientation {}'
              .format(orientation))

    return A
```

We obtain:


Amyg_acc KDE2D conditioned on orientation 2
250
200
150
100
50
−0.04
−0.03
−0.02
−0.01
0.00
0.01
0.02
0.03
acc
−0.02
0.00
0.02
0.04
0.06
Amygdala


Amyg_acc KDE2D conditioned on orientation 3
175
150
125
100
75
50
25
−0.03
−0.02
−0.01
0.00
0.01
0.02
0.03
0.04
0.05
acc
−0.06
−0.04
−0.02
0.00
0.02
0.04
0.06
0.08
Amygdala

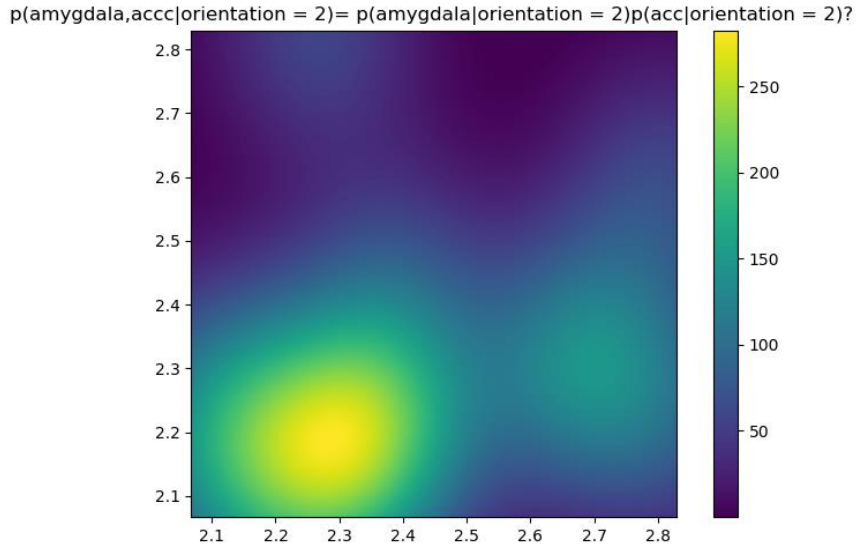

Amyg_acc KDE2D conditioned on orientation 4
200
150
100
50
−0.02
0.00
0.02
0.04
acc
−0.06
−0.04
−0.02
0.00
0.02
0.04
0.06
Amygdala

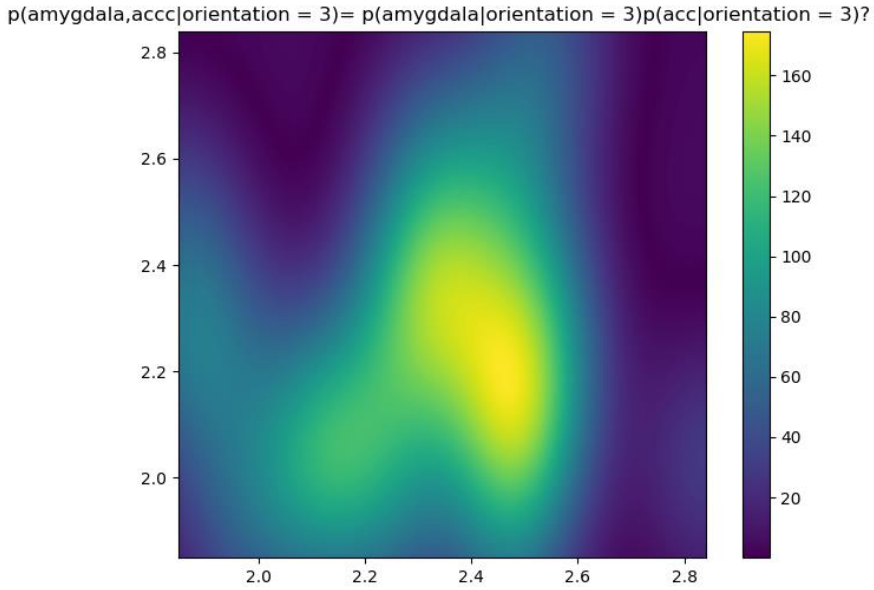

Amyg_acc KDE2D conditioned on orientation 5
250
200
150
100
50
−0.02
−0.01
0.00
0.01
0.02
0.03
0.04
acc
−0.06
−0.04
−0.02
0.00
0.02
Amygdala

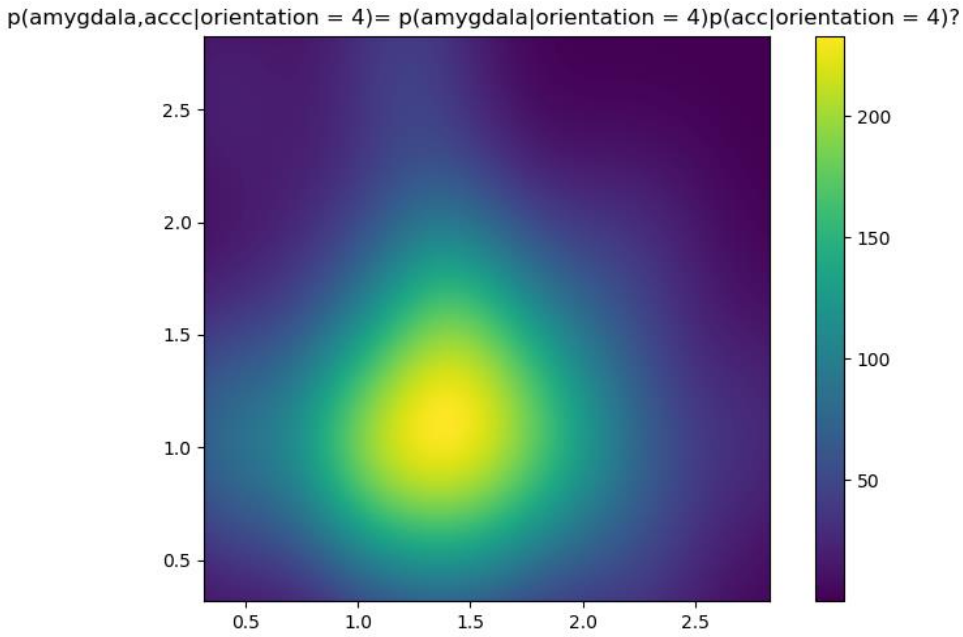
f)

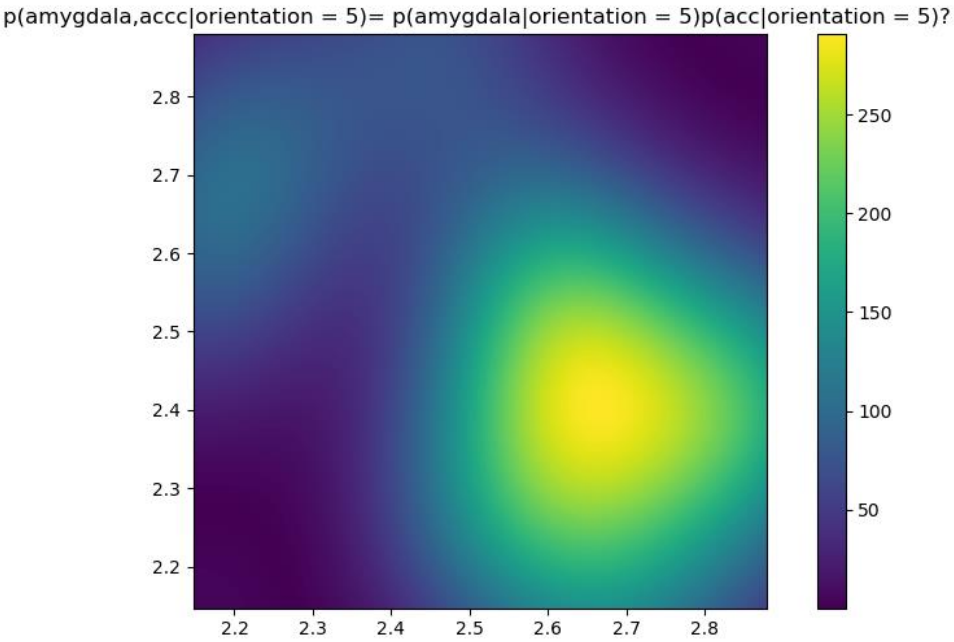
We have compute this function which basically is the same as the Independent one:

```
def independant_conditionned(n90pol,XY,X,Y,orientation):
    X_Y=np.zeros((len(X),len(X)))
    for i in range (0,len(X)):
        for j in range(0,len(X)):
            X_Y[i][j]=X[i]*X[j]
    Inde=abs(np.asarray(XY)-X_Y)
    fig,ax_1=plt.subplots()

    im=ax_1.imshow(np.rot90(Inde),
                   extent=[min(X), max(X), min(Y), max(Y)])
    fig.colorbar(im)
    plt.title('p(amygdala,accc|orientation = {})= p(amygdala|orientation
    return Inde
```









Conditionned on the political orientation, the two parts of the brain are still not independent as we get a non-close-to zero plot. This means for this example that our way of thinking and reflecting is independent of our political view. Our brain works independently from our political orientation.

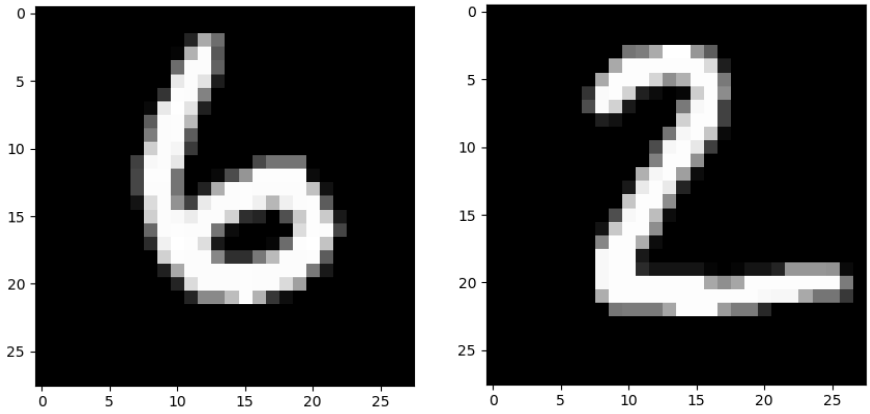
2. Implementing EM for MNIST dataset, with PCA for dimensionality reduction

a)

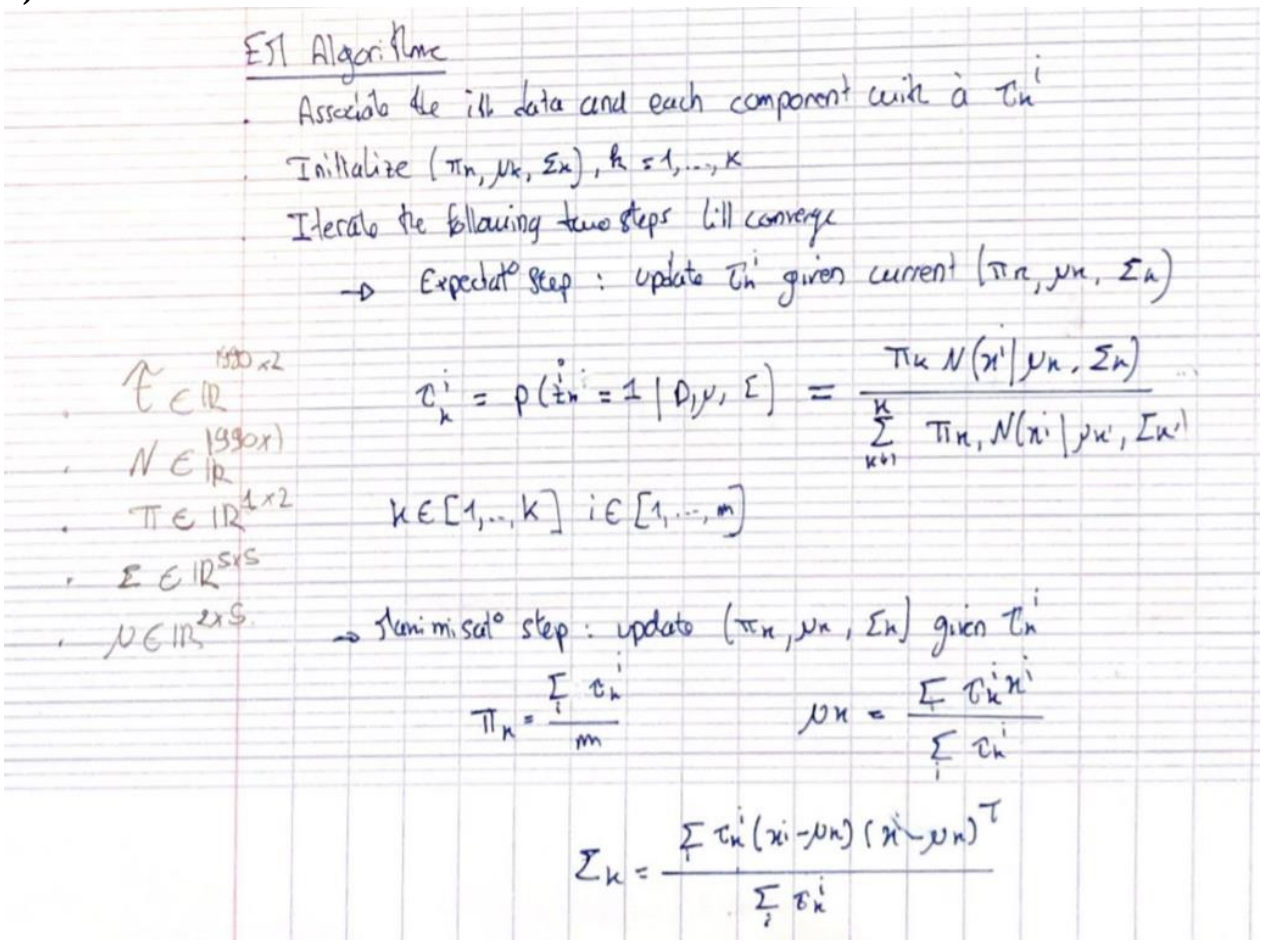
After importing the images we plot a 2 and a 6 :

```
from PIL import Image
two =np.reshape(np.asarray(data_set_data.iloc[:,0]),(28,28))
six =np.reshape(np.asarray(data_set_data.iloc[:,1060]),(28,28))

img_two=Image.fromarray(np.uint8(two*255).T,'L')
img_six=Image.fromarray(np.uint8(six*255).T,'L')
#plt.imshow(img_two,cmap='gray',vmin=0, vmax=255)
#plt.imshow(img_six,cmap='gray',vmin=0, vmax=255)
```

We get:



b)



c)

After pre-processing the data, we initialize the mean, the pi, the sigma and the tau as follow (with C=2)

```
pi_1= 0.5
pi_2= 0.5

mean_1=np.random.randn(5)
mean_2=np.random.randn(5)

Sigma1=np.random.randn(5,5)
Sigma2=np.random.randn(5,5)

Sigma1=Sigma1.T.dot(Sigma1)+np.identity(5)
Sigma2=Sigma2.T.dot(Sigma2)+np.identity(5)
```

Then, we create the M_step function:

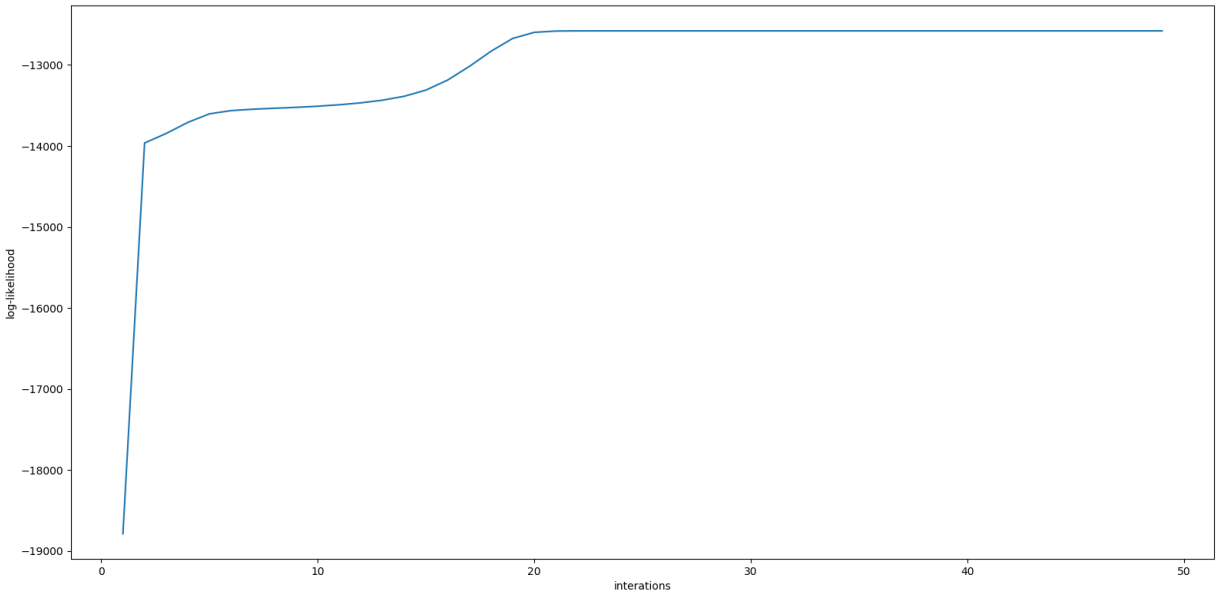
*

```
def M_step(Projected_data,tau_0,tau_1,Sigma1,Sigma2,mean_1,mean_2,pi_1,pi_2):

    pi_1=np.sum(tau_0)/nbr_row
    pi_2=np.sum(tau_1)/nbr_row

    mean_1=tau_0.T.dot(np.asarray(Projected_data))/np.sum(tau_0)
    mean_2=tau_1.T.dot(np.asarray(Projected_data))/np.sum(tau_1)


    Centered_data_1=np.asarray(Projected_data-np.tile(mean_1,(nbr_row,1)))
    Centered_data_2=np.asarray(Projected_data-np.tile(mean_2,(nbr_row,1)))

    Sigma1=0
    Sigma2=0
    for i in range (0,1990):
        Sigma1+=tau_0.tolist()[i][0]*np.reshape(Centered_data_1[i],(5,1)).dot(np.reshape(Centered_data_1[i],(5,1)).T)
        Sigma2+=tau_1.tolist()[i][0]*np.reshape(Centered_data_2[i],(5,1)).dot(np.reshape(Centered_data_2[i],(5,1)).T)

    Sigma1=Sigma1/np.sum(tau_0)
    Sigma2=Sigma2/np.sum(tau_1)


    return pi_1,pi_2,mean_1,mean_2,Sigma1,Sigma2
```

Finally, we set up a loop to iterate the E_step and the M_step as follow :

```
f=[]
X=[j for j in range(1,50)]

for loop in range(0,50):
    a=mvn.pdf(Projected_data,np.asarray(mean_1.tolist()).ravel(), Sigma1)
    b=mvn.pdf(Projected_data,np.asarray(mean_2.tolist()).ravel(), Sigma2)

    if loop>0:
        f.append(np.sum(np.log(pi_1*a+pi_2*b)))

        tau_0= pi_1 * a/(pi_1*a+pi_2*b)
        tau_0=np.reshape(tau_0, (1990,1))
        tau_1=pi_2 * b/(pi_1*a+pi_2*b)
        tau_1=np.reshape(tau_1, (1990,1))

        pi_1,pi_2,mean_1,mean_2,Sigma1,Sigma2= M_step(Projected_data,tau_0,tau_1,Sigma1,Sigma2,mean_1,mean_2,pi_1,pi_2)
        print('loop {}'.format(loop))


plt.figure()
plt.plot(X,f)
plt.xlabel('interations')
plt.ylabel('log-likelihood')
plt.show()
```

For 50 iterations we get the following graph of the likelihood function:



We can see that the algorithm is converging to a value that is close to -1250.

d)

Weigh of each component:

Pi_1 = 0.49313428755452005

Pi_2= 0.50686571244548

tau_0= array([[9.72233136e-01] , [1.00000000e+00], [9.84633391e-01], ..., [1.87174295e-08]])

tau_1=([[2.77668639e-02], [4.63038039e-10], [1.53666094e-02], ...,[9.99999981e-01]])

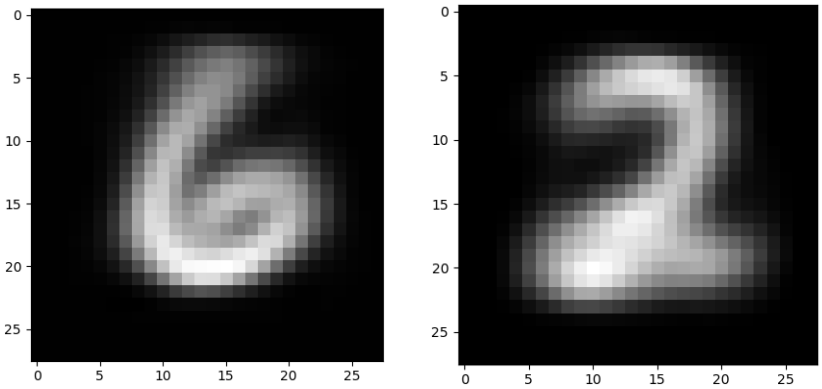
To reconstruct the data we first compute:

- Matrix of eigenvector
- Diagonal matrix with the eigenvalues
- Mean of the data for C=1 and 2

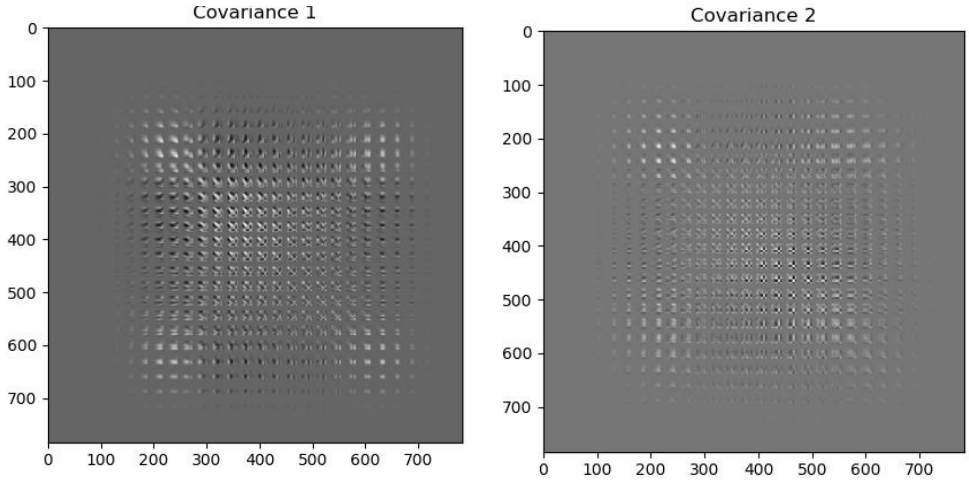
```
#c)
Diag_matrix=np.diag([eig**(1/2) for eig in Eigen_values])
U_T=  Eigen_vector

reconstruc_data=[]
for i in range (0,nbr_row):
    reconstruc_data.append(np.asarray(Projected_data.iloc[i,:].dot(Diag_matrix.dot(U_T))+ mean_data))

Reconstruct_data=pd.DataFrame(np.asarray(reconstruc_data))


Cov1=U_T.T@Sigma1@U_T
Cov2=U_T.T@Sigma2@U_T

mean1=(np.asarray(mean_1)@Diag_matrix@U_T).T.ravel() +mean_data
mean2=(np.asarray(mean_2)@Diag_matrix@U_T).T.ravel() +mean_data

two_reconstruct =np.reshape(np.asarray(mean1),(28,28))
six_reconstruct =np.reshape(np.asarray(mean2),(28,28))
```

Then we can plot the average reconstructed images:

```
plt.figure()
plt.imshow(two_reconstruct.T,cmap='gray')
plt.figure()
plt.imshow(six_reconstruct.T,cmap='gray')

Cov1_reconstruct=Image.fromarray(np.uint8(Cov1*255).T,'L')
Cov2_reconstruct=Image.fromarray(np.uint8(Cov2*255).T,'L')

plt.figure()
plt.imshow(Cov1,cmap='gray')
plt.title("Covariance 1")
plt.figure()
plt.imshow(Cov2,cmap='gray')
plt.title("Covariance 2")
```



We also get the covariance matrix associated:



e)

We first implemented an augmented matrix in the previous data set with the values of tau1 and tau2 and the true label:

```
tau1=tau[:,0]
tau2=tau[:,1]


Augmented_data=data_set_data.T
Augmented_data['tau1']=tau1
Augmented_data['tau2']=tau2
Augmented_data['True_label']=data_set_label.T
```

Using the library Sklearn we import K-means and compute the algorithm on the previous data set:

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=2, random_state=0).fit(data_set_data.T)


nbr_2=np.asarray(data_set_label).tolist()[0].count(2)
nbr_6=np.asarray(data_set_label).tolist()[0].count(6)

found_2=0
found_6=0
ite=0
for i in kmeans.labels_.tolist():
    if i==1 and np.asarray(data_set_label).tolist()[0][ite]==2:
        found_2+=1
    if i==0 and np.asarray(data_set_label).tolist()[0][ite]==6:
        found_6+=1
    ite+=1

k_means_miss=[ found_2/nbr_2,found_6/nbr_6]
```

We count the number of 2 and 6 that correspond to the true labels. Then we divide this amount by the number of “True 2” and “True 6” in the label data set. We finally get this ratio:

[found_2/nbr_2, found_6/nbr_6] = [0.9457364341085271, 0.9290187891440501]

For the GMM, if tau1 >tau2 then the data point is associated with a 2 and if tau2>tau1 the data point is a 6. Then we can do the same thing that we have done for the K-means:

```
found_2_G=0
found_6_G=0
row=0
for i in Augmented_data['True_label']:
     if i==2 and Augmented_data.iloc[row,784]> Augmented_data.iloc[0,785]:
        found_2_G+=1
     if i==6 and Augmented_data.iloc[row,784]< Augmented_data.iloc[0,785]:
        found_6_G+=1
     row+=1

GMM_miss=[ found_2_G/nbr_2,found_6_G/nbr_6]
```

We obtain the following ration:

[found_2_G/nbr_2, found_6_G/nbr_6] = [0.9903100775193798, 0.9592901878914405]

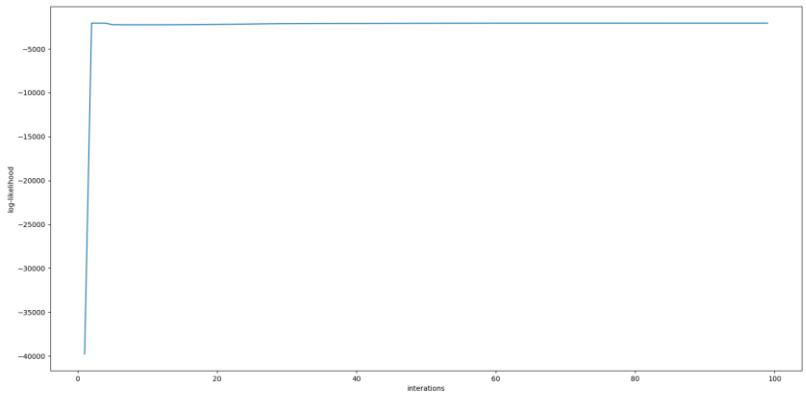
We clearly see that the GMM is more accurate than the K-means algorithm.

BONUS

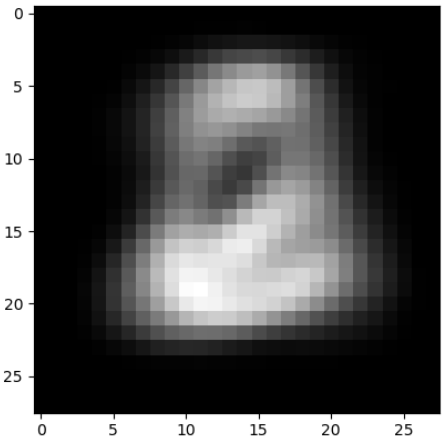
Here is the function to compute the multivariate normal distribution with the low-rank approximation of the covariance matrix Sigma:

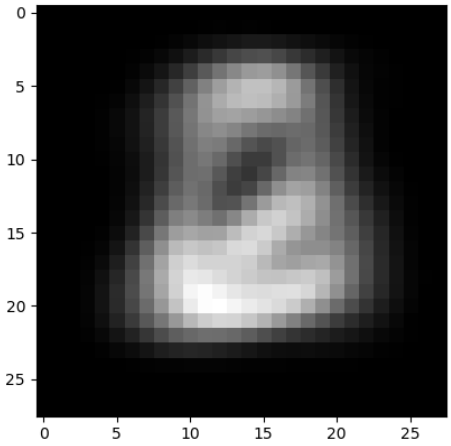
```
def new_gaussian(Sigma,mean,data_set_data):
    Eigen_vector, Eigen_values=Eigen_Value_Vect_selected(Sigma,5)
    Centered_data=np.asarray(data_set_data.T-np.tile(mean,(1990,1)))
    Product=Eigen_vector.dot(Centered_data.T)
    Diag_values=np.diag([eig**(-1/2) for eig in Eigen_values])
    P=Diag_values.dot(Product)
    PP=[]
    for i in P.T :
        PP.append(np.linalg.norm(i))
    PP=np.asarray(PP)


    a=np.exp(PP)/np.prod( Eigen_values)**(1/2)
    return a
```

Using this code and adapting the other lines of code with the new dimension we get :



Which is not relevant because we get these means images for 100 iterations :





I did not had time to find the problem in my code. But try to do the maximum then I could for the deadline.