```python
In [2]: import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
from scipy.stats import multivariate_normal as mn
from scipy.stats import chi2 as chi2
import pandas as pd
import math
```

Exercise 1: Discrete distributions

1. Explain how to generate a random variable X having the discrete distribution on X given by (pi)1≤i≤n

We construct the segment [0,1] by concatenating the bins of size pi respectivly. Then We pick a random real number r between [0,1], we return the x_i corresponding to the bin containing r.

2. Write (in Matlab, Python, Octave. . .) the corresponding algorithm.

```python
In [3]: def simulateRV(probabilities,values,n):
    bins = np.add.accumulate(probabilities)
    return values[np.digitize(np.random.rand(n),bins)-1]
```

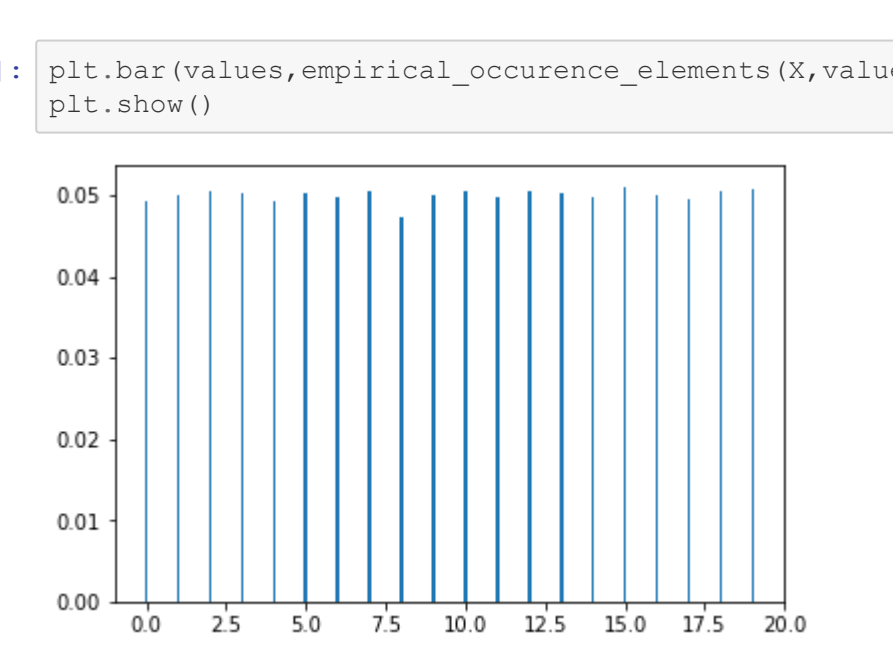
```python
In [233]: def empirical_occurence_elements(X,values):
    Z=[0]*len(values)
    for i in range(len(values)):
        for j in range(len(X)):
            if X[j]==values[i]:
                Z[i]+=1
    return np.array(Z)/len(X)
```

3. Generate a sequence (Xi)1≤i≤N of i.i.d. random variables having the same distribution as X for large values of N. Compare the empirical distribution to the theoretical distribution of X. (In Matlab, you can use the function histogram).

```python
In [938]: probabilities=[1/20]*20
values=np.arange(20)
X=simulateRV(probabilities,values,100000)
empirical_distribution=empirical_occurence_elements(X,values)
print(X[0:10])
print(empirical_distribution)
```

```
[13 13  4 16 11  8 15  9  3 18]
[ 0.04926  0.04994  0.05039  0.05017  0.04931  0.05026  0.04969  0.05054
  0.0474   0.05006  0.05061  0.04983  0.05042  0.0503   0.04987  0.05106
  0.04997  0.04956  0.0506   0.05076]
```

```python
In [939]: plt.bar(values,empirical_occurence_elements(X,values),width=0.1)
plt.show()
```



For N=400, The empirical distribution is not far from the theoritical one.

Exercise 2: Gaussian mixture model and the EM algorithm

1. Identify the parameters, denoted θ, of the model and write down the likelihood of θ given the outcomes (xi)i of the i.i.d n-sample (Xi)i, i.e the p.d.f of (Xi)i

Please see in the attached pdf scanned images.

2. Sample a set of observation according to a Gaussian mixture law, with the parameters of your choice. Use the hierarchical model and the first exercise.

```python
In [238]: probabilities=np.array([0.5,0.25,0.25])
values=np.array([0,1,2])
gaussian_means=np.array([[10,2],[0,1],[3,6]])
gaussian_variances=np.array([np.eye(2),[[5,0],[0,3]],[[10,1],[1,9]]])

simulation=simulateRV(probabilities,values,400)
data=[]
for i in range(len(simulation)):
    data.append(list(np.random.multivariate_normal(gaussian_means[simulation[i]],gaussian_vari
ances[simulation[i]])))
data=np.array(data)
```

We consider a mixture of 3 gaussians, to generate the data, we first simulate a random variable having the distribution 'probabilities' using the exercise 1 and then use the correspondant gaussian to generate a point in R^2.

3. Implement the EM algorithm in order to estimate the parameters of this model from your obser- vations and plot the log-likelihood over the number of iteration of the algorithm.

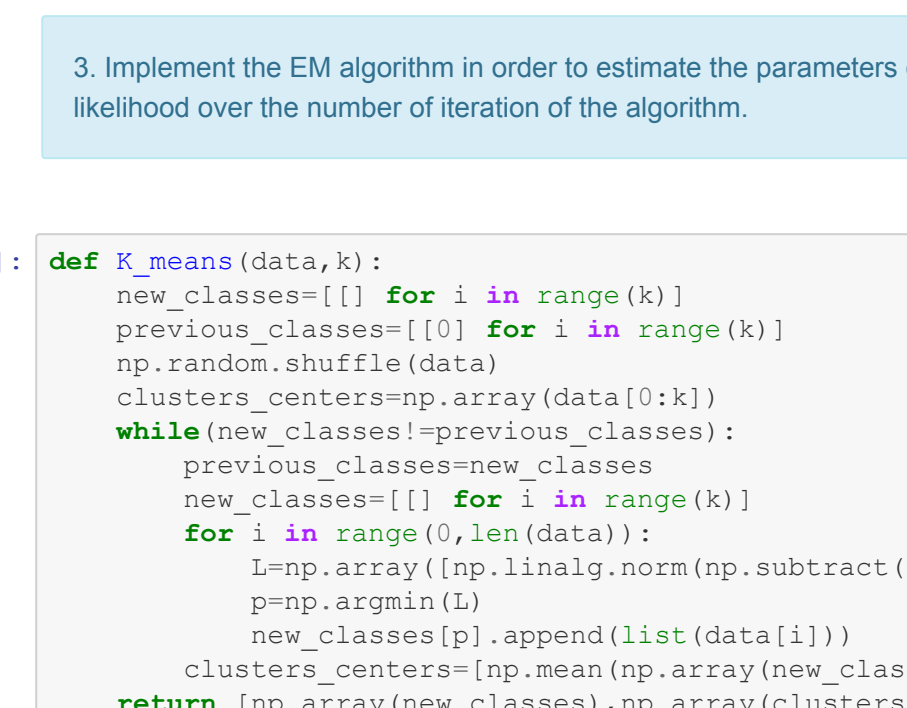
```python
In [257]: def K_means(data,k):
    new_classes=[[] for i in range(k)]
    previous_classes=[[0] for i in range(k)]
    np.random.shuffle(data)
    clusters_centers=np.array(data[0:k])
    while(new_classes!=previous_classes):
        previous_classes=new_classes
        new_classes=[[] for i in range(k)]
        for i in range(0,len(data)):
            L=np.array([np.linalg.norm(np.subtract(clusters_centers,data[i]),axis=1)])
            p=np.argmin(L)
            new_classes[p].append(list(data[i]))
        clusters_centers=[np.mean(np.array(new_classes[j]),axis=0) for j in range(k)]
    return [np.array(new_classes),np.array(clusters_centers)]

def intializationStep(data,k):
    k_m=K_means(data,k)
    gaussian_means=k_m[1]
    cov=np.array([np.sum([list(np.dot(np.array([k_m[0][i][j]-gaussian_means[i]]).T,[k_m[0][i][
j]-gaussian_means[i]])) for j in range(len(k_m[0][i]))],axis=0)/len(k_m[0][i]) for i in range(
k)])
    weights=[len(k_m[0][i])/len(data) for i in range(k)]
    return [gaussian_means,cov,weights]

def stepE(data,means,cov,weights,k):
    g_w=[[weights[j]*mn.pdf(data[i],means[j],cov[j]) for j in range(k)] for i in range(len(dat
a))]  #weight*Gaussian
    resp=np.array([[g_w[i][j]/np.sum(g_w[i]) for j in range(k)] for i in range(len(data))]) #R
esponsabilities(Posteriors)
    log_llh=np.sum([math.log2(np.sum(g_w,axis=1)[i]) for i in range(len(data))]) #Loglikelihoo
d
    return [resp,log_llh]


def stepM(data,resp,k):
    resp_sum=np.sum(resp,axis=0) #Summation over data for each i dans {1,...,k}
    weights=resp_sum/len(data)
    means=np.divide(np.dot(resp.T,data),resp_sum[:,None])
    d=np.shape(means)[1]
    cov=np.array([np.sum([resp[:,i][j]*np.dot(np.array([data[j]-means[i]]).T,[data[j]-means[i
]]) for j in range(len(data))],axis=0)/resp_sum[i] for i in range(k)])
    return [means,cov,weights]


def algorithmEM(data,k):
    [means,covariances,weights]=intializationStep(data,k)
    [responsabilities,log_llh]=stepE(data,means,covariances,weights,k)
    loglist=[log_llh]
    log_llh2=log_llh+1
    iteration=0
    while(log_llh2!=log_llh):
        iteration+=1
        log_llh2=log_llh
        [means,covariances,weights]=stepM(data,responsabilities,k)
        [responsabilities,log_llh]=stepE(data,means,covariances,weights,k)
        loglist.append(log_llh)
    return [means,covariances,weights,responsabilities,loglist,iteration]
```

Used K-means to intialize the GMM for fast and consistent convergence.

The function algorithmEM returns the estimated means, covariance matrices, weights and responsabilities (posterior probabilities) after convergence and also the list of log-likelihood calculated over each iteration and finally the number of iterations taken by the while loop till convergence.

```python
In [240]: [means,covariances,weights,responsabilities,loglist,iteration]=algorithmEM(data,3)
```

```python
In [241]: plt.plot(loglist)
plt.grid()
plt.xlabel('iterations')
plt.ylabel('Log-likelihood')
plt.title('Evolution of log-likelihood as a function of iteration until convergence')
plt.show()
```




```python
In [242]: loglist[iteration-10:iteration-1]
```

```
Out[242]: [-2959.1735379946404,
 -2959.1735379946385,
 -2959.1735379946376,
 -2959.1735379946367,
 -2959.1735379946358,
 -2959.1735379946349,
 -2959.173537994634,
 -2959.1735379946335,
 -2959.1735379946331]
```

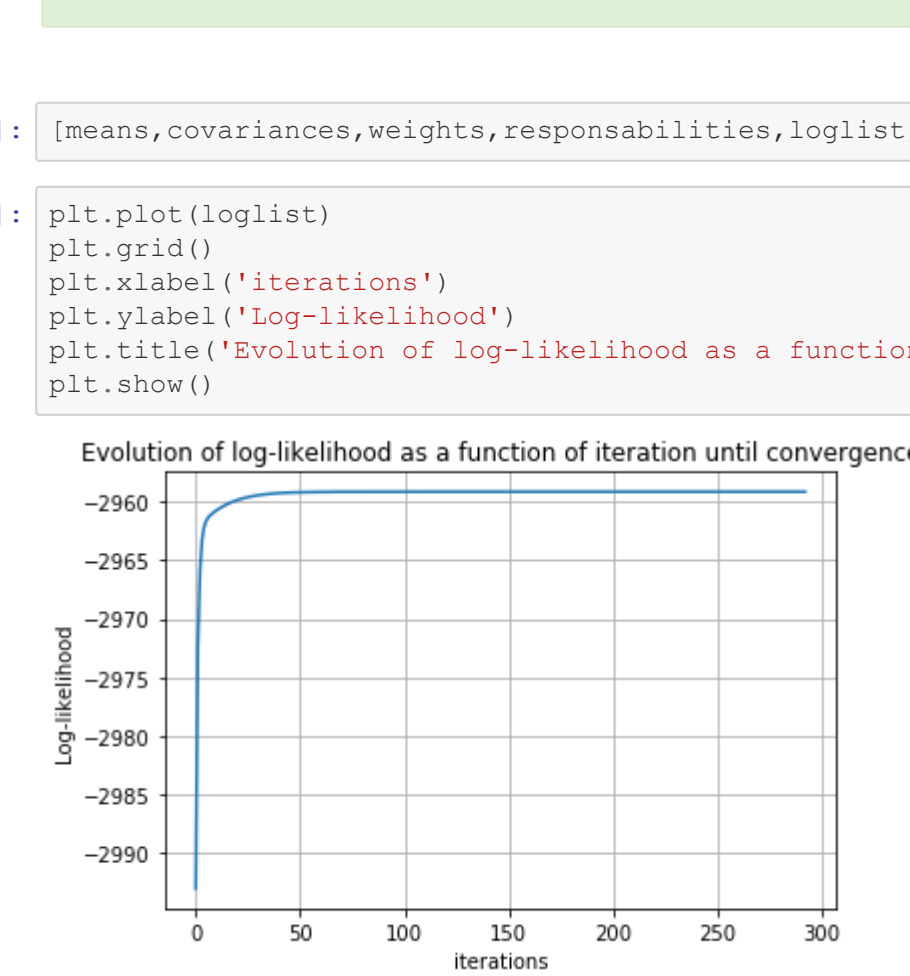
The log-likelihood converges rapidely (around 30 iterations) to the minimal value with 0.001 precision.

4. Are the estimated parameters far from the original ones ?

```python
In [243]: print('The estimated gaussian centers are:', means)
print('The estimated covariance matrices are:', covariances)
print('The estimated gaussian weights are:', weights)
```

```
The estimated gaussian centers are: [[ 3.10957689  6.32613561]
 [-0.14000232  1.14234627]
 [ 9.89306219  1.99913091]]
The estimated covariance matrices are: [[[ 9.43931686  1.10732111]
  [ 1.10732111  9.66169468]]

 [[ 4.94554023  0.13833876]
  [ 0.13833876  2.97770611]]

 [[ 1.16601164  0.08600689]
  [ 0.08600689  1.31552191]]]
The estimated gaussian weights are: [ 0.44570487  0.28839188  0.26590325]
```

```python
In [244]: clusters=[[] for i in range(3)]
for i in range(np.shape(responsabilities)[0]):
    p=np.argmax(responsabilities[i])
    clusters[p].append(data[i])

fig = plt.figure()
ax = fig.add_subplot(111, aspect='auto')
for i in range(3):
    ax.scatter(np.array(clusters[i])[:,0],np.array(clusters[i])[:,1],alpha=0.5)
#Estimated gaussian_means with black cross color
ax.scatter(np.array(means)[:,0],np.array(means)[:,1],c='black',marker='x')
#True gaussian_means with red cross color
ax.scatter(np.array(gaussian_means)[:,0],np.array(gaussian_means)[:,1],c='r',marker='x')
#Estimated 90% of ellipses gaussian_covariances
#with black color
eigenElements=[np.linalg.eig(covariances[i]) for i in range(3)]
quantile=chi2.ppf(0.90, 2)
ellipse_axis=[2*np.sqrt(quantile*eigenElements[i][0]) for i in range(3)]
ellipse_angles=[math.degrees(math.atan2(eigenElements[i][1][0][0],eigenElements[i][1][0][1]))
for i in range(3)]
ellipses=[Ellipse(means[i],ellipse_axis[i][1],ellipse_axis[i][0],ellipse_angles[i]) for i in r
ange(3)]
for e in ellipses:
    e.set_fill(False)
    e.set_linewidth(2)
    e.set_alpha(1)
    ax.add_artist(e)
#True 90% of ellipses gaussian_covariances
#with red color
eigenElements_1=[np.linalg.eig(gaussian_variances[i]) for i in range(3)]
quantile=chi2.ppf(0.90, 2)
ellipse_axis_1=[2*np.sqrt(quantile*eigenElements_1[i][0]) for i in range(3)]
ellipse_angles_1=[math.degrees(math.atan2(eigenElements_1[i][1][0][0],eigenElements_1[i][1][0]
[1])) for i in range(3)]
ellipses_1=[Ellipse(gaussian_means[i],ellipse_axis_1[i][1],ellipse_axis_1[i][0],ellipse_angles
_1[i]) for i in range(3)]
for e in ellipses_1:
    e.set_fill(False)
    e.set_linewidth(2)
    e.set_alpha(1)
    e.set_ec('r')
    ax.add_artist(e)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Comparison between true and estimated variables with different colors for latent va
riables')
plt.show()
```

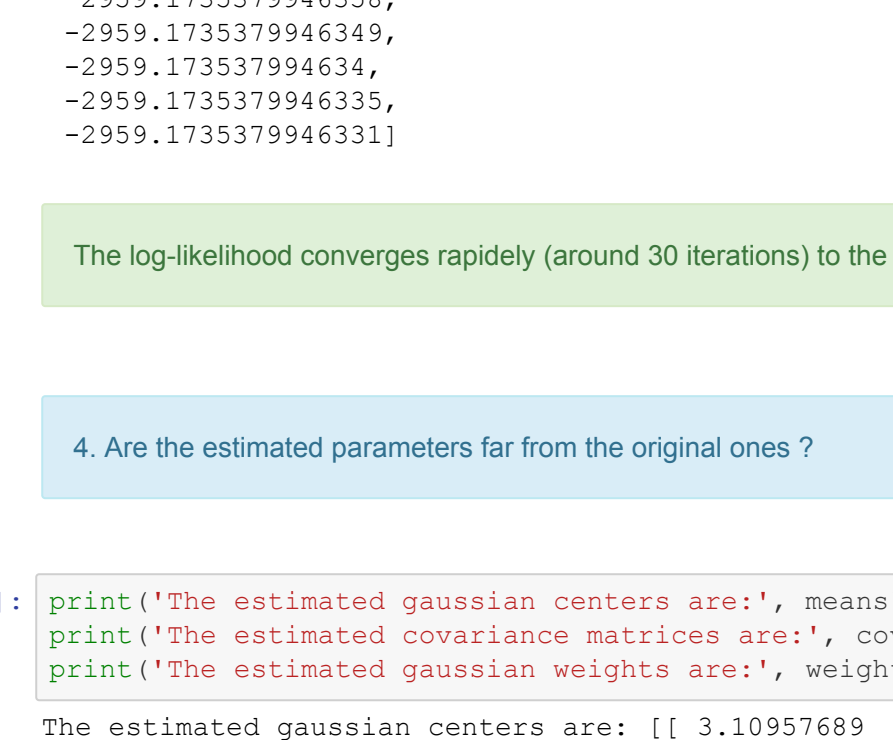


The estimated parameters are in red color and not far from the original ones with black color for means and covariances and also the estimated weights [0.52434153, 0.24785219, 0.22780628] aren't far from the original ones.

5. Application : Download the data Crude Birth/Death Rate – See esa.un.org/unpd/wpp/ for instance – and plot the associated scatter graph. What do you think about using a Gaussian mixture model ?

```python
In [245]: crudedata=pd.read_table('/Users/adilrhiulam/Downloads/Crude-BirthDeath-Rate.csv',sep=',')
```

```python
In [246]: plt.scatter(crudedata.Birth.values,crudedata.Death.values)
plt.xlabel('Crude Birth')
plt.ylabel('Crude Death')
plt.show()
```



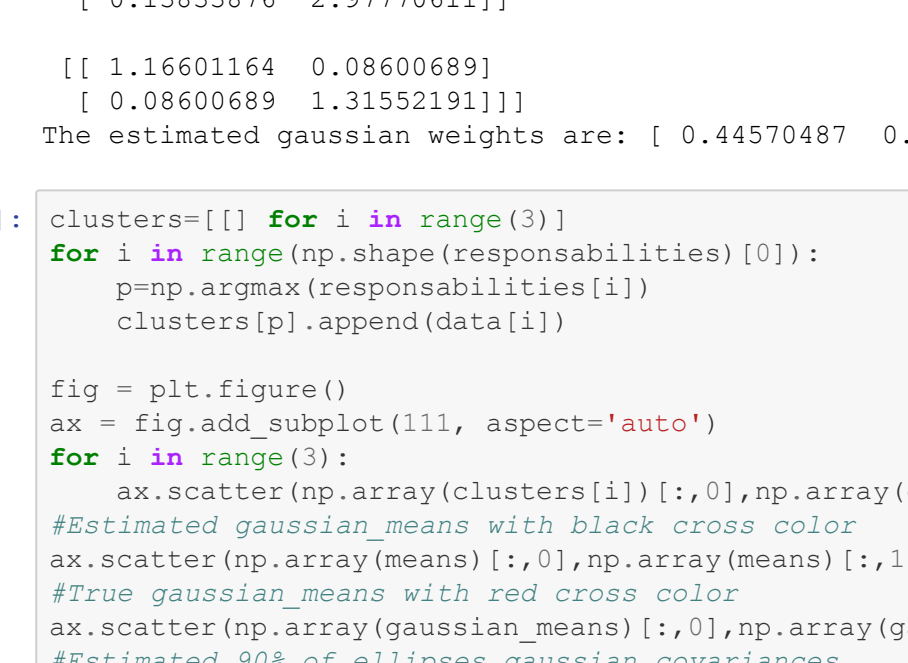
This model seems to have a mixture of gaussian distributions since it has a compact distribution (not a random one).

6. Estimate the parameters θ for different values of M, try to interpret them and compute the BIC. Plot the corresponding p.d.f over the scatter plot. (In Matlab, you can use the functions aicbic and contour).

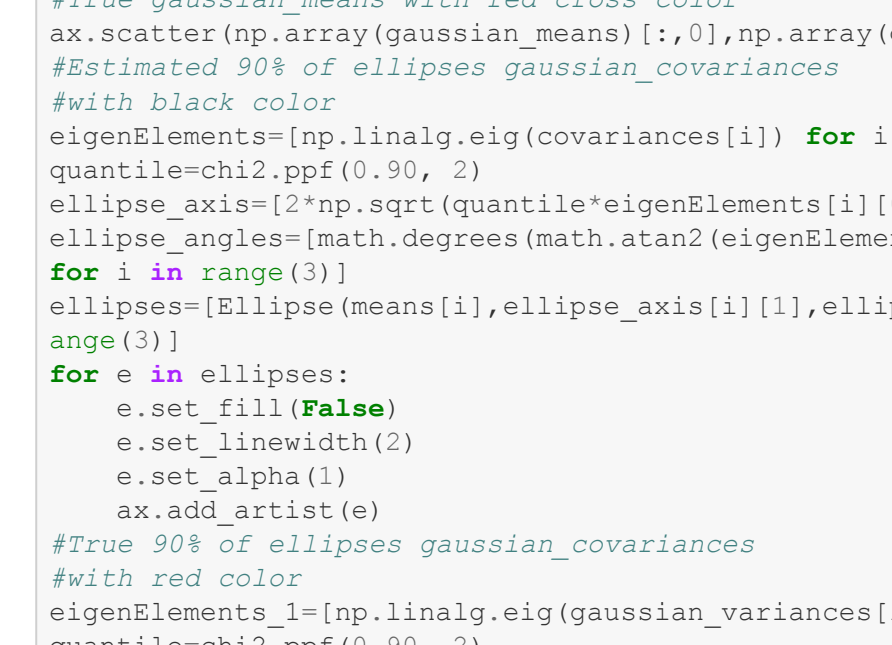
```python
In [247]: crudedata=crudedata.values.tolist()
```

```python
In [248]: def clustersEM(data,k):
    convergenceEM=algorithmEM(data,k)
    resp=convergenceEM[3]
    clusters_center=convergenceEM[0]
    clusters=[[] for i in range(k)]
    for i in range(np.shape(resp)[0]):
        p=np.argmax(resp[i])
        clusters[p].append(data[i])
    return [np.array(clusters),np.array(clusters_center),convergenceEM[1],convergenceEM[4]]

def plotGMM(data,k):
    result=clustersEM(data,k)
    logllh=result[3][-1]
    dim_data=len(data[0])
    df=dim_data*k+(dim_data*(dim_data+1)/2)*k+k #n(n+1)/2 is the number of parameters of a sym
metric
                #covariance matrices and n is the number of parameters of a gaussian means an
d 1 is the
               #dimension of a gaussian weights (we multiply by k for k GM)
    BIC=-logllh+df*math.log2(len(data))/2
    fig = plt.figure()
    ax = fig.add_subplot(111, aspect='auto')
    for i in range(k):
        ax.scatter(np.array(result[0][i])[:,0],np.array(result[0][i])[:,1],alpha=0.5)
    ax.scatter(np.array(result[1])[:,0],np.array(result[1])[:,1],c='black',marker='x')
    eigenElements=[np.linalg.eig(result[2][i]) for i in range(k)]
    quantile=chi2.ppf(0.90, 2)
    ellipse_axis=[2*np.sqrt(quantile*eigenElements[i][0]) for i in range(k)]
    ellipse_angles=[math.degrees(math.atan2(eigenElements[i][1][0][0],eigenElements[i][1][0][1
])) for i in range(k)]
    ellipses=[Ellipse(result[1][i],ellipse_axis[i][1],ellipse_axis[i][0],ellipse_angles[i]) fo
r i in range(k)]
    for e in ellipses:
        e.set_fill(False)
        e.set_linewidth(2)
        e.set_alpha(1)
        ax.add_artist(e)
    plt.xlabel('x')
    plt.ylabel('y')
    num_cluster=str(k)+' clusters'
    title='Data plotted with different colors for latent variables using '+ num_cluster
    plt.suptitle(title)
    plt.title('The corresponding BIC is: '+ str(BIC))
    plt.show()
```

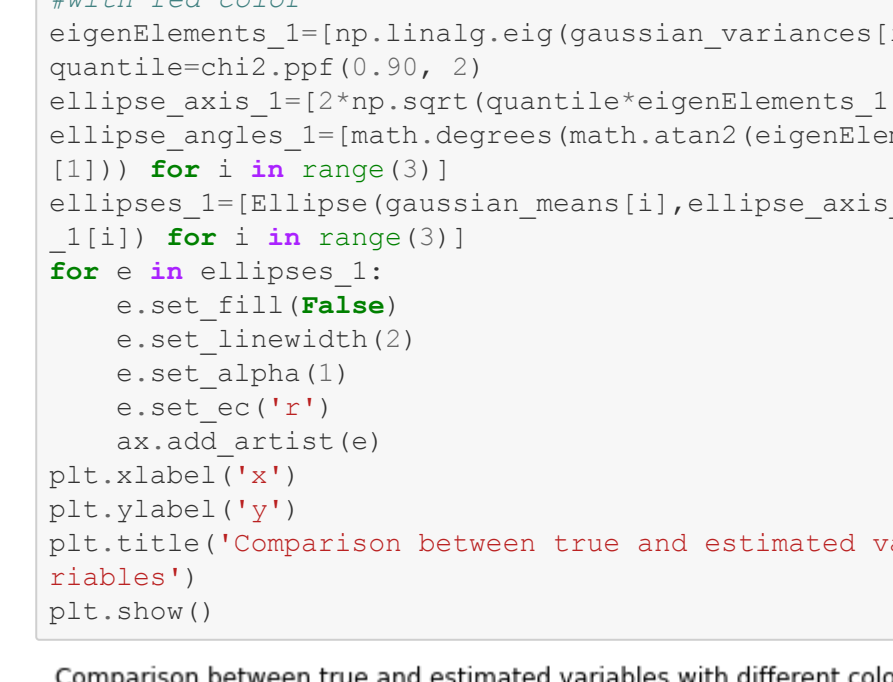
```python
In [249]: plotGMM(crudedata,1)
```



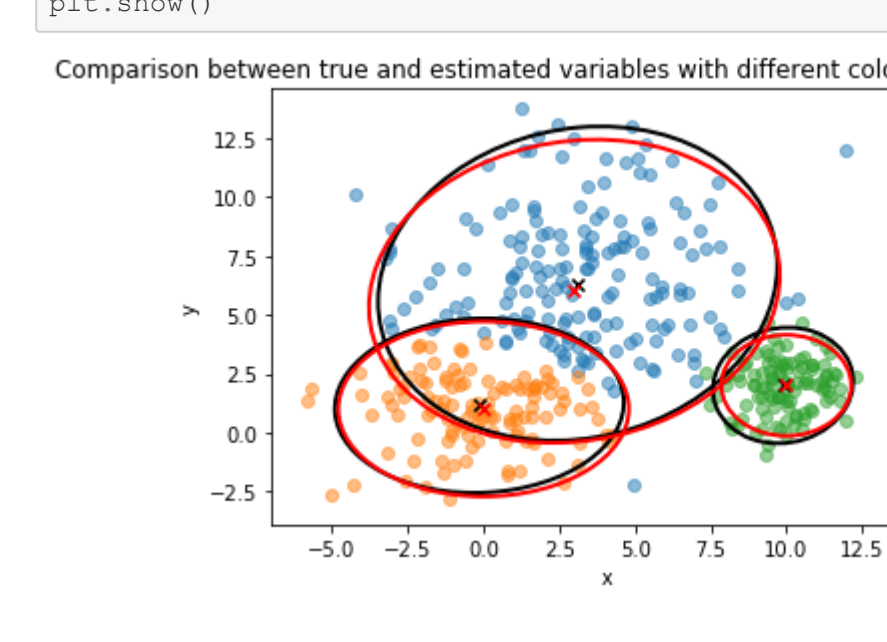

```python
In [250]: plotGMM(crudedata,2)
```



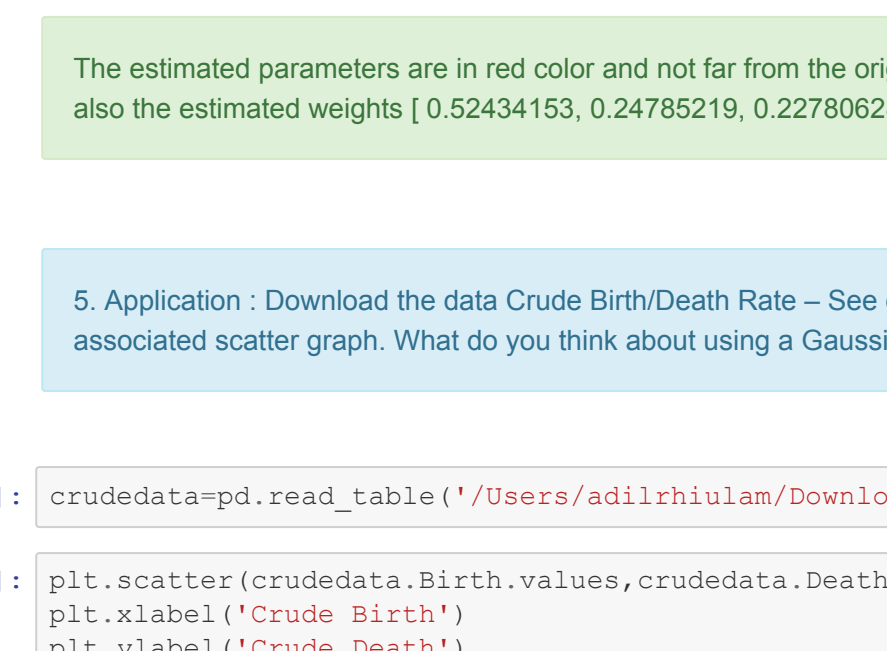

```python
In [251]: plotGMM(crudedata,3)
```



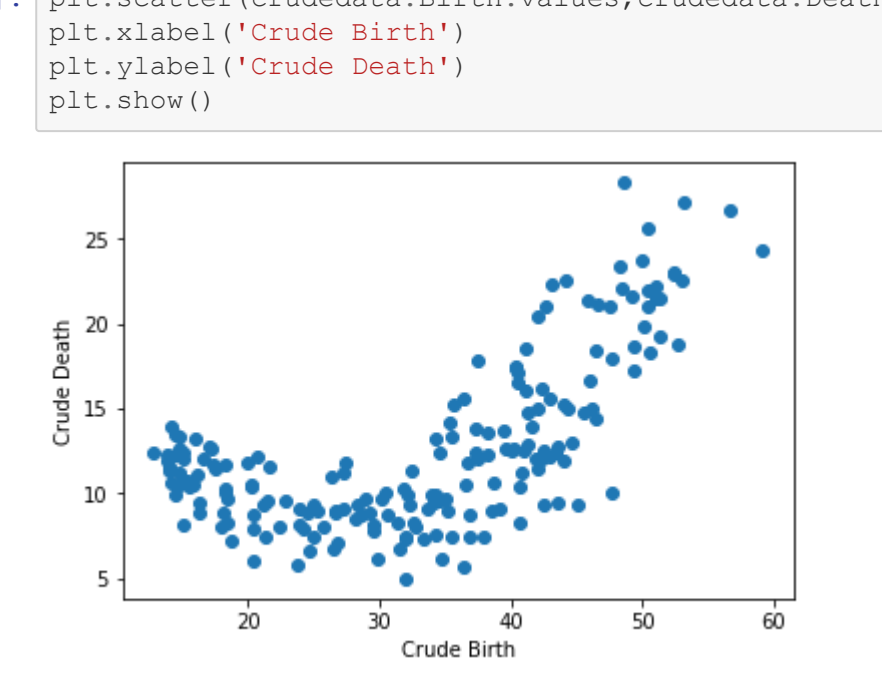

```python
In [252]: plotGMM(crudedata,4)
```




```python
In [279]: plotGMM(crudedata,5)
```




```python
In [280]: plotGMM(crudedata,6)
```



We can notice then that a mixture of gaussians of 3 gives the minimum BIC and succeded by 2,4 GM, then relying on the criterion: minimizing BIC. We conclude that 3 GMM is the best model for our data.

Exercise 3: Importance sampling

3.A – Poor Importance Sampling

1. Implement a simple importance scheme for the previous functions.

```python
In [90]: p = lambda x: (x**(1.65-1)*math.exp(-x**2/2)*int(x>=0)).real
q = lambda x: mn.pdf(x,0.8,1.5)
f = lambda x: 2*math.sin((math.pi/1.5)*x)*int(x>=0)
```

```python
In [99]: def simpleImportanceSampling(n,p,q,f):
    samples=[]
    while(len(samples)<n):
        sample=np.random.normal(0.8,1.5)
        if sample>=0:
            samples.append(sample)
    weights=[round(p(samples[i])/q(samples[i]),2) for i in range(n)]
    expectation_estimate=(1/n)*np.sum([(p(samples[i])*f(samples[i]))/q(samples[i]) for i in ran
ge(n)])
    return [expectation_estimate,weights]
```

2. Compare the estimate and the importance weight for several sample size, N = 10, 100, 10^3, 10^4 for instance.

```python
In [111]: ImpoSam=simpleImportanceSampling(10,p,q,f)
print('The estimate for N=10 is: ',ImpoSam[0])
print('The last 5 importance weights for N=10 are: ',ImpoSam[1][-5:-1])
```

```
The estimate for N=10 is:  0.845834930163
The last 5 importance weights for N=10 are:  [1.6399999999999999, 1.9099999999999999, 0.02,
1.8899999999999999]
```

```python
In [119]: ImpoSam=simpleImportanceSampling(100,p,q,f)
print('The estimate for N=100 is: ',ImpoSam[0])
print('The last 5 importance weights for N=100 are: ',ImpoSam[1][-5:-1])
```

```
The estimate for N=100 is:  0.765778366725
The last 5 importance weights for N=100 are:  [1.3600000000000001, 1.9099999999999999, 1.649
9999999999999, 0.17999999999999999]
```

```python
In [123]: ImpoSam=simpleImportanceSampling(1000,p,q,f)
print('The estimate for N=1000 is: ',ImpoSam[0])
print('The last 5 importance weights for N=1000 are: ',ImpoSam[1][-5:-1])
```

```
The estimate for N=1000 is:  0.864560624729
The last 5 importance weights for N=1000 are:  [0.54000000000000004, 1.9299999999999999, 1.3
500000000000001, 1.49]
```

```python
In [120]: ImpoSam=simpleImportanceSampling(10000,p,q,f)
print('The estimate for N=10000 is: ',ImpoSam[0])
print('The last 5 importance weights for N=10000 are: ',ImpoSam[1][-5:-1])
```

```
The estimate for N=10000 is:  0.854884480305
The last 5 importance weights for N=10000 are:  [0.85999999999999999, 1.6399999999999999, 1.
0700000000000001, 0.44]
```

We notice that for small values of N the Importance Sampling gives values in a large range comparing it with the range of values of the one given by N=1000 or N=10000 which gives every time almost the same value (it converges to its limit when N-->+oo).

The importance weigths are in the same range of values for all N (the value of a weight is independente from N)

3. Shift the mean of q, μ = 6, so that the centers of mass for each distribution are far apart and repeat the experiment.

```python
In [124]: q = lambda x: mn.pdf(x,6,1.5)
```

```python
In [125]: ImpoSam=simpleImportanceSampling(10,p,q,f)
print('The estimate for N=10 is: ',ImpoSam[0])
print('The last 5 importance weights for N=10 are: ',ImpoSam[1][-5:-1])
```

```
The estimate for N=10 is:  19160.5497246
The last 5 importance weights for N=10 are:  [0.0, 200.63, 13629.950000000001, 13.73]
```

```python
In [126]: ImpoSam=simpleImportanceSampling(100,p,q,f)
print('The estimate for N=100 is: ',ImpoSam[0])
print('The last 5 importance weights for N=100 are: ',ImpoSam[1][-5:-1])
```

```
The estimate for N=100 is:  24066.538048
The last 5 importance weights for N=100 are:  [10358.620000000001, 15654.309999999999, 1894
7.84, 1414.0999999999999]
```

```python
In [147]: ImpoSam=simpleImportanceSampling(1000,p,q,f)
print('The estimate for N=1000 is: ',ImpoSam[0])
print('The last 5 importance weights for N=1000 are: ',ImpoSam[1][-5:-1])
```

```
The estimate for N=1000 is:  18866.2982705
The last 5 importance weights for N=1000 are:  [112.27, 128.93000000000001, 30086.5299999999
99, 6697.6099999999997]
```

```python
In [129]: mpoSam=simpleImportanceSampling(10000,p,q,f)
print('The estimate for N=10000 is: ',ImpoSam[0])
print('The last 5 importance weights for N=10000 are: ',ImpoSam[1][-5:-1])
```

```
The estimate for N=10000 is:  18516.3674632
The last 5 importance weights for N=10000 are:  [565.17999999999995, 266.77999999999997, 275
68.619999999999, 31389.139999999999]
```

The same thing that we have said in the previous question applies here also. However, the values of the estimate and the importance weights have become larger (from 0.85 to 18516 for the estimate) which tell us that the importance sampling depends strongly on the choice of the importance density (We are estimating the same expectation but for two different importance densities, we got a large difference between the estimations of the two)

3.B – Adaptative Importance Sampling

4. Explain how the EM algorithm can be used to maximize the empirical criterion in step (iii). Derive the parameters update.

We can see that the update (iii) is similar to the log-likelihood of a mixture of gaussian, but here each random variable Xi which has a mixture of gaussian distribution is weighted with an importance weights, so this can be maximized only using the M step of the EM algorithm, see the attached pdf images for the derivation of parameters update.

5. Using the Adaptive Importance Sampling, write an algorithm which allows to exploring the density p. You may display the results for the banana-shaped density in the first two coordinates.

```python
In [186]: def p(x):
    d,b,var=10,0.03,100
    mean=[0]*d
    x[1]=x[1]+b*(x[1]**2-var)
    sigma=np.eye(d)
    sigma[0,0]=var
    return mn.pdf(x,mean,sigma)
```

```python
In [210]: def intializationStepPMC(M,d=10): #PMC stands for Population Monte Carlo
    alpha=np.array([1/M]*M)
    means=np.random.multivariate_normal([0]*d,np.diag([200,5]+[4]*8),M)
    cov=np.array([np.diag([200,5]+[4]*8)]*M)
    return [alpha,means,cov]

def importance_sampling(alpha,means,cov,n):
    M=len(alpha)
    values=np.arange(M)
    simulation=simulateRV(alpha,values,n)
    data=[]
    for i in range(len(simulation)):
        data.append(list(np.random.multivariate_normal(means[simulation[i]],cov[simulation[i
]])))
    samples=np.array(data)
    q = lambda x: np.sum([alpha[i]*mn.pdf(x,means[i],cov[i]) for i in range(M)])
    impo_weights=np.array([p(samples[i])/q(samples[i]) for i in range(n)])
    normalized_weights=impo_weights/np.sum(impo_weights)
    return [samples,normalized_weights]

def stepE_PMC(samples,means,cov,alpha):
    M=len(alpha)
    n=len(samples)
    g_w=[[alpha[j]*mn.pdf(samples[i],means[j],cov[j]) for j in range(M)] for i in range(n)]  #
weight*Gaussian
    resp=np.array([[g_w[i][j]/np.sum(g_w[i]) for j in range(M)] for i in range(n)]) #Responsab
ilities(Posteriors)
    approximation=np.sum([normalized_weights[i]*math.log2(np.sum(g_w,axis=1)[i]) for i in rang
e(n)])
    return [resp,approximation]


def stepM_PMC(samples,normalized_weights,resp):
    n=len(samples)
    M=np.shape(resp)[1]
    alpha=np.dot(normalized_weights,resp)
    multip=np.array([normalized_weights[i]*np.array(samples[i]) for i in range(n)])
    means=np.array([np.dot(multip.T,resp[:,i])/alpha[i] for i in range(M)])
    cov=np.array([np.sum([normalized_weights[j]*resp[j,k]*np.dot(np.array([samples[j]-means[k
]]).T,[samples[j]-means[k]]) for j in range(n)],axis=0)/alpha[k] for k in range(M)])
    return [alpha,means,cov]


def populationMC(M,n):
    [alpha,means,cov]=intializationStepPMC(M,d=10)
    [samples,normalized_weights]=importance_sampling(alpha,means,cov,n)
    [resp,approximation]=stepE_PMC(samples,means,cov,alpha)
    appr=[approximation]
    approximation2=approximation+1
    iteration=0
    while((approximation2-approximation)>0.01):
        iteration+=1
        approximation2=approximation
        [alpha,means,cov]=stepM_PMC(samples,normalized_weights,resp)
        [samples,normalized_weights]=importance_sampling(alpha,means,cov,n)
        [resp,approximation]=stepE_PMC(samples,means,cov,alpha)
        appr.append(approximation)
    return [means,cov,alpha,resp,appr,iteration]
```

```python
In [211]: [means,cov,alpha,resp,appr,iteration]=populationMC(20,1000)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-211-35f5668f7b18> in <module>()
----> 1 [means,cov,alpha,resp,appr,iteration]=populationMC(20,1000)

<ipython-input-210-6d764996607c> in populationMC(M, n)
     48         approximation2=approximation
     49         [alpha,means,cov]=stepM_PMC(samples,normalized_weights,resp)
---> 50         [samples,normalized_weights]=importance_sampling(alpha,means,cov,n)
     51         [resp,approximation]=stepE_PMC(samples,means,cov,alpha)
     52         appr.append(approximation)

<ipython-input-210-6d764996607c> in importance_sampling(alpha, means, cov, n)
     14     samples=np.array(data)
     15     q = lambda x: np.sum([alpha[i]*mn.pdf(x,means[i],cov[i]) for i in range(M)])
---> 16     impo_weights=np.array([p(samples[i])/q(samples[i]) for i in range(n)])
     17     normalized_weights=impo_weights/np.sum(impo_weights)
     18     return [samples,normalized_weights]

<ipython-input-210-6d764996607c> in <listcomp>(.0)
     14     samples=np.array(data)
     15     q = lambda x: np.sum([alpha[i]*mn.pdf(x,means[i],cov[i]) for i in range(M)])
---> 16     impo_weights=np.array([p(samples[i])/q(samples[i]) for i in range(n)])
     17     normalized_weights=impo_weights/np.sum(impo_weights)
     18     return [samples,normalized_weights]

<ipython-input-210-6d764996607c> in <lambda>(x)
     13         data.append(list(np.random.multivariate_normal(means[simulation[i]],cov[simu
lation[i]])))
     14     samples=np.array(data)
---> 15     q = lambda x: np.sum([alpha[i]*mn.pdf(x,means[i],cov[i]) for i in range(M)])
     16     impo_weights=np.array([p(samples[i])/q(samples[i]) for i in range(n)])
     17     normalized_weights=impo_weights/np.sum(impo_weights)

<ipython-input-210-6d764996607c> in <listcomp>(.0)
     13         data.append(list(np.random.multivariate_normal(means[simulation[i]],cov[simu
lation[i]])))
     14     samples=np.array(data)
---> 15     q = lambda x: np.sum([alpha[i]*mn.pdf(x,means[i],cov[i]) for i in range(M)])
     16     impo_weights=np.array([p(samples[i])/q(samples[i]) for i in range(n)])
     17     normalized_weights=impo_weights/np.sum(impo_weights)

/Users/adilrhiulam/anaconda3/lib/python3.6/site-packages/scipy/stats/_multivariate.py in pdf
(self, x, mean, cov, allow_singular)
    504         dim, mean, cov = self._process_parameters(None, mean, cov)
    505         x = self._process_quantiles(x, dim)
--> 506         psd = _PSD(cov, allow_singular=allow_singular)
    507         out = np.exp(self._logpdf(x, mean, psd.U, psd.log_pdet, psd.rank))
    508         return _squeeze_output(out)
```