In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import pickle as pkl
import math as mt
import os
```

A. EM et le geyser Old Faithful

La durée des éruptions varie dans le temps ainsi que le délai entre deux éruptions consécutives. Notre hypothèse est que ces éruptions suivent en réalité deux distributions distinctes que nous allons chercher à identifier. Il s'agit donc bien d'apprentissage non supervisée d'une variable indiquant à quelle distribution se rapporte chaque éruption.

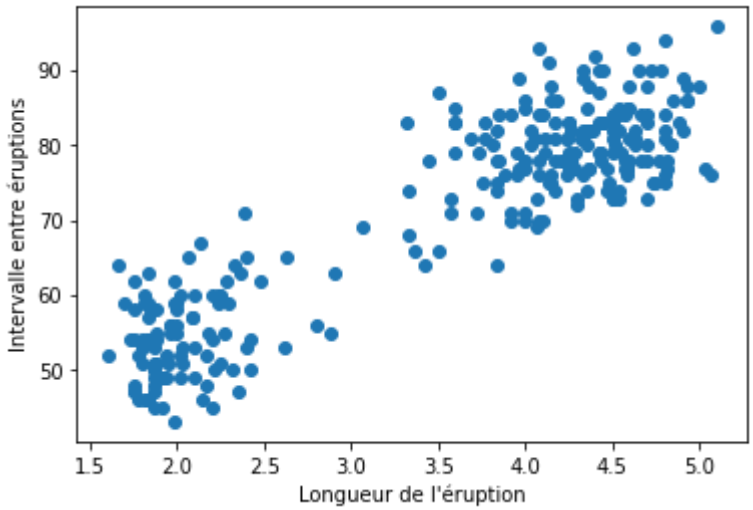
In [2]:

```
# chargement des données:

data = pkl.load( open('faithful.pkl', 'rb'))
X = data["X"]

print("taille des données", X.shape)
plt.figure()
plt.scatter(X[:,0], X[:,1])
plt.xlabel("Longueur de l'éruption")
plt.ylabel("Intervalle entre éruptions")
# on identifie sans peine les deux distributions: c'est un cas d'école...
# ... Reste à voir si on arrive à les retrouver automatiquement
```

```
taille des données (272, 2)
```

Out[2]:

```
Text(0, 0.5, 'Intervalle entre éruptions')
```



A.1 Modélisation

Suite à la prise en main des données précédentes, nous choisissons de modéliser les deux distributions latentes par des lois normales dont il va falloir déterminer les paramètres. La vraisemblance d'une loi normale de dimension N pour une observation $\mathbf{x} \in \mathbb{R}^N$ est la suivante:

$$p(\mathbf{x}|\mu,\Sigma) = \frac{1}{(2\pi)^{N/2}|\Sigma|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\mu)\Sigma^{-1}(\mathbf{x}-\mu)^\top}, \qquad \mu \in \mathbb{R}^N, \Sigma \in \mathbb{R}^{N\times N}$$

où $|\Sigma|$ désigne le déterminant de la matrice.

Petite vérification dimensionnelle

Vérifier rapidement au brouillon que $(x-\mu)\Sigma^{-1}(\mathbf{x}-\mu)^\top$ est bien un scalaire. Ca vous donnera des indications pour éviter les problèmes d'implémentation.

Coder la fonction de vraisemblance

Donner le code de la `normale_bidim` : `array(float) x array(float) x array(float) -> float` qui prend en argument `x, mu, Sig` et qui retourne la vraisemblance.

Note: vous chercherez par vous même les fonctions numpy pour calculer le déterminant et l'inversion de matrice

In [3]:

```
# retour: vraissamblance d'une loi normale
def normale_bidim(x, mu, Sig):
    # votre code
    N=x.shape[0]
    #print(N)
    #print(mt.pow((2*mt.pi),(N/2)))
    #print(mt.pow(np.linalg.det(Sig),1/2))
    alpha=(1/(mt.pow((2*mt.pi),(N/2))*(mt.pow(np.linalg.det(Sig),1/2))))

    return alpha* np.exp((-1/2)*np.dot(np.dot((x-mu),np.linalg.inv(Sig)),np.tran
spose(x-mu)))
    pass
```

In [4]:

```
mu  = np.array([1.,2])
Sig = np.array([[3., 0.],[0., 3.]])

x = np.array([1.,2])
print(normale_bidim(x, mu, Sig)) # 0.053051647697298435
x = np.array([0,0])
print(normale_bidim(x, mu, Sig)) # 0.023056151047594564
```

```
0.053051647697298435
0.023056151047594564
```

oui résultat prévisible car la loi est appliquée sur x=moyenne sur la première valeur ça donne la valeur maximum de proba et l'autre point= (0,0) est loin de la moyenne donc selon la distribution de la loi normale prend une valeur plus petite

Validation des résultats précédents

Au delà des chiffres obtenus, l'ordonnancement des deux valeurs était-il prévisible? pourquoi?

Visualisation des isocontours de la Gaussienne

Visualiser des isocontours sur une distribution continue en 2D implique -implicitement- de travailler en 3D. La fonction suivante donne le code pour réaliser cette opération (aucun code à ajouter de votre part).

In [5]:

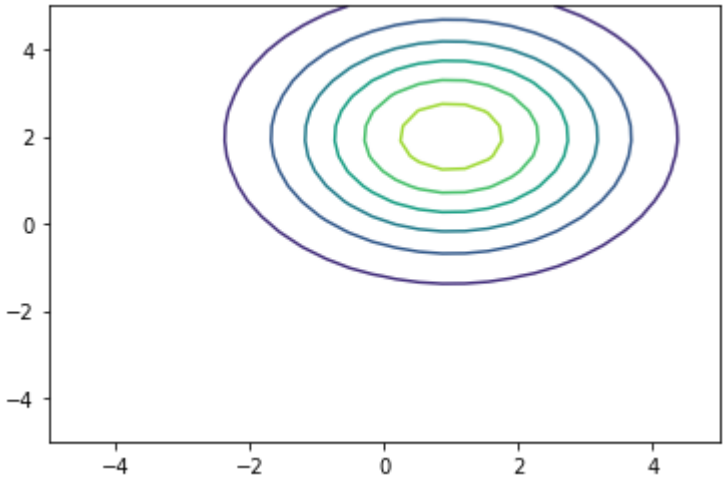
```
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
```

In [6]:

```
def plot_norm_2D(mu, Sig, bounds_min = np.array([-5, -5]), bounds_max = np.array
([5, 5])):
    ngrid = 30
    x = np.linspace(bounds_min[0], bounds_max[0], ngrid)
    y = np.linspace(bounds_min[1], bounds_max[1], ngrid)
    X,Y = np.meshgrid(x,y)
    Z = np.array([normale_bidim(np.array([x,y]), mu, Sig)
                  for x,y in zip(X.flatten(), Y.flatten())]).reshape(ngrid, ngri
d)
    fig = plt.gcf() # recuperation de la figure courante
    ax = fig.gca()
    ax.contour(X,Y,Z)
```

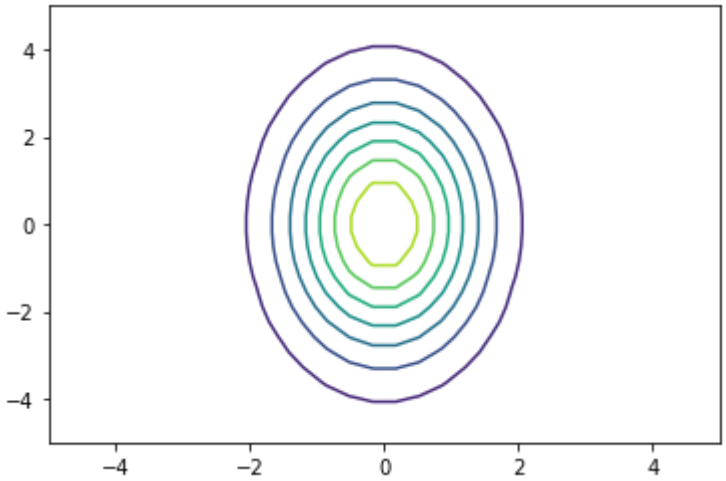
In [7]:

```
# test sur la distribution utilisée ci-dessus
plt.figure()
plot_norm_2D(mu, Sig)
```



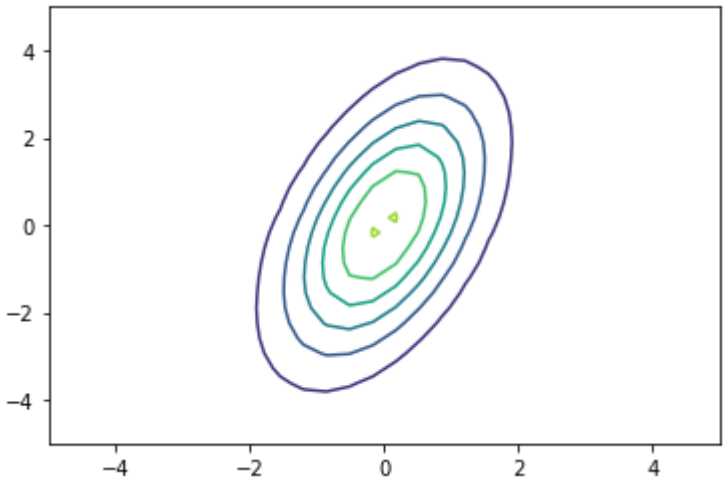
In [8]:

```
# autre test avec des variances très différentes sur les deux axes
mu2  = np.array([0.,0.])
Sig2 = np.array([[1., 0.],[0., 4.]])
plt.figure()
plot_norm_2D(mu2, Sig2)
```



In [9]:

```
# dernier test avec une covariance non nulle
# Note: Attention à garder une matrice de Variance symétrique !!!
mu3  = np.array([0.,0.])
Sig3 = np.array([[1., 1.],[1., 4.]])
plt.figure()
plot_norm_2D(mu3, Sig3)
```



Construction à la main d'une Gaussienne qui colle à nos données

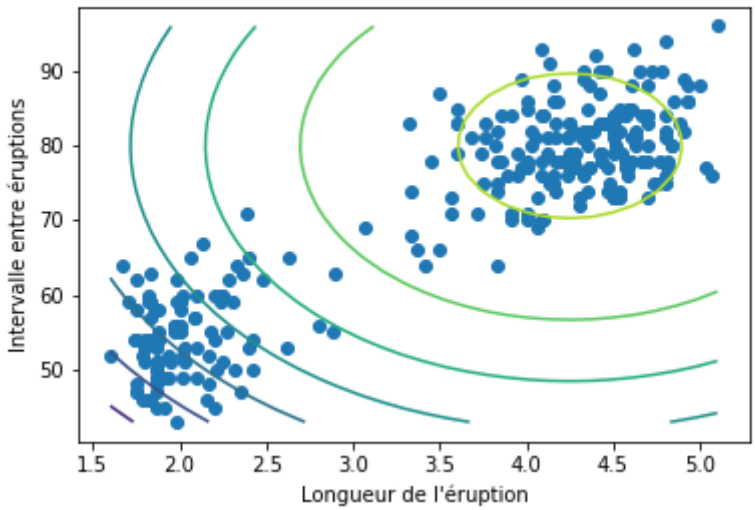
Estimation de la moyenne: (4.25, 80) à la louche sur la première figure

Estimation de l'écart-type en axe 1 : ??? (2/3 des données en mu+std et mu-std)

Estimation de l'écart-type en axe 2 : ??? (2/3 des données en mu+std et mu-std)

Estimation de la co-variance: ???

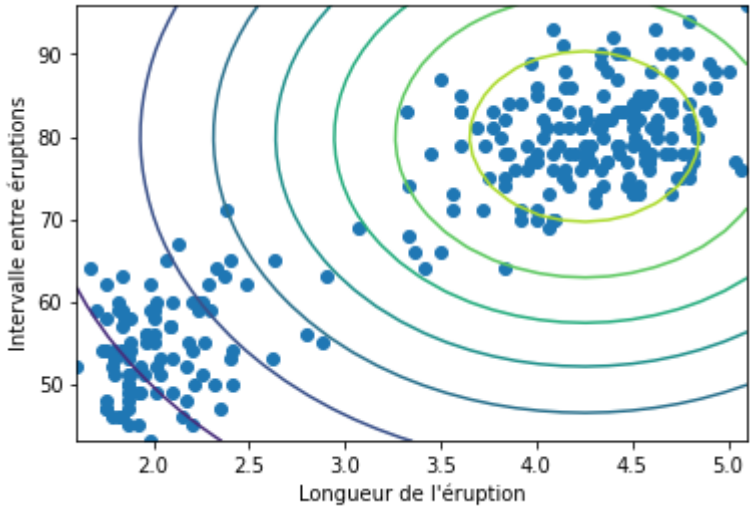
Une fois la matrice des variances estimée, vous devez obtenir:



Note: Attention à ne pas confondre écart-types et variances.

In [10]:

```
# votre code ici
mu4  = [4.25, 80]
Sig4 = [[2, 0], [0, 600]]

plt.figure()
# tracé des points
plt.scatter(X[:,0], X[:,1])
plt.xlabel("Longueur de l'éruption")
plt.ylabel("Intervalle entre éruptions")
plot_norm_2D(mu4, Sig4, X.min(0), X.max(0) )
# plt.savefig('manual_gauss_2d.png')
```



demander au prof la logique de ces valeurs !!!!!!!!!! elles ne sont pas normales

A.2 Apprentissage automatique

Arrêtons maintenant de faire des essais à la main et tentons une approche automatique qui sera:

- Robuste à des données bruitées
- Robuste à des données multi-dimensionnelles qu'on ne peut pas afficher

Nous allons utiliser une approche EM et nous utiliserons les notations suivantes.

Soit Θ les paramètres de notre modélisation normale à deux classes:

$$\Theta = \{(\pi_0 \in \mathbb{R}, \mu_0 \in \mathbb{R}^2, \Sigma_0 \in \mathbb{R}^{2\times 2}), (\pi_1, \mu_1, \Sigma_1)\}$$

où π désigne les probabilités a priori des classes.

Pour plus de simplicité, nous proposons de stocker tous les π dans une seule structure de données, tous les μ dans une seule structure de données et tous les Σ dans une structure afin de pouvoir facilement augmenter ou diminuer le nombre de classes à trouver dans le futur.

EM : Etape 0, initialisation

$\pi = [0.5, 0.5]$: initialisation la plus neutre et cohérente par rapport à l'observation des données

Pour μ, une stratégie classique consiste à partir de la moyenne du nuage et à ajouter 1 sur les dimensions pour μ_1 et retrancher 1 sur toutes les dimensions pour μ_2.

Pour Σ, on propose de partir de la matrice de variance de l'ensemble du nuage pour les deux modèles. On propose d'initialiser une matrice 3D afin d'accéder aux matrices des modèles en faisant `Sig[0]` et `Sig[1]`

La méthode `init` prend en argument `X` et retourne `pi`, `mu` et `Sig`.

Note: Si on veut une initialisation plus générique, il faut prendre en argument suppélmentaire le nombre de classes à prédire (et éventuellement d'autres choses).

In [11]:

```
def init(X):
    pi = np.array([0.5, 0.5]) # le plus raisonnable dans l'absolu et cohérent av
ec les observations
    mu=np.zeros((2,1, 2))
    mean=np.zeros((1,2))
    mean=np.mean(X,axis=0)
    #mu.append(mean+1)
    #mu.append(mean-1)
    mu[0]=mean+1
    mu[1]=mean-1
    Sig=np.zeros((2,2,2))
    Sig[0]=np.cov(X.T)
    Sig[1]=np.cov(X.T)
    return pi, mu, Sig

pi, mu, Sig = init(X)
print(pi,"\n", mu,"\n", Sig)
# Check:
#[0.5 0.5]
# [[ 4.48778309 71.89705882]
# [ 2.48778309 69.89705882]]
# [[[  1.30272833  13.97780785]
#  [ 13.97780785 184.82331235]]
#
# [[  1.30272833  13.97780785]
#  [ 13.97780785 184.82331235]]]
```

```
[0.5 0.5]
 [[[ 4.48778309 71.89705882]]

 [[ 2.48778309 69.89705882]]]
 [[[  1.30272833  13.97780785]
  [ 13.97780785 184.82331235]]

 [[  1.30272833  13.97780785]
  [ 13.97780785 184.82331235]]]
```

EM : l'étape E

Nous allons maintenant écrire l'étape E de l'algorithme EM, qui va nous permettre d'estimer les paramètres de la mixture de lois normales bidimensionnelles. Ainsi, comme vu en cours, soit Y une variable aléatoire indiquant quelle classe/loi normale bidimensionnelle a généré le couple de données $\mathbf{x} \in \mathbb{R}^2$

Soit $\Theta^t = \{\pi^t, \mu^t, \Sigma^t\}$ les paramètres à l'itération t.

Pour chaque observation $\mathbf{x}$, on définit les $Q_i^{t+1}(y_0) = P(y_0|\mathbf{x}, \Theta^t)$

$$Q_i^{t+1}(y_0) = \frac{p(\mathbf{x}|\mu_0, \Sigma_0)\pi_0}{\sum_{i=0}^{1} p(\mathbf{x}|\mu_i, \Sigma_i)\pi_i}, \qquad Q_i^{t+1}(y_1) = \frac{p(\mathbf{x}|\mu_1, \Sigma_1)\pi_1}{\sum_{i=0}^{1} p(\mathbf{x}|\mu_i, \Sigma_i)\pi_i}$$

Ecrire la fonction `Q_i: np.array x np.array x np.array x np.array -> np.array` qui prend en argument `X, pi, mu, Sig` et qui retourne le tableau des Q_i.

Attention, `X` correspond à l'ensemble des données, `pi` aux deux probabilités a priori, `mu` et `Sig` à des matrices contenant les paramètres des deux classes.

In [12]:

```
def Q_i(X, pi, mu, Sig):
    # votre code
    sh=X.shape
    nb_classes=mu.shape[0]
    Q=np.zeros((nb_classes,sh[0]))
    #maqam=np.sum(, axis=1)
    #Q[0]=
    #Q[1]=
    for i in range (sh[0]):
        temp=0
        for j in range(nb_classes):
            temp+=normale_bidim(X[i], mu[j], Sig[j])*pi[j]
        #temp=normale_bidim(X[i], mu[0], Sig[0])*pi[0]+normale_bidim(X[i], mu
[1], Sig[1])*pi[1]
        for j in range(nb_classes):
            Q[j,i]=normale_bidim(X[i], mu[j], Sig[j])*pi[j]/temp
        #Q[1, i]=normale_bidim(X[i], mu[1], Sig[1])*pi[1]/temp
    return Q
    pass

q = Q_i(X, pi, mu, Sig)

print(q[:,:5]) # q pour les 5 premiers points de la base
# [[0.02459605 0.03668168 0.05226123 0.01630238 0.49795917]
# [0.97540395 0.96331832 0.94773877 0.98369762 0.50204083]]
```

```
[[0.02459605 0.03668168 0.05226123 0.01630238 0.49795917]
 [0.97540395 0.96331832 0.94773877 0.98369762 0.50204083]]
```

EM : Etape M (La démonstration des formules est à la fin)

La maximisation de la vraisemblance est l'occasion de revoir le calcul des paramètres de la loi normale bidimensionnelle en intégrant la pondération q.

Ainsi:

$$\mu_0 = \frac{\sum_i Q_i^{t+1}(y_0)\mathbf{x_i}}{\sum_i Q_i^{t+1}(y_0)}, \qquad \mu_1 = \frac{\sum_i Q_i^{t+1}(y_1)\mathbf{x_i}}{\sum_i Q_i^{t+1}(y_1)}$$

$$\Sigma_0 = \frac{\sum_i Q_i^{t+1}(y_0)(\mathbf{x_i}-\mu_0)^T(\mathbf{x_i}-\mu_0)}{\sum_i Q_i^{t+1}(y_0)}, \qquad \Sigma_1 = \frac{\sum_i Q_i^{t+1}(y_1)(\mathbf{x_i}-\mu_1)^T(\mathbf{x_i}-\mu_1)}{\sum_i Q_i^{t+1}(y_1)}$$

Prenez le temps de dessiner les matrices pour vérifier les dimensions de Σ

Les probabilités a priori correspondent aux rapports des masses de probabilité Q:

$$\pi_0 = \frac{\sum_i Q_i^{t+1}(y_0)}{\sum_i Q_i^{t+1}(y_0) + Q_i^{t+1}(y_1)}, \qquad \pi_1 = \frac{\sum_i Q_i^{t+1}(y_1)}{\sum_i Q_i^{t+1}(y_0) + Q_i^{t+1}(y_1)}$$

Ecrire la fonction `update_param: np.array x np.array x np.array x np.array x np.array -> np.array x np.array x np.array` qui prend en argument `X`, `q`, `pi`, `mu`, `Sig` et qui retourne une nouvelle version de `pi`, `mu`, `Sig`

In [13]:

```
print(np.array([[1,1], [2,2], [3,3]])-np.array([1,1]))
print(np.sum(np.array([[1,1], [2,2], [3,3]])))
```

```
[[0 0]
 [1 1]
 [2 2]]
12
```

In [14]:

```
def update_param(X, q, pi, mu, Sig):
    sh=X.shape[0]
    nb_classes=mu.shape[0]
    mu=np.zeros((nb_classes,1, 2))
    Sig=np.zeros((nb_classes,2,2))
    pi=np.zeros((nb_classes,1))
    #sum1=np.sum(q[0])
    #sum2=np.sum(q[1])

    for j in range(nb_classes):
        mu[j]=np.dot(q[j], X)/np.sum(q[j]) # donne du (1*2)
        pi[j]=np.sum(q[j])/(np.sum(q))
        for i in range(sh):
            Sig[j]+=q[j, i]*np.dot(np.transpose(X[i]-mu[j]), X[i]-mu[j])/np.sum(
q[j])
    #mu[1]=np.dot(q[1], X)/sum2
    '''for i in range(sh):
        for j in range(nb_classes):
            Sig[j]+=q[j, i]*np.dot(np.transpose(X[i]-mu[j]), X[i]-mu[j])/np.sum
(q[j])
        Sig[1]+=q[1, i]*np.dot(np.transpose(X[i]-mu[1]), X[i]-mu[1])/sum2'''


    #pi[1]=sum2/(sum1+sum2)
    return  pi, mu, Sig
    pass

pi_u, mu_u, Sig_u = update_param(X, q, pi, mu, Sig)
#update_param(X, q, pi, mu, Sig)

print(pi_u,"\n", mu_u,"\n", Sig_u)
# check:
# [0.51132321 0.48867679]
# [[ 3.88361418 71.3886521 ]
# [ 3.07360826 70.38268397]]
# [[[  1.04337668  12.40444673]
#  [ 12.40444673 162.96851264]]
#
# [[  1.22881404  15.10227603]
#  [ 15.10227603 205.78298546]]]
```

```
[[0.51132321]
 [0.48867679]]
 [[[ 3.88361418 71.3886521 ]]

 [[ 3.07360826 70.38268397]]]
 [[[  1.04337668  12.40444673]
  [ 12.40444673 162.96851264]]

 [[  1.22881404  15.10227603]
  [ 15.10227603 205.78298546]]]
```

Algorithme EM: la boucle

Alterner les itérations jusqu'à convergence, c'est à dire jusqu'à ce que les nouveaux paramètres soit très proches des anciens. Arbitrairement, on pourra faire le test de convergence sur `mu` qui est assez stable.

La méthode `EM` prend en argument `X` et retourne `pi`, `mu` et `Sig`. `nIterMax` est un paramètre par défaut pour éviter les boucles infinies. Généralement, les algorithmes itératifs de type EM sont codés avec des boucles for plutot que des while pour plus de fiabilité.

Le paramètre `saveParam` sera utile dans les questions suivantes: ne vous en occupez pas dans un premier temps.

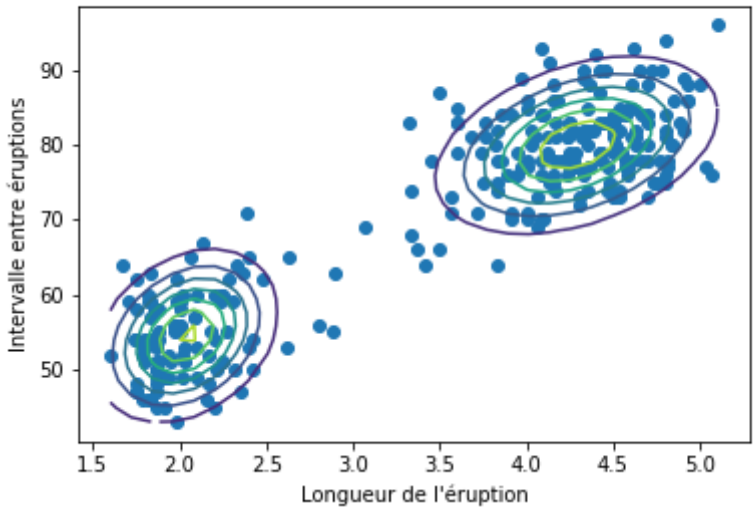
In [15]:

```
import os

def EM(X, nIterMax=100, saveParam=None):
    # votre code
    #initialisation
    pi, mu, Sig=init(X)

    for i in range(nIterMax):
        #Expectation
        q = Q_i(X, pi, mu, Sig)
        #Maximization
        pi_u, mu_u, Sig_u = update_param(X, q, pi, mu, Sig)
        if saveParam != None:                                          # détec
tion de la sauvergarde
            if not os.path.exists(saveParam[:saveParam.rfind('/')]):     # créat
ion du sous-répertoire
                os.makedirs(saveParam[:saveParam.rfind('/')])
            pkl.dump({'pi':pi_u, 'mu':mu_u, 'Sig': Sig_u},\
                     open(saveParam+str(i)+".pkl",'wb'))                 # séria
lisation
        if(np.sum(np.abs(mu_u-mu))<1e-3):
            break
        pi, mu, Sig=pi_u, mu_u, Sig_u

    print('nb iterations of EM algorithm : <'+str(i)+' >')
    return pi_u, mu_u,  Sig_u
    pass

pi, mu, Sig = EM(X)

print(pi,"\n", mu,"\n", Sig)

# CVG en 22 iterations pour un critère à 1e-3 sur la somme des écarts en valeur
 absolue sur les moyennes
# [0.64411944 0.35588056]
# [[ 4.28967856 79.96831569]
# [ 2.0364072  54.47870502]]
# [[[ 0.16994739  0.94034165]
#  [ 0.94034165 36.04319866]]
#
# [[ 0.06918256  0.43532305]
#  [ 0.43532305 33.69834324]]]
```

```
nb iterations of EM algorithm : <22 >
[[0.64412529]
 [0.35587471]]
 [[[ 4.28966596 79.96816344]]

 [[ 2.03639296 54.47856174]]]
 [[[ 0.16996337  0.94054489]
  [ 0.94054489 36.04548599]]

 [[ 0.06917125  0.43520499]
  [ 0.43520499 33.69753693]]]
```

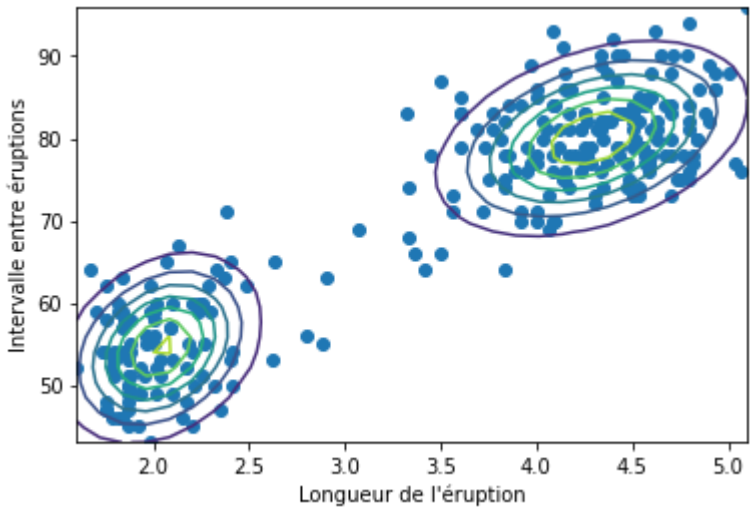
Affichage du résultat

A l'issue des itérations, vous devez obtenir la figure suivante:



In [16]:

```
plt.figure()

plt.scatter(X[:,0], X[:,1])
plt.xlabel("Longueur de l'éruption")
plt.ylabel("Intervalle entre éruptions")
plot_norm_2D(mu[0], Sig[0], X.min(0), X.max(0) ) # affichage modèle 1
plot_norm_2D(mu[1], Sig[1], X.min(0), X.max(0) ) # affichage modèle 2
plt.savefig('res_EM.png')
```



A-3 Réalisation d'une animation sur la convergence du modèle

L'idée est d'utiliser le paramètre `saveParam` de la méthode EM pour savegarder les paramètres du modèle à chaque itération et pouvoir retracer toutes les figures correspondant à toutes les étapes de l'algorithme.

- il faut sauver les figures dans un répertoire à part, sinon ça va être le bazar dans votre répertoire
- il faut donner des noms de fichiers explicites pour pouvoir récupérer les informations dans l'ordre ensuite
- il faut utiliser pickle, car la sérialisation c'est fantastique dans ce genre de cas de figure.

L'idée est d'invoquer: `pi, mu, Sig = EM(X, saveParam="params/faithful")`

Il faudra créer le sous-répertoire `params` puis sauver les paramètres dans: `params/faithful1.pkl` , `params/faithful2.pkl` , ...

Afin de respecter toutes les contraintes, nous vous proposons d'insérer le code suivant dans la méthode EM:

```
import os # ajout de bibliothèque utile avant la méthode

        if saveParam != None:
 # détection de la sauvergarde
            if not os.path.exists(saveParam[:saveParam.rfind('/')]):
 # création du sous-répertoire
                os.makedirs(saveParam[:saveParam.rfind('/')])
            pkl.dump({'pi':pi_u, 'mu':mu_u, 'Sig': Sig_u},\
                     open(saveParam+str(i)+".pkl",'wb'))
 # sérialisation
```

Note: tout le code est donné... Mais il faut vérifier que vous sauvez bien les bonnes variables (j'ai mis les noms que j'ai utilisé dans la correction et qui ne correspondent pas forcément.

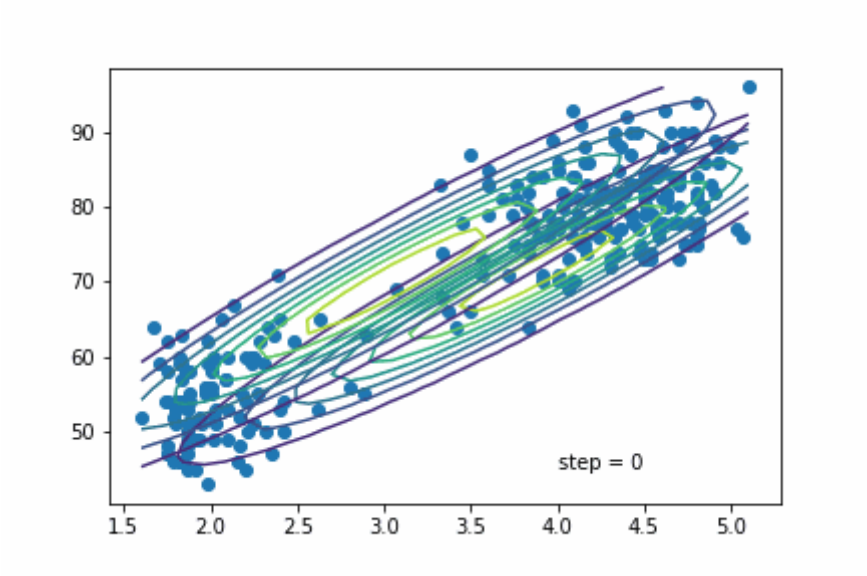
In [17]:

```
pi, mu, Sig = EM(X, saveParam="params/faithful")
```

```
nb iterations of EM algorithm : <22 >
```

Il faut ensuite créer la méthode qui va charger ces paramètres et créer l'animation... Le code est fourni ci-dessous.

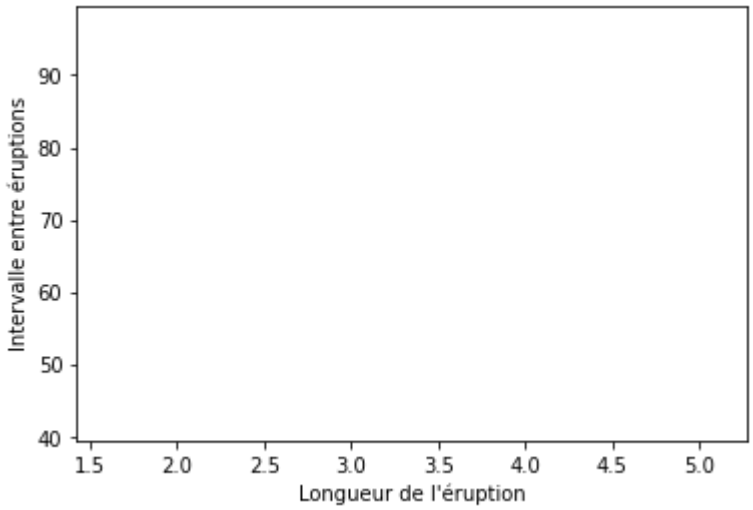
Le code est capricieux... Mais si ça marche, vous devez obtenir:



In [18]:

```
import matplotlib.animation as animation
import glob


def creer_animation(X, params_path, fname):
    nbiter = len(glob.glob(params_path+"/*.pkl"))
    print(nbiter)
    fig = plt.figure()
    plt.xlim(X[:,0].min()-(X[:,0].mean()*0.05), X[:,0].max()+(X[:,0].mean()*0.05
))
    plt.ylim(X[:,1].min()-(X[:,1].mean()*0.05), X[:,1].max()+(X[:,1].mean()*0.05
))
    plt.xlabel("Longueur de l'éruption")
    plt.ylabel("Intervalle entre éruptions")

    def animate(i):
        print(i)
        data = pkl.load(open(params_path+"/"+fname+str(i)+".pkl","rb")) # récupé
ration d'un dictionnaire
        plt.clf()
        ax  = fig.gca()
        ax.scatter(X[:,0], X[:,1])
        ax.text(X[:,0].max()*0.75, X[:,1].min()*1.25, 'step = ' + str ( i ))
        for i in range(len(data['mu'])):
            plot_norm_2D(data['mu'][i], data['Sig'][i], X.min(0), X.max(0) ) # a
ffichage modèle i

        return ax

    ani = animation.FuncAnimation(fig, animate, frames = nbiter, interval=300, r
epeat=True )
    ani.save(params_path+'/animation.gif', bitrate=4000)


creer_animation(X, "params", "faithful")
```

```
100

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-cb9f49b1709e> in <module>()
     28
     29
---> 30 creer_animation(X, "params", "faithful")

<ipython-input-18-cb9f49b1709e> in creer_animation(X, params_path, fname)
     25
     26     ani = animation.FuncAnimation(fig, animate, frames = nbiter, interval=300, repeat=True )
---> 27     ani.save(params_path+'/animation.gif', bitrate=4000)
     28
     29

/home/inas/.local/lib/python3.6/site-packages/matplotlib/animation.py in save(self, filename, writer, fps, dpi, codec, bitrate, extra_args, metadata, extra_anim, savefig_kwargs, progress_callback)
   1100                                                 metadata=metadata)
   1101             else:
-> 1102                 alt_writer = next(writers, None)
   1103                 if alt_writer is None:
   1104                     raise ValueError("Cannot save animation: no writers are "

TypeError: 'MovieWriterRegistry' object is not an iterator
```

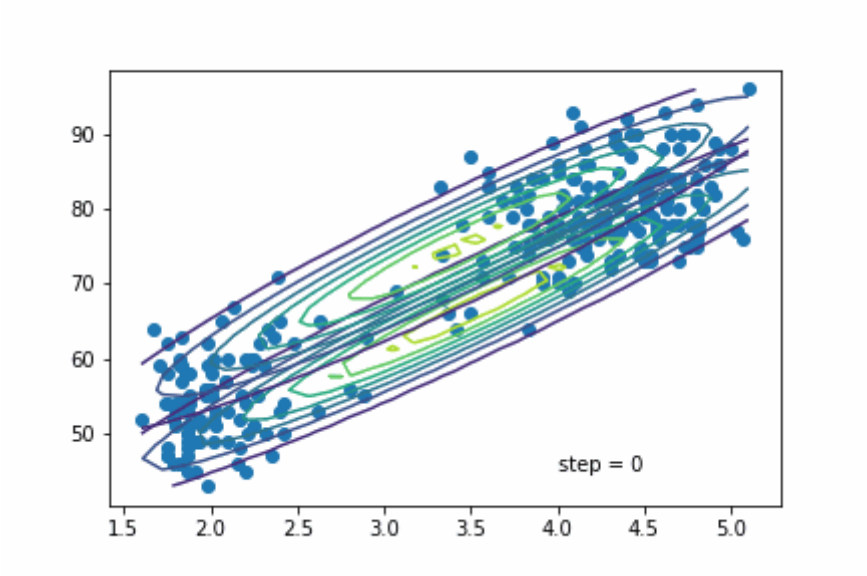


A-4 Impact de l'initialisation

EM est très souvent associé à des formalismes non convexes... Ca sinifie que les maxima sont multiples et qu'une mauvaise initialisation peut avoir des conséquences désastreuses...

Imaginons une nouvelle fonction d'initialisation où les moyennes initiales sont initialisées de manière défavorable , c'est à dire orthogonalement à la séparabilité naturelle des données.

Les données sont vraiment très facile à séparer et le code va continuer à fonctionner... Mais on note que la convergence est bien moins rapide.

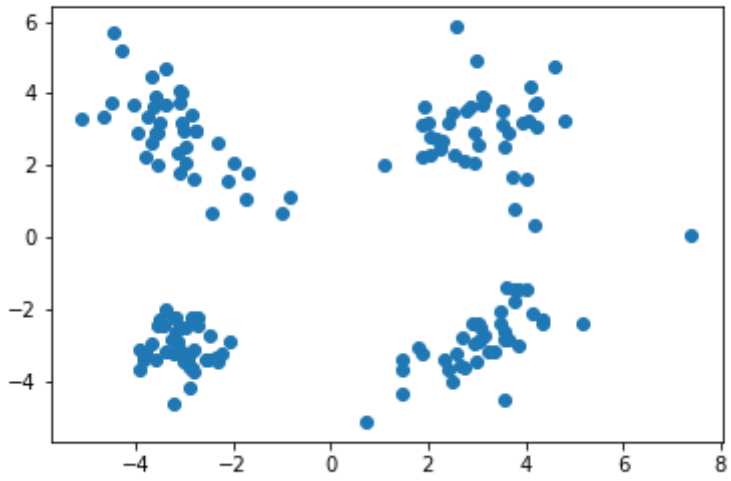


In [ ]:

```
def init(X):
    pi = np.array([0.5, 0.5]) # le plus raisonnable dans l'absolu et cohérent av
ec les observations

    mu1 = X.mean(0)+ [0.1,-15] # initialisation défavorable
    mu2 = X.mean(0)+ [-0.1,15]
    mu  = np.vstack((mu1, mu2))

    Sig = np.zeros((2, 2, 2))
    Sig[0,:,:] = np.cov(X.T)
    Sig[1,:,:] = np.cov(X.T)
    return pi, mu, Sig


# code un peu moche... Mais rappel utile sur les notebooks
# redefinition de init
# appel de EM qui fait appel à init => nouvelle version de init => nouvelle vers
ion de EM

pi, mu, Sig = EM(X, saveParam="params_bad/faithful")
creer_animation(X, "params_bad", "faithful")
```

B- Passage en 4 classes

Nous vous proposons un nouveau jeu de données (jouet) où il y a 4 classes à découvrir. Adapter votre code pour faire face à ce cas de figure.

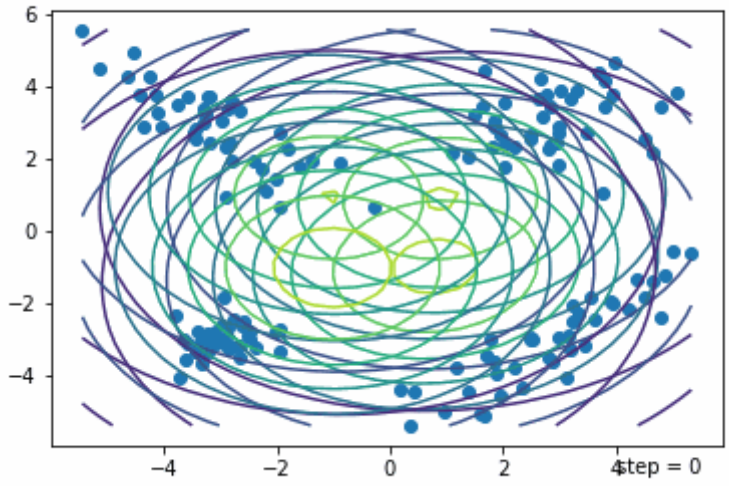


In [ ]:

```
def generation_4_gaussiennes():
    n = 50 # nb points par gaussienne

    Sig1 = np.array([[1, 0],[0, 1]])
    mu1  = np.array([-3, -3])
    Sig2 = np.array([[1.5, 0.5],[0.5, 1.5]])
    mu2  = np.array([3, -3])
    Sig3 = np.array([[1.2, -0.5],[-0.5, 1.2]])
    mu3  = np.array([-3, 3])
    Sig4 = np.array([[1.2, 0],[0, 1.2]])
    mu4  = np.array([3, 3])

    X = np.vstack((np.random.randn(n,2)@Sig1 +mu1, np.random.randn(n,2)@Sig2 +mu
2,\
                   np.random.randn(n,2)@Sig3 +mu3, np.random.randn(n,2)@Sig4 +mu
4))
    return X

Xg = generation_4_gaussiennes()

plt.figure()
plt.scatter(Xg[:,0],Xg[:,1])
```

Si votre code est robuste, il suffit de changer l'initialisation:

- pi = 4 modalités équi-probables
- mu = 4 vecteurs de dimension 2 décalés de la moyenne globale de [1,1], [-1,1], [1,-1], [-1,-1]
- Sig = Matrice des variances de l'ensemble du nuage pour toutes les classes



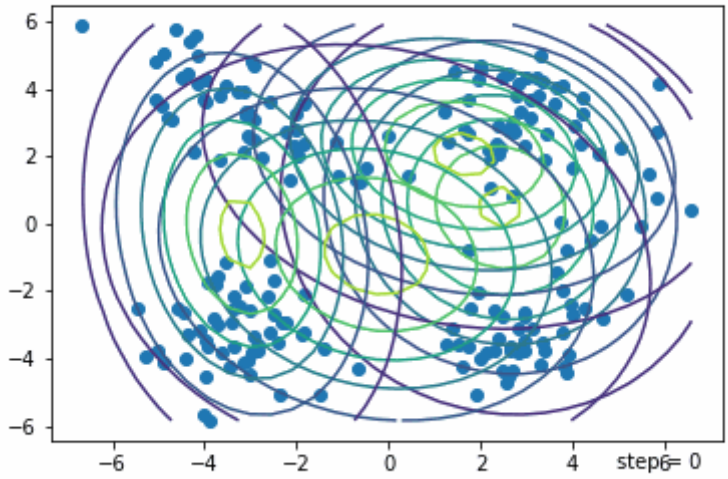
In [ ]:

```
# si le code est robuste, il suffit de changer le code d'initialisation.
# Les dimensions des matrices doivent ensuite se propager

def init(X):
    # à compléter
    return pi, mu, Sig

def init(X):
    pi = np.array([0.25, 0.25, 0.25, 0.25]) # modalités équi-probables
    mu=np.zeros((4,1, 2))
    #mean=np.zeros((1,2)
    mean=np.mean(X,axis=0)
    mu[0]=mean+np.array( [1,1])
    mu[1]=mean+np.array( [-1,1])
    mu[2]=mean+np.array( [1,-1])
    mu[3]=mean+np.array( [-1,-1])
    Sig=np.zeros((4,2,2))
    for i in range(4):
        Sig[i]=np.cov(X.T)
    #Sig[1]=np.cov(X.T)
    return pi, mu, Sig

pi, mu, Sig = init(X)


pi, mu, Sig = EM(Xg, saveParam="params_gauss/gauss")
creer_animation(Xg, "params_gauss", "gauss")
```

passage à une initialisation dégradée

- mu = 4 vecteurs de dimension 2 décalés de la moyenne globale de [4,2], [3,4], [0,0], [-5,0]

Quand on part de n'importe où, on n'arrive pas forcément au bon endroit



In [ ]:

```
def init(X):
    # A compléter
    pi = np.array([0.25, 0.25, 0.25, 0.25]) # lmodalités équi-probables
    mu=np.zeros((4,1, 2))
    #mean=np.zeros((1,2)
    mean=np.mean(X,axis=0)
    mu[0]=mean+np.array( [4,2])
    mu[1]=mean+np.array( [3,4])
    mu[2]=mean+np.array( [0,0])
    mu[3]=mean+np.array( [-5,0])
    Sig=np.zeros((4,2,2))
    for i in range(4):
        Sig[i]=np.cov(X.T)
    return pi, mu, Sig

pi, mu, Sig = init(X)
pi, mu, Sig = EM(Xg, saveParam="params_gauss_bad/gauss")
creer_animation(Xg, "params_gauss_bad", "gauss")
```

C- Passage en haute dimension

Au delà de 2 ou 3 dimensions, il n'est plus question de visualiser le nuage de points... Cela peut poser des problèmes pour la compréhension générale du problème, la détermination du nombre de classes, les intuitions pour l'initialisation des paramètres.

Bref, on franchit un pallier de complexité important.

C-1 Application sur les données USPS de la semaine 3

Reprenons les données de la semaine 3 mais en mode non supervisé: nous observons les X mais il n'y a plus de Y (du moins, plus pour l'algorithme d'apprentissage, on pourra toujours s'en servir pour l'évaluation).

C-1.2 Choix d'une modélisation

Nous n'allons pas faire l'hypothèse d'une gaussienne en 256 dimensions pour représenter une image... Il faut donc choisir un modèle. Par défaut, nous allons reprendre la modélisation de Bernoulli (mais vous pouvez optez pour la modélisation gaussienne par pixel, ça ne change pas grand chose).

Il y a donc un paramètre p_j par pixel et par classe à déterminer.

Note: le modèle étant différent, on doit refaire toutes les méthodes

- Nous vous donnons le code de chargement des données
- Nous vous donnons la fonction de calcul de la log-vraisemblance d'une image pour un modèle de classe de type Bernoulli (calcul de $\log p(X|\theta)$)
- Vous devez refaire une fonction d'initialisation `init_B(X)`
  - par exemple en moyennant 3 images consécutives à partir du début de la base 10 fois de suite pour obtenir des paramètre theta
- Vous devez refaire une fonction `Q_i_B(X, pi , theta)` qui comporte une difficulté non négligeable.

En effet, nous utilisons normalement:

$$p(\theta|X) = \frac{p(X|\theta)p(\theta)}{\sum_c p(X|\theta_c)p(\theta_c)}$$

Ne disposant que de $\log(p(X|\theta))$, la formule devient:

$$\log p(\theta|X) = \log(p(X|\theta)) + \log(p(\theta)) - \log\left(\sum_c p(X|\theta_c)p(\theta_c)\right)$$

Le dénominateur (ou partie négative du log) est compliqué à calculer car on ne peut pas faire rentrer le log dans la somme.

Nous allons utiliser le *logsumexp trick* pour résoudre ce problème:

$$\log \sum_z p(x,z) = s^* + \log \sum_z \exp(\log p(x,z) - s^*), \qquad s^* = \max_z \log p(x,z)$$

- la fonction `update_param_B(X, q, pi, theta)` est quasi-identique à la précédente, les theta étant obtenus comme les mu en pondérant le calcul de la moyenne par l'appartenance du point à la classe.
- La fonction `EM_B` est quasi-inchangée... Mais la convergence est trop lente. Le critère prévu initialement était trop strict (une somme de valeurs absolues augmente mécaniquement avec le nombre de paramètres). Ca vaut le coup de changer et de regarder seulement le plus grand changement:

```
np.abs(theta-theta_u).sum()<1e-3   =>   np.abs(theta-theta_u).max()<1e
-3
```

Note importante: Prendre le temps de réfléchir à la raison qui rend impossible le retour vers la formulation initiale en prenant simplement $p(\theta|X) = \exp(\log p(X|\theta))$.

In [28]:

```
# chargement des données
data = pkl.load(open("usps.pkl",'rb'))
# data est un dictionnaire contenant les champs explicites X_train, X_test, Y_tr
ain, Y_test
Xu = np.array(data["X_train"],dtype=float) # changement de type pour éviter les
 problèmes d'affichage
Xu = np.where(Xu>0, 1, 0) # binarisation
Yu = data["Y_train"]
```

In [29]:

```
# vraisemblance d'une image x pour un modèle de classe theta
# X de dimension 256
# theta de dimension 256 (pour une classe)
# out : une log-vraisemblance (scalaire)

def logpobsBernoulli(X, theta):
    seuil = 1e-5
    theta = np.maximum(np.minimum(1-seuil, theta),seuil)
    logp = (X*np.log(theta)+(1-X)*np.log(1-theta)).sum()
    return np.array(logp)
```

In [30]:

```
A=np.array([[1, 1], [2, 2], [3,3], [4, 4]])
print (np.sum(A, 0))
```

```
[10 10]
```

In [31]:

```
print(A[0:2,:])
```

```
[[1 1]
 [2 2]]
```

In [32]:

```
def init_B(X):
    # A compléter
    nb_classes=10
    theta=np.zeros(( nb_classes, X.shape[1]) )
    for i in range(nb_classes):
        theta[i]=np.sum(X[i*3:i*3+3, :], 0)/3
    # pi modalités equi-probables
    pi=np.array([(1/nb_classes) for i in range (nb_classes)])
    return pi, theta
# pour une initialisation avec les 30 premiers points (3 points x 10 classes)
pi, theta = init_B(Xu)

print(pi,"\n", theta)

# [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
# [[0.         0.         0.         ... 0.         0.         0.        ]
# [0.         0.         0.         ... 0.33333333 0.33333333 0.        ]
# [0.         0.         0.         ... 0.         0.         0.        ]
# ...
# [0.         0.         0.         ... 0.33333333 0.         0.        ]
# [0.         0.         0.         ... 0.         0.         0.        ]
# [0.         0.33333333 0.33333333 ... 0.         0.         0.        ]]
# => il semble que les groupes de 3 correspondent bien à des chiffres différents
```

```
[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
 [[0.         0.         0.         ... 0.         0.         0.
]
 [0.         0.         0.         ... 0.33333333 0.33333333 0.
]
 [0.         0.         0.         ... 0.         0.         0.
]
 ...
 [0.         0.         0.         ... 0.33333333 0.         0.
]
 [0.         0.         0.         ... 0.         0.         0.
]
 [0.         0.33333333 0.33333333 ... 0.         0.         0.
]]
```

In [33]:

```
print(logpobsBernoulli(Xu[0], theta[0]))
```

```
-114.53301719853128
```

In [34]:

```
def Q_i_B(X, pi, theta):
    # A compléter
    nb_classes=10
    nb_images=X.shape[0]
    q=np.zeros((nb_classes,nb_images ))
    for i in range(nb_images):
        value=0
        s=np.max(np.array([(logpobsBernoulli(X[i], theta[k])+np.log(pi[k])) for
k in range(nb_classes)]))
        sum=0
        for j in range(nb_classes):
            sum+=np.exp(logpobsBernoulli(X[i], theta[j])+np.log(pi[j])-s)
        value=s+np.log(sum)
        for j in range(nb_classes):
            intermediate=logpobsBernoulli(X[i], theta[j])+np.log(pi[j])-value
            q[j,i]=np.exp(intermediate)
    return q

q = Q_i_B(Xu, pi, theta)
print(q)

#[[1.00000000e+000 1.00000000e+000 1.00000000e+000 ... 1.58812497e-106
#  9.16928453e-094 1.00470165e-087]
# [1.95535205e-148 5.29887946e-063 4.23956990e-117 ... 7.57852443e-068
#  1.59179416e-056 5.99285928e-035]
# [1.81219302e-076 7.67516622e-113 6.28806207e-154 ... 1.75623379e-086
#  6.04505236e-181 1.69526450e-052]
# ...
# [1.24908969e-160 3.04862065e-014 7.80634218e-132 ... 3.17279444e-085
#  2.29003336e-118 1.56806005e-043]
# [3.31958161e-022 8.99909374e-117 1.15205680e-189 ... 1.22724138e-047
#  4.12431878e-035 1.89549430e-032]
# [1.92053768e-143 1.74647432e-085 5.58866193e-079 ... 3.81153805e-115
#  2.20065029e-102 3.08644347e-089]]

# Les 3 premières images sont attribuées au cluster 0 de manière quasi-certain
e...
# ... C'est normal, elle a servi à initialiser le cluster !
```

```
[[1.00000000e+000 1.00000000e+000 1.00000000e+000 ... 1.58812497e-10
6
  9.16928453e-094 1.00470165e-087]
 [1.95535205e-148 5.29887946e-063 4.23956990e-117 ... 7.57852443e-06
8
  1.59179416e-056 5.99285928e-035]
 [1.81219302e-076 7.67516622e-113 6.28806207e-154 ... 1.75623379e-08
6
  6.04505236e-181 1.69526450e-052]
 ...
 [1.24908969e-160 3.04862065e-014 7.80634218e-132 ... 3.17279444e-08
5
  2.29003336e-118 1.56806005e-043]
 [3.31958161e-022 8.99909374e-117 1.15205680e-189 ... 1.22724138e-04
7
  4.12431878e-035 1.89549430e-032]
 [1.92053768e-143 1.74647432e-085 5.58866193e-079 ... 3.81153805e-11
5
  2.20065029e-102 3.08644347e-089]]
```

In [35]:

```
def update_param_B(X, q, pi, theta):

    #-----------------------------------
    sh=X.shape[0]
    nb_classes=pi.shape[0]
    theta_u=np.zeros((nb_classes, X.shape[1]))
    pi_u=np.zeros((nb_classes,1))
    #sum1=np.sum(q[0])
    #sum2=np.sum(q[1])

    for j in range(nb_classes):
        theta_u[j]=np.dot(q[j], X)/np.sum(q[j]) # donne du (1*nb_pixels)
        pi_u[j]=np.sum(q[j])/(np.sum(q))


    #-------------------------------------------------------------------------
---
    return pi_u, theta_u

pi_u, theta_u = update_param_B(Xu, q, pi, theta)
print(pi_u,"\n", theta_u)

#[0.1429148  0.0529933  0.01089579 0.2132248  0.09166928 0.1990869
# 0.04188381 0.12184295 0.10426259 0.02122579]
# [[5.61774149e-03 2.44345060e-02 1.06284941e-01 ... 7.14836506e-03
#  1.19248146e-03 1.12332257e-03]
# [3.02933044e-03 3.02935034e-03 3.53098758e-02 ... 2.26634861e-01
#  1.72759601e-01 9.62523737e-02]
# [6.34695440e-39 3.21493798e-34 8.03847237e-15 ... 1.04112751e-06
#  2.27933690e-10 2.16581572e-46]
# ...
# [1.31759291e-03 7.90378929e-03 1.23672936e-02 ... 3.53305058e-02
#  7.83213119e-03 5.26816105e-03]
# [1.54511878e-03 7.77811601e-03 2.19550033e-02 ... 4.90627331e-02
#  1.58317275e-02 6.16099079e-03]
# [1.66752649e-01 3.46648571e-01 4.14299680e-01 ... 7.50657938e-03
#  7.50657924e-03 5.32960363e-48]]
```

```
  [[0.1429148 ]
   [0.0529933 ]
   [0.01089579]
   [0.2132248 ]
   [0.09166928]
   [0.1990869 ]
   [0.04188381]
   [0.12184295]
   [0.10426259]
   [0.02122579]]
   [[5.61774149e-03 2.44345060e-02 1.06284941e-01 ... 7.14836506e-03
    1.19248146e-03 1.12332257e-03]
   [3.02933044e-03 3.02935034e-03 3.53098758e-02 ... 2.26634861e-01
    1.72759601e-01 9.62523737e-02]
   [6.34695440e-39 3.21493798e-34 8.03847237e-15 ... 1.04112751e-06
    2.27933690e-10 2.16581572e-46]
   ...
   [1.31759291e-03 7.90378929e-03 1.23672936e-02 ... 3.53305058e-02
    7.83213119e-03 5.26816105e-03]
   [1.54511878e-03 7.77811601e-03 2.19550033e-02 ... 4.90627331e-02
    1.58317275e-02 6.16099079e-03]
   [1.66752649e-01 3.46648571e-01 4.14299680e-01 ... 7.50657938e-03
    7.50657924e-03 5.32960363e-48]]
```

In [36]:

```
def EM_B(X, nIterMax=100, saveParam=None):
    # A compléter
        #initialisation
    pi, theta=init_B(X)

    for i in range(nIterMax):
        #Expectation
        q = Q_i_B(X, pi, theta)
        #Maximization
        pi_u, theta_u = update_param_B(X, q, pi, theta)
        if saveParam != None:                                          # détec
tion de la sauvergarde
            if not os.path.exists(saveParam[:saveParam.rfind('/')]):     # créat
ion du sous-répertoire
                os.makedirs(saveParam[:saveParam.rfind('/')])
            pkl.dump({'pi':pi_u, 'mu':mu_u, 'Sig': Sig_u},\
                     open(saveParam+str(i)+".pkl",'wb'))                 # séria
lisation
        if(np.abs(theta-theta_u).max()<1e-3):
            break
        pi, theta=pi_u, theta_u

    print('nb iterations of EM algorithm : <'+str(i)+' >')

    return pi_u, theta_u

pi, theta = EM_B(Xu)
print(pi,"\n", theta)

'''pi, theta = EM_B(Xu)
print(pi,"\n", theta)'''

# résultats obtenus au bout de 53 iterations avec le critère d'arret
# np.abs(theta-theta_u).max()<1e-3

# [0.11204358 0.11290604 0.09168978 0.11606624 0.08083578 0.14108101
# 0.09640892 0.09443431 0.09441848 0.06011585]
# [[1.23776431e-060 1.17073232e-024 1.44528441e-003 ... 4.52135684e-046
#  2.97691733e-083 3.56806303e-100]
# [8.91923457e-003 3.37172393e-002 1.13397739e-001 ... 3.77805198e-002
#  2.81742587e-003 5.39756643e-024]
# [2.27528639e-002 6.99418725e-002 1.40382224e-001 ... 2.43335597e-001
#  2.01275409e-001 1.20720447e-001]
# ...
# [1.06592297e-006 1.74318535e-003 3.52223476e-002 ... 4.24154250e-002
#  8.49929686e-003 3.40126603e-003]
# [1.89969401e-039 6.80091280e-003 4.09086094e-002 ... 4.58582862e-002
#  1.02027794e-002 1.70029643e-003]
# [9.07953380e-002 2.21645819e-001 3.74532494e-001 ... 3.75578862e-036
#  3.75515652e-036 8.49004067e-084]]
```

```
nb iterations of EM algorithm : <54 >
[[0.11202121]
 [0.11286288]
 [0.09170415]
 [0.1161186 ]
 [0.0807734 ]
 [0.14107999]
 [0.09642712]
 [0.09443751]
 [0.0944622 ]
 [0.06011294]]
 [[1.22895359e-060 1.18035568e-024 1.44544572e-003 ... 4.51785215e-0
46
  7.45534527e-084 1.06652180e-100]
 [8.90157097e-003 3.36994406e-002 1.13357965e-001 ... 3.77770751e-00
2
  2.81759146e-003 5.40264482e-024]
 [2.27490045e-002 6.99327987e-002 1.40365620e-001 ... 2.43308899e-00
1
  2.01245732e-001 1.20703133e-001]
 ...
 [1.06641714e-006 1.74229923e-003 3.52208774e-002 ... 4.24139635e-00
2
  8.49898988e-003 3.40114863e-003]
 [2.04126418e-039 6.79777460e-003 4.08904884e-002 ... 4.58464560e-00
2
  1.01981320e-002 1.69950960e-003]
 [9.07997465e-002 2.21656578e-001 3.74548201e-001 ... 3.81149711e-03
6
  3.81086525e-036 8.49336897e-084]]
```

Out[36]:

```
'pi, theta = EM_B(Xu)\nprint(pi,"\n", theta)'
```

C-2 Reflexion sur les métriques d'évaluation en clustering

Etape d'affectation des points à une classe: il s'agit juste de faire tourner `Q_i_B` une fois de plus pour trouver des $\hat{Y}$.

Problème : rien ne garantit que le $Y == 0$ corresponde au $\hat{Y} == 0$

Proposition de métrique: la pureté

1. Déterminer la classe majoritaire dans chaque cluster
2. Calculer la pureté du cluster = le pourcentage d'image dans le cluster qui appartiennent à la classe majoritaire du cluster.
   - L'idée est de travailler sur `Y_hat_c = Y[Y_hat == c]` : les vraies étiquettes associées à un cluster
   - Pour compter les étiquettes dans `Y_hat_c` , le plus simple est d'utiliser: `val, count = np.unique(Y_hat_c, return_counts=True)`
3. Agréger les puretés en pondérant par la taille du cluster pour obtenir un indicateur unique

Note: la pureté est une métrique intéressante pour comparer deux partitionnements des données... A condition de travailler à nombre de partitions fixes. En effet, plus il y a de clusters, plus c'est facile d'avoir une pureté élevée!

In [37]:

```
B=A[:,0]
print(B)
print(B==3)
print(B[B==3])
```

```
[1 2 3 4]
[False False  True False]
[3]
```

In [40]:

```
def calcul_purete(X, Y, pi, theta):
    # A compléter
    print(Y.shape)
    q = Q_i_B(X, pi, theta)
    # affectation des points à une classe
    Y_EM=np.zeros((Y.shape))
    # Y_EM associe un cluster à chaque image
    Y_EM=np.argmax(q, axis=0)
    # pour chaque cluster
    purete=np.zeros((q.shape[0]))
    poids=np.zeros((q.shape[0]))
    for i in range (q.shape[0]):
        # récupérer les images associées à ce cluster
        img_clus=Y[Y_EM==i]
        val, counts=np.unique(img_clus, return_counts=True)
        major_class=val[np.argmax(counts)]
        purete[i]=np.size(img_clus[img_clus==major_class])/img_clus.size
        poids[i]=img_clus.size/Y.size
    return purete, poids

purete, poids = calcul_purete(Xu, Yu, pi, theta)

print(purete, poids, ((purete*poids)/poids.sum()).sum())

# on trouve une pureté agrégée de 0.753
```

```
(6229,)
[0.56241033 0.55064194 0.85989492 0.55862069 0.93426295 0.95795455
 0.76039933 0.65704584 0.8811545  0.90909091] [0.11189597 0.11253813
0.091668   0.11639107 0.08059079 0.14127468
 0.09648419 0.09455771 0.09455771 0.06004174] 0.7527693048643442
```

D- Démonstration des calculs pour A-2 (étape M)

Attention, la formule de la normale bidimensionnelle a été aplatie: les deux dimensions des vecteurs sont explicitées en (x, z), les moyennes sont μ_x, μ_z et la matrice des variances est remplacée par les écarts types sur les deux dimensions et le coefficient de corrélation linéaire

On suppose ici que les couples de durée et de délai (x, z) suivent une mixture de lois normales bidimensionnelles. La distribution de probabilité que l'on cherche à estimer est donc :

$$P(x, z|\Theta) = \pi_0 \mathcal{N}(\mu_{x0}, \mu_{z0}, \sigma_{x0}, \sigma_{z0}, \rho_0)(x, z) + \pi_1 \mathcal{N}(\mu_{x1}, \mu_{z1}, \sigma_{x1}, \sigma_{z1}, \rho_1)(x, z)$$

On notera $f(\mu_x, \mu_z, \sigma_x, \sigma_z, \rho)$ la fonction de densité d'une loi normale bidimensionnelle :

$$f_{\mu_x,\mu_z,\sigma_x,\sigma_z,\rho}(x, z) = \frac{1}{2\pi\sigma_x\sigma_z\sqrt{1-\rho^2}} \exp\left\{ -\frac{1}{2(1-\rho^2)} \left[\left(\frac{x-\mu_x}{\sigma_x}\right)^2 - 2\rho\frac{(x-\mu_x)(z-\mu_z)}{\sigma_x\sigma_z} + \left(\frac{z-\mu_z}{\sigma_z}\right)^2 \right] \right\}.$$

Par conséquent, le logarithme de cette fonction est :

$$\log f_{\mu_x,\mu_z,\sigma_x,\sigma_z,\rho}(x, z) = -\log(2\pi) - \log(\sigma_x) - \log(\sigma_z) - \frac{1}{2}\log(1-\rho^2) - \frac{1}{2(1-\rho^2)} \left[\left(\frac{x-\mu_x}{\sigma_x}\right)^2 - 2\rho\frac{(x-\mu_x)(z-\mu_z)}{\sigma_x\sigma_z} + \left(\frac{z-\mu_z}{\sigma_z}\right)^2 \right].$$

L'étape M consiste à calculer :

$$\text{Argmax}_\Theta \log(L^{t+1}(\mathbf{x}^o, \Theta)) = \text{Argmax}_\Theta \sum_{i=1}^{n} \sum_{k=0}^{1} Q_i^{t+1}(y_k) \log\left(\frac{p(x_i, z_i, y_k|\Theta)}{Q_i^{t+1}(y_k)} \right)$$

$$= \text{Argmax}_\Theta \sum_{i=1}^{n} Q_i^{t+1}(y_0) \log(\pi_0 f_{\mu_{x0},\mu_{z0},\sigma_{x0},\sigma_{z0},\rho_0}(x, z)) + Q_i^{t+1}(y_1) \log(\pi_1 f_{\mu_{x1},\mu_{z1},\sigma_{x1},\sigma_{z1},\rho_1}(x, z))$$

Donc, étant donné que $\pi_1 = 1 - \pi_0$

$$\frac{\partial \log(L^{t+1}(\mathbf{x}^o, \Theta))}{\partial \pi_0} = \sum_{i=1}^{n} Q_i^{t+1}(y_0) \frac{1}{\pi_0} - \sum_{i=1}^{n} Q_i^{t+1}(y_1) \frac{1}{1-\pi_0} = 0$$

D'où :

$$\pi_0 = \frac{\sum_{i=1}^{n} Q_i^{t+1}(y_0)}{\sum_{i=1}^{n} Q_i^{t+1}(y_0) + Q_i^{t+1}(y_1)} \qquad (1)$$

et

$$\pi_1 = 1 - \pi_0 = \frac{\sum_{i=1}^{n} Q_i^{t+1}(y_1)}{\sum_{i=1}^{n} Q_i^{t+1}(y_0) + Q_i^{t+1}(y_1)} \qquad (2)$$

Calculons maintenant les expressions des μ:

$$\frac{\partial \log(L^{t+1}(\mathbf{x}^o, \Theta))}{\partial \mu_0} = \frac{\partial}{\partial \mu_{x0}} \sum_{i=1}^{n} Q_i^{t+1}(y_0) \left(-\frac{1}{2(1-\rho_0^2)} \right) \left[\left(\frac{x_i - \mu_{x0}}{\sigma_{x0}}\right)^2 - 2\rho_0 \frac{(x_i - \mu_{x0})(z_i -}{\sigma_{x0}\sigma_{z0}} \right.$$

$$\iff \sum_{i=1}^{n} Q_i^{t+1}(y_0) \left(-\frac{1}{2(1-\rho_0^2)} \right) \left[2\left(\frac{x_i - \mu_{x0}}{\sigma_{x0}^2}\right) - 2\rho_0 \frac{z_i - \mu_{z0}}{\sigma_{x0}\sigma_{z0}} \right] = 0$$

ce qui est équivalent à :

$$\sum_{i=1}^{n} Q_i^{t+1}(y_0) \left[2 \left(\frac{x_i - \mu_{x0}}{\sigma_{x0}} \right) - 2\rho_0 \frac{z_i - \mu_{z0}}{\sigma_{z0}} \right] = 0 \qquad (3)$$

Par symmétrie, en dérivant par rapport à $\{\mu_{z0}\}$ on a également :

$$\sum_{i=1}^{n} Q_i^{t+1}(y_0) \left[2 \left(\frac{z_i - \mu_{z1}}{\sigma_{z1}} \right) - 2\rho_0 \frac{x_i - \mu_{x0}}{\sigma_{x0}} \right] = 0 \qquad (4)$$

Si l'on ajoute ρ_0 fois l'équation (4) à l'équation (3), on obtient :

$$\sum_{i=1}^{n} Q_i^{t+1}(y_0) \times 2 \left[\frac{1 - \rho_0^2}{\sigma_{x0}} \right] (\mu_{x0} - x_i) = 0$$

ce qui est équivalent à :

$$\mu_{x0} = \frac{\sum_{i=1}^{n} Q_i^{t+1}(y_0) x_i}{\sum_{i=1}^{n} Q_i^{t+1}(y_0)} \qquad (5)$$

Par symmétrie, on a :

$$\mu_{z0} = \frac{\sum_{i=1}^{n} Q_i^{t+1}(y_0) z_i}{\sum_{i=1}^{n} Q_i^{t+1}(y_0)} \qquad (6)$$

Calculons maintenant les expressions des σ et de ρ :

$$\frac{\partial \log(L^{t+1}(\mathbf{x}^o, \Theta))}{\partial \sigma_{x0}} = \frac{\partial}{\partial \sigma_{x0}} \sum_{i=1}^{n} Q_i^{t+1}(y_0) \left\{ -\log(\sigma_{x0}) - \frac{1}{2(1-\rho_0^2)} \left[\left(\frac{x_i - \mu_{x0}}{\sigma_{x0}} \right)^2 - 2\rho_0 \frac{(x_i - \mu_{x0})(z_i - \mu_{z0})}{\sigma_{x0}\sigma_{z0}} \right] \right\}$$

$$= \sum_{i=1}^{n} Q_i^{t+1}(y_0) \left\{ -\frac{1}{\sigma_{x0}} - \frac{1}{2(1-\rho_0^2)} \left[-2\frac{(x_i - \mu_{x0})^2}{\sigma_{x0}^3} + 2\rho_0 \frac{(x_i - \mu_{x0})(z_i - \mu_{z0})}{\sigma_{x0}\sigma_{z0}^2} \right] \right\} = 0$$

$$\iff \sum_{i=1}^{n} Q_i^{t+1}(y_0) \left\{ -1 - \frac{1}{2(1-\rho_0^2)} \left[-2\frac{(x_i - \mu_{x0})^2}{\sigma_{x0}^2} + 2\rho_0 \frac{(x_i - \mu_{x0})(z_i - \mu_{z0})}{\sigma_{x0}\sigma_{z0}} \right] \right\} = 0$$

ce qui est équivalent à :

$$2(1-\rho_0^2) \sum_{i=1}^{n} Q_i^{t+1}(y_0) = \sum_{i=1}^{n} Q_i^{t+1}(y_0) \left\{ \left[2\frac{(x_i - \mu_{x0})^2}{\sigma_{x0}^2} - 2\rho_0 \frac{(x_i - \mu_{x0})(z_i - \mu_{z0})}{\sigma_{x0}\sigma_{z0}} \right] \right\}$$

Par symmétrie, quand on dérive par rapport à σ_{z0}, on obtient :

$$2(1-\rho_0^2) \sum_{i=1}^{n} Q_i^{t+1}(y_0) = \sum_{i=1}^{n} Q_i^{t+1}(y_0) \left\{ \left[2\frac{(z_i - \mu_{z0})^2}{\sigma_{z0}^2} - 2\rho_0 \frac{(x_i - \mu_{x0})(z_i - \mu_{z0})}{\sigma_{x0}\sigma_{z0}} \right] \right\}$$

En additionnant les 2 équations (7) et (8), on obtient:

$$4(1-\rho_0^2) \sum_{i=1}^{n} Q_i^{t+1}(y_0) = 2\sum_{i=1}^{n} Q_i^{t+1}(y_0) \left\{ \left[\frac{(x_i - \mu_{x0})^2}{\sigma_{x0}^2} - 2\rho_0 \frac{(x_i - \mu_{x0})(z_i - \mu_{z0})}{\sigma_{x0}\sigma_{z0}} + \frac{(z_i - \mu_z}{\sigma_{z0}^2} \right.\right.$$

Enfin, dérivons la log-vraisemblance par rapport à ρ_0:

$$\frac{\partial \log(L^{t+1}(\mathbf{x}^o, \Theta))}{\partial \rho_0} = \frac{\partial}{\partial \rho_{x0}} \sum_{i=1}^{n} Q_i^{t+1}(y_0) \left\{ -\frac{1}{2}\log(1-\rho_0^2) \right.$$
$$\left. - \frac{1}{2(1-\rho_0^2)} \left[\frac{(x_i - \mu_{x0})^2}{\sigma_{x0}^2} - 2\rho_0 \frac{(x_i - \mu_{x0})(z_i - \mu_{z0})}{\sigma_{x0}\sigma_{z0}} + \frac{(z_i - \mu_{z0})^2}{\sigma_{z0}^2} \right] \right\}$$

$$= \sum_{i=1}^{n} Q_i^{t+1}(y_0) \left\{ \frac{\rho_0}{1-\rho_0^2} \right.$$
$$- \frac{\rho_0}{(1-\rho_0^2)^2} \left[\frac{(x_i-\mu_{x0})^2}{\sigma_{x0}^2} - 2\rho_0 \frac{(x_i-\mu_{x0})(z_i-\mu_{z0})}{\sigma_{x0}\sigma_{z0}} + \frac{(z_i-\mu_{z0})^2}{\sigma_{z0}^2} \right]$$
$$\left. + \frac{1}{1-\rho_0^2} \left[\frac{(x_i-\mu_{x0})(z_i-\mu_{z0})}{\sigma_{x0}\sigma_{z0}} \right] \right\} = 0$$

$$\iff \sum_{i=1}^{n} Q_i^{t+1}(y_0) \{ \rho_0$$
$$- \frac{\rho_0}{(1-\rho_0^2)} \left[\frac{(x_i-\mu_{x0})^2}{\sigma_{x0}^2} - 2\rho_0 \frac{(x_i-\mu_{x0})(z_i-\mu_{z0})}{\sigma_{x0}\sigma_{z0}} + \frac{(z_i-\mu_{z0})^2}{\sigma_{z0}^2} \right]$$
$$\left. + \left[\frac{(x_i-\mu_{x0})(z_i-\mu_{z0})}{\sigma_{x0}\sigma_{z0}} \right] \right\} = 0$$

En remplaçant le 2ème terme par le membre gauche de l'équation (9), on obtient :

$$\sum_{i=1}^{n} Q_i^{t+1}(y_0) \left\{ \rho_0 - 2\rho_0 + \left[\frac{(x_i-\mu_{x0})(z_i-\mu_{z0})}{\sigma_{x0}\sigma_{z0}} \right] \right\} = 0$$

Par conséquent,

$$\rho_0 = \frac{\sum_{i=1}^{n} Q_i^{t+1}(y_0) \frac{(x_i-\mu_{x0})(z_i-\mu_{z0})}{\sigma_{x0}\sigma_{z0}}}{\sum_{i=1}^{n} Q_i^{t+1}(y_0)} \qquad (10)$$

Appelons $\eta = \sum_{i=1}^{n} Q_i^{t+1}(y_0)(x_i-\mu_{x0})(z_i-\mu_{z0})$. Alors:

$$\rho_0 = \frac{\eta}{\sigma_{x0}\sigma_{z0} \sum_{i=1}^{n} Q_i^{t+1}(y_0)}$$

En remplaçant ρ_0 par son expression dans l'équation (8), on obtient:

$$2 \left(1 - \left(\frac{1}{\sum_{i=1}^{n} Q_i^{t+1}(y_0)} \right)^2 \frac{\eta^2}{\sigma_{x0}^2\sigma_{z0}^2} \right) \sum_{i=1}^{n} Q_i^{t+1}(y_0) =$$
$$2 \left(\sum_{i=1}^{n} Q_i^{t+1}(y_0) \frac{(x_i-\mu_{x0})^2}{\sigma_{x0}^2} \right) - 2 \frac{\eta^2}{\sigma_{x0}^2\sigma_{z0}^2 \sum_{i=1}^{n} Q_i^{t+1}(y_0)}$$

ce qui est équivalent à

$$\sigma_{x0}^2 \sum_{i=1}^{n} Q_i^{t+1}(y_0) = \sum_{i=1}^{n} Q_i^{t+1}(y_0)(x_i-\mu_{x0})^2.$$

D'où

$$\sigma_{x0}^2 = \frac{\sum_{i=1}^{n} Q_i^{t+1}(y_0)(x_i-\mu_{x0})^2}{\sum_{i=1}^{n} Q_i^{t+1}(y_0)} \qquad (11)$$

Par symmétrie,

$$\sigma_{z0}^2 = \frac{\sum_{i=1}^{n} Q_i^{t+1}(y_0)(z_i-\mu_{z0})^2}{\sum_{i=1}^{n} Q_i^{t+1}(y_0)} \qquad (12)$$