École Normale Supérieure Paris-Saclay

Master MVA

Computational Statistics

TP 2: Expectation-Maximisation algorithm – Importance sampling

Student:
Théo Danielou

Professors:
Stéphanie Allassonnière
Pierre Clavier
Maxence Noble

November 9, 2023

1 Exercise 1: Discrete distributions

Question 1 :
Pour générer une variable aléatoire X avec une distribution discrète, nous allons utiliser la méthode de la transformée inverse. Cependant, étant donné que nous avons ici une distribution discrète, nous obtiendrons l'inverse généralisée F^{-1}.
Par définition, dans le cas d'une distribution discrète (telle que celle de l'énoncé), la fonction de répartition s'écrit :

$$F(x_k) = P(X \leq x_k) = P((X = x_1) \cup ... \cup (X = x_k)) = \sum_{i=1}^{k} p_i \tag{1.1}$$

Par définition, l'inverse généralisée de la fonction de répartition s'écrit :

$$F^{-1}(u) = inf_{x_k}\{F(x_k) \geq u\} \tag{1.2}$$

Il nous suffit maintenant de générer une variable aléatoire $U \sim \mathcal{U}(0,1)$, et on aura grâce à la méthode de la transformée inverse $X \sim F^{-1}(U)$.

Question 2 : (voir notebook) **Question 3 :** (voir notebook)

2 Exercise 2: Gaussian mixture model and the EM algorithm

Question 1 :
(i) On remarque que le problème ainsi posé θ correspond aux paramètres qui caractérisent le couple (X, Z) avec $X = (X_i)_{1 \leq i \leq n}$ et $Z = (Z_i)_{1 \leq i \leq n}$. L'ensemble des paramètres sont donc :

$$\theta = \{\{\alpha_i\}_{1 \leq i \leq n}, \{\mu_i\}_{1 \leq i \leq n}, \{\Sigma_i\}_{1 \leq i \leq n}\}$$

(ii) A l'aide de la formule des probabilités totales, et connaissant les lois des Z_i et $X_i|\theta, Z_i$, on obtient :

$$p_\theta(x) = \sum_{j=1}^{p} p_\theta(x|z=j)p\theta(z=j) = \sum_{j=1}^{p} \frac{\alpha_j}{(2\pi)^{d/2}|det\Sigma_j|^{1/2}} exp[-\frac{1}{2}(x-\mu_j)^T\Sigma_j^{-1}(x-\mu_j)]$$

(iii) Les $(x_i)_{1 \leq i \leq n}$ étant i.i.d, on obtient alors :

$$\mathcal{L}(x_1, ..., x_n; \theta) = \prod_{i=1}^{n}\sum_{j=1}^{p} \frac{\alpha_j}{(2\pi)^{d/2}|det\Sigma_j|^{1/2}} exp[-\frac{1}{2}(x_i-\mu_j)^T\Sigma_j^{-1}(x_i-\mu_j)]$$

Question 2 : (voir notebook)

Question 3 :

Notre but est de trouver les paramètres θ qui maximisent la vraisemblance $\mathcal{L} = p_\theta(x)$.
La vraisemblance pouvant se réécrire :

$$p_\theta(x) = \int p_\theta(x,z)dz = \mathbb{E}_{q(Z)}[\frac{p_\theta(x,Z)}{q(Z)}]$$

avec q une densité sur l'espace de Z.
On remarque que maximiser $p_\theta(x)$ est équivalent à maximiser $log(p_\theta(x))$. Et grâce à l'inégalité de Jensen sur l'espérance, on obtient et on identifie l'Evidence Lower Bound :

$$log(p_\theta(x)) = log(\mathbb{E}_{q(Z)}[\frac{p_\theta(x,Z)}{q(Z)}]) \geq \mathbb{E}_{q(Z)}[log(\frac{p_\theta(x,Z)}{q(Z)})] = ELBO(q,\theta)$$

Ainsi on obtient d'une part que :

$$ELBO(q,\theta) = \mathbb{E}_{q(Z)}[log(p_\theta(x,Z))] - \int q(z)log(q(z))dz$$

Et d'autre part, en faisant apparaître la divergence de Kullback Leiber $KL(.||.)$:

$$\begin{aligned} ELBO(q,\theta) &= \int q(z)log(\frac{p_\theta(z|x)}{q(z)})dz + \int q(z)log(\frac{p_\theta(x)}{q(z)})dz \\ &= \int q(z)log(\frac{p_\theta(z|x)}{q(z)})dz + log(p_\theta(x)) \\ &= -KL(q(z)||p_\theta(z|x)) + log(p_\theta(x)) \end{aligned}$$

Etant donné que c'est une divergence, on a $KL(q(z)||p_\theta(z|x)) \geq 0$ avec égalité pour $q(z) = p_\theta(z|x)$, on en déduit que si on fixe $q(z) = p_\theta(z|x)$, alors maximiser $log(p_\theta)$ revient à maximiser $ELBO(q,\theta)$.

Or maximiser $ELBO(q,\theta)$ en θ revient à maximiser $\mathbb{E}_{q(Z)}[log(p_\theta(x,Z))]$ car l'autre terme ne dépend pas de θ.

En rassemblant ces deux informations, on en retrouve l'algorithme Expectation-Maximisation qui s'écrit à l'itération $t+1$, sachant connu θ^t, et en l'appliquant à notre problème :

$$\begin{aligned} \theta^{t+1} &= argmax_\theta \mathbb{E}_{p_\theta^t(Z|x)}[logp_\theta(x,Z)] \\ &= argmax_\theta \sum_{i=1}^{n} \mathbb{E}_{p_\theta^t(Z_i|x_i)}[logp_\theta(x_i,Z_i)] \end{aligned}$$

Sachant que nous voulons maximiser selon θ, remarquons que $p_\theta^t(Z|x)$ ne dépend pas de θ et notons $\tau_{j,i} = p_\theta^t(Z_i = j|x_i)$. En remarquant que $P(Z_i = j) = \alpha_j \forall i$, calculons :

$$f(\theta) = \sum_{i=1}^{n} \mathbb{E}_{\tau_{j,i}}[log(p_\theta(x_i, Z_i)] = \sum_{i=1}^{n}\sum_{j=1}^{p}[logp_\theta(x_i|z_i = j) + log\alpha_j]\tau_{j,i}$$
$$\approx \sum_{i=1}^{n}\sum_{j=1}^{p}\tau_{j,i}[-\frac{1}{2}log|det\Sigma_j| - \frac{1}{2}(x_i - \mu_j)^T\Sigma_j^{-1}(x_i - \mu_j) + log\alpha_j]$$

(à la constante 2π près, non utile pour la suite)

Maintenant, nous devons maximiser $f(\theta)$, sous les contraintes : $\sum_i \alpha_j = 1$, $\alpha_j > 0$, Σ_j définie symétrique postive.
Le Lagrangien s'écrit alors :

$$L(\theta, \lambda) = f(\theta) + \lambda(1 - \sum_j \alpha_j)$$

On remarque que f est concave en μ_j car on a l'opposé d'une forme quadratique, de même f est concave en Σ_j car on a la somme de l'opposé d'une forme quadratique et de l'opposé du $logdet(\Sigma)$ qui est convexe, f est aussi concave en α_j car $log\alpha_j$ est concave.
Ainsi f est concave en μ_j, Σ_j et α_j individuellement, ce qui nous assure alors d'obtenir au moins un maximum local.

Grâce aux règles de dérivation : $\frac{\partial log(|detM|)}{\partial M} = M^{-1}$, $\frac{\partial Tr(AM)}{\partial M} = A$. On obtient :

$$\frac{\partial L(\theta, \lambda)}{\partial \mu_j} = \sum_{i=1}^{n} \tau_{j,i}\Sigma_j^{-1}(x - \mu_j)$$
$$\frac{\partial L(\theta, \lambda)}{\partial \Sigma_j^{-1}} = \sum_{i=1}^{n} \tau_{j,i}[\frac{1}{2}\Sigma_j - \frac{1}{2}(x_i - \mu_j)(x_i - \mu_j)^T]$$
$$\frac{\partial L(\theta, \lambda)}{\partial \alpha_j} = \frac{\sum_{i=1}^{n} \tau_{j,i}}{\alpha_j} + \lambda$$

On peut remarquer que $\tau_{j,i}$ ne dépend que de variables calculées à l'instant t, et on peut alors désormais le noter $\tau_{j,i}^t$.
Lorsqu'on annule ces dérivées partielles, on obtient :

$$\mu_j^{t+1} = \frac{\sum_{i=1}^{n} \tau_{j,i}^t x_i}{\sum_{i=1}^{n} \tau_{j,i}^t}$$
$$\Sigma_j^{t+1} = \frac{\sum_{i=1}^{n} \tau_{j,i}^t (x_i - \mu_j)(x_i - \mu_j)^T}{\sum_{i=1}^{n} \tau_{j,i}^t}$$
$$\alpha_j^{t+1} = \frac{\sum_{i=1}^{n} \tau_{j,i}^t}{\lambda}$$

Or, on sait que $\sum_j \alpha_j = 1$, ainsi $\lambda = \sum_{j=1}^{p}\sum_{i=1}^{n} \tau_{j,i}^t$, et on obtient :

$$\alpha_j^{t+1} = \frac{\sum_{i=1}^{n} \tau_{j,i}^t}{\sum_{j=1}^{p}\sum_{i=1}^{n} \tau_{j,i}^t}$$

On remarque que μ_j^{t+1}, Σ_j^{t+1} et α_j^{t+1} respectent bien les contraintes, ainsi on peut dire que $\theta^{t+1} = \{\mu_j^{t+1}, \Sigma_j^{t+1}, \alpha_j^{t+1}\}$ est un maximiseur global de $f(\theta)$.

Ainsi pour déterminer θ^{t+1} à chaque étape, il nous faut connaître l'expression des $\tau_{j,i}^t$. Pour cela, grâce à la formule de Bayes et des probabilités totales, on obtient :

$$\tau_{j,i}^t = p_{\theta^t}(Z_i = j|x_i) = \frac{p_{\theta^t}(x_i|Z_i = j)\alpha_j^t}{\sum_{k=1}^{p} p_{\theta^t}(x_i|Z_i = k)\alpha_k^t}$$

Pour simplifier les caculs dans le code nous calculerons $\tau_{j,i}^t = exp(log(\tau_{j,i}^t))$, le numérateur $\widetilde{\tau}_{j,i}^t$ s'écrira ainsi (et nous obtiendrons le dénominateur de la même manière) :

$$log\widetilde{\tau}_{j,i}^t = log\alpha_j^t - \frac{d}{2}nlog(2\pi) - \frac{1}{2}nlog|det\Sigma_j^t| - \frac{1}{2}(x_i - \mu_j^t)^T(\Sigma_j^t)^{-1}(x_i - \mu_j^t)$$

Maintenant les expressions des différentes variables connus, nous pouvons réaliser l'implémentation. (voir notebook pour l'implémentation)

Question 3 :
En notant que nos paramètres estimés ne gardent pas l'ordre, c'est à dire que $\hat{\mu}_1$ peut correspondre à l'estimation de μ_2 par exemple. On remarque alors que la proximité de nos paramètres estimés avec les paramètres réels va dépendre de l'initialisation, les paramètres sont en moyennes assez proches mais peuvent parfois être très éloignés. De plus, nous pouvons noter que nous avons du rajouter des constantes ϵ afin d'éviter des valeurs interdites (division par 0) qui peuvent notamment provenir du fait que les valeurs de notre matrice de covariance devient trop faible. On peut alors remarquer qu'en fonction de la valeur donnée à cette constante et la manière de l'ajouter les résultats peuvent varier.
Pour finir, l'algorithme fournit dans le notebook pourrait être optimisé (beaucoup de double boucles for) en utilisant mieux les caractéristiques des numpy arrays, ce qui réduirait grandement le temps de calcul.

Question 4 : (voir notebook)

Question 5 : (voir notebook)

3 Exercise 3: Importance sampling

Question 1 : (voir notebook)

Question 2 : (voir notebook)

Question 3 : (voir notebook)

Question 4 :
L'étape (iii) correspond à trouver θ^{t+1} vérifiant :

$$\theta^{t+1} = argmax_\theta \sum_{i=1}^{n} \widetilde{\omega}_i^t log q_\theta(X_i^t) \tag{3.1}$$

Or dans l'algorithme EM, on cherche à trouver les paramètres θ qui maximisent la vraisemblance $q_\theta(x)$, ce qui est équivalent à maximiser la logvraisemblance $log(q_\theta(x))$. Etant donné que nous sommes toujours dans le cas d'un mélange de gaussiennes, nous pouvons reprendre les résultats de l'exercice 2, tout en faisant attention que cette fois nous avons un paramètre $\widetilde{\omega}_i^t$ dont dépend θ^{t+1} et qui va varier à chaque étape. On réécrivant cette expression, on obtient alors que nous allons finalement devoir trouver à chaque étape θ^{t+1} tel que :

$$\theta^{t+1} = argmax_\theta \sum_{i=1}^{n} \widetilde{\omega}_i^t \mathbb{E}_{q_\theta^t(Z|X_i)}[log q_\theta(X_i, Z)]$$

En ayant introduit la variable latente Z telle que la probabilité à postériori que l'échantillon X_i appartienne la gaussienne j (que je note par la suite $\tau_{j,i}$) s'écrit :

$$\tau_{j,i} = P(Z_i = j|X_i) = \frac{\varphi(X_i; \mu_j^t, \Sigma_j^t)\alpha_j^t}{q_\theta(X_i)}$$

De la même manière qu'à l'exercice 2, nous avons :

$$f(\theta) = \sum_{i=1}^{n} \widetilde{\omega}_i^t \mathbb{E}_{\tau_{j,i}}[log(q_\theta(x_i, Z_i)] = \sum_{i=1}^{n}\sum_{j=1}^{K} [log q_\theta(x_i|z_i = j) + log\alpha_j]\tau_{j,i}\widetilde{\omega}_i^t$$

En remarquant alors que $\widetilde{\omega}_i^t$ ne dépend pas de j, il devient un facteur multiplicatif devant la somme sur i, on va donc pouvoir retrouver θ^{t+1} directement en reprenant les mêmes étapes que précédemment (introduction du lagrangien, calcul du gradient et des valeurs pour lesquelles il s'annule,...) telles que :

$$\mu_j^{t+1} = \frac{\sum_{i=1}^n \widetilde{\omega}_i^t \tau_{j,i}^t X_i^t}{\sum_{i=1}^n \widetilde{\omega}_i^t \tau_{j,i}^t}$$

$$\Sigma_j^{t+1} = \frac{\sum_{i=1}^n \widetilde{\omega}_i^t \tau_{j,i}^t (X_i^t - \mu_j)(X_i^t - \mu_j)^T}{\sum_{i=1}^n \widetilde{\omega}_i^t \tau_{j,i}^t}$$

$$\alpha_j^{t+1} = \frac{\sum_{i=1}^n \widetilde{\omega}_i^t \tau_{j,i}^t}{\sum_{j=1}^K \sum_{i=1}^n \widetilde{\omega}_i^t \tau_{j,i}^t}$$

Question 5 : (voir notebook)

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg as la
from scipy.special import logsumexp
import pandas as pd
from tqdm import tqdm
```

Exercise 1 : Discrete distributions

Question 2 :

In [ ]:
```
# Génération de p valeurs aléatoires différentes qui correspondront à nos x_i, il
p = 10
x_i = np.random.randint(100,size=p)
x_i = np.sort(x_i)

# Générons p valeurs aléatoires qui correspondront à nos probabilités alpha_true
alpha_true = np.random.rand(p)
alpha_true = alpha_true/np.sum(alpha_true)
print("p probabilités :", alpha_true)
print("p valeurs de x :", x_i)
```

```
p probabilités : [0.08034899 0.07692466 0.14363876 0.23048222 0.02460988 0.083196
81
 0.0148661  0.10508626 0.16699519 0.07385114]
p valeurs de x : [ 9 24 28 31 38 45 64 69 88 92]
```

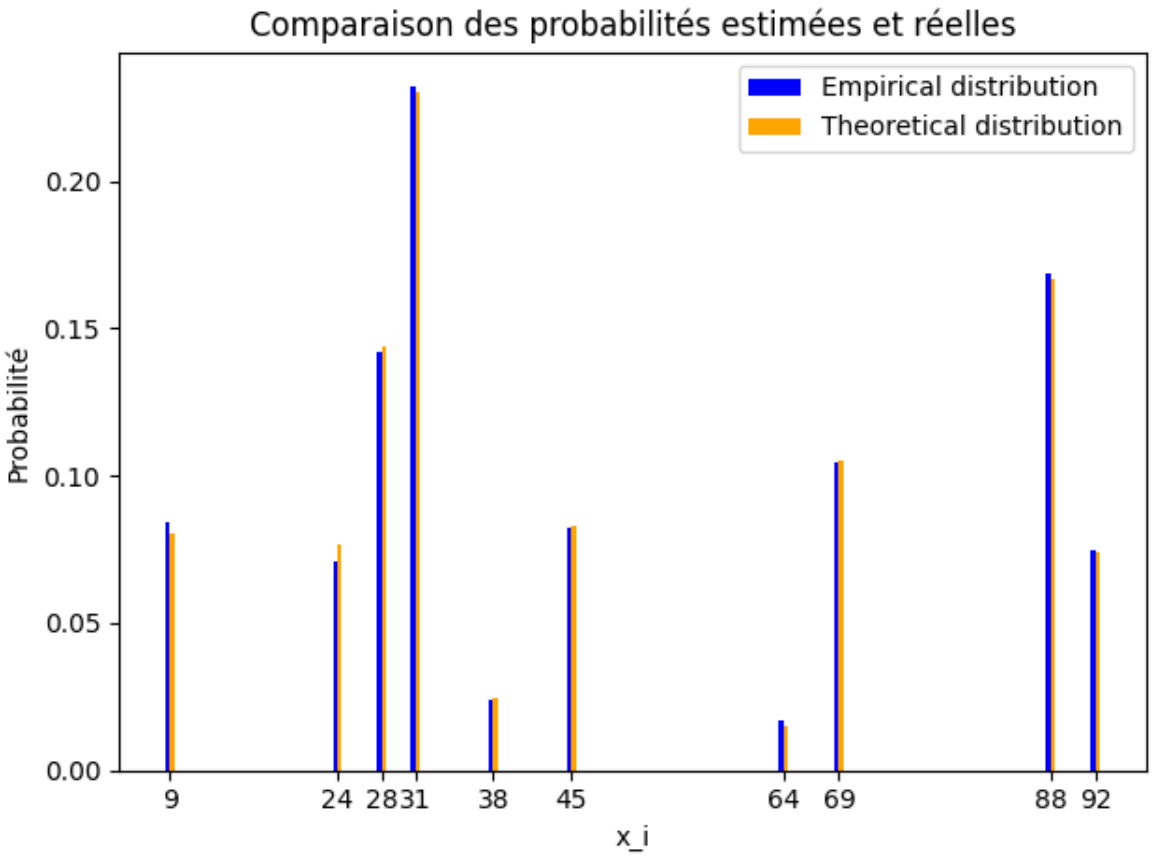
In [ ]:
```
def F(x, alpha_true, x_i): # F(x) = P(X <= x) = proba
    k = 0
    proba = 0
    while (x >= x_i[k]):
        proba += alpha_true[k]
        k += 1
        if k == len(x_i):
            break
    return proba

print("proba", F(64, alpha_true, x_i))

# Pour créer notre fonction de répartition inverse généralisée d'une variable al
def F_1(u, pi, x_i):
    k = 0
    """while (u > F(x_i[k], pi)):
        k += 1
        if k == len(x_i):
            break"""
    while (u > F(x_i[k], pi, x_i) and k < len(x_i) - 1):
        k += 1
    return x_i[k]

print("inverse :", F_1(0.65, alpha_true, x_i))
```

```
proba 0.6540674093620881
inverse : 64
```

Question 3 :

In [ ]:
```
# Générons n variables aléatoires qui suivent la loi de probabilité pi
n = 10000
X = []
for i in range(n):
    X.append(F_1(np.random.rand(), alpha_true, x_i))

extended_bins = np.append(x_i, [x_i[-1] + 1]) #Sert à résoudre le problème sur l
X = np.array(X)

hist, bins = np.histogram(X, bins=extended_bins)
estimated_alpha_true = hist / n
print("estimated_alpha_true", estimated_alpha_true)
print("alpha_true", alpha_true)
print("x_i", x_i)

plt.bar(x_i - 0.4/2, estimated_alpha_true, width=0.4, label='Empirical distribut
plt.bar(x_i + 0.4/2, alpha_true, width=0.4, label='Theoretical distribution', co

plt.xlabel('x_i')
plt.ylabel('Probabilité')
plt.title('Comparaison des probabilités estimées et réelles')
plt.xticks(x_i)
plt.legend()
plt.tight_layout()
plt.show()
```

```
estimated_alpha_true [0.084  0.0706 0.1418 0.2322 0.0241 0.0821 0.0169 0.1046 0.1
687 0.075 ]
alpha_true [0.08034899 0.07692466 0.14363876 0.23048222 0.02460988 0.08319681
 0.0148661  0.10508626 0.16699519 0.07385114]
x_i [ 9 24 28 31 38 45 64 69 88 92]
```



On remarque que nos valeurs de probabilités estimées sont très proches des valeurs théoriques de probabilité.

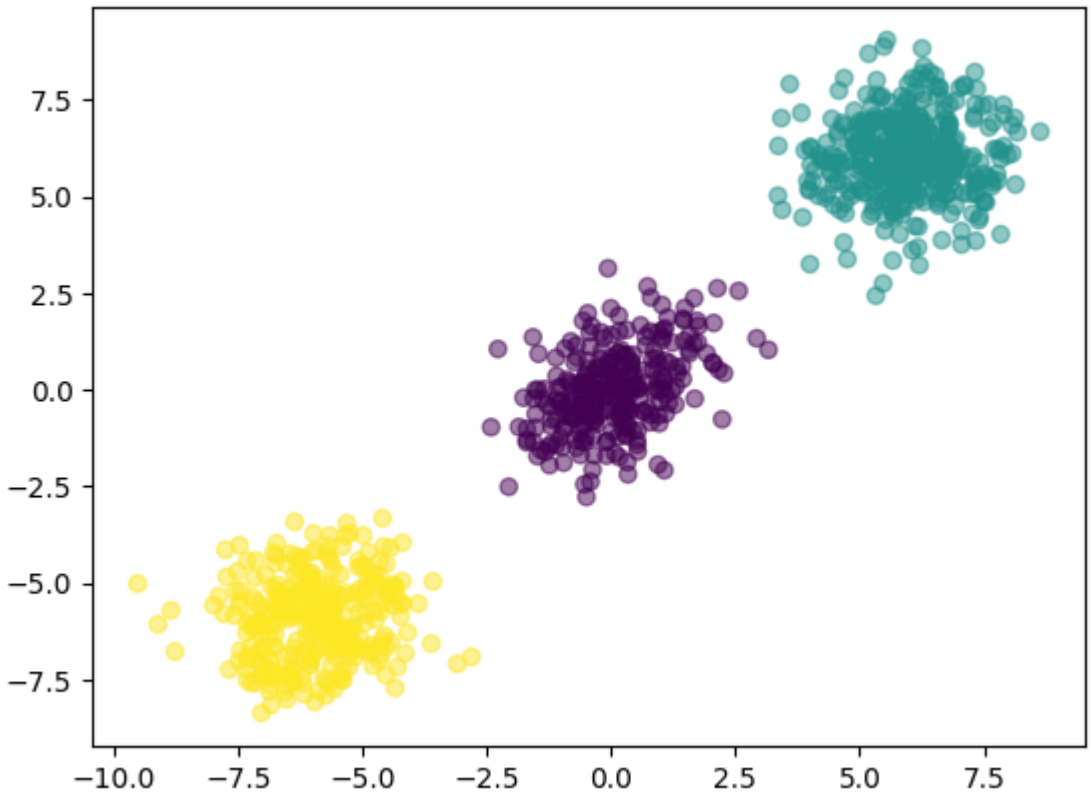
Exercise 2 : Gaussian mixture model and the EM algorithm

Question 2 :

In [ ]:
```
# Prenons le cas où nous avons 3 clusters, p = 3
p = 3
d = 2
z_values = [1,2,3]
alpha_true = np.random.rand(p)
alpha_true = alpha_true/np.sum(alpha_true)
mu_i = np.array([[0,0], [6, 6], [-6, -6]]) #Je prends des valeurs espacées afin
sigma_i = np.array([[[1,0.5],[0.5,1]], [[1, 0], [0, 1]], [[1, 0], [0, 1]]]) #Je



# Créons n tirages de la variable aléatoire Z qui peut prendre 3 valeurs [1,2,3]
n = 1000
Z = []
for i in range(n):
    Z.append(F_1(np.random.rand(), alpha_true, z_values))
Z = np.array(Z)

# Créons n tirages de la variable aléatoire X qui suit la loi normale N(mu_i, si
```

```
X = []
for i in range(n):
    X.append(np.random.multivariate_normal(mu_i[Z[i]-1], sigma_i[Z[i]-1])) #On s
X = np.array(X)

plt.scatter(X[:,0], X[:,1], c=Z, cmap='viridis', alpha=0.5)
```

Out[ ]: <matplotlib.collections.PathCollection at 0x7f80aea1a8c0>



Question 3 :

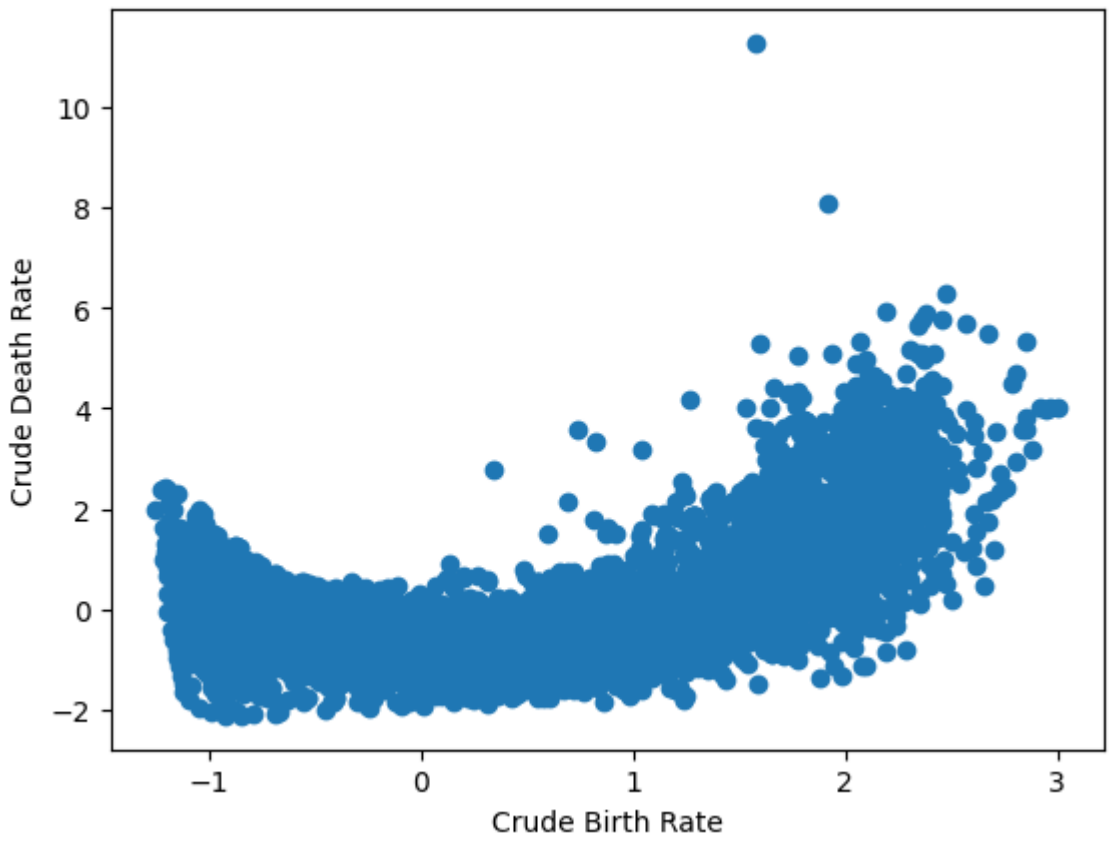
In [ ]:
```
def generate_random_sdp(d):
    """Génère une matrice symétrique définie positive aléatoire de taille d x d.
    A = np.random.randn(d, d)  # Matrice aléatoire
    sdp_matrix = np.dot(A.T, A) + np.eye(d) * 1e-1  # A^T * A + petite matrice i
    return sdp_matrix

def EM(X,p):
    n = len(X)
    d = len(X[0])
    #Initialisation
    #Il nous faut des valeurs aléatoires différentes pour les mu_j
    mu = np.random.rand(p, d)
    #Il nous faut des valeurs aléatoires différentes pour les sigma_j en respect
    sigma = np.array([generate_random_sdp(d) for _ in range(p)])
    #On initialise les probabilités alpha_j avec des valeurs aléatoires
    alpha = np.random.rand(p)
    alpha = alpha/np.sum(alpha)
    max_iter = 200
    epsilon = 0.01
    for _ in range(max_iter):
        #EXPECTATION
        tau=np.zeros((p,n))
        log_tau = np.zeros((p, n))
```

```
        for j in range(p):
            for i in range(n):
                log_tau[j, i] = np.log(alpha[j] + epsilon) - 0.5 * d * np.log(2
        logsum = logsumexp(log_tau, axis=0)
        tau = np.exp(log_tau - logsum)
        tau[np.isnan(tau)] = 0 #On remplace les valeurs nan par des 0

        #MAXIMISATION
        mu = np.zeros((p,d))
        sigma = np.zeros((p,d,d))
        alpha = np.zeros(p)
        sum_tau = np.sum(tau, axis=1)
        #On calcule les nouvelles valeurs de alpha_j
        alpha = sum_tau / np.sum(tau)
        alpha = alpha/np.sum(alpha) # On normalise alpha pour que la somme des a

        sum_tau = np.sum(tau, axis=1)
        #On calcule les nouvelles valeurs de alpha_j
        for j in range(p):
            for i in range(n):
                mu[j] += tau[j,i]*X[i] #On calcule les nouvelles valeurs de mu_j
            mu[j] = mu[j]/sum_tau[j]
        for j in range(p):
            for i in range(n):
                sigma[j] += tau[j,i]* np.outer((X[i] - mu[j]),(X[i] - mu[j])) +
            sigma[j] = sigma[j]/sum_tau[j]
    return alpha, mu, sigma

p=3
alpha, mu, sigma = EM(X,p)

print("alpha estimé:", alpha)
print("alpha réel:", alpha_true)

print("mu estimé:")
print(mu)
print("mu réel:")
print(mu_i)

print("sigma estimé:")
print(sigma)
print("sigma réel:")
print(sigma_i)
```

```
alpha estimé: [0.39799157 0.31699982 0.28500861]
alpha réel: [0.27526566 0.41227666 0.31245768]
mu estimé:
[[ 5.95551     6.01928759]
 [-5.95262185 -5.92169343]
 [ 0.04334014 -0.01512734]]
mu réel:
[[ 0  0]
 [ 6  6]
 [-6 -6]]
sigma estimé:
[[[1.01504163 0.05731493]
  [0.05731493 1.12583737]]

 [[1.04388066 0.10066563]
  [0.10066563 1.12813504]]

 [[0.95181962 0.43576803]
  [0.43576803 1.15895411]]]
sigma réel:
[[[1.  0.5]
  [0.5 1. ]]

 [[1.  0. ]
  [0.  1. ]]

 [[1.  0. ]
  [0.  1. ]]]
```

Question 4 :

In [ ]:
```
file = "WPP2019_Period_Indicators_Medium.csv"
df = pd.read_csv(file)
df = df[["CDR", "CBR"]]
df = df.dropna()
mean = df.mean()
std = df.std()
df = (df-mean)/std
plt.scatter(df["CBR"], df["CDR"])
plt.ylabel("Crude Death Rate")
plt.xlabel("Crude Birth Rate")
X = df[["CDR", "CBR"]].to_numpy()
```



Les données montrent une structure complexe. La distribustion des données ne semble pas linéairment séparable. On peut identifier à priori plusieurs groupes d'agrégats avec des formes elliptiques. On peut alors penser qu'utiliser une mélange de gaussiennes ici pour représenter les données paraît adéquat.

Question 5 :

In [ ]:
```
theta = {"alpha":[], "mu":[], "sigma":[]}
for p in tqdm(range(1, 8), desc="p"):
    alpha, mu, sigma = EM(X,p)
    theta["alpha"].append(alpha)
    theta["mu"].append(mu)
    theta["sigma"].append(sigma)
    print("mu:", mu)
```

```
p:  14%|██          | 1/7 [01:13<07:19, 73.21s/it]
mu: [[ 2.03570126e-16 -2.63913964e-16]]
p:  29%|████        | 2/7 [03:39<09:41, 116.27s/it]
mu: [[-0.21161612 -0.60771031]
 [ 0.42906821  1.23218012]]
p:  43%|██████      | 3/7 [07:16<10:48, 162.09s/it]
mu: [[ 1.18830487  1.68486607]
 [ 0.02568702 -0.77713508]
 [-0.73413644  0.08532858]]
p:  57%|████████    | 4/7 [12:09<10:41, 213.68s/it]
mu: [[-0.47007977  0.64327593]
 [-0.73250155 -0.11419915]
 [ 1.54376866  1.79135501]
 [ 0.06792126 -0.79653415]]
p:  71%|██████████  | 5/7 [18:14<08:56, 268.30s/it]
```

```
mu: [[-0.73138953 -0.12179442]
 [ 1.79687218  1.16660978]
 [ 0.06995009 -0.79782924]
 [ 1.46014942  1.77706233]
 [-0.50158015  0.60713913]]
p:  86%|████████▌ | 6/7 [25:30<05:25, 325.45s/it]
mu: [[-0.05741317  0.66830788]
 [ 1.48788089  1.78391686]
 [ 0.06856089 -0.79745321]
 [-0.73422038 -0.11740559]
 [ 1.87163903  1.20433148]
 [-0.50020388  0.61351669]]
p: 100%|██████████| 7/7 [34:01<00:00, 291.63s/it]
mu: [[ 0.06836569 -0.79761066]
 [ 1.90885072  1.26067839]
 [ 0.04147973  0.69493426]
 [-0.50254631  0.61256331]
 [ 1.49812802  1.78660182]
 [ 0.12648265  0.71498247]
 [-0.73436113 -0.11927897]]
```

In [ ]:
```
theta["alpha"]
```

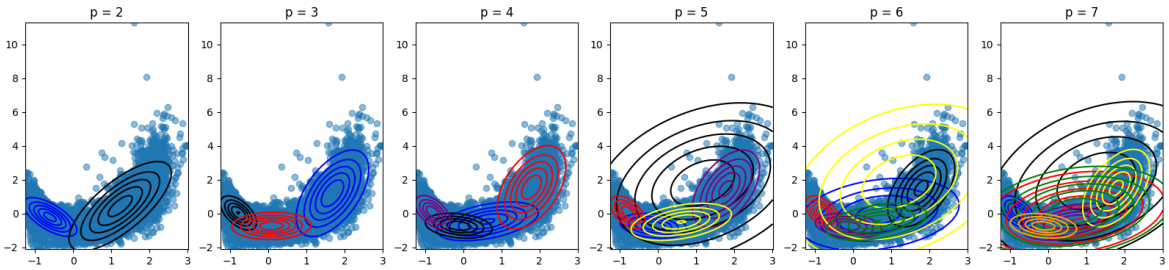
Out[ ]:
```
[array([1.]),
 array([0.66970299, 0.33029701]),
 array([0.19779041, 0.46576079, 0.33644879]),
 array([0.15444336, 0.26251166, 0.15265111, 0.43039387]),
 array([0.2605968 , 0.00893171, 0.42660895, 0.15145753, 0.15240501]),
 array([0.0179002 , 0.14657029, 0.42601705, 0.25962071, 0.00815219,
        0.14173956]),
 array([0.42432947, 0.00764816, 0.0145292 , 0.13890474, 0.14276262,
        0.01316878, 0.25865704])]
```

On remarque que plus on augmente p, i.e le nombre de gaussiennes différentes, plus certains centres de ces gaussiennes paraissent très proches. Nous pouvons de plus reamrquer qu'à part pour le cas p = 4, on a toujours un cluster qui a une probabilité très faible d'apparition (de l'odre de 1%). Afin de mieux comprendre les paramètres obtenus, affichons les différents contours générées par ces mélanges de gaussiennes.

In [ ]:
```
def gaussian_density(x, mu, sigma):
    d = len(mu)
    return 1 / ((2 * np.pi)**(d / 2) * np.linalg.det(sigma)**0.5) * np.exp(-0.5


fig, axes = plt.subplots(1, 6, figsize=(20, 4))
grid_x, grid_y = np.mgrid[min(X[:,0]):max(X[:,0]):100j, min(X[:,1]):max(X[:,1]):
for u in range(2, 8):
    mu_p = theta["mu"][u-1]
    sigma_p = theta["sigma"][u-1]
    axes[u-2].scatter(X[:,1], X[:,0], cmap='viridis', alpha=0.5)
    Z = np.zeros(grid_x.shape)
    # Créons un contour plot pour chaque composante de la mixture d'une couleur
    colors = ['blue', 'black', 'red', 'purple', 'yellow', 'green', 'orange']
    for i in range(u):
        # On calcule la densité de la composante courante pour chaque point du g
        Z_component = np.zeros_like(grid_x)
        for j in range(grid_x.shape[0]):
            for k in range(grid_x.shape[1]):
```

```
                x_point = np.array([grid_x[j, k], grid_y[j, k]])
                Z_component[j, k] = gaussian_density(x_point, mu_p[i], sigma_p[i

        # Plot the contours for the current Gaussian density
        axes[u-2].contour(grid_y, grid_x, Z_component, levels=5, colors=colors[i
        axes[u-2].set_title("p = " + str(u))
```

```
/tmp/ipykernel_240/4155921695.py:17: UserWarning: No data for colormapping provid
ed via 'c'. Parameters 'cmap' will be ignored
  axes[u-2].scatter(X[:,1], X[:,0], cmap='viridis', alpha=0.5)
```



On remarque après visualisation de ces différents mélanges de gaussiennes, qu'à partir de p=5 il y a l'introduction de gaussiennes qui "chevauchent" toutes les autres, sans doute dans l'optique de pouvoir représenter les outliers. Si on prend en compte cette remarque, on peut donc penser que le mélange de gaussienne optimal serait pour p=4. Regardons maintenant si le Bayesian Information Criterion confirme cette hypothèse.

In [ ]:
```
def df(p):
    return 6*p -1

def log_vraisemblance(X, theta, p):
    alpha = theta["alpha"][p-1]
    mu = theta["mu"][p-1]
    sigma = theta["sigma"][p-1]
    n = len(X)
    d = len(X[0])
    l = 0
    for i in range (n):
        test = 0
        for j in range(p):
            test += alpha[j] / ((2*np.pi)**(d/2) * np.linalg.det(sigma[j])**(1/2
        l += np.log(test)
    return l

def BIC(X, theta):
    minimum = np.inf
    p_min = 0
    for p in range(1, 8):
        test = df(p) * np.log(len(X))/2 - log_vraisemblance(X, theta, p)
        if minimum > test:
            minimum = test
            p_min = p
    return p_min

p_min = BIC(X, theta)
print("p_min", p_min)
```

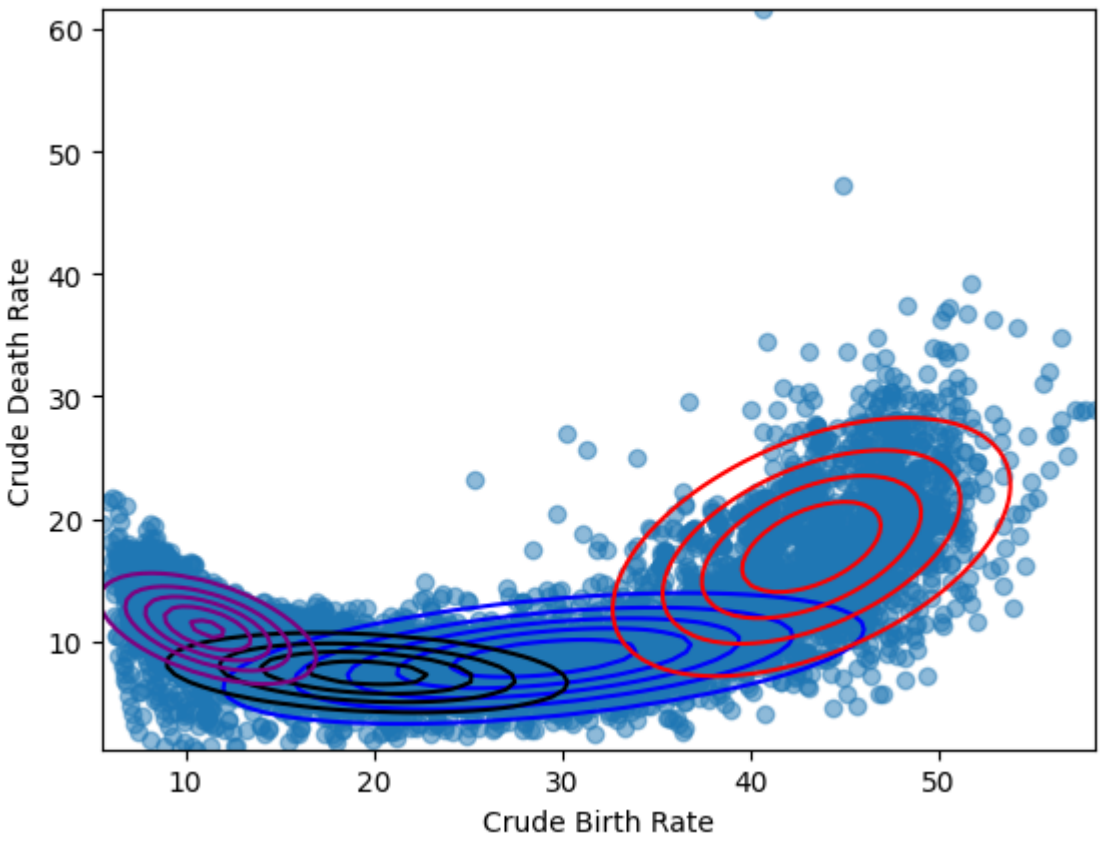
```
BIC 4
```

Le Bayesian Information Criterion confirme alors l'hypothèse que nous avions formulé. Affichons désormais le meilleur mélange de gaussiennes, au sens du BIC, sur nos données.

In [ ]:
```
alpha_p = theta["alpha"][p_min-1]
mu_p = theta["mu"][p_min-1]
sigma_p = theta["sigma"][p_min-1]

# Calculer les mu dénormalisés
mu_denormalized = np.empty_like(mu_p)
mu_denormalized = mu_p * std + mean

# Calculer les sigmas dénormalisés
sigma_p_denormalized = np.empty_like(sigma_p)
for k, sigma in enumerate(sigma_p):
    for i in range(sigma.shape[0]):
        for j in range(sigma.shape[1]):
            sigma_p_denormalized[k, i, j] = sigma[i, j] * std[i] * std[j]

# Calculer les X dénormalisés
X_denormalized = np.empty_like(X)
for i in range(X.shape[0]):
    for j in range(X.shape[1]):
        X_denormalized[i, j] = X[i, j] * std[j] + mean[j]
```

In [ ]:
```
plt.scatter(X_denormalized[:,1], X_denormalized[:,0], cmap='viridis', alpha=0.5)

Z = np.zeros(grid_x.shape)

grid_x, grid_y = np.mgrid[min(X_denormalized[:,0]):max(X_denormalized[:,0]):100j
colors = ['blue', 'black', 'red', 'purple']
for i in range(p_min):
    Z_component = np.zeros_like(grid_x)
    for j in range(grid_x.shape[0]):
        for k in range(grid_x.shape[1]):
            x_point = np.array([grid_x[j, k], grid_y[j, k]])
            Z_component[j, k] = gaussian_density(x_point, mu_denormalized[i], si

    plt.contour(grid_y, grid_x, Z_component, levels=5, colors=colors[i])

plt.ylabel("Crude Death Rate")
plt.xlabel("Crude Birth Rate")
plt.show()
```

```
/tmp/ipykernel_240/1514166660.py:12: UserWarning: No data for colormapping provid
ed via 'c'. Parameters 'cmap' will be ignored
  plt.scatter(X_denormalized[:,1], X_denormalized[:,0], cmap='viridis', alpha=0.
5)
```



Exercise 3 : Importance sampling

3-A Poor importance sampling

Question 1 :

In [ ]:
```
n = 1000

def p(x):
    return x**0.65 * np.exp(-x**2/2)

def q(x):
    return np.exp(-(0.8-x)**2/2)/np.sqrt(2*np.pi*1.5)

def f(x):
    return 2 * np.sin(2*np.pi*x/3)

def importance_sampling(n):
    # Générons n valeurs aléatoires suivant la loi normal N(0.8, 1.5)
    X = np.random.normal(0.8, 1.5, n)
    #Vérifions que tous les X_i sont bien positifs
    for i in range(n):
        while X[i] < 0:
            X[i] = np.random.normal(0.8, 1.5)
    weights_normalized = p(X)/q(X) / np.sum(p(X)/q(X))
    esperance = 1/n * np.sum(f(X) * weights_normalized)
    return esperance
esperance = importance_sampling(n)
print("Estimation de E_p[f(X)]:", esperance)
```

```
Estimation de E_p[f(X)]: 0.0004964051235655967
```

Question 2 :

In [ ]:
```
Ns = [10, 100, 1000, 10000]
esperances = []

for N in Ns:
    esperances = [importance_sampling(N) for _ in range(100)]
    moyenne_esperances = np.mean(esperances)
    variance_esperances = np.var(esperances)
    print(f"Pour N={N}, Moyenne de l'estimation: {moyenne_esperances}, Variance
```

```
Pour N=10, Moyenne de l'estimation: 0.05069611956098941, Variance de l'estimatio
n: 0.0023745423002229024
Pour N=100, Moyenne de l'estimation: 0.0053148031088937975, Variance de l'estimat
ion: 1.7232848233004652e-06
Pour N=1000, Moyenne de l'estimation: 0.000544695271699419, Variance de l'estimat
ion: 1.7366651488726422e-09
Pour N=10000, Moyenne de l'estimation: 5.420140997976712e-05, Variance de l'estim
ation: 1.7242063722240916e-12
```

Question 3 :

In [ ]:
```
def q(x, mu):
    return np.exp(-(mu-x)**2/2)/np.sqrt(2*np.pi*1.5)

def importance_sampling(n, mu):
    # Générons n valeurs aléatoires suivant la loi normal N(0.8, 1.5)
    X = np.random.normal(mu, 1.5, n)
    #Vérifions que tous les X_i sont bien positifs
    for i in range(n):
        while X[i] < 0:
            X[i] = np.random.normal(mu, 1.5)
    weights_normalized = p(X)/q(X, mu) / np.sum(p(X)/q(X, mu))
    esperance = 1/n * np.sum(f(X) * weights_normalized)
    return esperance, weights_normalized

Ns = [10, 100, 1000, 10000]
esperances = []

for N in Ns:
    esperances = [importance_sampling(N, 6)[0] for _ in range(100)]
    moyenne_esperances = np.mean(esperances)
    variance_esperances = np.var(esperances)
    print(f"Pour N={N}, Moyenne de l'estimation: {moyenne_esperances}, Variance
```

```
Pour N=10, Moyenne de l'estimation: 0.058667924287771386, Variance de l'estimatio
n: 0.015085560284653124
Pour N=100, Moyenne de l'estimation: -0.007710395054266999, Variance de l'estimat
ion: 0.0001310734148998352
Pour N=1000, Moyenne de l'estimation: 0.00041044807902220626, Variance de l'estim
ation: 1.2968084980469294e-06
Pour N=10000, Moyenne de l'estimation: 0.0001310226052871473, Variance de l'estim
ation: 1.4536257766099126e-09
```

In [ ]:
```
_, weights_q3 = importance_sampling(1000, 0.8)
_, weights_q3_6 = importance_sampling(1000, 6)
```

```
print("Poids d'importance normalisé, pour la moyenne 0.8", weights_q3[:10])
print("Poids d'importance normalisé, pour la moyenne 6", weights_q3_6[:10])
```

```
Poids d'importance normalisé, pour la moyenne 0.8 [0.00125352 0.0004137  0.000857
76 0.00076707 0.00136857 0.00126147
 0.00100559 0.00096247 0.00109153 0.00051627]
Poids d'importance normalisé, pour la moyenne 6 [1.57698300e-14 5.70798291e-06 7.
38477379e-04 4.30828232e-21
 1.21535766e-05 1.25267578e-14 1.04934469e-17 1.19679614e-14
 2.64777166e-06 1.98419104e-14]
```

On peut donc noter que les poids d'importances normalisés sont bien plus faibles pour la moyenne égale à 6 que pour la moyenne égale à 0.8. Par soucis de lisibilité, je n'ai affiché que les 10 premiers, mais c'est valable pour tous les autres aussi.

Question 5 :

In [ ]:

```
def mixture_model_sample(theta, n_samples):
    """
    Fonction générant un sample selon un mélange de gaussiennes, c'est une génér

    """

    alpha = theta[0]
    mu = theta[1]
    sigma = theta[2]

    d = len(mu[0])
    # Grâce au code l'exercice 1 on peut générer un sample selon une loi multino
    z_values = np.linspace(1, len(alpha), len(alpha))
    Z = []
    for i in range(n_samples):
        Z.append(F_1(np.random.rand(), alpha, z_values))
    Z = np.array(Z)

    samples = []
    # On calcule le nombre de samples à générer selon chaque gaussienne
    n_samples_per_component = [np.sum(Z == k) for k in range(1, len(alpha)+1)]

    # Pour chaque gaussienne on génère des samples selon la loi normale correspo
    for i in range(len(alpha)):
        samples_component = np.random.multivariate_normal(mu[i], sigma[i] +np.ey
        samples.append(samples_component)

    # On regroupe tous les samples dans un seul array
    samples = np.vstack(samples)
    np.random.shuffle(samples)

    return samples

def banana_density(x, b, sigma_square):
    """
    Fonction calculant la densité de la distribution en banane en un point x.
    """

    d = len(x)

    mean = np.zeros(d)
```

```
    cov = np.diag([sigma_square] + [1] * (d - 1))

    new_x = x
    new_x[1] = new_x[1] + b * (x[0]**2 - sigma_square)

    density = 1 / ((2 * np.pi)**(d / 2) * np.linalg.det(cov)**0.5) * np.exp(-0.5

    return density

def mixture_model_density(x, theta):
    """
    Fonction calculant la densité d'un mélange de gaussiennes en un point x.

    """

    alpha = theta[0]
    mu = theta[1]
    sigma = theta[2]

    d = len(x)

    density = 0
    for i in range(len(alpha)):
        sigma_reg = sigma[i] + np.eye(d) * 1e-6
        component_density = 1 / ((2 * np.pi)**(d / 2) * np.linalg.det(sigma_reg)
        density += alpha[i] * component_density

    return density



def EM_weights(X, weights, p):
    """
    Algorithme EM de la question 4. les tau_ij ne dépendant pas des poids w_i, i

    """
    n = len(X)
    d = len(X[0])
    #Initialisation
    #Il nous faut des valeurs aléatoires différentes pour les mu_j
    mu = np.random.rand(p, d)
    #Il nous faut des valeurs aléatoires différentes pour les sigma_j en respect
    sigma = np.array([generate_random_sdp(d) for _ in range(p)])
    #On initialise les probabilités alpha_j avec des valeurs aléatoires
    alpha = np.random.rand(p)
    alpha = alpha/np.sum(alpha)
    max_iter = 100
    epsilon = 1e-6
    for _ in range(max_iter):
        #EXPECTATION
        tau=np.zeros((p,n))
        log_tau = np.zeros((p, n))
        for j in range(p):
            for i in range(n):
                sigma_reg = sigma[j] + np.eye(d) * epsilon
                log_tau[j, i] = np.log(alpha[j]) - 0.5 * d * np.log(2 * np.pi) -
        logsum = logsumexp(log_tau, axis=0)
        tau = np.exp(log_tau - logsum)
        tau[np.isnan(tau)] = 0 #On remplace les valeurs nan par des 0
```

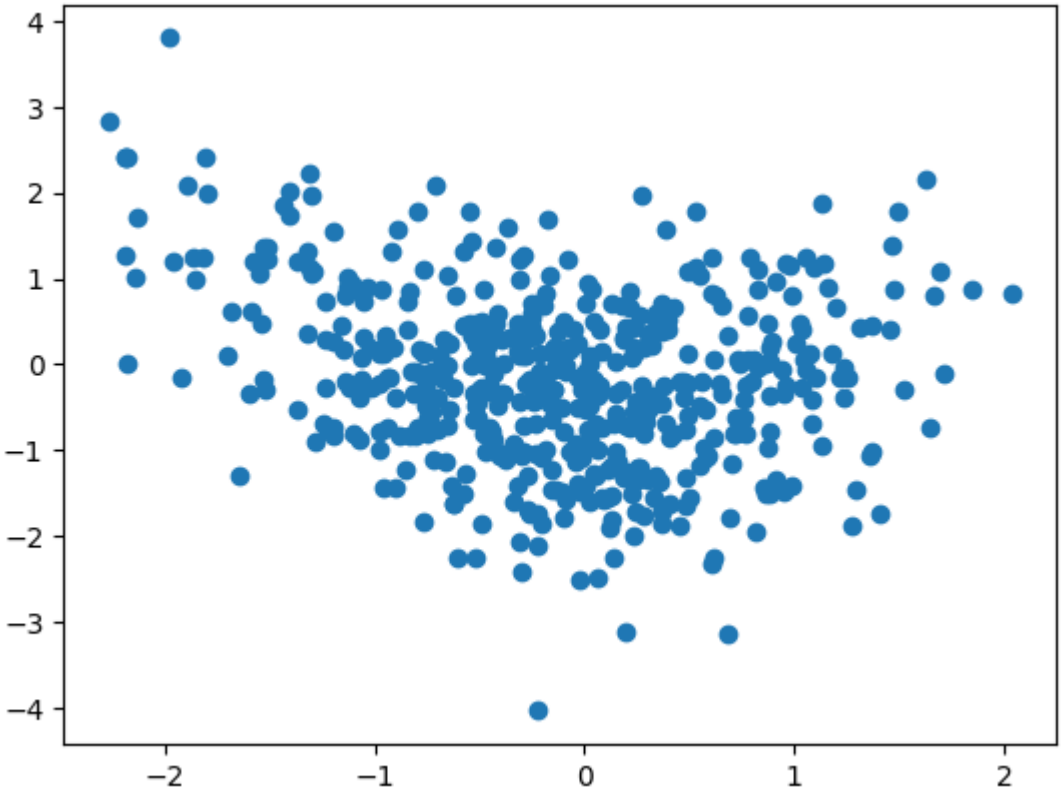
```
        #MAXIMISATION
        mu = np.zeros((p,d))
        sigma = np.zeros((p,d,d))
        alpha = np.zeros(p)
        sum_tau = np.sum(tau, axis=1)

        # On calcule les nouveaux paramètres en utilisant les poids w_i, pour ça
        tau = tau * weights[None, :]

        #On calcule les nouvelles valeurs de alpha_j
        alpha = sum_tau / np.sum(tau)
        alpha = alpha/np.sum(alpha) # On normalise alpha pour que la somme des a

        sum_tau = np.sum(tau, axis=1)
        #On calcule les nouvelles valeurs de alpha_j
        for j in range(p):
            for i in range(n):
                mu[j] += tau[j,i]*X[i] #On calcule les nouvelles valeurs de mu_j
            mu[j] = mu[j]/sum_tau[j]
        for j in range(p):
            for i in range(n):
                sigma[j] += tau[j,i]* np.outer((X[i] - mu[j]),(X[i] - mu[j])) #C
            sigma[j] = sigma[j]/sum_tau[j]
    return alpha, mu, sigma


def population_monte_carlo(n_samples, d, b, sigma_square):
    """
    Algorithme de la question 5.

    """

    # On initialise X selon une loi normale N(0, sigma)
    X = np.random.normal(0, np.sqrt(sigma_square), (n_samples, d))

    # On définit un certain nombre d'itérations max (on considèrera qu'il y a co
    max_iterations = 100
    theta = EM(X, d)
    for _ in tqdm(range(max_iterations), desc="EM"):
        # On génère de nouvelles données X selon le mélange de gaussiennes défin
        X = mixture_model_sample(theta, n_samples)

        # On cherche les poids w_i de l'étape t+1 en utilisant les paramètres th
        densities_mixture = np.array([mixture_model_density(x, theta) for x in X
        densities_banana = np.array([banana_density(x, b, sigma_square) for x in
        weights = densities_banana / densities_mixture

        # On normalise les poids
        weights = weights / np.sum(weights)

        # On calcule les nouveaux paramètres theta t+1 en utilisant les poids w_
        theta = EM_weights(X, weights, d)

    return X
```

In [ ]:
```
samples = population_monte_carlo(500, d=5, b=0.4, sigma_square=1)
```

```
EM: 100%|██████████| 100/100 [29:20<00:00, 17.60s/it]
```

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

Out[ ]:
```
<matplotlib.collections.PathCollection at 0x7f80aee55c00>
```



On remarque qu'il n'y a pas assez de points pour l'affirmer mais on peut dire que les points générés grâce à l'algorithme Population Monte Carlo ont bien l'air de suivre la distribution en banane attendue. Du au long temps de calcul de cet algorithme, je ne tenterai pas d'afficher plus de points.