TP2_Colin_Coerchon

22 Novembre 2024

1 TP2 - Computational statistics

[38]:
```
import numpy as np
import pandas as pd
import random
import matplotlib.pyplot as plt
from matplotlib import colormaps
from scipy.stats import multivariate_normal
from scipy.spatial.distance import cdist
from sklearn.cluster import KMeans
from matplotlib.patches import Ellipse
from sklearn.preprocessing import StandardScaler

# J'ai un warning embêtant en utilisant la fonction K-Means...
import warnings
warnings.filterwarnings("ignore", message="KMeans is known to have a memory leak␣
 ↪on Windows with MKL")


def rand():
    return random.random()

# Fonction pour tracer une ellipse correspondant à une covariance
def plot_ellipse(ax, mean, cov, color):
    v, w = np.linalg.eigh(cov)
    angle = np.arctan2(w[0][1], w[0][0])
    angle = np.degrees(angle)
    v = 2. * np.sqrt(2.) * np.sqrt(v)
    ell = Ellipse(xy=mean, width=v[0], height=v[1], angle=angle, color=color,␣
 ↪alpha=0.5)
    ax.add_patch(ell)
```

1.1 Exercice 1

1.1.1 Question 1

Soit un ensemble $X = \{x_1, \ldots, x_n\}$ de n valeurs réelles distinctes, et $(p_i)_{i\in[\![1,n]\!]}$ une suite de réels vérifiant : - $\forall i \in [\![1, n]\!]$, $p_i > 0$, - $\sum_{i=1}^{n} p_i = 1$.

Nous voulons générer une variable aléatoire X ayant une distribution discrète sur X donnée par $(p_i)_{i\in[\![1,n]\!]}$ telle que :

$$\forall i \in [\![1, n]\!], \quad \mathbb{P}(X = x_i) = p_i.$$

Explication de la méthode :

Pour générer cette variable aléatoire discrète, nous utilisons **la méthode d'inversion**. Voici les étapes de cette méthode :

1. On calcule les probabilités cumulées $P_k = \sum_{i=1}^{k} p_i$ pour $k \in [\![1, n]\!]$. En ajoutant une valeur initiale $P_0 = 0$, nous obtenons la suite $(P_k)_{k\in[\![0,n]\!]}$ avec $P_n = 1$.
2. On génère une variable aléatoire $U \sim \mathcal{U}([0, 1])$.
3. On trouve l'indice i tel que $P_{i-1} \leq U < P_i$. En assignant la valeur x_i lorsque U tombe dans cet intervalle, nous respectons les probabilités définies p_i.

1.1.2 Question 2

On peut alors écrire la fonction correspondante en python :

La fonction `tirage_va_Discrete` prend en entrée :

- `valeurs` : un tableau de valeurs possibles pour la variable aléatoire,
- `proba` : un tableau de probabilités associées à chaque valeur de `valeurs`,
- `N` : le nombre de tirages indépendants souhaités (par défaut 1).

Elle retourne :

- une valeur aléatoire si `N = 1`, ou un tableau de `N` valeurs aléatoires suivant la distribution définie par `proba`.

[12]:
```python
def tirage_va_Discrete(valeurs, proba, N=1):
    # Calcul des probabilités cumulées
    cum_proba = np.concatenate(([0], np.cumsum(proba)))
    res = np.zeros(N)

    for k in range(N):
        U = rand()  # Tirage d'une variable uniforme entre 0 et 1
        for i in range(len(valeurs)):
            if U >= cum_proba[i] and U < cum_proba[i + 1]:
                res[k] = valeurs[i]

    if N==1 : return res[0]
    else: return res
```

1.1.3 Question 3

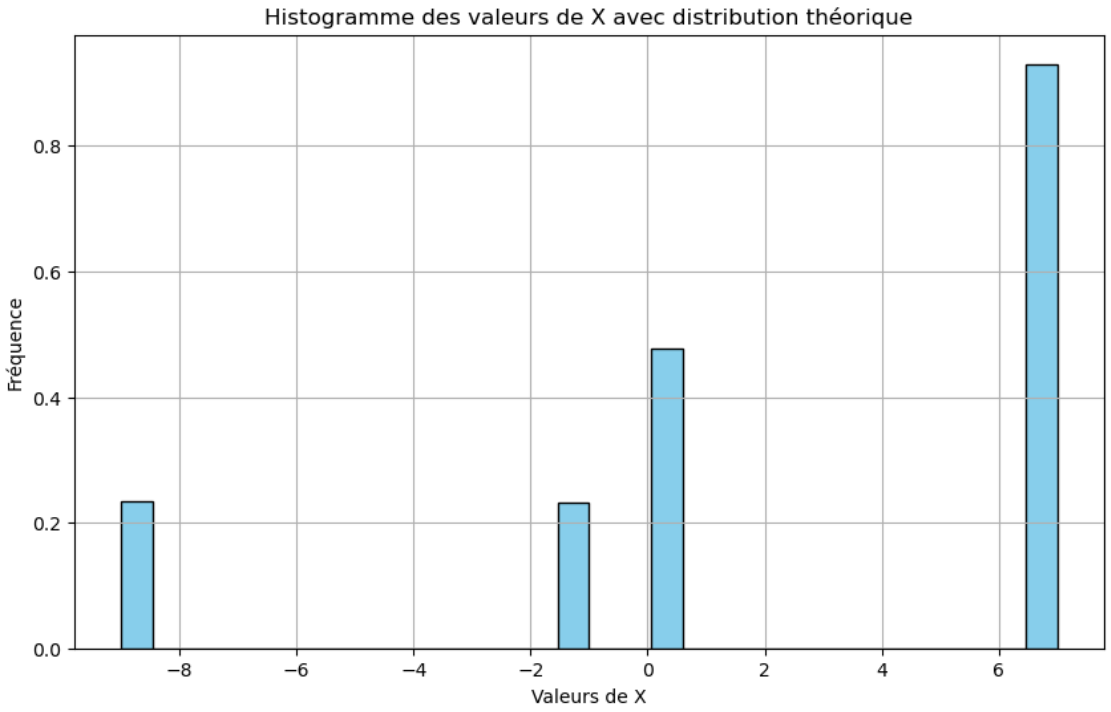
Cet exemple est tiré de mon cours de l'an dernier de *méthodes de simulation*. On considère une variable aléatoire X prenant ses valeurs dans $\{a_1, a_2, a_3, a_4\}$ avec

$$a_1 := 0.5, \quad a_2 := -9, \quad a_3 := -1.5, \quad a_4 := 7$$

et soit $(p_i)_{i=1,\dots,4}$ les poids associés définis par

$$\begin{cases} p_1 = \mathbb{P}(X = a_1) = 1/4 \\ p_2 = \mathbb{P}(X = a_2) = 1/8 \\ p_3 = \mathbb{P}(X = a_3) = 1/8 \\ p_4 = \mathbb{P}(X = a_4) = 1/2 \end{cases}$$

[13]:
```
valeurs = [0.5, -9, -1.5, 7]
probas = [1/4, 1/8, 1/8, 1/2]
N = 10000

# Génération des valeurs aléatoires
X = tirage_va_Discrete(valeurs, probas, N)

plt.figure(figsize=(10, 6))
plt.hist(X, bins=30, density=True, color='skyblue', edgecolor='black')


plt.title('Histogramme des valeurs de X avec distribution théorique')
plt.xlabel('Valeurs de X')
plt.ylabel('Fréquence')
plt.grid(True)
plt.show()
```



[14]:
```
empirical_probabilities = [(X == valeur).sum() / N for valeur in valeurs]

# Affichage des probabilités théoriques et empiriques
for i, valeur in enumerate(valeurs):
    print(f"Valeur: {valeur}, Probabilité théorique: {probas[i]:.3f},␣
 ↪Probabilité empirique: {empirical_probabilities[i]:.3f}")
```

```
Valeur: 0.5, Probabilité théorique: 0.250, Probabilité empirique: 0.255
Valeur: -9, Probabilité théorique: 0.125, Probabilité empirique: 0.124
Valeur: -1.5, Probabilité théorique: 0.125, Probabilité empirique: 0.124
Valeur: 7, Probabilité théorique: 0.500, Probabilité empirique: 0.496
```

1.2 Exercice 2

1.2.1 Question 1

Soit un modèle de mélange gaussien où nous avons un échantillon de taille n noté $(X_i)_{i\in[1,n]}$. Chaque observation X_i est associée à une variable cachée Z_i qui encode la classe de X_i. Nous considérons un mélange de m gaussiennes avec les paramètres :

- $(\alpha_1, \ldots, \alpha_m) \in \mathbb{R}_+^m$ tels que $\sum_{i=1}^m \alpha_i = 1$, où α_j représente la probabilité qu'une observation appartienne à la j-ème composante gaussienne.
- μ_j et Σ_j sont respectivement la moyenne et la matrice de covariance de la j-ème composante gaussienne.

Nous supposons également pour le modèle :

$$\forall i \in [\![1, n]\!],\ \forall j \in [\![1, m]\!], \quad \mathbb{P}_\theta(Z_i = j) = \alpha_j$$

et

$$\forall i \in [\![1, n]\!],\ \forall j \in [\![1, m]\!], \quad (X_i \mid \theta, \{Z_i = j\}) \sim \mathcal{N}(\mu_j, \Sigma_j).$$

Les paramètres du modèle, notés θ, sont donc : - les proportions $\alpha_1, \ldots, \alpha_m$, - les moyennes $\mu_1, \ldots, \mu_m$, - les matrices de covariance $\Sigma_1, \ldots, \Sigma_m$.

On a donc : $\boxed{\theta = \{\pi_k, \mu_k, \sigma_k \,|\, k \in [\![1, m]\!]\}}$.

La fonction de vraisemblance de θ étant donnée par les réalisations $(x_i)_{i \in [\![1,n]\!]}$ de l'échantillon i.i.d. $(X_i)_{i \in [\![1,n]\!]}$, on a :

$$\mathcal{L}(x_1, \ldots, x_n; \theta) = \prod_{i=1}^{n} f_\theta(x_i).$$

Pour calculer la fonction de vraisemblance, nous devons exprimer $f_\theta(x_i)$ en tenant compte du **modèle de mélange**. En effet, comme chaque X_i peut appartenir à l'une des m composantes gaussiennes, la densité $f_\theta(x_i)$ est une combinaison pondérée des densités de chaque composante, pondérées par les proportions α_j.

Ainsi, pour chaque $i \in [\![1, n]\!]$:

$$f_\theta(x_i) = \sum_{j=1}^{m} \alpha_j \, \mathcal{N}(x_i \mid \mu_j, \Sigma_j)$$

où $\mathcal{N}(x_i \mid \mu_j, \Sigma_j)$ désigne la densité de la loi normale multivariée de paramètres μ_j et Σ_j, évaluée en x_i.

En remplaçant cette expression dans la fonction de vraisemblance, on obtient :

$$\boxed{\mathcal{L}(x_1, \ldots, x_n; \theta) = \prod_{i=1}^{n} \left(\sum_{j=1}^{m} \alpha_j \, \mathcal{N}(x_i \mid \mu_j, \Sigma_j) \right)}$$

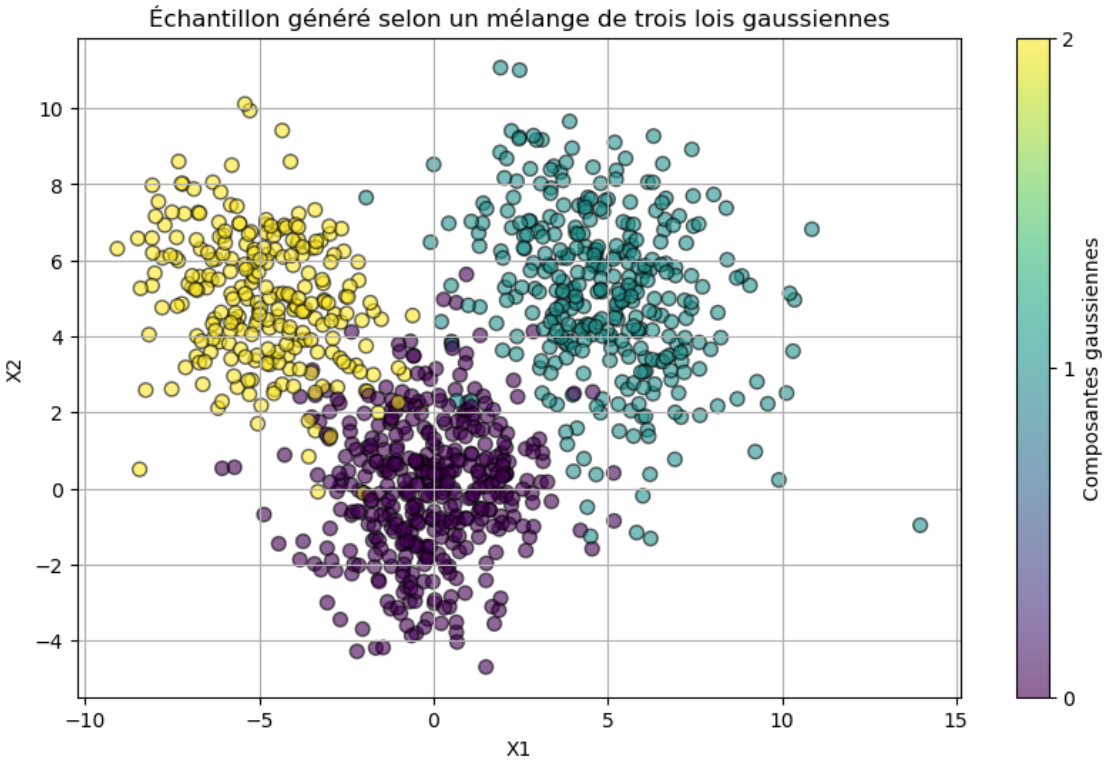
1.2.2 Question 2

J'ai choisi de prendre pour cette question un modèle de mélange de 3 gausiennes, de paramètres : - **1** : $\mu_1 = [0, 0]$, $\Sigma_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, proportion $\alpha_1 = 0.4$

- **2** : $\mu_2 = [5, 5]$, $\Sigma_2 = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$, proportion $\alpha_2 = 0.35$
- **3** : $\mu_3 = [-5, 5]$, $\Sigma_3 = \begin{bmatrix} 1 & -0.3 \\ -0.3 & 1 \end{bmatrix}$, proportion $\alpha_3 = 0.25$

[15]:
```
n_samples = 1000  # Taille de l'échantillon
```

```
# Paramètres des 3 gaussiennes, avec des covariances augmentées pour plus de␣
 ↪chevauchement
means = [np.array([0, 0]), np.array([5, 5]), np.array([-5, 5])]
covariances = [
    np.array([[3, 0], [0, 3]]),
    np.array([[5, -2], [-2, 5]]),
    np.array([[3, -1], [-1, 3]])
]
# covariances = [
#     np.array([[0.01, 0], [0, 0.01]]),
#     np.array([[0.01, 0], [0, 0.01]]),
#     np.array([[0.01, 0], [0, 0.01]])
# ]
proportions = [0.4, 0.35, 0.25]

# Choix de la composante (0, 1 ou 2) pour chaque échantillon. Utilisation de␣
 ↪l'exercice 1
components = tirage_va_Discrete([0, 1, 2], proportions, n_samples)

samples = np.zeros((n_samples, 2))
colors = np.zeros(n_samples, dtype=int)

for i in range(3):  # Pour chaque composante gaussienne
    n_i = np.sum(components == i)  # Nombre de points dans cette composante
    samples[components == i] = np.random.multivariate_normal(means[i],␣
 ↪covariances[i], n_i)
    colors[components == i] = i  # Assigner une couleur différente à chaque␣
 ↪composante

# Affichage des données avec des couleurs différentes pour chaque composante
plt.figure(figsize=(10, 6))
scatter = plt.scatter(samples[:, 0], samples[:, 1], c=colors, cmap='viridis',␣
 ↪alpha=0.6, edgecolor='k', s=50)
plt.colorbar(scatter, ticks=[0, 1, 2], label="Composantes gaussiennes")
plt.title("Échantillon généré selon un mélange de trois lois gaussiennes")
plt.xlabel("X1")
plt.ylabel("X2")
plt.grid(True)
plt.show()
```



Note : On aurait pu utiliser des méthodes vues dans le premier TP pour simuler des points i.i.d de loi normale (comme la méthode de Box-Muller par exemple). Mais on se contentera ici d'utiliser plus simplement `np.random.multivariate_normal` car cela n'est pas précisé dans la question.

1.2.3 Question 3

L'algorithme EM (Expectation-Maximization) permet d'estimer les paramètres θ d'un mélange de m gaussiennes à partir de données observées $(X_i)_{i\in[\![1,n]\!]}$.

Initialisation Avant de commencer les itérations, il est nécessaire d'initialiser les paramètres $\theta = \{\alpha_j, \mu_j, \Sigma_j \mid j \in [\![1, m]\!]\}$: - α_j représente la proportion de la j-ème composante gaussienne, - μ_j est la moyenne de la j-ème gaussienne, - Σ_j est la matrice de covariance de la j-ème gaussienne.

L'initialisation peut être aléatoire ou basée sur une certaine heuristique choisie.

Au début, j'avais choisi de me concentrer sur le premier choix simple : l'**aléatoire**. J'ai finalement changé d'avis pour une option plus robuste : - Proportion uniforme pour les α_j - Faire un **K-Means pour initialiser les moyennes** μ_j. - Les Σ_j sont initialisées comme des matrices unités.

Étape E (Expectation) L'étape E consiste à calculer l'espérance de la log-vraisemblance conditionnelle des variables latentes en fonction des paramètres estimés à l'itération précédente, notés $\theta^{(q)}$.

$$Q(\theta, \theta^{(q)}) = \mathbb{E}_{Z \mid X, \theta^{(q)}} \left[\log \mathbb{P}(X, Z; \theta)\right]$$

Dans le cadre du mélange de gaussiennes, il s'agit de déterminer la probabilité d'appartenance de chaque observation X_i à la j-ème composante gaussienne, que l'on note $\omega_{ij}^{(q)}$:

$$\omega_{ij}^{(q)} = \mathbb{P}\left(Z_i = j \mid X_i = x_i, \theta^{(q)}\right).$$

En utilisant la règle d'inversion de Bayes, on obtient :

$$\begin{aligned}
\omega_{ij}^{(q)} &= \mathbb{P}\left(Z_i = j \mid X_i = x_i, \theta^{(q)}\right) \\
&= \frac{\mathbb{P}\left(Z_i = j, X_i = x_i \mid \theta^{(q)}\right)}{\mathbb{P}\left(X_i = x_i \mid \theta^{(q)}\right)} \\
&= \frac{\mathbb{P}(Z_i = j \mid \theta^{(q)})\,\mathbb{P}(X_i = x_i \mid Z_i = j, \theta^{(q)})}{\sum_{\ell=1}^{m} \mathbb{P}(Z_i = \ell \mid \theta^{(q)})\,\mathbb{P}(X_i = x_i \mid Z_i = \ell, \theta^{(q)})} && \text{(formule des probabilités totales)} \\
&= \frac{\alpha_j^{(q)}\,\mathcal{N}(x_i \mid \mu_j^{(q)}, \Sigma_j^{(q)})}{\sum_{\ell=1}^{m} \alpha_\ell^{(q)}\,\mathcal{N}(x_i \mid \mu_\ell^{(q)}, \Sigma_\ell^{(q)})}.
\end{aligned}$$

Et donc :

$$\boxed{\omega_{ij}^{(q)} = \frac{\alpha_j^{(q)}\,\mathcal{N}(x_i \mid \mu_j^{(q)}, \Sigma_j^{(q)})}{\sum_{\ell=1}^{m} \alpha_\ell^{(q)}\,\mathcal{N}(x_i \mid \mu_\ell^{(q)}, \Sigma_\ell^{(q)})}}$$

Le calcul n'est pas encore terminé, on doit déterminer $Q(\theta, \theta^{(q)})$.

Calcul de $\mathbb{P}(X, Z; \theta)$:

$$\mathbb{P}(X, Z; \theta) = \prod_{i=1}^{n} \mathbb{P}(x_i, z_i; \theta) = \prod_{i=1}^{n} \prod_{j=1}^{m} \mathbb{P}(x_i, z_i; \theta)^{\mathbb{1}_{\{z_i = j\}}}$$

La notation sous forme de puissances à l'aide des variables latentes z_i permet de faciliter le calcul de la log-vraisemblance.

Prenons maintenant le logarithme de $\mathbb{P}(X, Z; \theta)$ pour obtenir la log-vraisemblance des données complètes :

$$\begin{aligned}
\log \mathbb{P}(X, Z; \theta) &= \sum_{i=1}^{n} \sum_{j=1}^{m} \mathbb{1}_{\{z_i = j\}} \log \mathbb{P}(x_i, z_i; \theta) \\
&= \sum_{i=1}^{n} \sum_{j=1}^{m} \mathbb{1}_{\{z_i = j\}} \log \left(\mathbb{P}(z_i = j; \theta)\,\mathbb{P}(x_i \mid z_i = j; \theta)\right) \\
&= \sum_{i=1}^{n} \sum_{j=1}^{m} \mathbb{1}_{\{z_i = j\}} \log \left(\alpha_j\,\mathcal{N}(x_i \mid \mu_j, \Sigma_j)\right) && \text{(par définition)}
\end{aligned}$$

Passage à l'espérance conditionnelle par rapport aux $Z\_\{ij\}$:

L'étape suivante consiste à prendre l'espérance conditionnelle de cette log-vraisemblance des données complètes par rapport à la distribution conditionnelle des Z sachant X et $\theta^{(q)}$:

$$\begin{aligned} Q(\theta, \theta^{(q)}) &= \mathbb{E}_{Z|X,\theta^{(q)}} \left[\log \mathbb{P}(X, Z; \theta)\right] \\ &= \mathbb{E}_{Z|X,\theta^{(q)}} \left[\sum_{i=1}^{n}\sum_{j=1}^{m} \mathbb{1}_{\{z_i=j\}} \log\left(\alpha_j \mathcal{N}(x_i \mid \mu_j, \Sigma_j)\right)\right] \\ &= \sum_{i=1}^{n}\sum_{j=1}^{m} \mathbb{E}_{Z|X,\theta^{(q)}} \left[\mathbb{1}_{\{z_i=j\}}\right] \log\left(\alpha_j \mathcal{N}(x_i \mid \mu_j, \Sigma_j)\right). \end{aligned}$$

Puisque $\mathbb{E}_{Z|X,\theta^{(q)}} \left[\mathbb{1}_{\{z_i=j\}}\right] = \mathbb{P}\left(Z_i = j \mid X_i = x_i, \theta^{(q)}\right) = \omega_{ij}^{(q)}$, on obtient finalement :

$$\boxed{Q(\theta, \theta^{(q)}) = \sum_{i=1}^{n}\sum_{j=1}^{m} \omega_{ij}^{(q)} \log\left(\alpha_j \mathcal{N}(x_i \mid \mu_j, \Sigma_j)\right)}$$

Cette expression de $Q(\theta, \theta^{(q)})$ sera utilisée dans l'étape M pour mettre à jour les paramètres θ en maximisant cette fonction.

Étape M (Maximization) L'étape M consiste à maximiser la fonction Q par rapport aux paramètres θ.

$$\theta^{(q+1)} = \arg\max_{\theta} Q(\theta, \theta^{(q)})$$

Pour cela, on utilise les valeurs de $\omega_{ij}^{(q)}$ calculées dans l'étape E pour obtenir les nouvelles estimations des paramètres $\theta^{(q+1)}$:

$$Q(\theta, \theta^{(q)}) = \sum_{i=1}^{n}\sum_{j=1}^{m} \omega_{ij}^{(q)} \log\left(\alpha_j \mathcal{N}(x_i \mid \mu_j, \Sigma_j)\right).$$

En dérivant cette expression par rapport aux paramètres, on obtient les mises à jour suivantes :

1. **Proportions** :

$$\boxed{\alpha_j^{(q+1)} = \frac{1}{n}\sum_{i=1}^{n} \omega_{ij}^{(q)}}$$

2. **Moyennes** :

$$\boxed{\mu_j^{(q+1)} = \frac{\sum_{i=1}^{n} \omega_{ij}^{(q)} x_i}{\sum_{i=1}^{n} \omega_{ij}^{(q)}}}$$

3. **Covariances** :

$$\Sigma_j^{(q+1)} = \frac{\sum_{i=1}^{n} \omega_{ij}^{(q)} (x_i - \mu_j^{(q+1)})(x_i - \mu_j^{(q+1)})^\top}{\sum_{i=1}^{n} \omega_{ij}^{(q)}}$$

Implémentation en Python Passons à l'implémentation en Python :

[16]:
```
def calculate_log_likelihood(X, pi, mu, sigma, m):
    n = X.shape[0]
    likelihood = np.zeros(n)

    for j in range(m):
        # Calcul de la densité de probabilité pour chaque cluster
        likelihood += pi[j] * multivariate_normal.pdf(X, mean=mu[j],␣
↪cov=sigma[j])

    log_likelihood = np.sum(np.log(likelihood))
    return log_likelihood


### Initialisation des paramètres pour un mélange de m gaussiennes ###
def initialisation_parametres(X, m):
    _, d = X.shape
    kmeans = KMeans(n_clusters=m, n_init=10, random_state=42)
    kmeans.fit(X)

    pi = np.ones(m) / m     # Proportions initiales égales
    mu = kmeans.cluster_centers_     # Initialisation des moyennes avec K-means
    sigma = [np.eye(d) for _ in range(m)]     # Initialisation des covariances␣
↪par des matrices identités
    return pi, mu, sigma

### Étape E : Calcul des probabilités d'appartenance (poids) ω_ij pour chaque␣
↪observation ###
def etape_E(X, pi, mu, sigma, m):
    n = X.shape[0]
    omega = np.zeros((n, m))

    for j in range(m):
        # Calcul de la densité pour la j-ième gaussienne
        rv = multivariate_normal(mean=mu[j], cov=sigma[j], allow_singular=True)
        omega[:, j] = pi[j] * rv.pdf(X)

    # Normalisation pour que la somme des poids pour chaque observation soit␣
↪égale à 1
    omega = omega / omega.sum(axis=1, keepdims=True)
    return omega
```

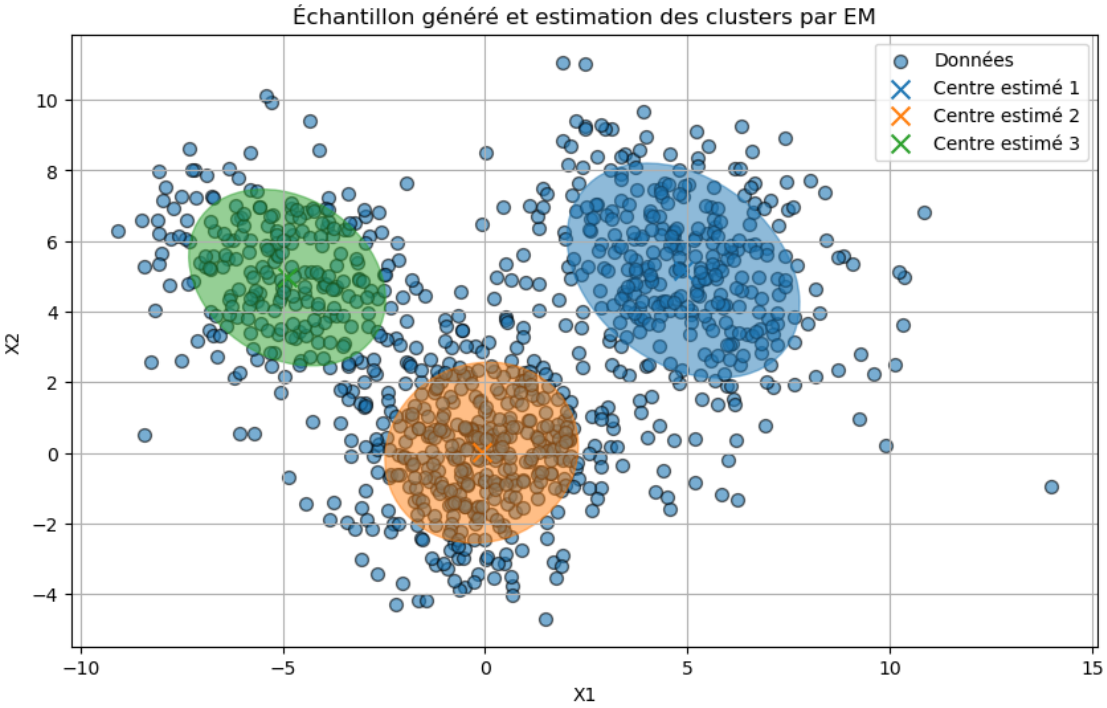
```python
### Étape M : Mise à jour des paramètres en maximisant Q ###
def etape_M(X, omega, m):
    n, d = X.shape
    pi = omega.mean(axis=0)  # Mise à jour des proportions

    mu = np.zeros((m, d))
    sigma = []
    reg = 1e-6

    for j in range(m):
        # Mise à jour des moyennes
        mu[j] = np.sum(omega[:, j].reshape(-1, 1) * X, axis=0) / np.sum(omega[:,␣
↪j])

        # Mise à jour des matrices de covariance
        diff = X - mu[j]
        sigma_j = (omega[:, j].reshape(-1, 1) * diff).T @ diff / np.sum(omega[:,␣
↪j])
        sigma_j += reg * np.eye(d)  # Ajout de régularisation (pour éviter les␣
↪bugs)
        sigma.append(sigma_j)

    return pi, mu, sigma


### Algorithme EM complet ###

def em_algorithme(X, m, max_iterations=1000, tolerance=1e-4, min_iterations=10):
    # Initialisation des paramètres
    pi, mu, sigma = initialisation_parametres(X, m)
    log_likelihoods = []

    for iteration in range(max_iterations):
        # Enregistrement des paramètres précédents pour vérifier la convergence
        pi_old, mu_old, sigma_old = pi.copy(), mu.copy(), sigma.copy()

        # Étape E
        omega = etape_E(X, pi, mu, sigma, m)

        # Calcul de la log-vraisemblance pour suivi de la convergence
        log_likelihood = calculate_log_likelihood(X, pi, mu, sigma, m)
        log_likelihoods.append(log_likelihood)

        # Étape M
        pi, mu, sigma = etape_M(X, omega, m)
```

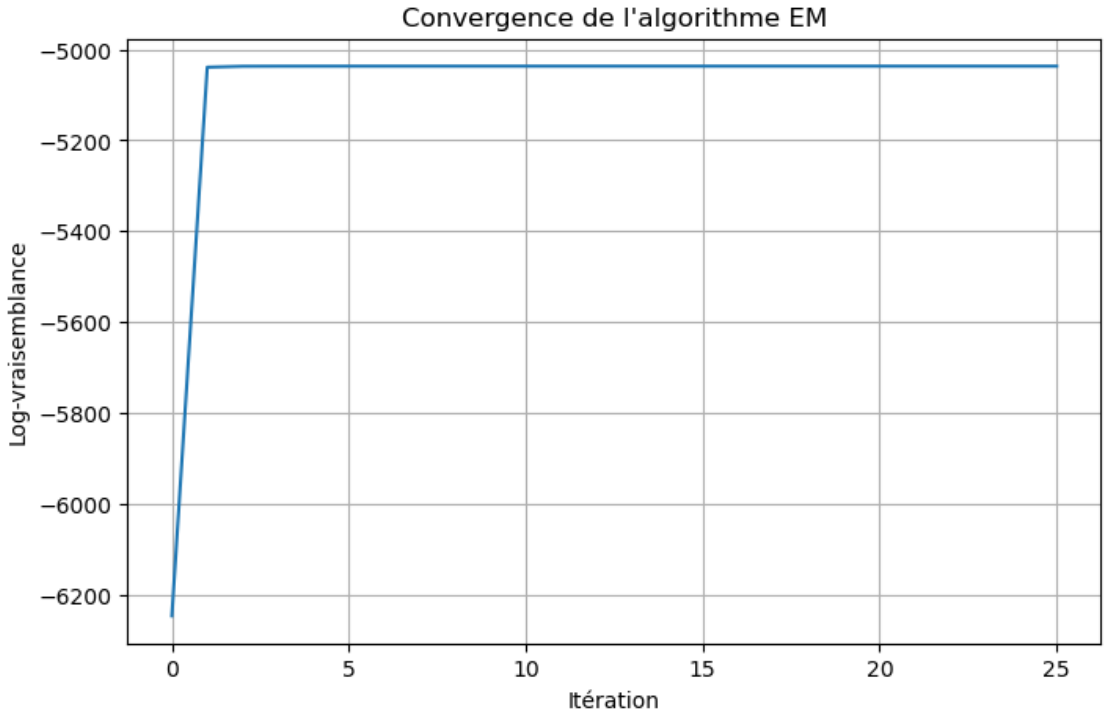
```
        # Vérification de la convergence (seulement après min_iterations)
        if iteration >= min_iterations and \
           np.allclose(mu, mu_old, atol=tolerance) and \
           np.allclose(sigma, sigma_old, atol=tolerance) and \
           np.allclose(pi, pi_old, atol=tolerance):
            print(f"Convergence atteinte après {iteration + 1} itérations.")
            break

    return pi, mu, sigma, log_likelihoods
```

Test sur un exemple

[17]:
```
cmap = colormaps["tab10"]

### Exécution de l'algorithme EM sur les données générées à la question 2 ###

m = 3
pi, mu, sigma, log_likelihoods = em_algorithme(samples, m)

### Affichage des données et des clusters estimés ###

plt.figure(figsize=(10, 6))
plt.scatter(samples[:, 0], samples[:, 1], alpha=0.6, edgecolor='k', s=50,␣
↪label="Données")

# Superposition des ellipses pour les gaussiennes estimées
ax = plt.gca()
for j in range(m):
    color = cmap(j % 10)
    plot_ellipse(ax, mu[j], sigma[j], color=color)
    plt.scatter(mu[j][0], mu[j][1], color=color, s=100, marker='x',␣
↪label=f'Centre estimé {j+1}')

plt.title("Échantillon généré et estimation des clusters par EM")
plt.xlabel("X1")
plt.ylabel("X2")
plt.legend()
plt.grid(True)
plt.show()

# Affichage de la convergence de la log-vraisemblance
plt.figure(figsize=(8, 5))
plt.plot(log_likelihoods)
plt.xlabel("Itération")
plt.ylabel("Log-vraisemblance")
plt.title("Convergence de l'algorithme EM")
plt.grid(True)
plt.show()
```

```
Convergence atteinte après 26 itérations.
```





L'algorithme EM converge bien, et fournit (quasiment) à chaque fois une excellente estimation de nos 3 gaussiennes, et cela s'opère **rarement en plus de 50 itérations** de l'algorithme.

Notre critère de converge se situe simplement sur la différences de valeurs de nos paramètres à estimer entre deux itérations. Nous considérons que l'algorithme a convergé si les paramètres estimés aux itérations q et $q+1$ vérifient les conditions suivantes, avec une tolérance définie ε (ici qui vaut 10^{-4}) :

$$\begin{cases} \|\mu^{(q+1)} - \mu^{(q)}\| < \varepsilon, \\ \|\Sigma^{(q+1)} - \Sigma^{(q)}\| < \varepsilon, \\ \|\pi^{(q+1)} - \pi^{(q)}\| < \varepsilon. \end{cases}$$

1.2.4 Question 4

[18]:
```
# Paramètres théoriques
theoretical_params = {
    "Proportions": proportions,
    "Moyennes": means,
    "Covariances": covariances
}

# Paramètres estimés par l'algorithme EM
estimated_params = {
    "Proportions": np.round(pi, 2),
    "Moyennes": [np.round(mean, 2) for mean in mu],
    "Covariances": [np.round(cov, 2) for cov in sigma]
}

# Calcul de la correspondance en utilisant la distance entre moyennes
mean_distances = cdist(theoretical_params["Moyennes"],␣
↪estimated_params["Moyennes"])
closest_indices = np.argmin(mean_distances, axis=1)

print("Comparaison des paramètres théoriques et estimés par EM avec␣
↪correspondance :\n")

print("Proportions (théoriques vs estimées) :")
for j, k in enumerate(closest_indices):
    print(f"  Composante théorique {j+1}: théorique =␣
↪{theoretical_params['Proportions'][j]:.2f}, estimée =␣
↪{estimated_params['Proportions'][k]:.2f}")

print("\nMoyennes (théoriques vs estimées) :")
for j, k in enumerate(closest_indices):
    print(f"  Composante théorique {j+1}: théorique =␣
↪{theoretical_params['Moyennes'][j]}, estimée =␣
↪{estimated_params['Moyennes'][k]}")
```

```
print("\nCovariances (théoriques vs estimées) :")
for j, k in enumerate(closest_indices):
    print(f"  Composante théorique {j+1}: théorique␣
 ↪=\n{theoretical_params['Covariances'][j]}")
    print(f"               estimée =\n{estimated_params['Covariances'][k]}\n")
```

```
Comparaison des paramètres théoriques et estimés par EM avec correspondance :

Proportions (théoriques vs estimées) :
  Composante théorique 1: théorique = 0.40, estimée = 0.41
  Composante théorique 2: théorique = 0.35, estimée = 0.34
  Composante théorique 3: théorique = 0.25, estimée = 0.25

Moyennes (théoriques vs estimées) :
  Composante théorique 1: théorique = [0 0], estimée = [-0.09  0.01]
  Composante théorique 2: théorique = [5 5], estimée = [4.9  5.16]
  Composante théorique 3: théorique = [-5  5], estimée = [-4.89  4.97]

Covariances (théoriques vs estimées) :
  Composante théorique 1: théorique =
[[3 0]
 [0 3]]
               estimée =
[[2.85 0.23]
 [0.23 3.28]]

  Composante théorique 2: théorique =
[[ 5 -2]
 [-2  5]]
               estimée =
[[ 4.13 -1.35]
 [-1.35  4.63]]

  Composante théorique 3: théorique =
[[ 3 -1]
 [-1  3]]
               estimée =
[[ 2.95 -0.75]
 [-0.75  3.12]]
```

Les paramètres estimés par l'algorithme EM sont remarquablement proches des valeurs initiales utilisées pour simuler les données. La correspondance est très frappante pour les **proportions** et les **moyennes**, où les écarts sont vraiment faibles. Pour les covariances, bien qu'il y ait de légères différences, les résultats restent très satisfaisants.

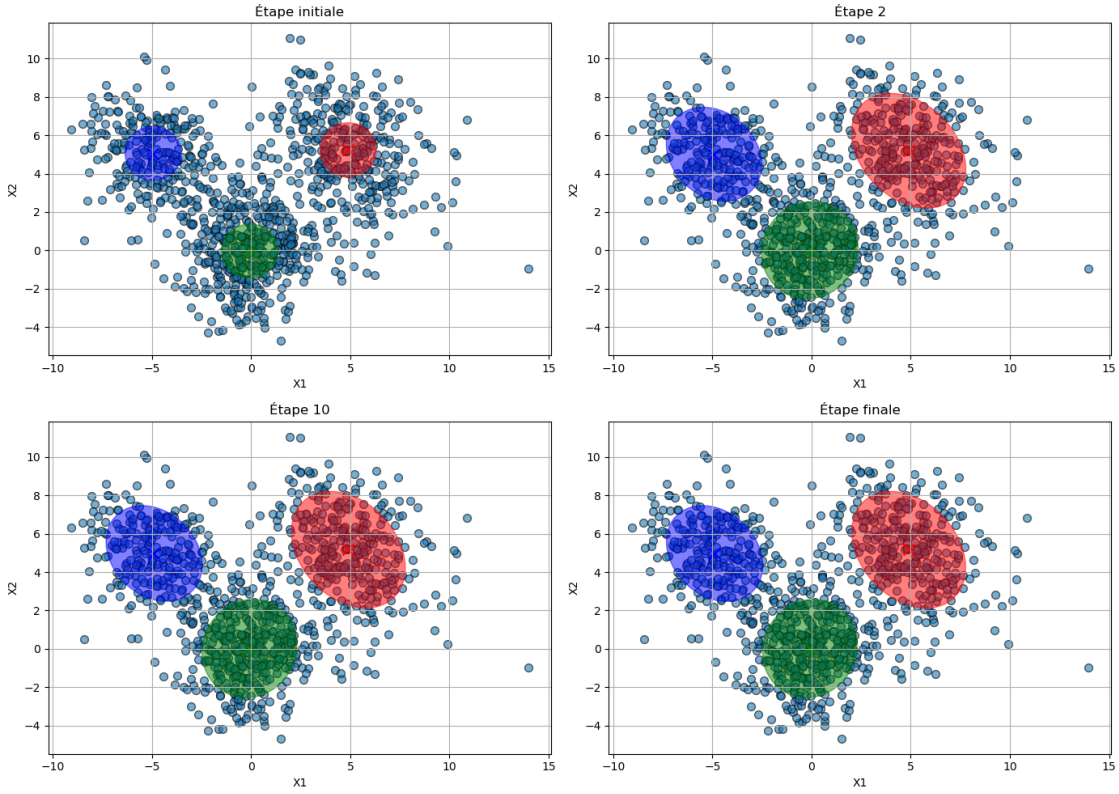
L'algorithme donne donc d'excellents résultats sur ce petit exemple.

J'ai également réalisé un petit affichage visuel des étapes 0, 2, 10 et de l'étape finale de notre algorithme sur notre exemple :

[19]:
```
def em_algorithme_etapes(X, m, max_iterations=1000, tolerance=1e-4,␣
↪min_iterations=10, save_steps=[0, 2, 10, -1]):
    pi, mu, sigma = initialisation_parametres(X, m)
    steps = []  # Pour enregistrer les paramètres à certaines étapes
    log_likelihoods = []

    for iteration in range(max_iterations):
        pi_old, mu_old, sigma_old = pi.copy(), mu.copy(), [s.copy() for s in␣
↪sigma]

        if iteration in save_steps or (save_steps[-1] == -1 and iteration ==␣
↪max_iterations - 1):
            steps.append((pi.copy(), mu.copy(), [s.copy() for s in sigma]))

        # Étape E
        omega = etape_E(X, pi, mu, sigma, m)

        # Étape M
        pi, mu, sigma = etape_M(X, omega, m)

        if iteration >= min_iterations and \
           np.allclose(mu, mu_old, atol=tolerance) and \
           np.allclose(sigma, sigma_old, atol=tolerance) and \
           np.allclose(pi, pi_old, atol=tolerance):
            print(f"Convergence atteinte après {iteration + 1} itérations.")
            break

    if not (np.allclose(steps[-1][0], pi) and np.allclose(steps[-1][1], mu) and␣
↪all(np.allclose(steps[-1][2][k], sigma[k]) for k in range(m))):
        steps.append((pi, mu, sigma))

    return pi, mu, sigma, steps


m = 3
save_steps = [0, 2, 10, -1]  # Étapes à afficher
pi, mu, sigma, steps = em_algorithme_etapes(samples, m, save_steps=save_steps)

fig, axs = plt.subplots(2, 2, figsize=(14, 10))
step_titles = ["Étape initiale", "Étape 2", "Étape 10", "Étape finale"]
colors = ['red', 'green', 'blue']

for i, (pi_step, mu_step, sigma_step) in enumerate(steps):
    ax = axs[i // 2, i % 2]
```

```
    ax.scatter(samples[:, 0], samples[:, 1], alpha=0.6, edgecolor='k', s=50,␣
↪label="Données")

    for j in range(m):
        plot_ellipse(ax, mu_step[j], sigma_step[j], color=colors[j])
        ax.scatter(mu_step[j][0], mu_step[j][1], c=colors[j], s=100, marker='x',␣
↪label=f'Centre estimé {j+1}')

    ax.set_title(step_titles[i])
    ax.set_xlabel("X1")
    ax.set_ylabel("X2")
    ax.grid(True)

plt.tight_layout()
plt.show()
```

```
Convergence atteinte après 26 itérations.
```



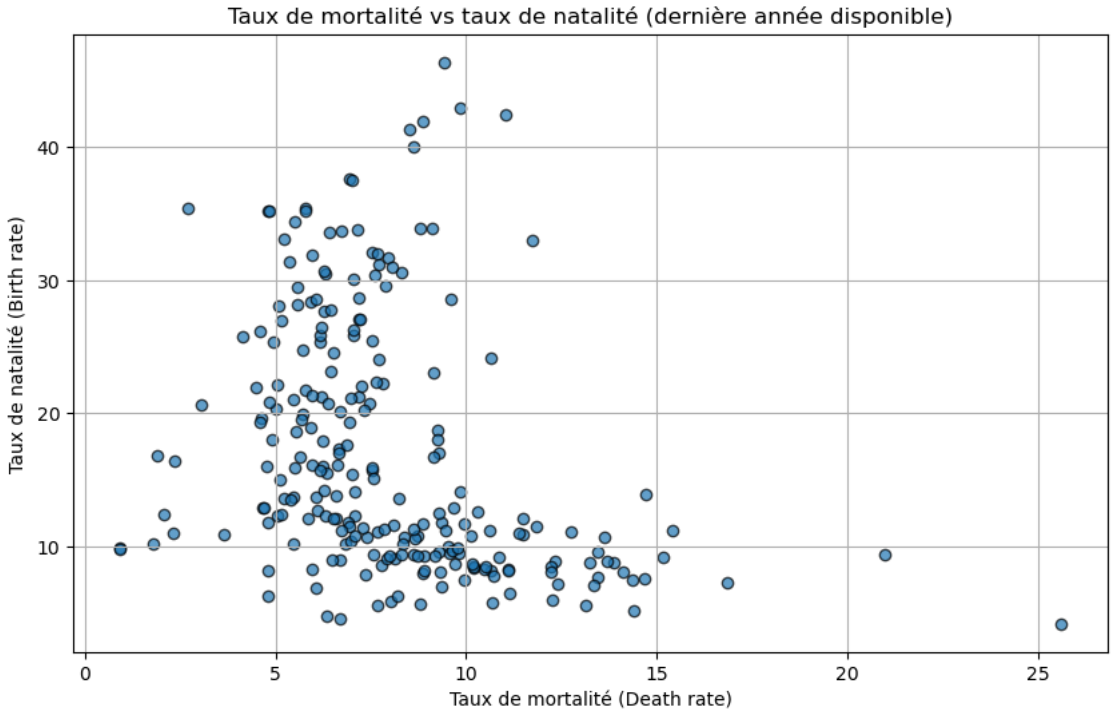
1.2.5 Question 5

Les données de cette question ont été téléchargées dans le fichier `birth-rate-vs-death-rate.csv`.

[20]:
```
file_path = "birth-rate-vs-death-rate.csv"
data = pd.read_csv(file_path)

# Noms exacts des colonnes pour les taux de natalité et mortalité
birth_rate_column = "Birth rate - Sex: all - Age: all - Variant: estimates"
death_rate_column = "Death rate - Sex: all - Age: all - Variant: estimates"
population_column = "Population - Sex: all - Age: all - Variant: estimates"

# On filtre les données pour obtenir la dernière année disponible pour chaque␣
↪pays (2023)
latest_data = data.sort_values(by="Year").groupby("Entity").tail(1)

# On supprime les lignes avec des valeurs manquantes dans les colonnes␣
↪pertinentes
required_columns = [birth_rate_column, death_rate_column, population_column,␣
↪"World regions according to OWID"]
latest_data_filtered = latest_data[required_columns].dropna()
```

Affichage classique

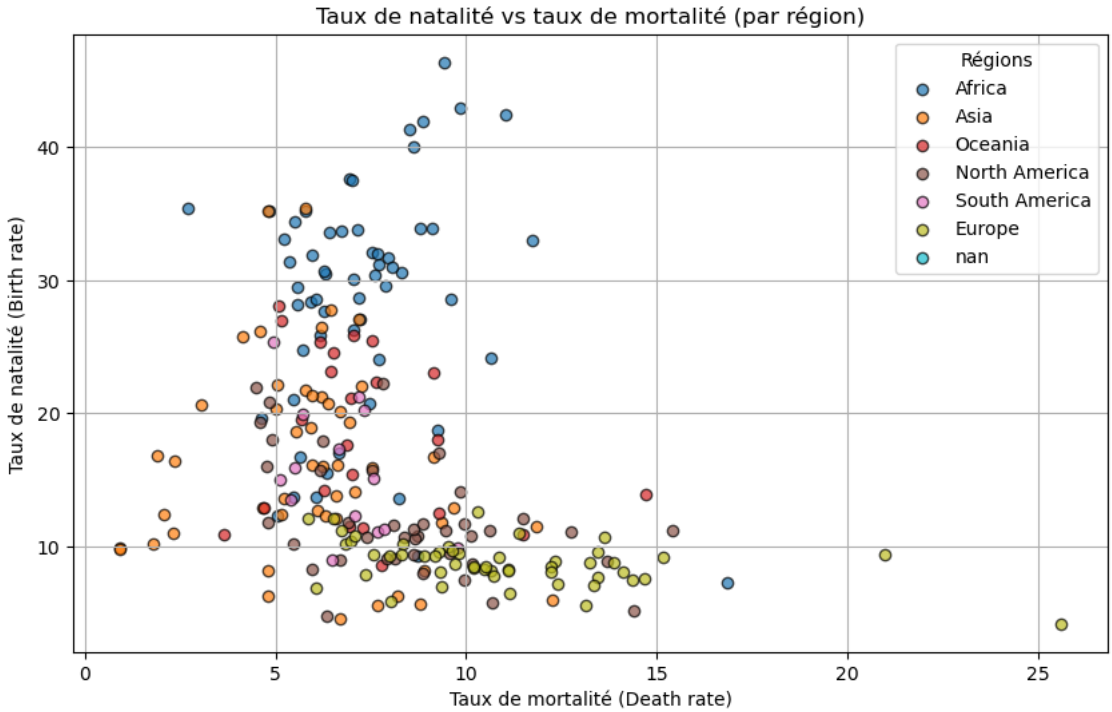
[21]:
```
plt.figure(figsize=(10, 6))
plt.scatter(
    latest_data_filtered[death_rate_column],
    latest_data_filtered[birth_rate_column],
    alpha=0.7,
    edgecolor='k'
)
plt.title("Taux de mortalité vs taux de natalité (dernière année disponible)")
plt.xlabel("Taux de mortalité (Death rate)")
plt.ylabel("Taux de natalité (Birth rate)")
plt.grid(True)
plt.show()
```



Affichage par continent

[22]:
```
### Couleur par continent ###

plt.figure(figsize=(10, 6))
unique_regions = latest_data["World regions according to OWID"].unique()
colors = plt.cm.tab10(np.linspace(0, 1, len(unique_regions)))  # Générer des␣
↪couleurs uniques

# Associer une couleur à chaque région
region_colors = {region: color for region, color in zip(unique_regions, colors)}

for region in unique_regions:
    region_data = latest_data[latest_data["World regions according to OWID"] ==␣
↪region]
    plt.scatter(
        region_data[death_rate_column],
        region_data[birth_rate_column],
        color=region_colors[region],
        alpha=0.7,
        label=region,
        edgecolor='k'
    )
```

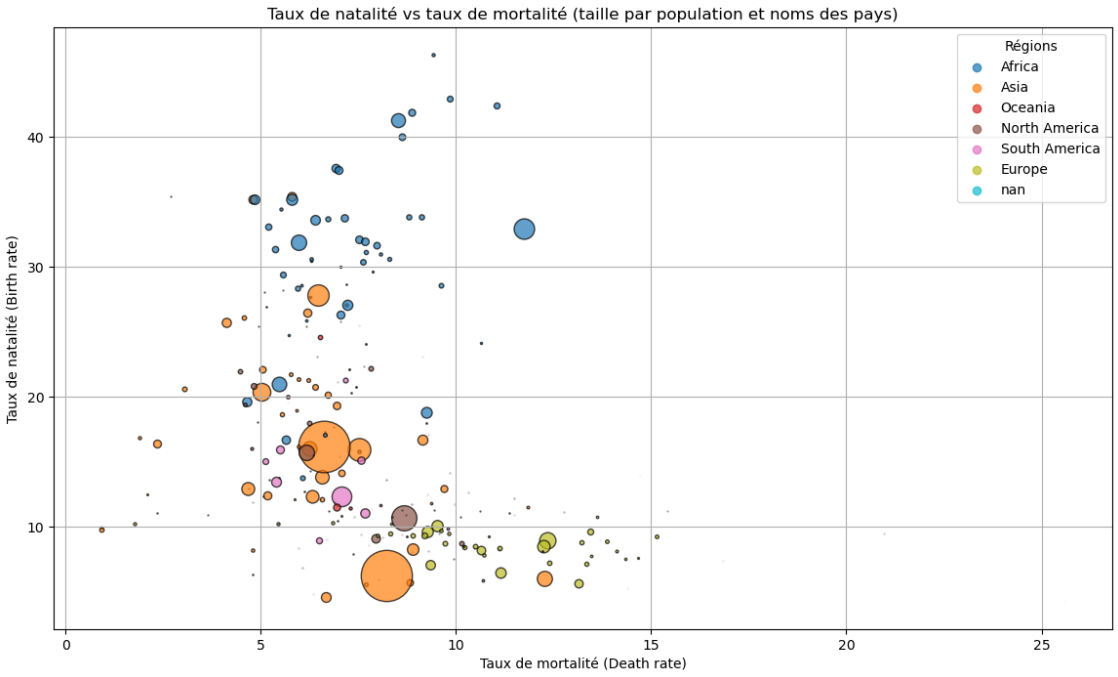
```
plt.title("Taux de natalité vs taux de mortalité (par région)")
plt.ylabel("Taux de natalité (Birth rate)")
plt.xlabel("Taux de mortalité (Death rate)")
plt.legend(title="Régions")
plt.grid(True)
plt.show()
```



Affichage par continent ET taille de la population

[23]:
```
### Taille des points et ajout des noms des pays ###
plt.figure(figsize=(14, 8))

for _, row in latest_data_filtered.iterrows():
    plt.scatter(
        row[death_rate_column],
        row[birth_rate_column],
        s=row[population_column] / 1e6,  # Taille proportionnelle à la␣
↪population (divisé par 1M pour lisibilité)
        color=region_colors[row["World regions according to OWID"]],
        alpha=0.7,
        edgecolor='k'
    )

for region, color in region_colors.items():
```

```
    plt.scatter([], [], color=color, alpha=0.7, label=region)

plt.title("Taux de natalité vs taux de mortalité (taille par population et noms␣
 ↪des pays)")
plt.ylabel("Taux de natalité (Birth rate)")
plt.xlabel("Taux de mortalité (Death rate)")
plt.legend(title="Régions", loc="upper right")  # Position de la légende
plt.grid(True)
plt.show()
```



Analyse de l'utilisation d'un modèle de mélange gaussien En observant le graphique, on peut distinguer **deux clusters potentiels** : un groupe majoritairement **africain** avec des taux de natalité élevés et un autre **européen** avec des taux plus faibles. Cela suggère qu'un modèle de mélange gaussien pourrait être une bonne approche pour capturer ces groupes principaux.

1.2.6 Question 6

Premièrement, voici la fonction pour calculer le critère BIC selon notre échantillon $(x_1, \ldots, x_n)$, et le nombre de paramètres contenus dans θ.

[24]:
```
# Calcul du critère BIC
def calculate_bic(log_likelihood, n_samples, n_clusters, d):
    # Nombre de paramètres : proportions + moyennes + covariances
    n_params = n_clusters - 1 + n_clusters * d + n_clusters * (d * (d + 1)) // 2
    bic = -2 * log_likelihood + n_params * np.log(n_samples)
```

```
    return bic
```

En effet, pour déterminer df(m), il suffit de remarquer que nos paramètres sont constitués de : - Proportions des clusters : $n\_\{\text{clusters}\} - 1$ (car la somme des proportions est égale à 1, imposant une contrainte). - Moyennes des clusters : $n\_\{\text{clusters}\} \times d$, où d est la dimension des données. - Covariances des clusters : Chaque matrice de covariance est **symétrique** $d \times d$, donc il y a $d \cdot (d+1)/2$ paramètres par cluster.

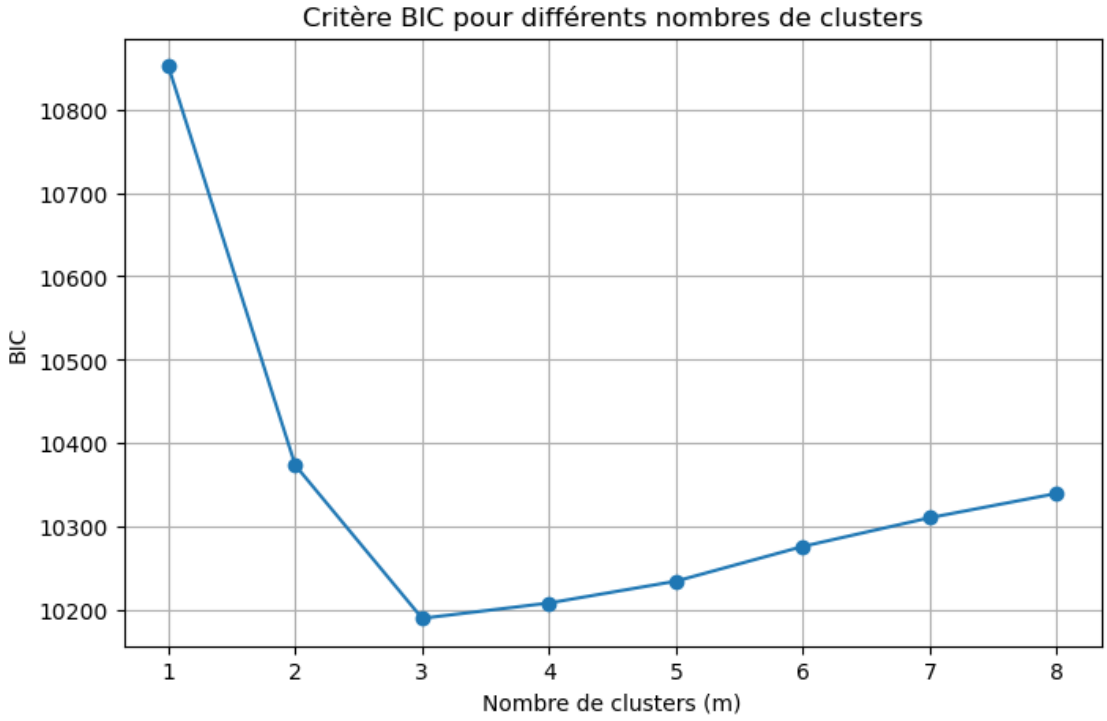
Ainsi, le total est :

$$\text{df}(m) = (n_{\text{clusters}} - 1) + n_{\text{clusters}} \cdot d + n_{\text{clusters}} \cdot \frac{d \cdot (d+1)}{2}$$

Nous pouvons déjà tester cette implémentation du critère BIC sur nos premières données :

[25]:
```
# Analyse BIC pour différents nombres de clusters
m_range = range(1, 9)  # Tester entre 1 et 8 clusters
bic_scores = []
log_likelihoods_all = []
models = []

for m in m_range:
    print(f"Évaluation pour m = {m}")

    # Exécuter l'algorithme EM
    pi, mu, sigma, log_likelihoods = em_algorithme(samples, m)
    final_log_likelihood = calculate_log_likelihood(samples, pi, mu, sigma, m)

    # Calcul du BIC
    n_samples, d = samples.shape
    bic = calculate_bic(final_log_likelihood, n_samples, m, d)
    bic_scores.append(bic)

    # Sauvegarder les modèles et log-likelihoods
    models.append((pi, mu, sigma))
    log_likelihoods_all.append(log_likelihoods)

### On trouve le meilleur nombre de clusters ! ###
best_m = m_range[np.argmin(bic_scores)]
print(f"\nMeilleur nombre de clusters selon le BIC : {best_m}")

# On trace le BIC pour chaque m
plt.figure(figsize=(8, 5))
plt.plot(m_range, bic_scores, marker='o')
plt.xlabel("Nombre de clusters (m)")
plt.ylabel("BIC")
plt.title("Critère BIC pour différents nombres de clusters")
plt.grid(True)
plt.show()
```

```
Évaluation pour m = 1
Convergence atteinte après 11 itérations.
Évaluation pour m = 2
Convergence atteinte après 23 itérations.
Évaluation pour m = 3
Convergence atteinte après 26 itérations.
Évaluation pour m = 4
Convergence atteinte après 476 itérations.
Évaluation pour m = 5
Évaluation pour m = 6
Convergence atteinte après 811 itérations.
Évaluation pour m = 7
Convergence atteinte après 856 itérations.
Évaluation pour m = 8
Convergence atteinte après 691 itérations.

Meilleur nombre de clusters selon le BIC : 3
```



On retrouve bien un nombre de clusters optimal égal à 3.

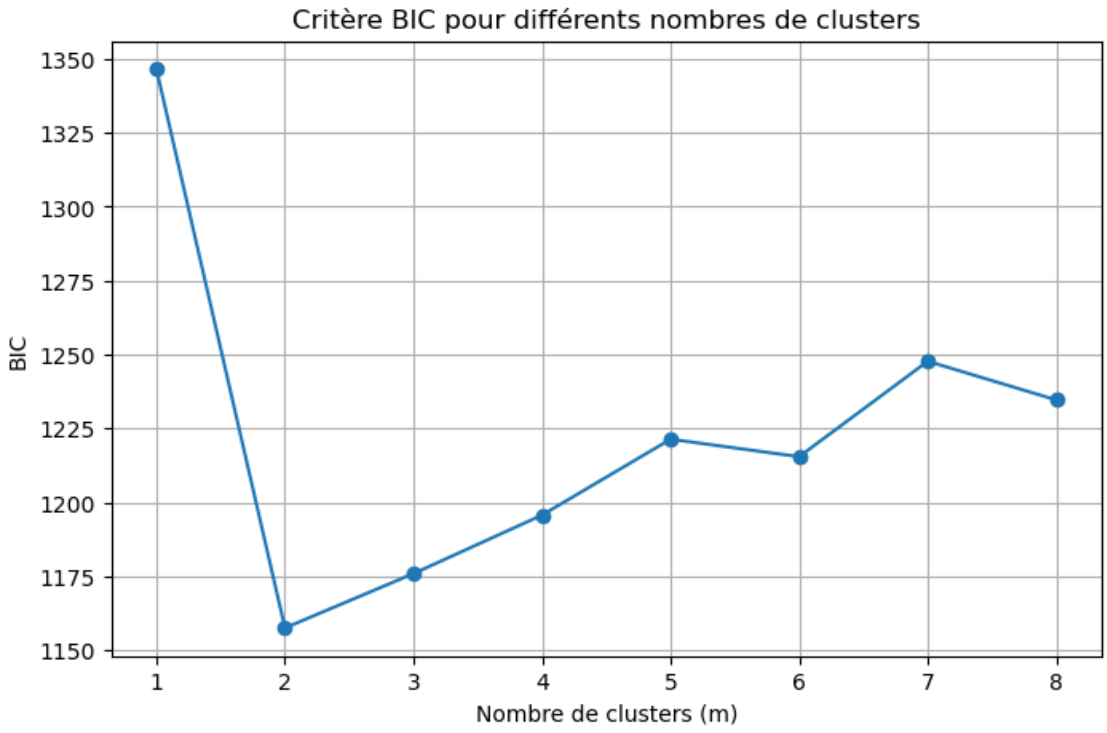
Pour ces nouvelles données, avant d'appliquer l'algorithme EM, il est souvent bien plus pertinent de normaliser les données.

[26]:
```
scaler = StandardScaler()
data_scaled = scaler.fit_transform(latest_data_filtered[[birth_rate_column,␣
↪death_rate_column]])
```

[27]:
```
bic_scores = []
log_likelihoods_all = []
models = []
m_range = range(1, 9)  # On teste entre 1 et 8 clusters

for m in m_range:
    print(f"Évaluation pour m = {m}")
    # Exécuter l'algorithme EM
    pi, mu, sigma, log_likelihoods = em_algorithme(data_scaled, m)
    final_log_likelihood = log_likelihoods[-1]

    log_likelihoods_all.append(log_likelihoods)

    # On calcule le BIC
    n_samples, d = data_scaled.shape
    bic = calculate_bic(final_log_likelihood, n_samples, m, d)
    bic_scores.append(bic)

    models.append((pi, mu, sigma))

best_m = m_range[np.argmin(bic_scores)]
print(f"\nMeilleur nombre de clusters selon le BIC : {best_m}")

plt.figure(figsize=(8, 5))
plt.plot(m_range, bic_scores, marker='o')
plt.xlabel("Nombre de clusters (m)")
plt.ylabel("BIC")
plt.title("Critère BIC pour différents nombres de clusters")
plt.grid(True)
plt.show()
```

```
Évaluation pour m = 1
Convergence atteinte après 11 itérations.
Évaluation pour m = 2
Convergence atteinte après 25 itérations.
Évaluation pour m = 3
Convergence atteinte après 142 itérations.
Évaluation pour m = 4
Convergence atteinte après 277 itérations.
Évaluation pour m = 5
Convergence atteinte après 215 itérations.
Évaluation pour m = 6
Convergence atteinte après 244 itérations.
Évaluation pour m = 7
```

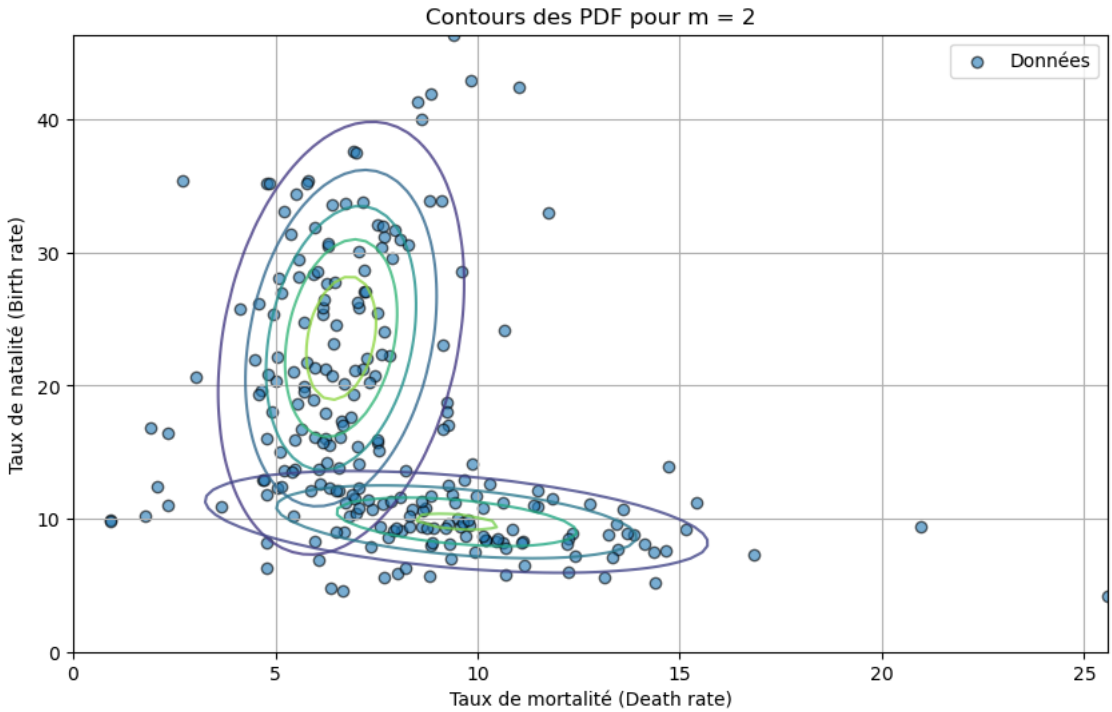
```
Convergence atteinte après 478 itérations.
Évaluation pour m = 8
Convergence atteinte après 466 itérations.

Meilleur nombre de clusters selon le BIC : 2
```



Le nombre de clusters qui minimise notre critère BIC est donc $m = 2$. On détermine donc nos paramètres contenus dans θ à l'aide de notre algorithme EM.

[28]:
```
# On récupère les paramètres du modèle optimal
pi_best, mu_best, sigma_best = models[best_m - 1]  # best_m - 1 car m_range␣
↪commence à 1

plt.figure(figsize=(10, 6))
plt.scatter(
    latest_data_filtered[death_rate_column],
    latest_data_filtered[birth_rate_column],
    alpha=0.6,
    edgecolor='k',
    label="Données"
)

x = np.linspace(0, latest_data_filtered[death_rate_column].max(), 100)  # x␣
↪commence à 0
```

```
y = np.linspace(0, latest_data_filtered[birth_rate_column].max(), 100)  # y␣
 ↪commence à 0
X, Y = np.meshgrid(x, y)
pos = np.dstack((Y, X))

pos_reshaped_df = pd.DataFrame(pos.reshape(-1, 2), columns=[birth_rate_column,␣
 ↪death_rate_column])

for j in range(best_m):
    rv = multivariate_normal(mean=mu_best[j], cov=sigma_best[j])
    Z = rv.pdf(scaler.transform(pos_reshaped_df)).reshape(X.shape)
    plt.contour(X, Y, Z, levels=5, cmap="viridis", alpha=0.8)

# Définir les limites des axes pour commencer à 0
plt.xlim(0, latest_data_filtered[death_rate_column].max())
plt.ylim(0, latest_data_filtered[birth_rate_column].max())

plt.title(f"Contours des PDF pour m = {best_m}")
plt.xlabel("Taux de mortalité (Death rate)")
plt.ylabel("Taux de natalité (Birth rate)")
plt.legend()
plt.grid(True)
plt.show()
```



Et voici les différents résultats estimés pour θ par notre algorithme EM :

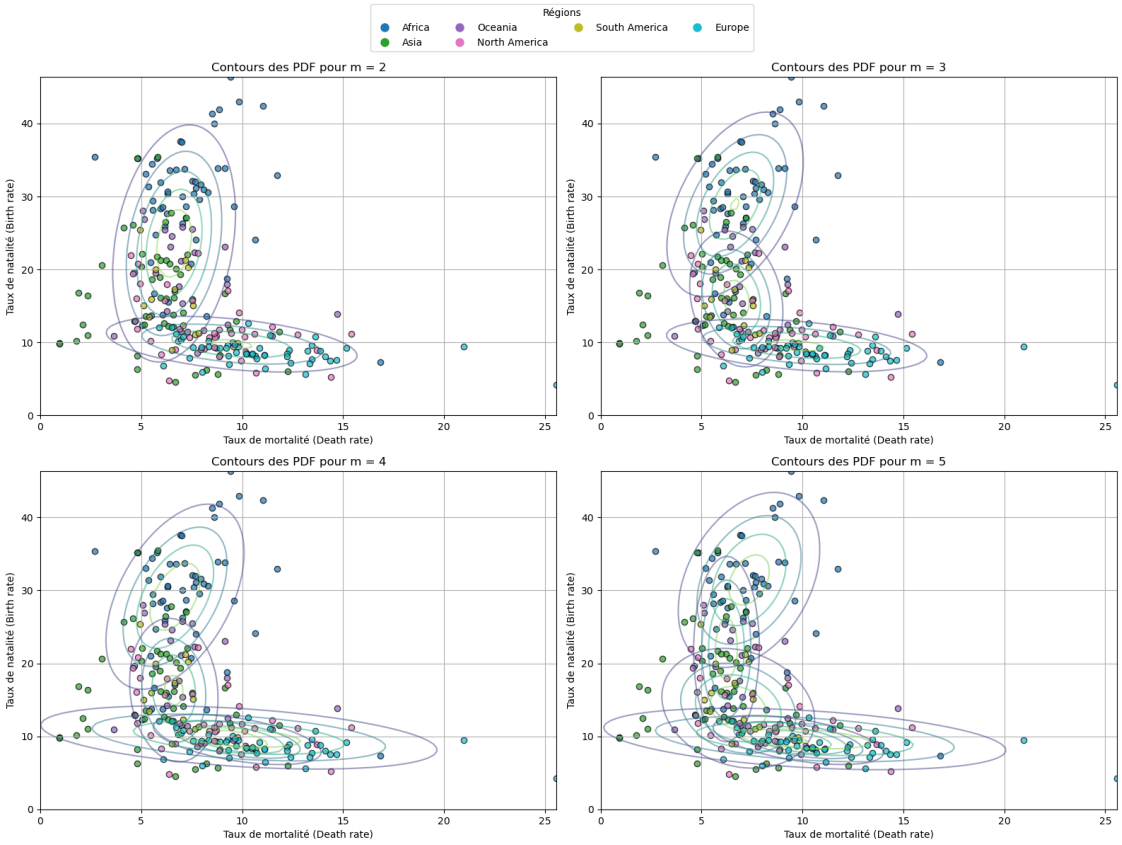
[29]:
```
m = best_m
pi, mu, sigma = models[m - 1]

print(f"Résultats pour m = {m} :\n")

print("Proportions :")
for j in range(m):
    print(f"  Cluster {j + 1}: {pi[j]:.4f}")

print("\nMoyennes :")
for j in range(m):
    print(f"  Cluster {j + 1}: {mu[j]}")

print("\nMatrices de covariance :")
for j in range(m):
    print(f"  Cluster {j + 1}:\n{sigma[j]}")
```

```
Résultats pour m = 2 :

Proportions :
  Cluster 1: 0.5315
  Cluster 2: 0.4685

Moyennes :
  Cluster 1: [ 0.68188807 -0.42776963]
  Cluster 2: [-0.77361813  0.48531475]

Matrices de covariance :
  Cluster 1:
[[0.83891152 0.11970049]
 [0.11970049 0.26936897]]
  Cluster 2:
[[ 0.05675483 -0.11689861]
 [-0.11689861  1.38578699]]
```

On peut également visualiser les résultats pour différentes valeurs de m. Ici, on teste pour $m = 2$, $m = 3$, $m = 4$ et $m = 5$.

[30]:
```
m_values = [2, 3, 4, 5]
fig, axes = plt.subplots(2, 2, figsize=(16, 12))
axes = axes.flatten()

x = np.linspace(0, latest_data_filtered[death_rate_column].max(), 100)
y = np.linspace(0, latest_data_filtered[birth_rate_column].max(), 100)
X, Y = np.meshgrid(x, y)
pos = np.dstack((Y, X))
```

```
pos_reshaped_df = pd.DataFrame(pos.reshape(-1, 2), columns=[birth_rate_column,␣
↪death_rate_column])

unique_regions = latest_data_filtered["World regions according to OWID"].unique()
colors = plt.cm.tab10(np.linspace(0, 1, len(unique_regions)))
region_colors = {region: color for region, color in zip(unique_regions, colors)}

# On parcourt les valeurs de m et on trace les graphiques correspondants
for idx, m in enumerate(m_values):
    ax = axes[idx]

    pi, mu, sigma = models[m - 1]

    for region in unique_regions:
        region_data = latest_data_filtered[latest_data_filtered["World regions␣
↪according to OWID"] == region]
        ax.scatter(
            region_data[death_rate_column],
            region_data[birth_rate_column],
            color=region_colors[region],
            alpha=0.7,
            label=region if idx == 0 else None,  # On affiche la légende␣
↪seulement dans le premier subplot
            edgecolor='k'
        )

    for j in range(m):
        rv = multivariate_normal(mean=mu[j], cov=sigma[j])
        Z = rv.pdf(scaler.transform(pos_reshaped_df)).reshape(X.shape)
        ax.contour(X, Y, Z, levels=5, cmap="viridis", alpha=0.5)

    ax.set_xlim(0, latest_data_filtered[death_rate_column].max())
    ax.set_ylim(0, latest_data_filtered[birth_rate_column].max())
    ax.set_title(f"Contours des PDF pour m = {m}")
    ax.set_xlabel("Taux de mortalité (Death rate)")
    ax.set_ylabel("Taux de natalité (Birth rate)")
    ax.grid(True)

# Ajouter la légende globale des régions
handles = [plt.Line2D([0], [0], marker='o', color=color, linestyle='None',␣
↪markersize=8, label=region)
           for region, color in region_colors.items()]
fig.legend(handles=handles, title="Régions", loc='upper center', ncol=4)

plt.tight_layout(rect=[0, 0, 1, 0.94])
plt.show()
```



Pour $m = 3$, on remarque bien que **le cluster du haut correspond globalement aux pays africains**, qui ont notamment un taux de natalité plus élevé. Tandis que **le cluster du bas correspond aux pays européens**, avec un taux de natalité très faible.

1.3 Exercice 3

1.3.1 Question 1

Voici la procédure pour réaliser notre échantillonnage préférentiel :

1. **Tirage des échantillons** :

   Générer n échantillons $X_1, \ldots, X_n$ à partir de $q(x)$, en rejetant les échantillons où $X_i < 0$, car $p(x) = 0$ pour ces valeurs.

2. **Calcul des poids** :

   Pour chaque échantillon X_i, calculer le poids :

$$\omega_i = \frac{p(X_i)}{q(X_i)}$$

3. **Estimation de l'espérance** :

Approximer l'espérance par :

$$\mathbb{E}_p[f(X)] \approx \frac{1}{n} \sum_{i=1}^{n} \omega_i f(X_i)$$

Cette procédure assure une approximation efficace de $\mathbb{E}_p[f(X)]$ à condition que $p(x)$ et $q(x)$ soient définis pour les mêmes valeurs supportées et que $q(x) > 0$ lorsque $p(x) > 0$.

[31]:
```
def f(x):
    return 2 * np.sin(np.pi / 1.5 * x) * (x >= 0)

def p(x):
    return x**(1.65 - 1) * np.exp(-x**2 / 2) * (x >= 0)

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

def sample_q(n_samples):
    samples = []
    while len(samples) < n_samples:
        new_samples = np.random.normal(loc=0.8, scale=np.sqrt(1.5),␣
↪size=n_samples - len(samples))
        # Ajout uniquement des échantillons valides (x >= 0)
        samples.extend(new_samples[new_samples >= 0])
    return np.array(samples)

def importance_sampling(n_samples, p, q, sample_q):
    samples = sample_q(n_samples)  # Échantillons tirés de q(x)
    weights = p(samples) / q(samples)  # Les poids omegas
    estimate = np.mean(weights * f(samples))
    return estimate
```

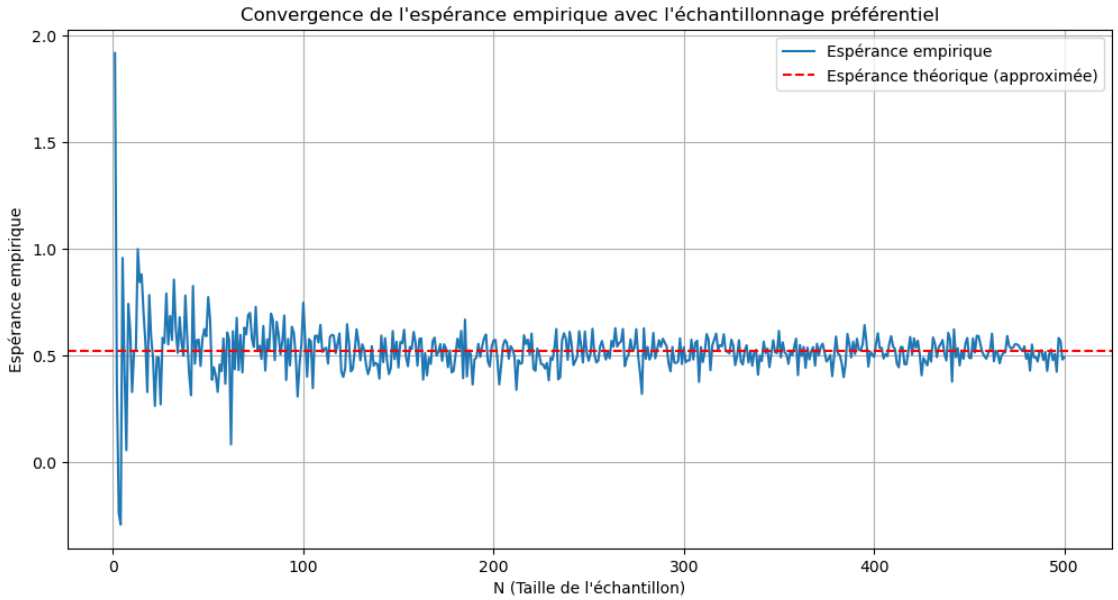
1.3.2 Question 2

[32]:
```
n_samples_list = [10, 100, 1000, 10000]

results = {n: importance_sampling(n, p, q, sample_q) for n in n_samples_list}

results
```

[32]:
```
{10: 0.4407758237212751,
 100: 0.3963029980174633,
 1000: 0.5380029622656904,
 10000: 0.5125848347020937}
```

On trouve une valeur approchée pour $\mathbb{E}_p[f(X)]$ proche de 0.521.

[33]:
```
N_values = np.arange(1, 501)  # Valeurs de 1 à 10000

empirical_means = [importance_sampling(N, p, q, sample_q) for N in N_values]

plt.figure(figsize=(12, 6))
plt.plot(N_values, empirical_means, label="Espérance empirique")
plt.axhline(y=0.521, color="r", linestyle="--", label="Espérance théorique␣
↪(approximée)")
plt.xlabel("N (Taille de l'échantillon)")
plt.ylabel("Espérance empirique")
plt.title("Convergence de l'espérance empirique avec l'échantillonnage␣
↪préférentiel")
plt.legend()
plt.grid(True)
plt.show()
```



On remarque que l'estimation met un peu de temps à converger. Il y a **une variance plutôt faible** dans l'échantillonnage, la fonction q ne semble pas optimale, mais elle fournit tout de même une bonne aproximation de notre espérance à partir de $N = 500$.

1.3.3 Question 3

[34]:
```
# Nouvelle fonction q(x) avec une moyenne modifiée
def q_shifted(x, mu=6):
    return (2 / np.sqrt(2 * np.pi * 1.5)) * np.exp(-(mu - x)**2 / (2 * 1.5))

def sample_q_shifted(n_samples, mu=6):
```

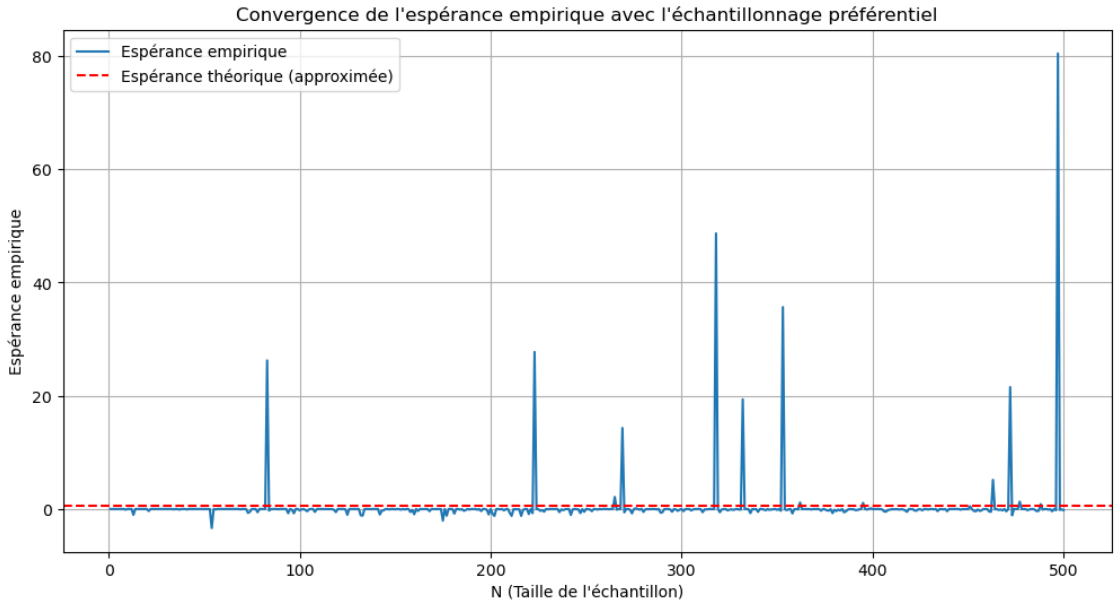
```
    samples = []
    while len(samples) < n_samples:
        new_samples = np.random.normal(loc=mu, scale=np.sqrt(1.5),␣
↪size=n_samples - len(samples))
        # Ajout uniquement des échantillons valides (x >= 0)
        samples.extend(new_samples[new_samples >= 0])
    return np.array(samples)
```

[35]:
```
n_samples_list = [10, 100, 1000, 10000]

results = {n: importance_sampling(n, p, q_shifted, sample_q_shifted) for n in␣
↪n_samples_list}

results
```

[35]:
```
{10: 0.002095173739290396,
 100: -0.05301229909479303,
 1000: -0.07003182931872054,
 10000: -0.02005773102610649}
```

Les résultats sont catastrophiques, notre estimation ne semble pas du tout converger malgré des tailles d'échantillon impoortants.

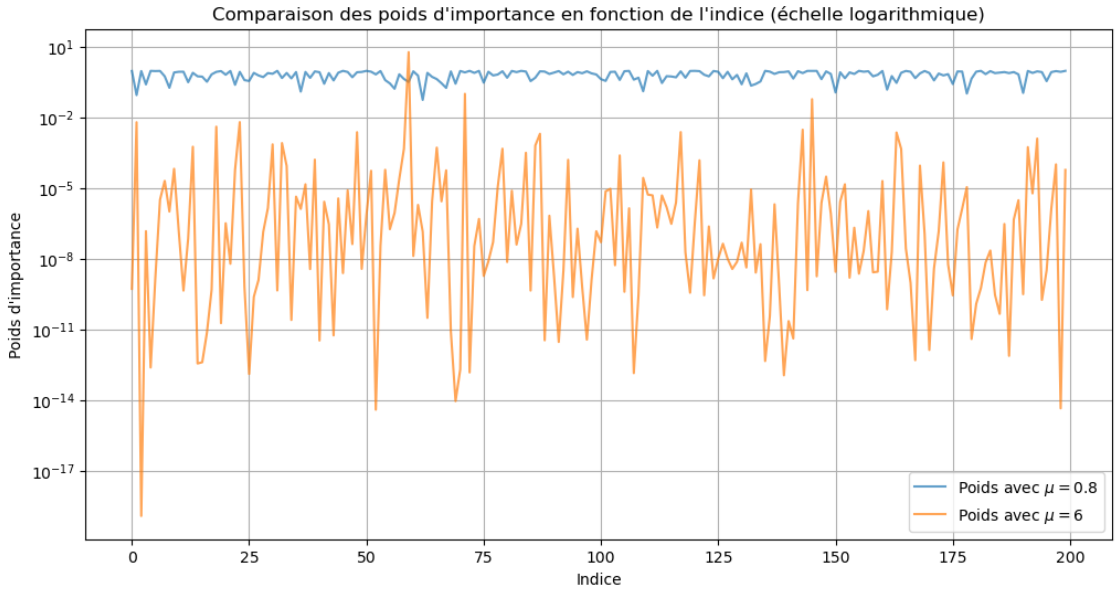
[36]:
```
N_values = np.arange(1, 501)  # Valeurs de 1 à 10000

empirical_means = [importance_sampling(N, p, q_shifted, sample_q_shifted) for N␣
↪in N_values]

plt.figure(figsize=(12, 6))
plt.plot(N_values, empirical_means, label="Espérance empirique")
plt.axhline(y=0.521, color="r", linestyle="--", label="Espérance théorique␣
↪(approximée)")
plt.xlabel("N (Taille de l'échantillon)")
plt.ylabel("Espérance empirique")
plt.title("Convergence de l'espérance empirique avec l'échantillonnage␣
↪préférentiel")
plt.legend()
plt.grid(True)
plt.show()
```



Ce **problème de variance** dans l'estimation se remarque particulièrement bien dans ce graphique.

[37]:
```
n_samples = 200

def compare_weights(n_samples, mu_shifted=6):
    samples_original = sample_q(n_samples)
    samples_shifted = sample_q_shifted(n_samples, mu=mu_shifted)

    weights_original = p(samples_original) / q(samples_original)
    weights_shifted = p(samples_shifted) / q_shifted(samples_shifted,␣
 ↪mu=mu_shifted)

    return weights_original, weights_shifted

weights_original, weights_shifted = compare_weights(n_samples)

def plot_weights_vs_index(weights_original, weights_shifted):
    plt.figure(figsize=(12, 6))
    plt.plot(weights_original, label='Poids avec $\\mu = 0.8$', linestyle='-',␣
 ↪alpha=0.7)
    plt.plot(weights_shifted, label='Poids avec $\\mu = 6$', linestyle='-',␣
 ↪alpha=0.7)
    plt.yscale("log")  # Échelle logarithmique pour mieux voir les différences
    plt.xlabel("Indice")
    plt.ylabel("Poids d'importance")
    plt.title("Comparaison des poids d'importance en fonction de l'indice␣
 ↪(échelle logarithmique)")
```

```
    plt.legend()
    plt.grid(True)
    plt.show()

plot_weights_vs_index(weights_original, weights_shifted)
```



- **Pour** $\mu = 0.8$: Les poids sont stables, ce qui garantit une estimation fiable et précise.
- **Pour** $\mu = 6$: La forte dispersion des poids rend l'estimation moins précise et plus sensible à des fluctuations aléatoires. De plus, presque tous les poids sont extrêmement faibles (proches de 0), ce qui est problématique. Lors du calcul de l'espérance avec la méthode de Monte Carlo :

$$\mathbb{E}_p[f(X)] \approx \frac{1}{n} \sum_{i=1}^{n} \omega_i f(X_i),$$

on se repose alors sur un **très petit sous-ensemble de poids significatifs**, augmentant fortement la variance.

Conclusion Un mauvais choix de $q(x)$ (ici avec $\mu = 6$) peut rendre l'estimation **impossible** en raison de l'inefficacité des poids. Cela souligne l'importance de sélectionner une fonction $q(x)$ bien alignée avec $p(x)$.

1.3.4 Question 4

L'objectif est de maximiser le critère empirique présenté à l'étape (iii) de l'algorithme de la page 6, donné par :

$$\sum_{i=1}^{n} \tilde{\omega}_i^{(0)} \log \left(\sum_{j=1}^{M} \alpha_j \varphi(X_i^{(0)}; \theta_j) \right)$$

où α_j, μ_j, et Σ_j doivent être mis à jour pour définir une nouvelle densité d'importance $q^{(1)}$. Ce problème peut être résolu efficacement en utilisant **l'algorithme EM**, qui est adapté ici pour prendre en compte les **poids d'importance** $\tilde{\omega}_i^{(0)}$. En effet, cette notion de poids d'importance est clairement présente dans l'algorithme EM. Et pour cela, on va grandement s'inspirer du travail réalisé dans **la question 3 de l'exercice 2**.

Voici les étapes phares de l'algorithme :

Initialisation L'algorithme commence par l'initialisation des paramètres $\{\alpha_j, \mu_j, \Sigma_j \mid j \in [\![1, M]\!]\}$: - α_j : proportion de la j-ème composante, - μ_j : moyenne de la j-ème composante, - Σ_j : matrice de covariance.

On peut initialiser ces paramètres de manière aléatoire ou avec une heuristique bien précise (ex: K-means pour les μ_j).

Étape E Comme dans la **question 3 de l'exercice 2**, l'étape E consiste à calculer les responsabilités pondérées γ_{ij} (la probabilité d'appartenance de chaque observation X_i à la j-ème composante gaussienne) :

$$\gamma_{ij} = \mathbb{P}(Z_i = j \mid X_i = x_i, \theta) \quad \underset{\text{cf. ex. 2 q. 3}}{\Longrightarrow} \quad \gamma_{ij} = \frac{\alpha_j \, \varphi(X_i^{(0)}; \mu_j, \Sigma_j)}{\sum_{k=1}^{M} \alpha_k \, \varphi(X_i^{(0)}; \mu_k, \Sigma_k)}$$

Étape M L'étape M met à jour les paramètres $\alpha_j, \mu_j, \Sigma_j$ en maximisant la fonction d'espérance pondérée (cf. question 3 exercice 2):

$$Q(\alpha, \mu, \Sigma) = \sum_{i=1}^{n} \tilde{\omega}_i^{(0)} \sum_{j=1}^{M} \gamma_{ij} \log \Big(\alpha_j \, \varphi(X_i^{(0)}; \mu_j, \Sigma_j) \Big)$$

En dérivant, puis en annulant, on trouve :

1. **Proportions :**

$$\boxed{\alpha_j^{(1)} = \frac{\sum_{i=1}^{n} \tilde{\omega}_i^{(0)} \gamma_{ij}}{\sum_{i=1}^{n} \tilde{\omega}_i^{(0)}}}$$

2. **Moyennes :**

$$\boxed{\mu_j^{(1)} = \frac{\sum_{i=1}^n \tilde{\omega}_i^{(0)} \gamma_{ij} X_i^{(0)}}{\sum_{i=1}^n \tilde{\omega}_i^{(0)} \gamma_{ij}}.}$$

3. **Covariances :**

$$\boxed{\Sigma_j^{(1)} = \frac{\sum_{i=1}^n \tilde{\omega}_i^{(0)} \gamma_{ij} \left(X_i^{(0)} - \mu_j^{(1)}\right)\left(X_i^{(0)} - \mu_j^{(1)}\right)^\top}{\sum_{i=1}^n \tilde{\omega}_i^{(0)} \gamma_{ij}}}$$

1.3.5 Question 5

Je n'ai malheureusement pas eu le temps de traiter cette question...

Cependant, l'entièreté du reste du sujet a été traitée.