TP3

20 Mars 2025

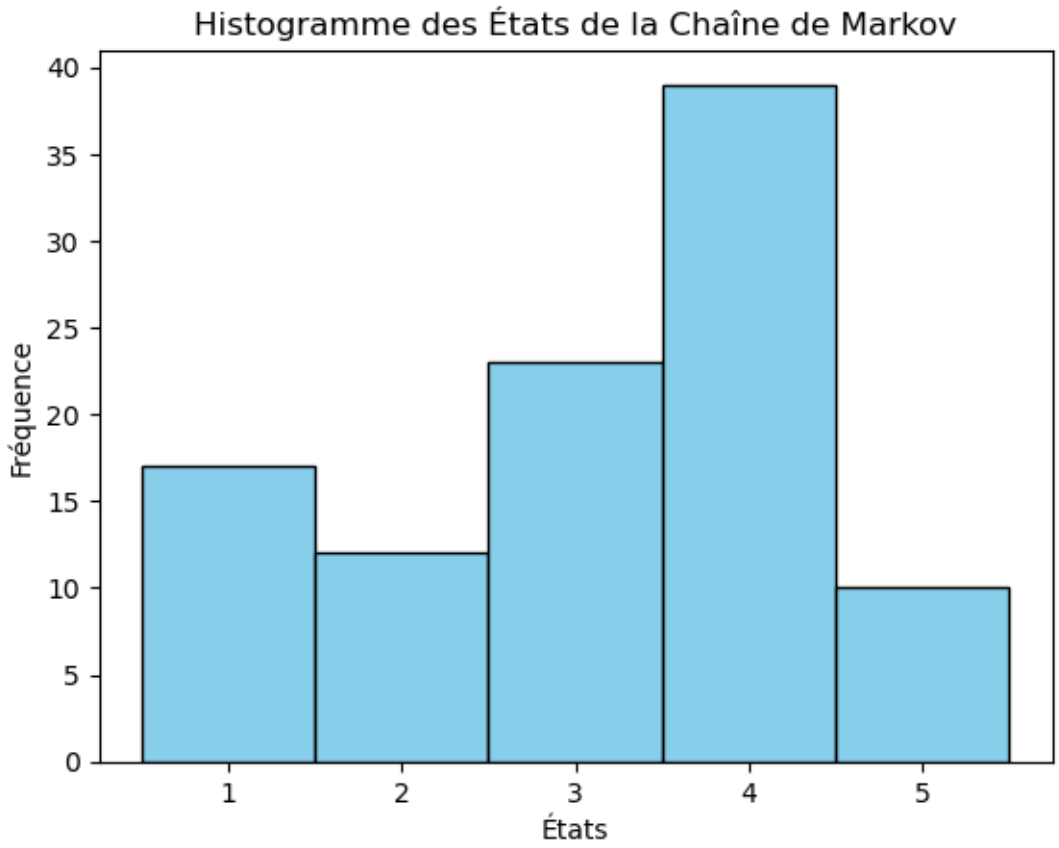
[2]:
```
import numpy as np
import random
import math
import matplotlib.pyplot as plt
from matplotlib import cm
from scipy.stats import norm,binom,gaussian_kde,t
from scipy.integrate import quad
from mpl_toolkits.mplot3d import Axes3D

def rand():
    return random.random()

def simuler_gaussienne_bidimensionnelle():
    u1 = rand()
    u2 = rand()
    R = -2*math.log(u1)
    Theta = u2*2*math.pi
    return (math.sqrt(R)*math.cos(Theta), math.sqrt(R)*math.sin(Theta))

def simuler_loi_normale(mu = 0, sigma = 1):
    x = simuler_gaussienne_bidimensionnelle()[0]
    return sigma*x + mu

def simuler_chaine_markov(P, etat_initial, N):
    n = P.shape[0]  # Nombre d'états
    etat_courant = etat_initial
    X = [etat_courant]
    for _ in range(N):
        u = rand()
        somme = 0
        for j in range(n):
            somme += P[etat_courant-1, j]  # -1 car les états se numérotent de 1␣
↪à n
            if u <= somme:
                etat_courant = j+1  # Obligé de faire +1 pour l'affichage des␣
↪états
                break
        X.append(etat_courant)
```

```
    return X

def tirage_va_Discrete_Uniforme(valeurs, N=1):
    n=len(valeurs)
    cum_proba = np.concatenate(([0], np.cumsum(np.ones(n)*1/n)))
    res = np.zeros(N, dtype = int)

    for k in range(N):
        U = rand()
        for i in range(n):
            if U >= cum_proba[i] and U < cum_proba[i + 1]:
                res[k] = valeurs[i]
    if N==1: return res[0]
    else: return res
```

0.1 Exercice 1

0.1.1 Question 1

[3]:
```
P = np.array([
    [0.1, 0.1, 0.5, 0.2, 0.1],
    [0.3, 0.1, 0.1, 0.4, 0.1],
    [0.2, 0.1, 0.1, 0.5, 0.1],
    [0.1, 0.1, 0.1, 0.6, 0.1],
    [0.1, 0.2, 0.1, 0.4, 0.2]
])

X0 = 1

X = simuler_chaine_markov(P, X0, 100)

plt.hist(X, bins=np.arange(1, P.shape[0]+2)-0.5, color='skyblue',␣
 ↪edgecolor='black')
plt.xlabel('États')
plt.ylabel('Fréquence')
plt.title('Histogramme des États de la Chaîne de Markov')
plt.xticks(np.arange(1, P.shape[0]+1))
plt.show()
```



La chaîne passe la plupart de son temps dans l'état 4.

0.1.2 Question 2

[4]:
```
def probabilite_invariante(P):
    P = np.asarray(P)

    A = np.vstack((P.T - np.eye(P.shape[0]), np.ones(P.shape[0])))
    b = np.append(np.zeros(P.shape[0]), 1)
    pi = np.linalg.solve(np.dot(A.T,A),  np.dot(A.T,b))

    return pi

probabilite_invariante(P)
```

[4]:
```
array([0.13773148, 0.11111111, 0.15509259, 0.4849537 , 0.11111111])
```

0.1.3 Question 3

[5]:
```
def estimer_probabilite_invariante(P, N):
    X = simuler_chaine_markov(P, 1, N)
    pi = [0] * len(P)
    for i in range(1, len(P) + 1):
        pi[i-1] = X.count(i) / N
    return pi

pi_approx = estimer_probabilite_invariante(P, 100000)
print(pi_approx)
```

```
[0.13798, 0.1112, 0.15534, 0.48358, 0.11191]
```

0.1.4 Question 4

[6]:
```
def estimer_matrice_transition(X):

    n = len(set(X))
    P_estimee = np.zeros((n, n))

    for i in range(len(X)-1):
        etat_actuel = X[i] - 1
        etat_suivant = X[i+1] - 1
        P_estimee[etat_actuel, etat_suivant] += 1

    for i in range(n):
        somme_transitions = np.sum(P_estimee[i, :])
        if somme_transitions > 0:
            P_estimee[i, :] /= somme_transitions

    P_estimee = np.round(P_estimee, 2)
    return P_estimee

N = 100000
X = simuler_chaine_markov(P, 1, N)

P_approx = estimer_matrice_transition(X)

print("Matrice P :\n", P)
print()
print("Matrice P approchée :\n", P_approx)
```

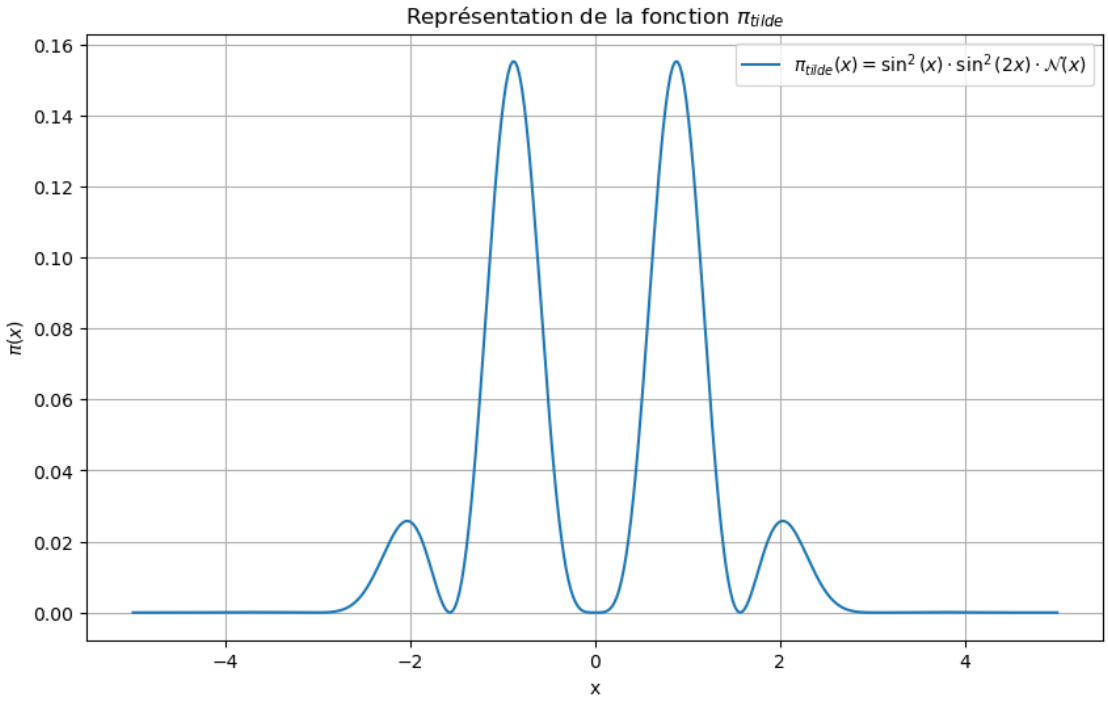
```
Matrice P :
 [[0.1 0.1 0.5 0.2 0.1]
 [0.3 0.1 0.1 0.4 0.1]
 [0.2 0.1 0.1 0.5 0.1]
 [0.1 0.1 0.1 0.6 0.1]
 [0.1 0.2 0.1 0.4 0.2]]
```

```
Matrice P approchée :
 [[0.1 0.1 0.5 0.2 0.1]
 [0.3 0.1 0.1 0.4 0.1]
 [0.2 0.1 0.1 0.5 0.1]
 [0.1 0.1 0.1 0.6 0.1]
 [0.1 0.2 0.1 0.4 0.2]]
```

0.2 Exercice 3

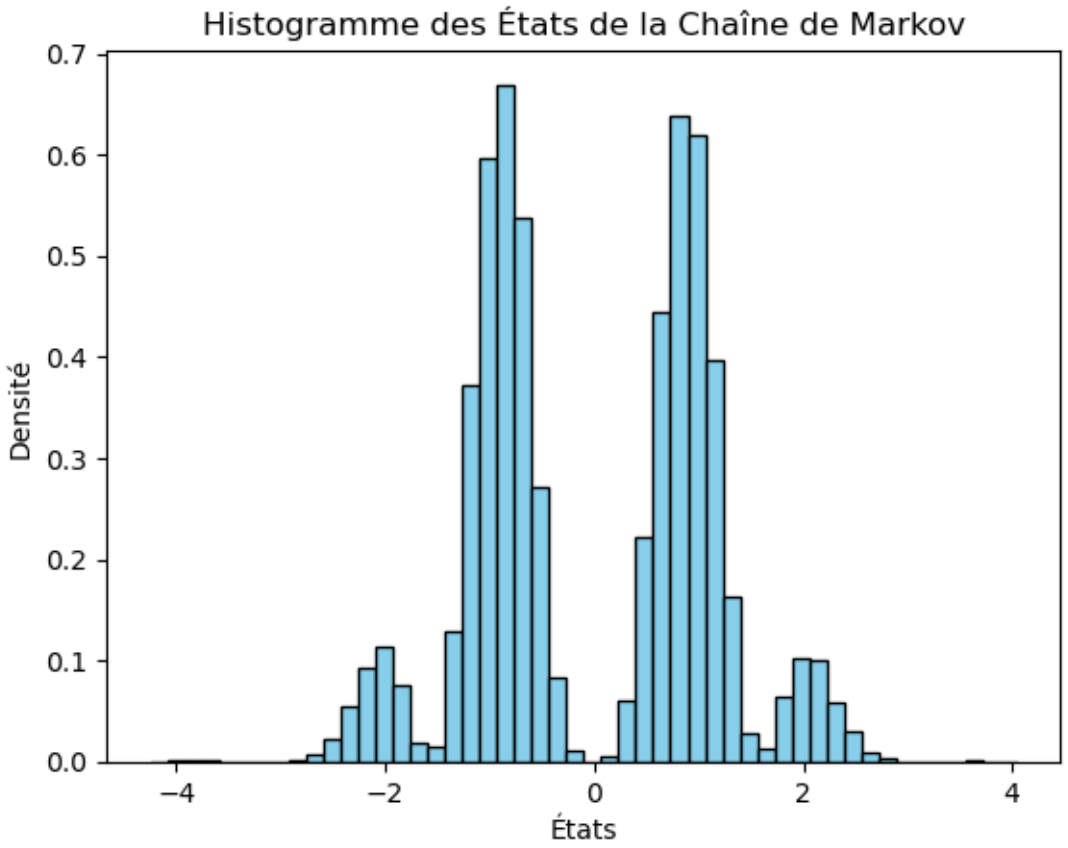
0.2.1 Question 2

[7]:
```
def pi_tilte(x):
    return math.sin(x)**2 * math.sin(2*x)**2 * norm.pdf(x)

x_values = np.linspace(-5, 5, 400)
y_values = [pi_tilte(x) for x in x_values]

plt.figure(figsize=(10, 6))
plt.plot(x_values, y_values, label='$\pi_{tilde}(x) = \sin^2(x) \cdot \sin^2(2x)␣
 ↪\cdot \mathcal{N}(x)$')
plt.title('Représentation de la fonction $\pi_{tilde}$')
plt.xlabel('x')
plt.ylabel('$\pi(x)$')
plt.legend()
plt.grid(True)
plt.show()
```



0.2.2 Question 3

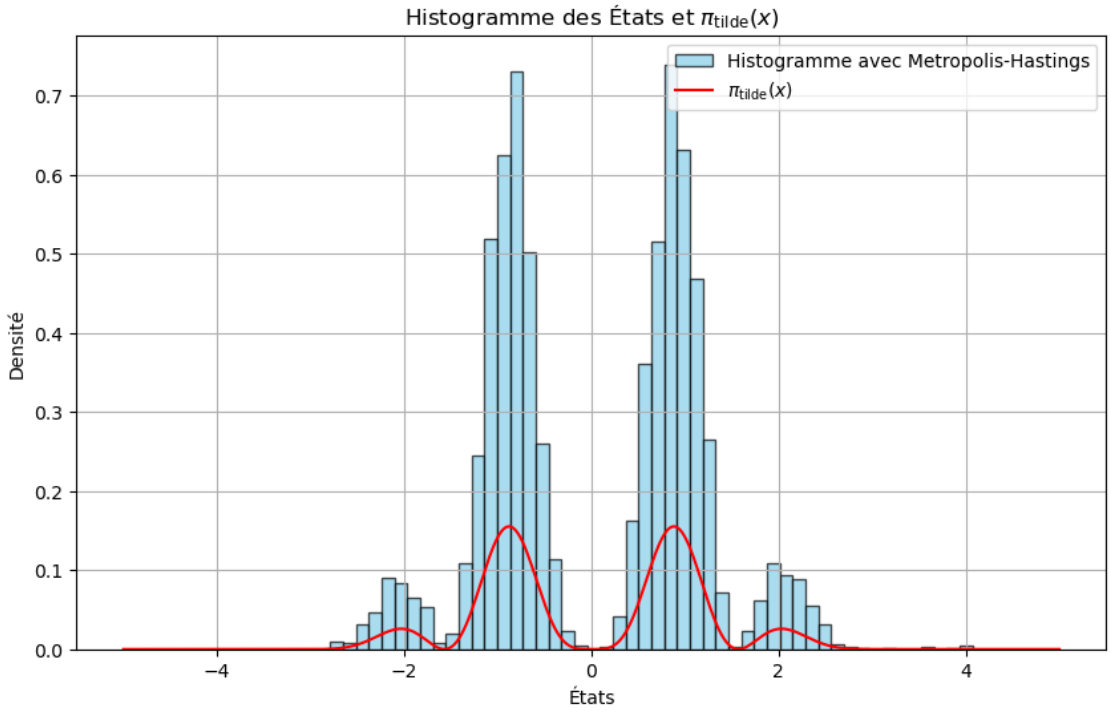
[8]:
```
alpha = 1
x0 = 3.24
N = 100000

def simuler_Y(x,alpha):
    u = rand()
    return x + alpha*(2*u - 1)

def simuler_loi_pi(x0, alpha, N):
    X = [x0]
    x_actuel = x0

    def h(x,y):
        return min(1, pi_tilte(y)/pi_tilte(x))

    for _ in range(N):
        y = simuler_Y(x_actuel, alpha)
        u = rand()
        if u <= h(x_actuel, y):
            x_actuel = y
        X.append(x_actuel)
    return X

X = simuler_loi_pi(x0, alpha, N)

plt.hist(X, bins=50, color='skyblue', edgecolor='black', density=True)
plt.xlabel('États')
plt.ylabel('Densité')
plt.title('Histogramme des États de la Chaîne de Markov')
plt.show()
```



0.2.3 Question 4

[9]:
```
alpha = 1
x0 = 3.24
N = 10000

X = simuler_loi_pi(x0, alpha, N)

plt.figure(figsize=(10, 6))

plt.hist(X, bins=50, color='skyblue', edgecolor='black', density=True, alpha=0.
↪7, label='Histogramme avec Metropolis-Hastings')

x_values = np.linspace(-5, 5, 400)
y_values = [pi_tilte(x) for x in x_values]
plt.plot(x_values, y_values, color='red', label='$\pi_{\mathrm{tilde}}(x)$')

plt.title('Histogramme des États et $\pi_{\mathrm{tilde}}(x)$')
plt.xlabel('États')
```

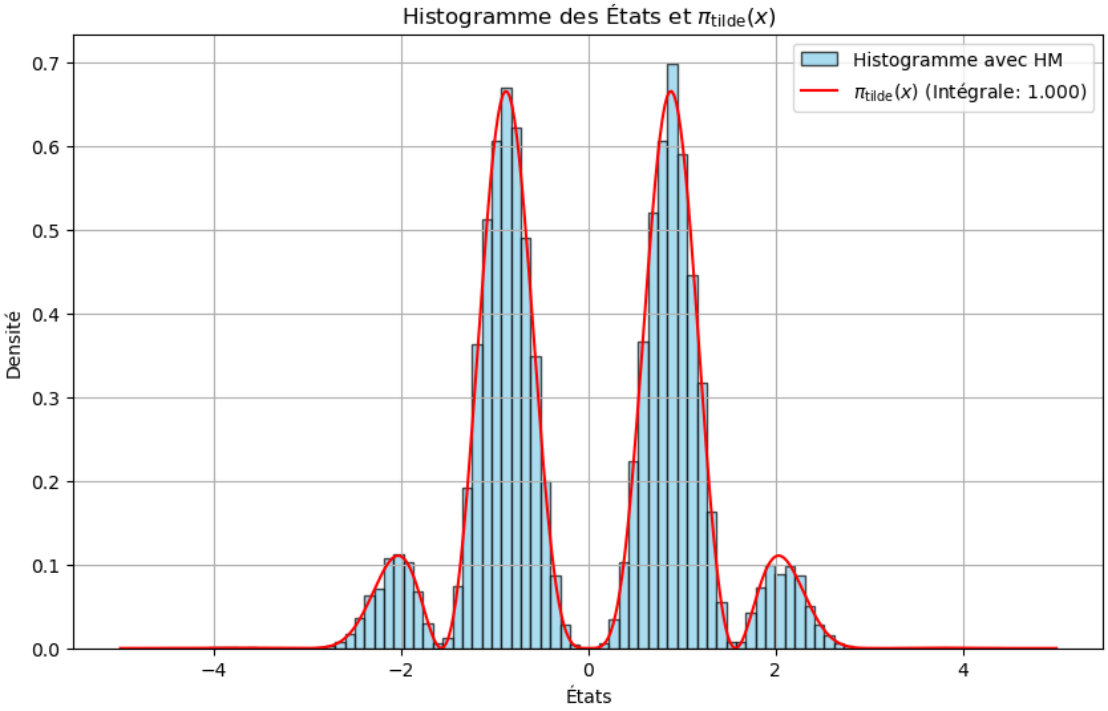
```
plt.ylabel('Densité')
plt.legend()
plt.grid(True)
plt.show()
```



0.2.4 Question 5

[10]:
```
alpha = 1
x0 = 3.24
N = 100000
c = 4.29

def pi(x):
    return c*pi_tilte(x)

X = simuler_loi_pi(x0, alpha, N)

plt.figure(figsize=(10, 6))

plt.hist(X, bins=80, color='skyblue', edgecolor='black', density=True, alpha=0.
↪7, label='Histogramme avec HM')

x_values = np.linspace(-5, 5, 400)
```

```
y_values = [pi(x) for x in x_values]
integrale_pi_tilte, _ = quad(lambda x: pi(x), -5, 5)
plt.plot(x_values, y_values, color='red', label=f'$\pi_{{\mathrm{{tilde}}}}(x)$␣
↪(Intégrale: {integrale_pi_tilte:.3f})')

plt.title('Histogramme des États et $\pi_{\mathrm{tilde}}(x)$')
plt.xlabel('États')
plt.ylabel('Densité')
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```



0.2.5 Question 6

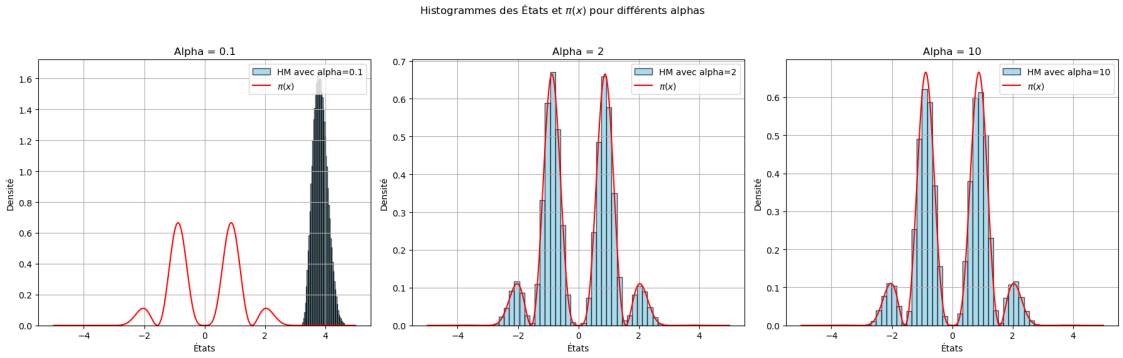
[11]:
```
N = 100000
alpha = 1
x0 = 3.24

def estimer_esperance(N):
    X = simuler_loi_pi(x0, alpha, N)
    Vec_X2 = [x**2 for x in X]
    M = np.mean(Vec_X2)
    return M
```

```
M = estimer_esperance(N)
print(f"E[X²] = {M:.3f}")
```

```
E[X²] = 1.324
```

0.2.6 Question 7

[12]:
```
alphas = [0.1, 2, 10]
x0 = 3.24
N = 100000
c = 4.29

fig, axs = plt.subplots(1, 3, figsize=(18, 6))

for ax, alpha in zip(axs, alphas):
    X = simuler_loi_pi(x0, alpha, N)

    ax.hist(X, bins=50, color='skyblue', edgecolor='black', density=True,␣
 ↪alpha=0.7, label=f'HM avec alpha={alpha}')

    x_values = np.linspace(-5, 5, 400)
    y_values = [pi(x) for x in x_values]
    ax.plot(x_values, y_values, color='red', label=f'$\pi(x)$')

    ax.set_title(f'Alpha = {alpha}')
    ax.set_xlabel('États')
    ax.set_ylabel('Densité')
    ax.legend(loc='upper right')
    ax.grid(True)

# Titre global pour la figure
plt.suptitle('Histogrammes des États et $\pi(x)$ pour différents alphas')
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



Plus α est grand, plus la chaîne de Markov a tendance à rapidement se déplacer sur notre intervalle : les y_n choisis par la loi $Q(x_n, \cdot)$ s'éloignent rapidement des x_n précédents.

À l'inverse, malgré un N choisi assez grand ($N = 10^5$), l'algorithme d'Hasting-Metropolis ne fonctionne pas bien pour $\alpha = 0.1$. Pourquoi ? Car les y_n choisis par la loi $Q(x_n, \cdot)$ sont extrêmement proches des précédents x_n, la chaîne de Markov se déplace donc très lentement sur notre intervalle. Cela explique aussi la présence de cette "bosse" vers $x_0 = 3.24$ puisque la chaîne de Markov met énormément d'itérations avant de quitter cette zone.

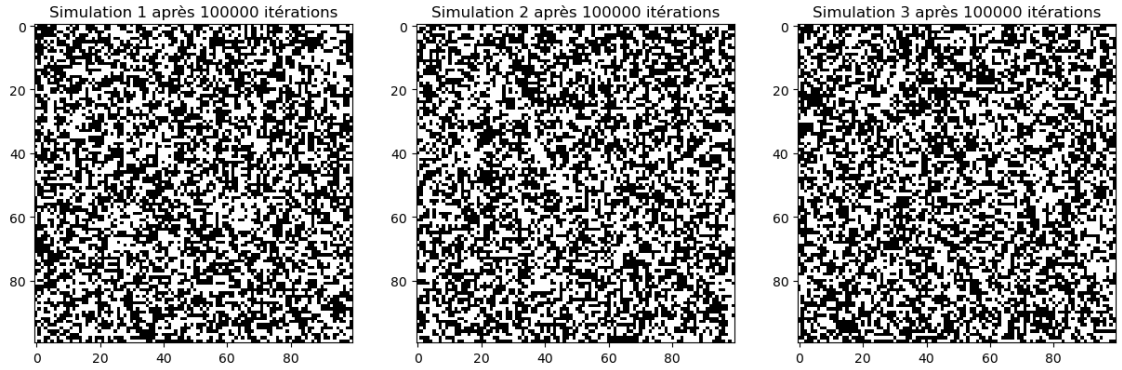
0.3 Exercice 4

0.3.1 Question 1

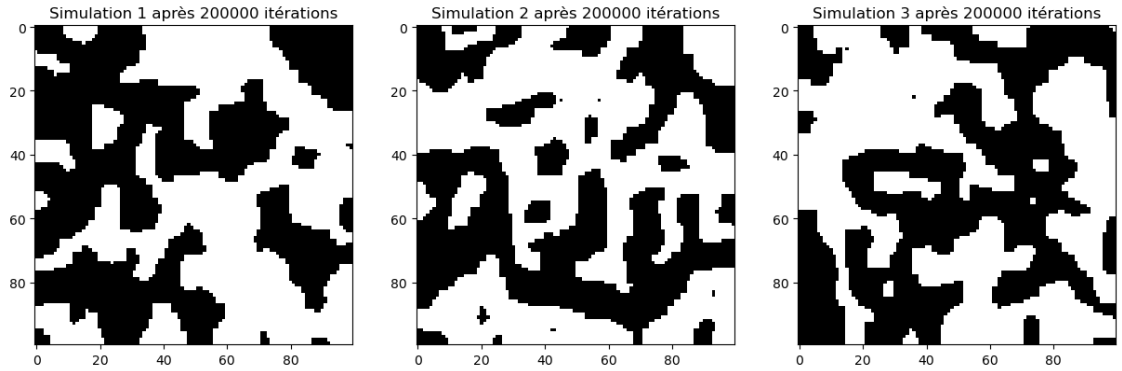
[50]:
```
N = 100
# K = -1 (rigolo, ça inverse totalement le style de résultat)

def initialiser_S(N):
    S = np.zeros((N,N))
    for i in range(N):
        for j in range(N):
            u = rand()
            if u <= 1/2:
                S[i,j] = -1
            else:
                S[i,j] = 1
    return S

S = initialiser_S(N)

def voisins_de(x, S):
    N = S.shape[0]
    i, j = x
    voisins = []
    if j > 0:
        voisins.append((i, j-1))
    if j < N-1:
        voisins.append((i, j+1))
    if i > 0:
        voisins.append((i-1, j))
    if i < N-1:
        voisins.append((i+1, j))
    return voisins

def Delta_H(x, S):
    return 2*K*S[x]*np.sum([S[y] for y in voisins_de(x,S)])
```

```
def h(x, S, beta):
    return math.exp(- beta * max(0, Delta_H(x, S)))     # Attention, erreur dans␣
 ↪l'énoncé

def simuler_V(S):   # Par la méthode de rejet dans le cadre les lois uniformes
    N = S.shape[0]
    valeurs = [i for i in range(N)]
    i = tirage_va_Discrete_Uniforme(valeurs)
    j = tirage_va_Discrete_Uniforme(valeurs)
    return (i,j)

def Ising_algorithme(S, beta, n):
    for _ in range(n):
        x = simuler_V(S)
        U = rand()
        if U <= h(x, S, beta):
            S[x] = - S[x]
    return S
```

[51]:
```
beta = 0.1
n = 100000

new_S1 = Ising_algorithme(S.copy(), beta, n)
new_S2 = Ising_algorithme(S.copy(), beta, n)
new_S3 = Ising_algorithme(S.copy(), beta, n)

# Visualisation des résultats
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

axs[0].imshow(new_S1, cmap='gray')
axs[0].set_title('Simulation 1 après ' + str(n) + ' itérations')

axs[1].imshow(new_S2, cmap='gray')
axs[1].set_title('Simulation 2 après ' + str(n) + ' itérations')

axs[2].imshow(new_S3, cmap='gray')
axs[2].set_title('Simulation 3 après ' + str(n) + ' itérations')

plt.show()
```



[52]:
```
beta = 10
n = 100000

new_S1 = Ising_algorithme(S.copy(), beta, n)
new_S2 = Ising_algorithme(S.copy(), beta, n)
new_S3 = Ising_algorithme(S.copy(), beta, n)

# Visualisation des résultats
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

axs[0].imshow(new_S1, cmap='gray')
axs[0].set_title('Simulation 1 après ' + str(n) + ' itérations')

axs[1].imshow(new_S2, cmap='gray')
axs[1].set_title('Simulation 2 après ' + str(n) + ' itérations')

axs[2].imshow(new_S3, cmap='gray')
axs[2].set_title('Simulation 3 après ' + str(n) + ' itérations')

plt.show()
```



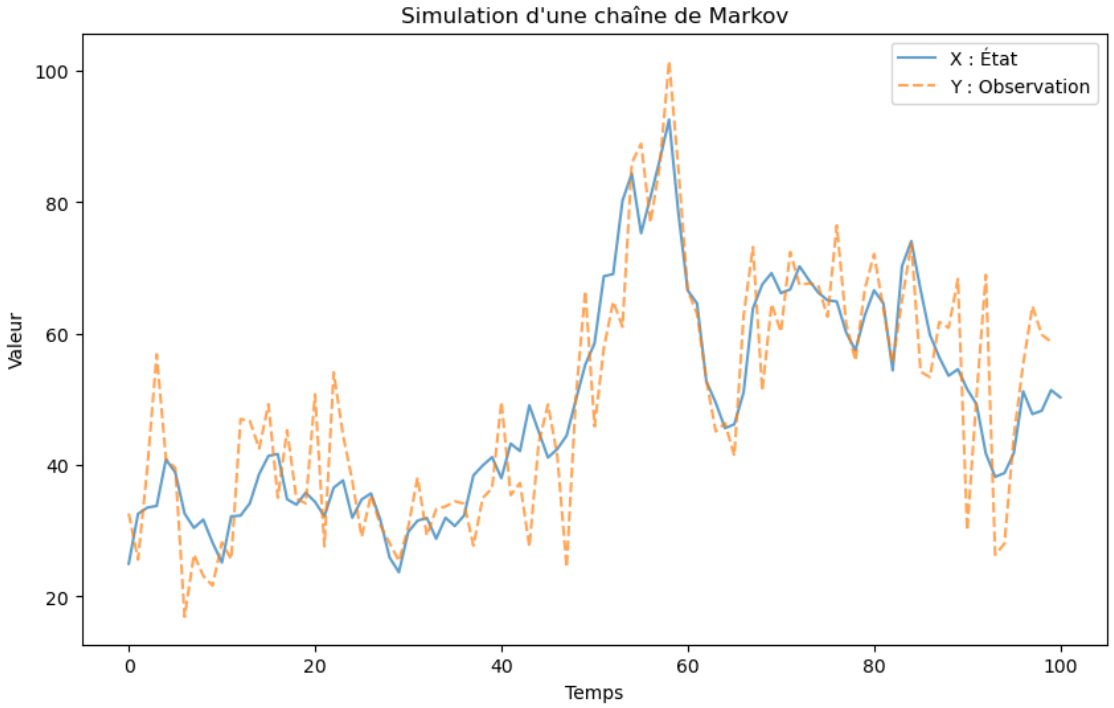
La quantité $T = 1/\beta$ est la température : lorsque la température T est élevée, les fluctuations thermiques dominent rendant le système désordonné tandis qu'en basse température (lorsque T est proche de 0), le système privilégie les configurations de basse énergie tendant à aligner les spins.

0.4 Exercice 6

0.4.1 Question 4

[53]:
```
def simuler_chaine_Markov(n, nu, mu, delta, X0):
    X = np.zeros(n+1)
    Y = np.zeros(n)
    X[0] = X0
    for k in range(n):
        Zk = simuler_loi_normale(0,nu)
        Vk = simuler_loi_normale(0,delta)
        X[k+1] = X[k]*(1+mu) + X[k]*Zk
        Y[k] = X[k] + Vk
    return (X,Y)
```

[55]:
```
nu = 0.1
mu = 0.03
X0 = 25
delta = 9
n = 1000

X,Y = simuler_chaine_Markov(n, nu, mu, delta, X0)

x1, x2 = 000, 100

# Tracé des résultats avec Y en pointillé
plt.figure(figsize=(10, 6))
# plt.plot(X, label='X: État', alpha=0.7)
# plt.plot(Y, label='Y: Observation', linestyle='--', alpha=0.7)
plt.plot(range(x1, x2+1), X[x1:x2+1], label='X : État', alpha=0.7)
plt.plot(range(x1, x2), Y[x1:x2], label='Y : Observation', linestyle='--',␣
 ↪alpha=0.7)
plt.xlabel('Temps')
plt.ylabel('Valeur')
plt.title('Simulation d\'une chaîne de Markov')
plt.legend()
plt.show()
```



[63]:
```
n = 10
N = 10000
C = 18

def simuler_chaine_Markov_upgrade(n, nu, mu, delta, X0):
    X = np.zeros(n+1)
    Y = np.zeros(n)
    X[0] = X0
    proba = 1

    def loi_normale(x):
        return (1/math.sqrt(2*math.pi))*math.exp(-(x**2)/2)

    for k in range(n):
        Zk = simuler_loi_normale(0,nu)
        Vk = simuler_loi_normale(0,delta)
        X[k+1] = X[k]*(1+mu) + X[k]*Zk
        Y[k] = X[k] + Vk
        proba *= (1/delta)*loi_normale((Y[k]-X[k])/delta)

    resultats = np.array([np.array(X[:n]),np.array(Y)])
    return resultats, proba

def simuler_N_chaine_markov_upgrade(N, n, nu, mu, delta, X0):
```

```
    l_resultats = np.zeros((N,2,n))
    l_proba = np.zeros(N)
    for i in range(N):
        res, pr = simuler_chaine_Markov_upgrade(n, nu, mu, delta, X0)
        l_resultats[i] = res
        l_proba[i] = pr
    return l_resultats, l_proba

simulations, l_proba = simuler_N_chaine_markov_upgrade(N, n, nu, mu, delta, X0)
w = l_proba / np.sum(l_proba)

def h(X):
    return np.prod(np.array(X) > C)

# print([np.max(simulations[i,0]) for i in range(N)])
# print([np.min(simulations[i,0]) for i in range(N)])
# print([h(simulations[i, 0]) for i in range(N)])
# print(l_proba)
# print(w)


temps_moyen_empirique = np.sum([h(simulations[i, 0]) * w[i] for i in range(N)])
print("P(tau > n | y_{0:n}) =", np.round(temps_moyen_empirique,3))
```

```
P(tau > n | y_{0:n}) = 0.935
```

Premièrement, on sait que : $\forall k \in \mathbb{N}^*$, $Y_k = X_k + V_k$ avec $V_k \sim \mathcal{N}\left(0, \delta^2\right)$.

Donc :

$$(Y_k \,|\, X_k = x_k) \sim \mathcal{N}(x_k, \delta^2)$$

On en déduit alors la variable aléatoire $(Y_k \,|\, X_k = x_k)$ admet pour densité la fonction suivante :

$$y_k \longmapsto \frac{1}{\delta}\, \varphi_0\left(\frac{y_k - x_k}{\delta}\right)$$

De plus, on rappelle cette hypothèse dans le modèle des chaînes de Markov cachées :

$$\forall k \in \mathbb{N}^*, \quad p\left(y_k \,|\, x_k, y_{0:k-1}\right) = p\left(y_n \,|\, x_k\right) \tag{1}$$

Ainsi, $\forall n \in \mathbb{N}^*$, on a :

$$\begin{aligned} p\left(y_{1:n} \,|\, x_{1:n}\right) &= p\left(y_n \,|\, x_{1:n}, y_{1:n-1}\right)\, p\left(y_{1:n-1} \,|\, x_{1:n}\right) \\ &= p\left(y_n \,|\, x_{1:n}\right)\, p\left(y_{1:n-1} \,|\, x_{1:n}\right) && \text{(par l'hypothèse (1))} \\ &= p\left(y_n \,|\, x_n\right)\, p\left(y_{1:n-1} \,|\, x_{1:n-1}\right) && \text{(car } \forall k \neq n,\, Y_n \perp X_k\text{)} \end{aligned}$$

\end{proof}

Par une relation de récurrence simple, on en déduit que $(Y_{1:k} \,|\, X_{1:k} = x_{1:k})$ admet pour densité la fonction suivante :

$$p\left(\cdot \,|\, x_{1:n}\right) \,:\, y_{1:n} \longmapsto \delta^{-n} \prod_{k=1}^{n} \varphi_0\left(\frac{y_k - x_k}{\delta}\right)$$

Or, comme on part de $X_0 = x_0$ et non de X_1, on peut conclure que :

$$\boxed{p\left(y_{1:n} \,|\, x_{0:n}\right) = \delta_{x_0}\, \delta^{-n} \prod_{k=1}^{n} \varphi_0\left(\frac{y_k - x_k}{\delta}\right)}$$