université de BORDEAUX

Rapport de Travail d'Étude et de Recherche

Débruitage et reconstruction d'images par méthodes de Monte-Carlo par Chaînes de Markov

Appriou Yves, Morton Eva, Straut Sebastian

Licence MIASHS
Université de Bordeaux
Mai 2025

Encadrant :
Mr. Poix

Remerciements

Nous tenons à remercier sincèrement notre professeur encadrant de ce travail d'étude et de recherche, Mr Jérôme Poix. Sa disponibilité et son écoute tout au long de ce semestre ont assurés la bonne continuité de ce travail, de la présentation des éléments sur lesquels ont voulait se pencher jusqu'à la finalisation de ce rapport. Nous apprécions grandement les conseils qu'il a pu nous apporter.

Nous exprimons également notre gratitude aux responsables de la licence MIASHS de l'Université de Bordeaux et à tous les enseignants nous ayant accompagés pendant ces trois années d'études. Ce projet est l'aboutissement d'un travail rigoureux fournit tout au long de la licence, fructueux de par le cadre académique et le corpus de connaissances enseigné.

Table des matières

1 Introduction

Ce Travail d'Etude et de Recherche a pour objectif d'introduire les méthodes de reconstruction d'images par approche probabiliste. Il se base sur des travaux de physique statistique et de biologie. À partir d'images, des relations physiques entre les éléments, représentés par des pixels, sont simulées. Notre travail a donc pour objectif de réutiliser ces différentes approches et de les appliquer directement aux images, et plus particulièrement à leur reconstruction. Nous nous intéressons aux images bruitées en noir et blanc ou en nuances de gris.

Il semble assez évident qu'une image n'est pas seulement constituée de pixels seuls et indépendants les uns des autres mais aussi de zones délimitées, de contrastes et de textures. Par exemple, sur un zèbre, les rayures sont verticales sur le corps et horizontales sur les pattes. Prenez un zèbre au hasard dans toute la population et vous êtes quasiment sûr que la probabilité qu'il présente le motif soit égale à 1. Ce phénomène s'explique par la morphogénèse : la loi veut que le zèbre soit rayé de cette manière là. L'organisation des rayures ou des pixels sur les images d'un zèbre est donc sous-tendue par des relations de voisinage entre les cellules et les pixels.

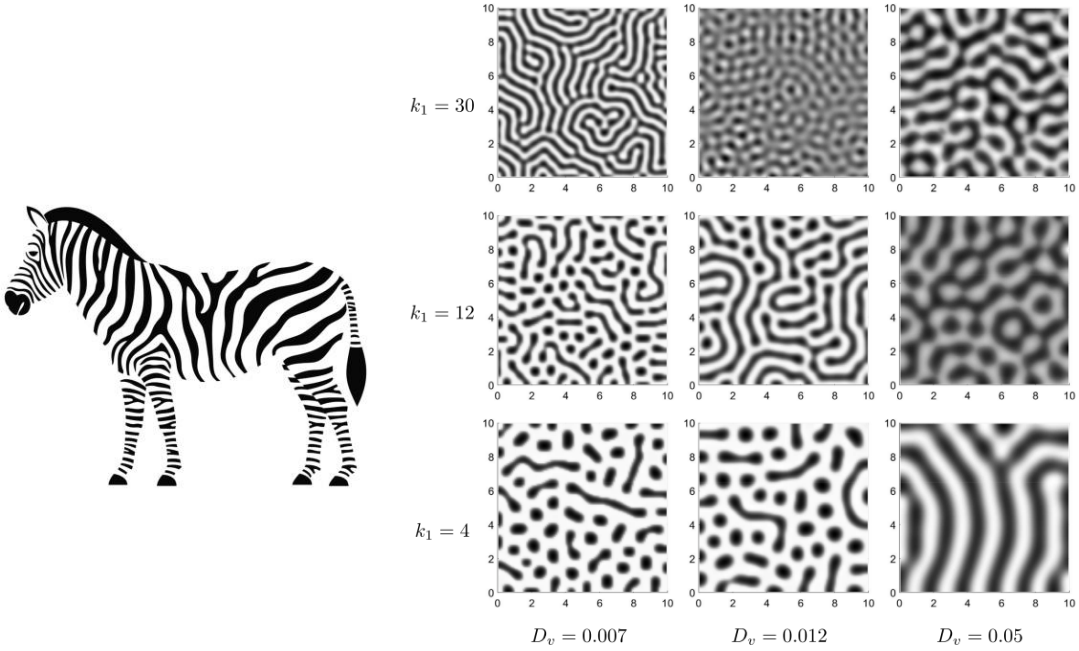


FIGURE 1 – À côté du zèbre sont représentés les résultats de l'étude de Junxiang Yang et Junseok Kim (2023) qui simule la formation d'un motif zébré non homogène à l'aide de structures de Turing, un modèle mathématique avec des paramètres spatiaux.

Le sens du formalisme markovien est donc de reprendre les relations spatiales, de voisinages, entre des zones et pixels. Un poil de zèbre au milieu d'une bande noire est quasi sûrement noir (sauf s'il est vieux, peut-être). La probabilité de l'état d'un site dépend donc du voisinage de ce même site. Pour modéliser ces relations spatiales, nous nous plaçons dans le cadre mathématique des champs de Markov. Une image est une modélisation d'un champ de Markov.

Toutefois, exploiter ces champs pour des tâches de traitement d'images, ici leur reconstruction, demande d'estimer des configurations optimales dans des espaces de grandes di-

mensions. C'est là l'intérêt des méthodes de Monte-Carlo par chaînes de Markov (MCMC). En effet, ne connaissant pas la loi d'un certain champ de Markov, on peut la simuler par l'intermédiaire des méthodes de Monte-Carlo. Mais le champ (l'image) ayant un grand nombre de configurations possibles, on ne peut appliquer la démarche de Monte-Carlo classique qui est de simuler un grand nombre de fois le champ et d'obtenir l'espérance du champ de Markov par la moyenne des simulations, et qui utilise donc la loi des grands nombres. Afin d'illustrer le problème, on peut prendre l'exemple d'un dé à 6 faces. Si on effectue 1000 lancers, on regarde le nombre de fois que l'on a obtenu chaque face afin d'avoir une idée de la loi de probabilité du dé. Cette démarche fonctionne bien en principe puisqu'il n'y a que six modalités possibles. Or, si nous avons par exemple $10^{2564785}$ modalités, on ne peut plus procéder de la sorte, puisque pour avoir un grand nombre d'occurrences de chaque modalité, il faudrait un nombre de simulations « démentiel ». Aussi, pour pallier cette limite de la méthode de Monte-Carlo classique, on introduit des algorithmes permettant de simuler un champ de Markov qui va converger vers la loi de probabilité qui nous intéresse. Nous nous intéressons à deux algorithmes : l'algorithme de Gibbs et l'algorithme de Metropolis-Hastings. On doit beaucoup du cadre théorique des méthodes de Monte-Carlo à Nicholas Metropolis ; il écrit aussi les prémices de l'algorithme à son nom que nous utilisons dans la suite de ce travail, ainsi que la méthode du recuit-simulé. Grâce à ses travaux, on peut donc estimer l'image que l'on souhaite reconstruire.

Finalement, nous comparons les différents algorithmes, méthodes et estimateurs. À travers l'analyse des différents estimateurs, des modèles d'attache aux données et des stratégies d'optimisation, nous discuterons des avantages et des limites des approches par méthodes de Monte-Carlo dans le domaine du traitement d'images.

2 Modélisation probabiliste de l'image

2.1 Définition d'une image

On définit l'image comme un ensemble discret de sites s_i. Ainsi, S est un ensemble fini de sites qui représentent les pixels de l'image, et les descripteurs des états de ces sites représentent la couleur des pixels, à valeur dans E, l'espace des états de la chaîne de Markov. On peut construire un système de voisinage des sites, pour générer des intéractions locales.

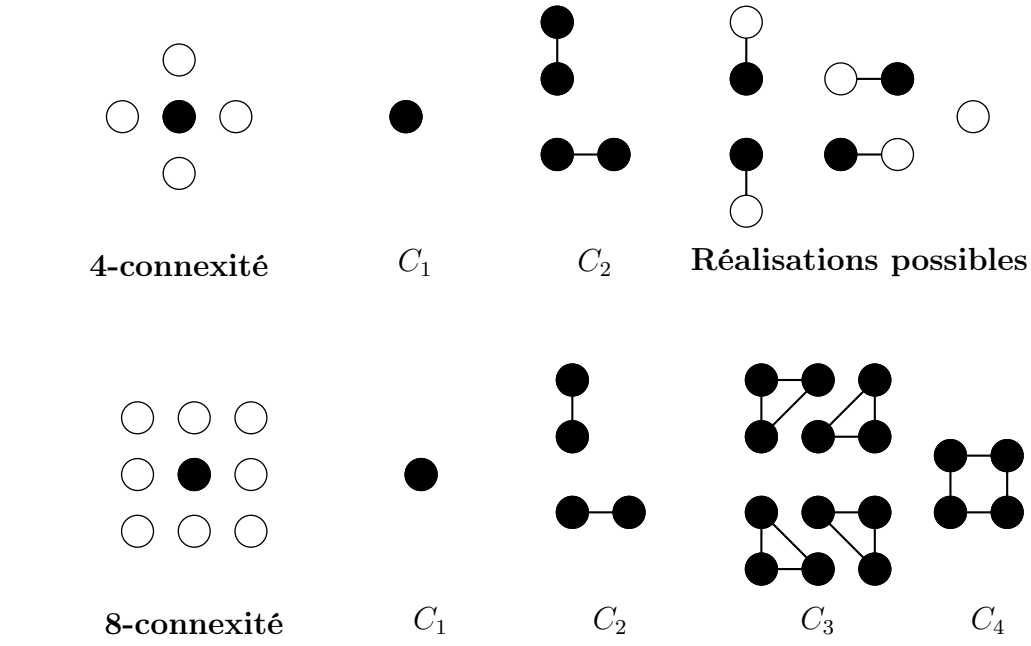
2.2 Voisinages et cliques

On définit comme voisinage d'un site $s \in S$, l'ensemble $V_s = \{t \in S\}$ défini par :

$$V_s = \begin{cases} s \notin V_s \\ t \in V_s \Rightarrow s \in V_t \end{cases}$$

Une clique c est un sous-ensemble de sites voisins entre eux. En théorie des graphes, une clique est un sous-graphe complet, dont les sommets sont deux a deux adjacents. Le système de cliques dépend du type de voisinage utilisé (4-connexité, 8-connexité), c'est-à-dire du graphe dont on va extraire les cliques. Une clique est d'ordre i si elle contient i éléments adjacents. On note finalement C l'ensemble des cliques de l'images. On retrouve

en-dessous de cette explication les ensembles de cliques possibles selon les systèmes de voisinages :



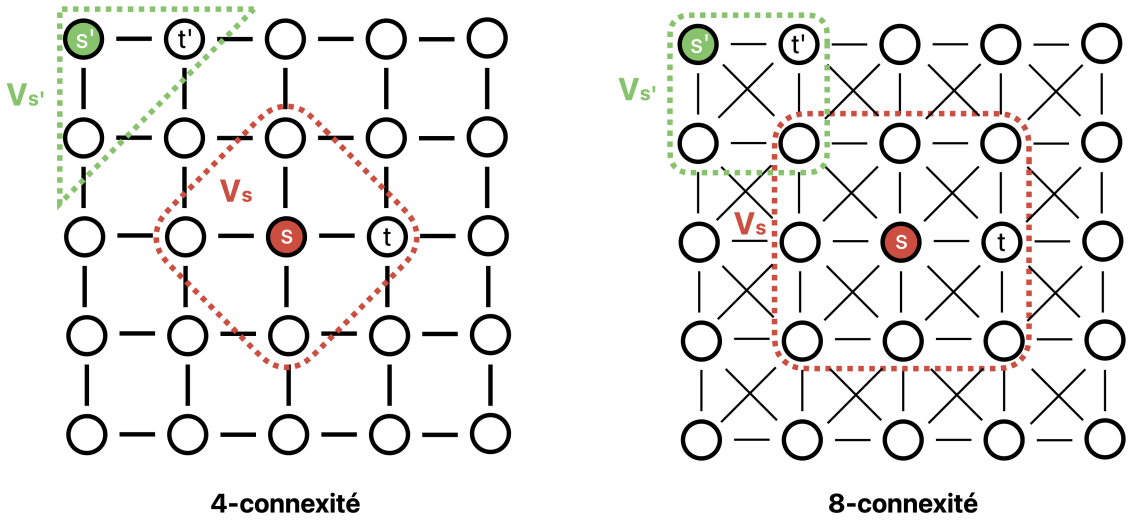


FIGURE 2 – Voisinnages possibles des sites s et s' selon les différents systemes de voisinnages.

On peut remarquer qu'à partir des configurations les plus restrictives en termes de voisins, ici au site s' qui n'a que 2 ou 3 voisins selon le système, on retrouve les conditions de création des cliques. Ainsi, les voisinnages contiennent toutes les réalisations des cliques possibles associées aux systèmes de voisinnage.

On comprend qu'à partir du système de voisinage, les interactions spatiales entre les sites sont formalisées par les cliques. Par exemple, dans le cadre de la 4-connexité, on peut en sortir des réalisations de voisinages possibles au site s_{ij}.

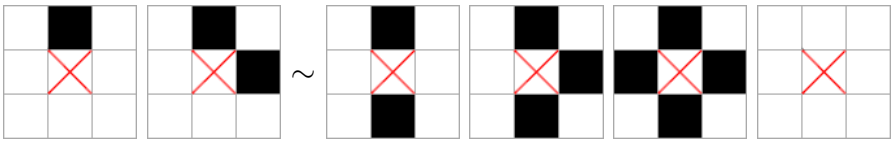


FIGURE 3 – Voisinages possibles au site s_{ij}

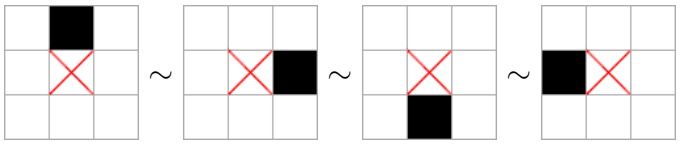


FIGURE 4 – Ensemble des voisinages à 1 pixel possibles au site s_{ij}

On définit à la suite des cliques les potentiels associés. On note U_c le potentiel de la clique c et dont la valeur dépend du niveau de gris des sites de la clique. Le potentiel global de l'image, aussi appelé énergie globale de l'image, est donc la somme des potentiels de toutes les cliques qui la composent :

$$U = \sum_{c \in C} U_c$$

Enfin, l'énergie en un site s (énergie locale) correspond à l'énergie des cliques auxquels le site s appartient :

$$U_s = \sum_{c \in C / s \in c} U_c$$

3 Champs de Markov et de Gibbs

3.1 Approche probabiliste de l'image

Nous avons besoin pour la suite de formaliser et de définir rigoureusement les variables utilisées pour nous placer dans le cadre des champs de Markov.

À partir de s un site de l'image S, on définit X_s une variable aléatoire a valeur dans E. Le descripteur x_s, le niveau de gris en s, est une réalisation de la variable aléatoire X_s. On peut alors definir $X = (X_s, X_t, ...)$ le champ de Markov a valeur dans $\Omega = E^{|S|}$, où $|S|$(ou $Card(S)$) est le nombre de sites qui composent l'image. Par exemple, pour une image à 4 pixels en noir et blanc, on à $|S| = 4$ et $E = \{0, 1\}$, et au final on a $\Omega = E^4$ et $Card(\Omega) = 2^4 = 16$. Il y a donc 16 images possibles issues de ces 2 conditions.

Dans ce cadre, on considère l'image simplement comme une de ses réalisations x du champs X. Ainsi, la probabilité globale de l'image x est donnée par $P(X = x)$, et les probabilités conditionnelles locales d'un état en un site s permettent de mesurer le lien statistique entre un descripteur x_s et le reste de l'image. L'hypothèse markovienne permet d'évaluer ces quantités (probabilités).

3.2 Définition d'un champ de Markov

Considérons x_s la valeur du descripteur du site s et $x^s = (x_t)_{t \neq s} \forall t \in S$ la configuration de l'image excepté le site s. La définition d'un champ de Markov est donc donnée par la

relation suivante :

$$P(X_i = x_i/x^i) = P(X_i = x_i/x_j, j \in V_i)$$

que l'on peut traduire par :

X est un champ de Markov si et seulement si la probabilité conditionnelle locale en un site n est fonction uniquement de la configuration du voisinage du site considéré.

3.3 Mesure et champ de Gibbs

Nous cherchons à introduire dans les champs markoviens des propriétés énergétiques pour accéder à l'expression de probabilités conditionelles locales, c'est-à-dire la probabilité d'un changement d'état au regard du voisinage du site.

Nous utilisons donc la mesure de Gibbs de fonction d'énergie définie comme suit. Soit $U : \Omega \to R$ et la probabilité P définie sur Ω par :

$$P(X = x) = \frac{1}{Z} exp(-U(x))$$

avec

$$U(x) = \sum_{c \in C} U_c(x)$$

la somme des énergies des cliques de C associées au système de voisinage V. On note par ailleurs $Z = \sum_{x \in \Omega} exp(-U(x))$ la constante de normalisation, appelée fonction de partition de Gibbs. On comprend bien que $\Omega = E^{Card(S)}$ est un ensemble de très grande taille. Par exemple, pour une image carrée de largeur 30 pixels en noir et blanc, on a donc $Card(S) = 30 \times 30 = 900$, aussi $Card(\Omega) = 2^{900}$. On a ainsi environ $8,45.10^{270}$ configurations possibles d'images. On peut donc remarquer que la fonction de partition de Gibbs est impossible à calculer, dans la suite nous verrons comment palier ce problème pour l'échantillonage du champ de Gibbs.

Le champ de Gibbs est le champ aléatoire X dont la probabilité est la mesure de Gibbs associée au système de voisinage V :

$$P(X = x) = \frac{1}{Z} exp(-U(x)) = \frac{1}{Z} exp(-\sum_{c \in C} U_c(x))$$

On remarque que la configuration d'énergie minimale du champs de Gibbs est celle qui maximise sa probabilité.

3.4 Théorème d'Hammersley-Clifford

Soient S un ensemble fini ou dénombrable, V un système de voisinage sur S et E un espace d'état discret. Soit (X) un processus aléatoire à valeurs dans $\Omega = E^{Card(S)}$. Alors :

X est un champ de Markov relativement a V et $P(X = x) > 0 \; \forall x \in \Omega$
$\Leftrightarrow$
X est un champs de Gibbs de potentiel associé à V

Ce théorème nous permet donc de passer d'un champ de Markov, avec une énergie globale difficile à calculer, à un champ de Gibbs, qui a pour propriété de décomposer l'énergie globale en somme d'énergies locales.

On obtient alors la probabilité conditionnelle locale suivante :

$$P(X_s = x_s | X_r = x_r, r \neq s) = \frac{\exp -U_s(X_s = x_s | X_r = x_r, r \in V_s)}{\sum_{\lambda \in E} \exp -U_s(X_s = \lambda | X_r = x_r, r \in V_s)}$$

où $U_s(X_s = x_s | X_r = x_r, r \in V_s)$ est l'énergie locale du site s, qui ne fait intervenir que ses sites voisins. On peut donc transformer la fonction de partition de Gibbs : elle ne repose maintenant que sur l'ensemble des états possibles. Pour une image en noir et blanc, $Card(E) = 2$, nous n'avons plus que deux termes à calculer, ce qui permet d'augmenter drastiquement la rapidité des algorithmes que nous décrivons dans la section suivante.

4 Méthodes de Monte-Carlo : Échantillonnage et réalisations de champs de Markov

Dans cette partie nous discutons des algorithmes développés pour simuler des réalisations d'un champ de Markov assimilé à un champ de Gibbs.

Les méthodes de Monte-Carlo nous permettent de faire converger le champ vers la loi de Gibbs (mesure de Gibbs) de ce même champ, celle qui minimise l'énergie globale du champ. On parle alors de convergence en loi (loi faible des grands nombres), et la mesure de Gibbs est considérée comme la probabilité invariante du champ :

$$X_n \xrightarrow{\mathcal{L}} X$$

4.1 Échantillonneur de Gibbs

L'échantillonneur de Gibbs est une méthode de Monte-Carlo par chaînes de Markov proposée par Geman et Geman en 1984. Elle repose sur la construction itérative d'images, en générant progressivement des configurations qui suivent une distribution de Gibbs. On considère que l'algorithme converge après un certain nombre d'itérations, et les images générées sont alors des réalisations de la loi de Gibbs. À chaque site s_{ij} de l'image, on associe un voisinage V_s.

À l'itération $n+1$, à partir de l'image de l'itération n, l'algorithme procède par mise à jour successive (relaxation) :

1. On choisit un pixel s_{ij} aléatoirement dans l'image.
2. Pour chaque état $\lambda_i \in E$ possible, on calcule l'énergie locale :

$$U_s(x_s = \lambda_i \mid V_s)$$

3. On construit alors le vecteur des énergies locales :

$$U(x_s) = \begin{bmatrix} U_s(x_s = \lambda_1 \mid V_s) \\ U_s(x_s = \lambda_2 \mid V_s) \\ \vdots \\ U_s(x_s = \lambda_k \mid V_s) \end{bmatrix}$$

4. On produit alors, à partir de cette mesure, une réalisation de la loi de Gibbs :

$$\mu(\lambda_i) = \frac{1}{Z} \exp\left(-U_s(x_s = \lambda_i \mid V_s)\right) \quad \text{avec} \quad Z = \sum_{\lambda \in E} \exp\left(-U_s(x_s = \lambda \mid V_s)\right)$$

   la probabilité que le site s_{ij} prenne la valeur λ_i à l'itération $n+1$ est donc donnée par le i-ème élément du vecteur de la loi de Gibbs.

5. On tire dans E un état λ_i selon la loi μ, et on remplace l'état du site s_{ij}.

4.2 Algorithme de Metropolis-Hastings

L'algorithme de Métropolis-Hastings, mis au point par W. K. Hastings en 1970, est issu de la physique statistique et est une généralisation de l'algorithme de Metropolis (Metropolis et al., 1953). Il est similaire à l'échantillonneur de Gibbs dans le sens où c'est aussi une méthode de Monte-Carlo par chaînes de Markov permettant une construction itérative d'images qui sont, après un assez grand nombre d'itérations, des réalisations d'une loi de champs de Markov. Mais il diffère de l'échantillonneur de Gibbs par le fait qu'il n'accepte pas toutes les transitions : il y a une condition aux changements des états.

En effet, l'algorithme repose sur les étapes suivantes :

1. On choisit aléatoirement un pixel s_{ij} dans l'image.
2. On récupère l'état du site s_{ij}, noté λ_s et à valeurs dans E, puis on calcule l'énergie locale du site :
$$U_s(x_s = \lambda_s \mid V_s)$$
3. On tire un état $\lambda_r \in E$ selon la loi uniforme U sur E.
4. On calcule l'énergie locale du nouveau site r_{ij} associée à ce nouvel état :
$$U_r(x_s = \lambda_r \mid V_r = V_s)$$
5. On calcule la variation des énergies locales :
$$\Delta U = U_r(x_s = \lambda_r \mid V_r) - U_s(x_s = \lambda_s \mid V_s)$$
6. Le changement d'état du site s_{ij} par λ_r est accepté si $\Delta U < 0$
7. Sinon, on tire une probabilité p (tirage aléatoire selon une loi uniforme sur $[0, 1]$, permet des tirages équiprobables), et si $p < \exp(-\Delta U)$, on accepte le changement d'état du site s_{ij} par λ_r.

Contrairement à l'échantillonneur de Gibbs, ici seules les transitions qui diminuent l'énergie ou qui satisfont une certaine probabilité d'acceptation sont retenues.

4.3 Algorithme du recuit simulé

L'algorithme du recuit simulé est dédié à la recherche des configurations les plus probables, qui sont obtenues à des états d'énergie minimale. Les images obtenues par recuit simulé correspondent en théorie aux minima globaux d'énergie et sont donc uniques. On introduit pour cela un paramètre de température $T > 0$.

En particulier, lorsque T est élevée, la recherche est exploratoire c'est-à-dire que les changements d'états sont acceptés mêmes s'ils sont sous optimaux. Lorsque $T \to 0$, l'algorithme se rapporche des minima globaux de l'énergie U.

Le recuit simulé prend donc en compte la température T, qui est introduite comme une suite appliquée à un processus. On choisit une température initiale T_0 plutôt grande puis, à chaque itération, la température diminue de façon lente avec :

$$T_n > \frac{c}{\log(2+n)}$$

où n est le nombre maximal d'itérations et c est une constante dépendant de la variation énergétique globale maximale sur l'espace des configurations (en pratique, on prend $c = T_0$). Appliquer un recuit simulé aux algorithmes de Gibbs ou de Metropolis-Hastings (le recuit simulé étant un optimisateur, il doit être associé à un algorithme) influence leur dernière étape de traitement. On accepte le changement de l'état du site s_{ij} :

— pour l'algorithme de Gibbs : selon la réalisation de la loi de Gibbs pour $\frac{U(x)}{T}$ sachant que :

$$P_T(X = x) = \frac{1}{Z(T)} exp\left(\frac{-U(x)}{T}\right)$$

avec $Z(T) = \sum_{x \in \Omega} exp\left(\frac{-U(x)}{T}\right)$.

— pour l'algorithme de Metropolis-Hastings : si $\frac{\Delta U}{T} < 0$, sinon si $p < \exp\left(-\frac{\Delta U}{T}\right)$ où p est une probabilité tirée sur une loi uniforme U.

Autrement dit, plus la température T diminue, plus les estimateurs seront forcés à sélectionner l'énergie la plus basse, puisque leur probabilité conditionnelle sera plus élevée par rapport aux configurations d'énergie plus élevée. L'algorithme s'arrête lorsque les changements des états des sites sont faibles (En pratique, l'algorithme s'arrête au bout d'un certain nombre d'itérations).

Contrairement aux autres algorithmes et modèles, le recuit simulé est le seul à avoir une convergence presque sure (loi forte des grands nombres) sous certaines conditions vers un minimum global, expliquant pourquoi on a de meilleurs résultats comme le montre les réalisations dans la section suivante.

4.3.1 Applications et Taux de restauration

On introduit avant de commencer à faire une application une mesure de taux de restauration, soit le taux de bonne restitution d'un pixel de l'image de départ par l'algorithme utilisé :

$$\varepsilon = \frac{1}{\text{Card}(S)} \sum_{s \in S} \mathbb{1}_{\{Y_s = X_s\}}$$

Dans la partie programmation, cet indice de restauration ε est calculé somme suit :

```
def taux_restauration(img_originale, img_restauree):
    pixels_corrects = np.sum(img_originale == img_restauree)
    total_pixels = img_originale.size
    taux = (pixels_corrects / total_pixels) * 100
    return taux
```

Finalement, on obtient les réalisations suivantes, d'images en noir et blanc et en nuances de gris.

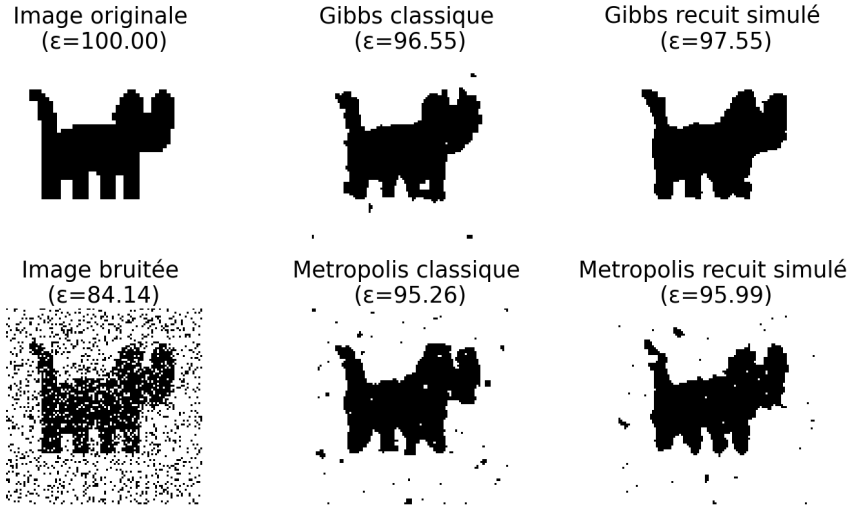


FIGURE 5 – Illustration d'une configuration comparant différentes simulations selon le modèle d'Ising avec et sans recuit simulé, bruit : ($\sigma_b^2 = 0.5$), nombre d'itérations : ($iter = 10^5$), nombre d'états : ($nbetats = 2$), valeur des paramètres : ($\beta = 0.8$)

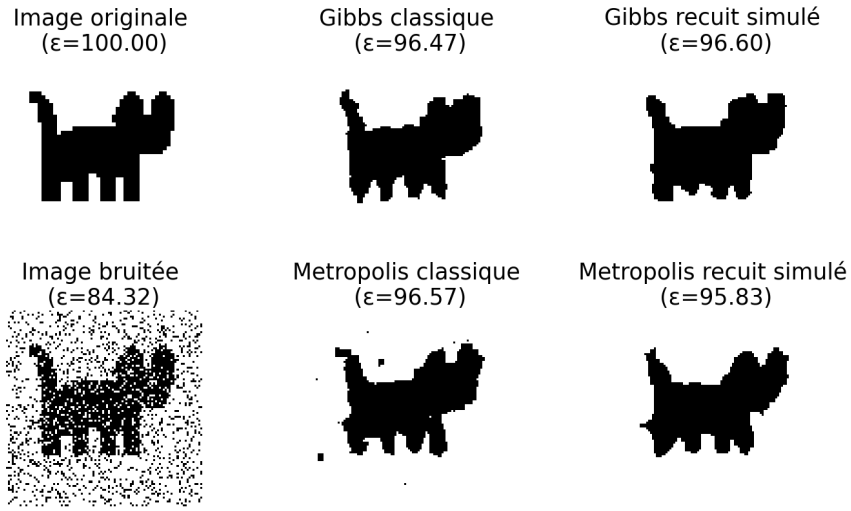


FIGURE 6 – Même configuration mais avec $iter = 2 \times 10^5$.

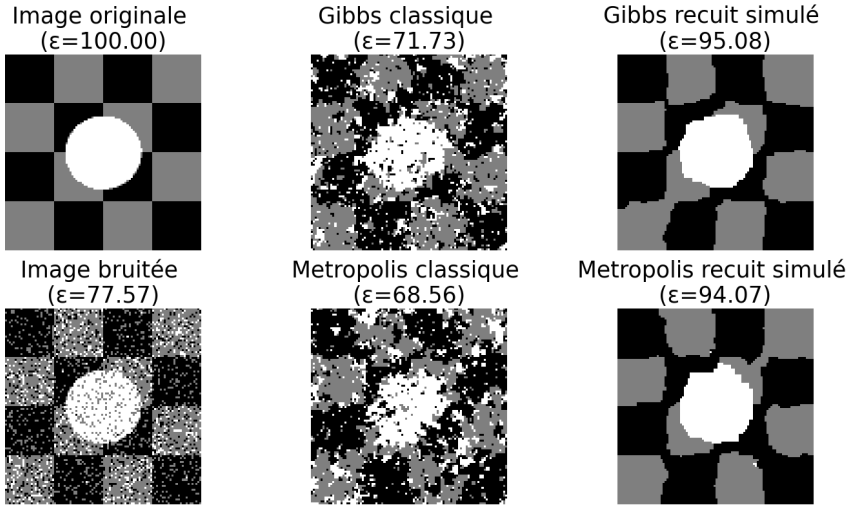


FIGURE 7 – Illustration d'une configuration comparant différentes simulations selon le modèle de Potts avec et sans recuit simulé, bruit : ($\sigma_b^2 = 0.5$), nombre d'itérations : ($iter = 2 \times 10^5$), nombre d'états : ($nbetats = 3$), valeur des paramètres : ($\beta = 0.5$)

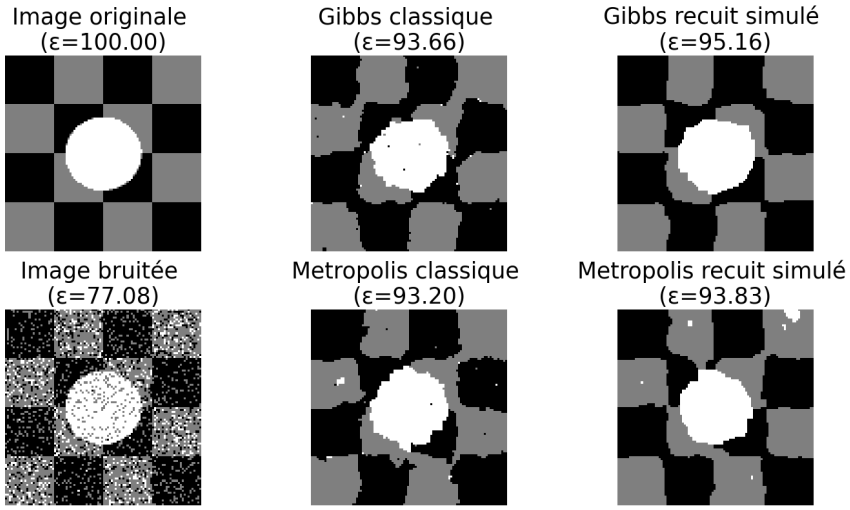


FIGURE 8 – Même configuration mais avec $\beta = 1$.

4.4 Algorithme des modes conditionnels itérés (ICM)

L'algorithme du recuit simulé est très coûteux en temps de calcul car il requiert la génération d'un grand nombre de configurations au fil des itérations. Aussi l'algorithme des modes conditionnels itérés (Iterated Conditional Mode, ICM) est utilisé car beaucoup plus rapide (caractère déterministe de l'algorithme) mais il n'y a pas de preuve de convergence vers un minimum global d'énergie comme pour le recuit simulé.

L'ICM est un algorithme itératif qui modifie le descripteur x_s de chaques sites s de l'image de manière déterministe. Il part d'une configuration initiale et construit une suite d'images qui converge vers une approximation du maximum a posteriori $\hat{x}$. En pratique, chaque descripteur est mis à jour avec la configuration qui permet la diminution d'énergie la plus importante. Aussi, l'ICM est plus rapide car on se contente de prendre la variation d'énergie locale minimale.

Aussi, à l'itération $n+1$, soit pour la mise à jour n de l'image, on parcourt tous les sites s de l'image et :

— on calcule les probabilités conditionnelles locales pour chaque état $\lambda \in E$ du site s, soient $P(X_s = \lambda \mid \hat{x_r}(n), r \in V_s)$. En pratique, on calcule les variations d'énergies locales : $\Delta U = U_s(\lambda \mid V_s) - U_s(x_s(n) \mid V_s)$

— on met à jour chacun des descripteurs $\hat{x}$ par l'état λ maximisant la probabilité conditionnelle locale :

$$\hat{x_s}(n+1) = Argmax_{\lambda \in E} P(X_s = \lambda \mid \hat{x_r}(n), r \in V_s)$$

En pratique, on minimise la variation d'énergie locale (le descripteur x_s est mis à jour en prenant l'état λ si $\Delta U < 0$).

L'algorithme s'arrête lorsque les changements des états des sites sont faibles.

Il est possible de montrer que l'énergie globale de la configuration $\hat{x}$ diminue et converge vers un minimum d'énergie local (et non global comme pour le recuit simulé de part le paramètre de température), l'algorithme de l'ICM dépendant fortement de la configuration initiale.

5 Application à la restauration d'images

5.1 Modèle d'Ising

Le modèle d'Ising est un modèle de physique statistique : il simule le spin (deux états) de particules dans un espace à deux dimensions, de température β et leurs interactions locales. L'espace des états est donc E=$\{-1, 1\}$ (espace binaire). Le modèle utilise les images comme un milieu à deux dimensions où chaque site correspond à une particule et la couleur (noir ou blanc) à son état. On utilise ainsi ce modèle pour la reconstruction d'images en noir et blanc.

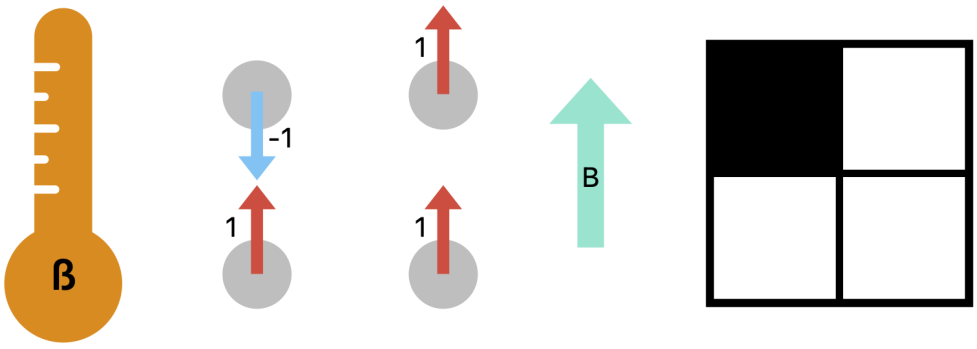


FIGURE 9 – Modèle d'Ising à 4 particules. Les flèches représentent le spin des atomes, β est la température du milieu et $\overrightarrow{B}$ le champ magnétique extérieur appliqué au modèle. On donne à droite la traduction en image dans le modèle.

Les potentiels des cliques d'ordre 2 sont définis de la manière suivante :

$$U_{c=(s,t)}(x_s, x_t) = -\beta x_s x_t = \begin{cases} -\beta & \text{si } x_s = x_t \\ \beta & \text{sinon} \end{cases}$$

où β est la constante de couplage entre sites voisins. Les potentiels des cliques d'ordre 1 (un seul site) sont de la forme $-\beta x_s$.

L'énergie globale s'écrit alors :

$$U(x) = -\sum_{c=(s,t)\in C} \beta x_s x_t - \sum_{s\in S} B x_s$$

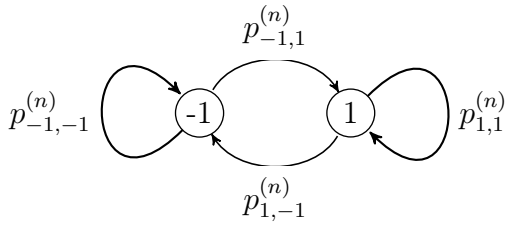
où B représente le champ magnétique externe. Dans le cas du traitement d'image, ce champ est considéré comme nul (c'est dans un cadre de physique statistique que l'on fait varier B).

En pratique, nous calculons l'énergie locale pour un site en particulier, ce qui nous donne la formule suivante :

$$U_s(x_s) = -\beta \sum_{c=(s,t)\in C} x_s x_t - B x_s$$

La valeur de β, la température du milieu modélisé, influence l'évolution du modèle. Si β est positif, les configurations les plus probables, donc celles d'énergie minimale, sont celles pour lesquelles les sites ont le même état ; ceci correspond à des spins de même signe soit un ferromagnétisme (capacité de certains corps à orienter tous leur spins dans un sens). Si β est négatif, sont privilégiées les configurations où les sites sont d'états opposés, soient les spins de signes opposés (anti-ferromagnétisme).

En nous replaçant dans le cadre markovien, au temps n+1 on établit les probabilités de transition du descripteur du site s_{ij} par $p_{i,j}^{(n)} = P(X_{n+1}^{(s)} = j \mid \mathcal{V}_n^{(s)} \cap X_n = i),\ \forall (i,j) \in E^2$. On obtient la matrice de transition de la chaîne de Markov, mise à jour a chaque itération de l'algorithme choisi.



$$M^{(n)} = \begin{pmatrix} p_{-1,-1} & p_{-1,1} \\ p_{1,-1} & p_{1,1} \end{pmatrix}$$

FIGURE 10 – Chaîne de Markov et matrice de transition du site s au temps $n+1$

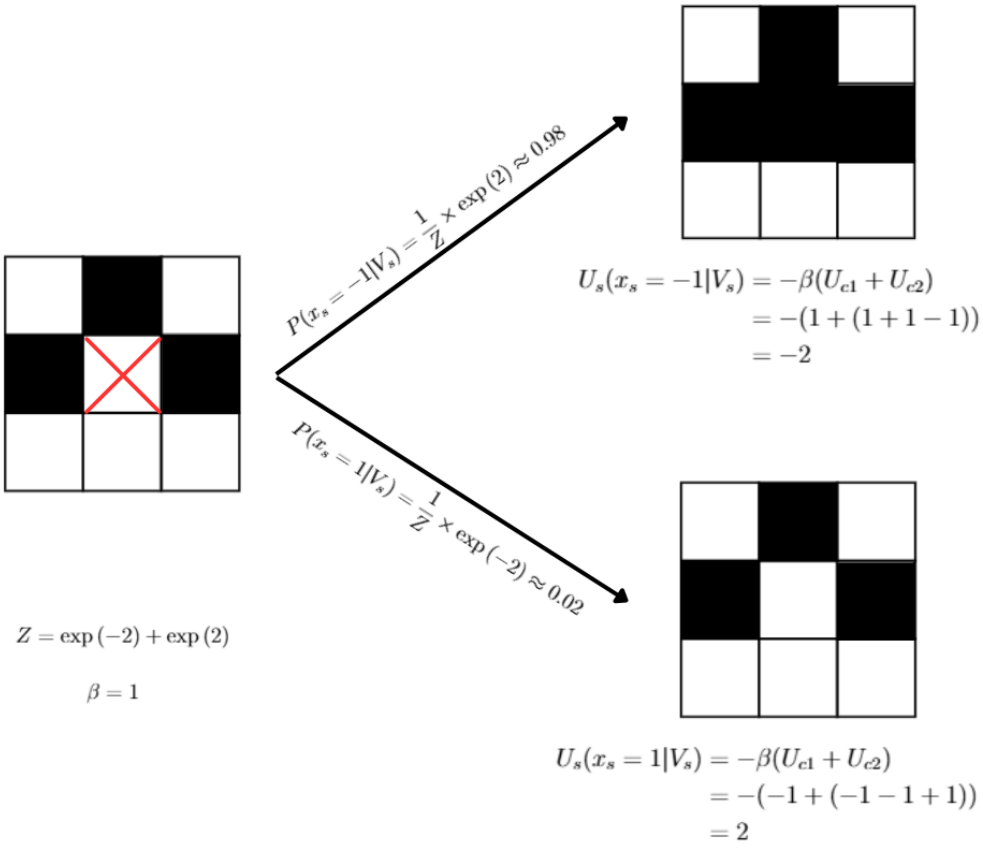


FIGURE 11 – Représentation d'un calcul d'énergie locale et de mesure de Gibbs dans le cadre d'un système de voisinage au site s et en 4-connexité.

Les figures suivantes montrent des réalisations du modèle d'Ising pour différents paramètres (différentes valeurs de β et donc une régularisation plus ou moins importante) avec les algorithmes de Gibbs et de Metropolis-Hastings.

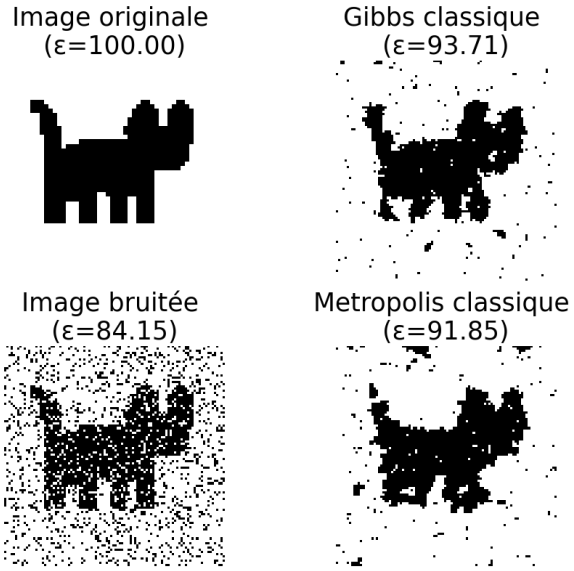


FIGURE 12 – Illustration d'une configuration simulée selon un modèle d'Ising, bruit : $(\sigma_b^2 = 1)$, nombre d'itérations : $(iter = 10^5)$, nombre d'états : $(nbetats = 2)$, valeur des paramètres : $(\beta = 0.6,\ B = 0)$

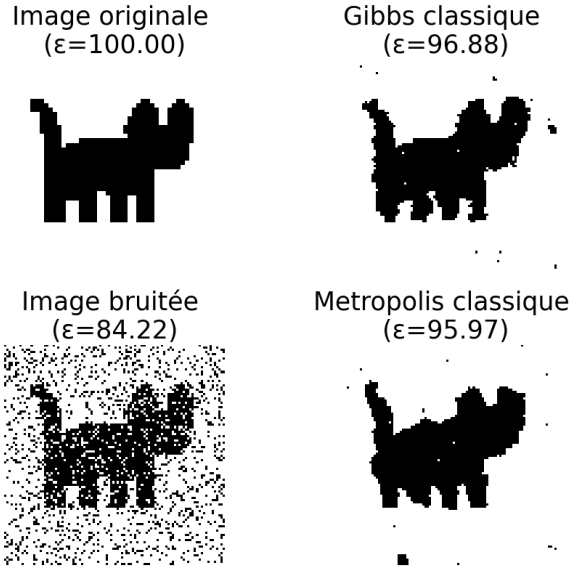


FIGURE 13 – Même configuration mais avec $\beta = 0.9$.

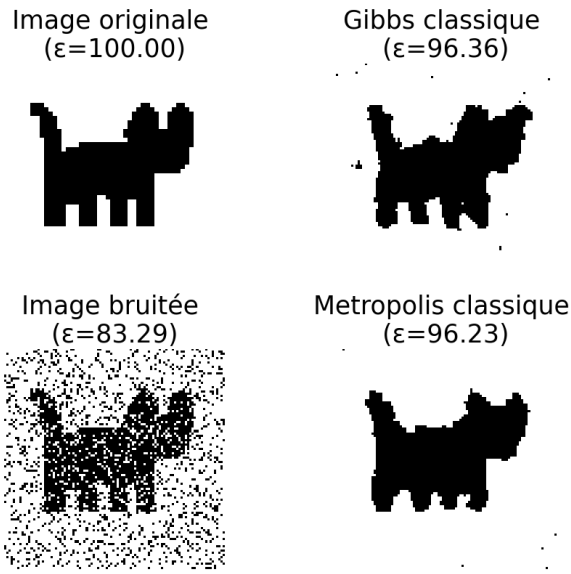


FIGURE 14 – Même configuration mais avec $\beta = 1$.

5.2 Modèle de Potts

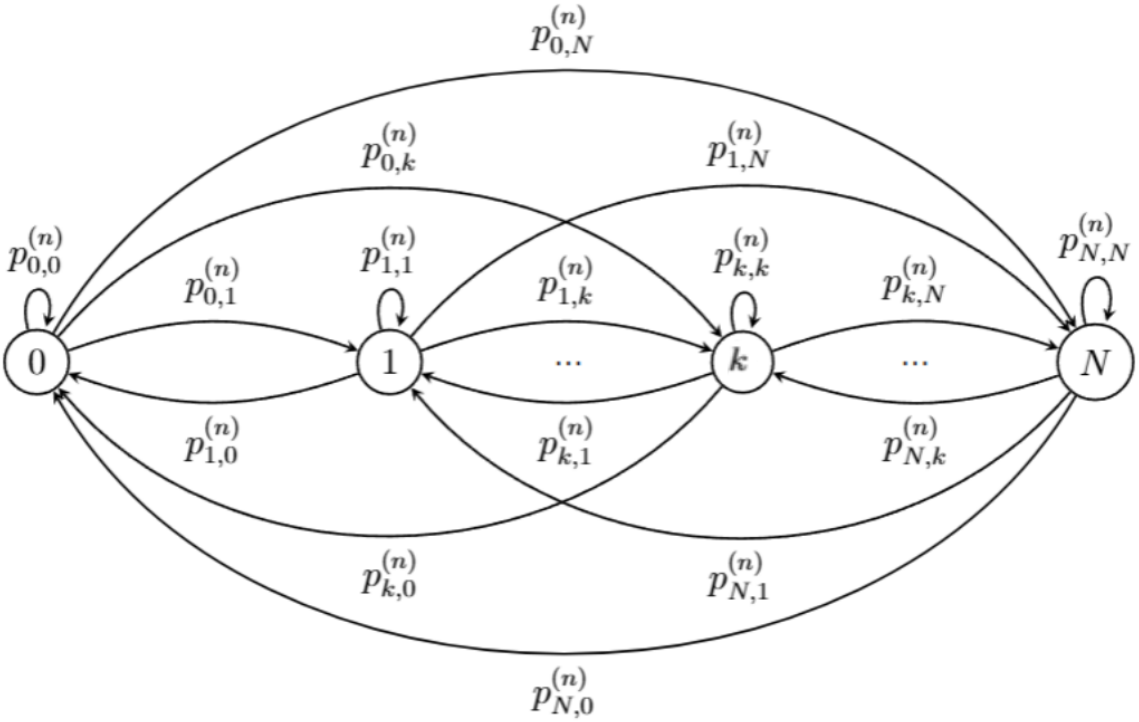
Le modèle de Potts ou modèle cellulaire de Potts est un modèle d'interaction et de comportement de cellules dans un environnement en deux dimensions. Le modèle de Potts est une généralisation du modèle d'Ising vers un espace d'états de taille supérieure à 2, on a donc $E = \{0, 1, \ldots, N\}$ un espace discret de taille $N + 1$. Le modèle utilise l'image comme un milieu de développement des cellules, et la couleur donne le type de cellules présentes dans une colonie.

Les potentiels ne sont cette fois-ci définis que pour des cliques d'ordre 2 :

$$U_{c=(s,t)}(x_s, x_t) = -\beta x_s x_t = \begin{cases} -\beta & \text{si } x_s = x_t \\ \beta & \text{sinon} \end{cases}$$

Ici β représente l'étalement du groupe de cellules sur le milieu. Si β est positif, les configurations les plus probables sont celles pour lesquelles les sites voisins ont les mêmes descripteurs (par exemple le même niveau de gris), ce qui en traitement d'images donne des réalisations constituées de larges zones homogènes dont la taille varie selon β.

Comme pour le modèle d'Ising, en nous replaçant dans le cadre markovien, au temps $n+1$ on établit les probabilités de transition du descripteur du site s_{ij} par $p_{i,j}^{(n)} = P(X_{n+1}^{(s)} = j \mid \mathcal{V}_n^{(s)} \cap X_n = i),\ \forall (i,j) \in E^2$. On obtient la matrice de transition de la chaîne de Markov, mise à jour à chaque itération de l'algorithme choisi.



$$M^{(n)} = \begin{pmatrix} p_{0,0}^{(n)} & p_{0,1}^{(n)} & p_{0,2}^{(n)} & \cdots & p_{0,N}^{(n)} \\ p_{1,0}^{(n)} & p_{1,1}^{(n)} & p_{1,2}^{(n)} & \cdots & p_{1,N}^{(n)} \\ p_{2,0}^{(n)} & p_{2,1}^{(n)} & p_{2,2}^{(n)} & \cdots & p_{2,N}^{(n)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{N,0}^{(n)} & p_{N,1}^{(n)} & p_{N,2}^{(n)} & \cdots & p_{N,N}^{(n)} \end{pmatrix}$$

FIGURE 15 – Chaîne de Markov et matrice de transition du site s au temps $n+1$

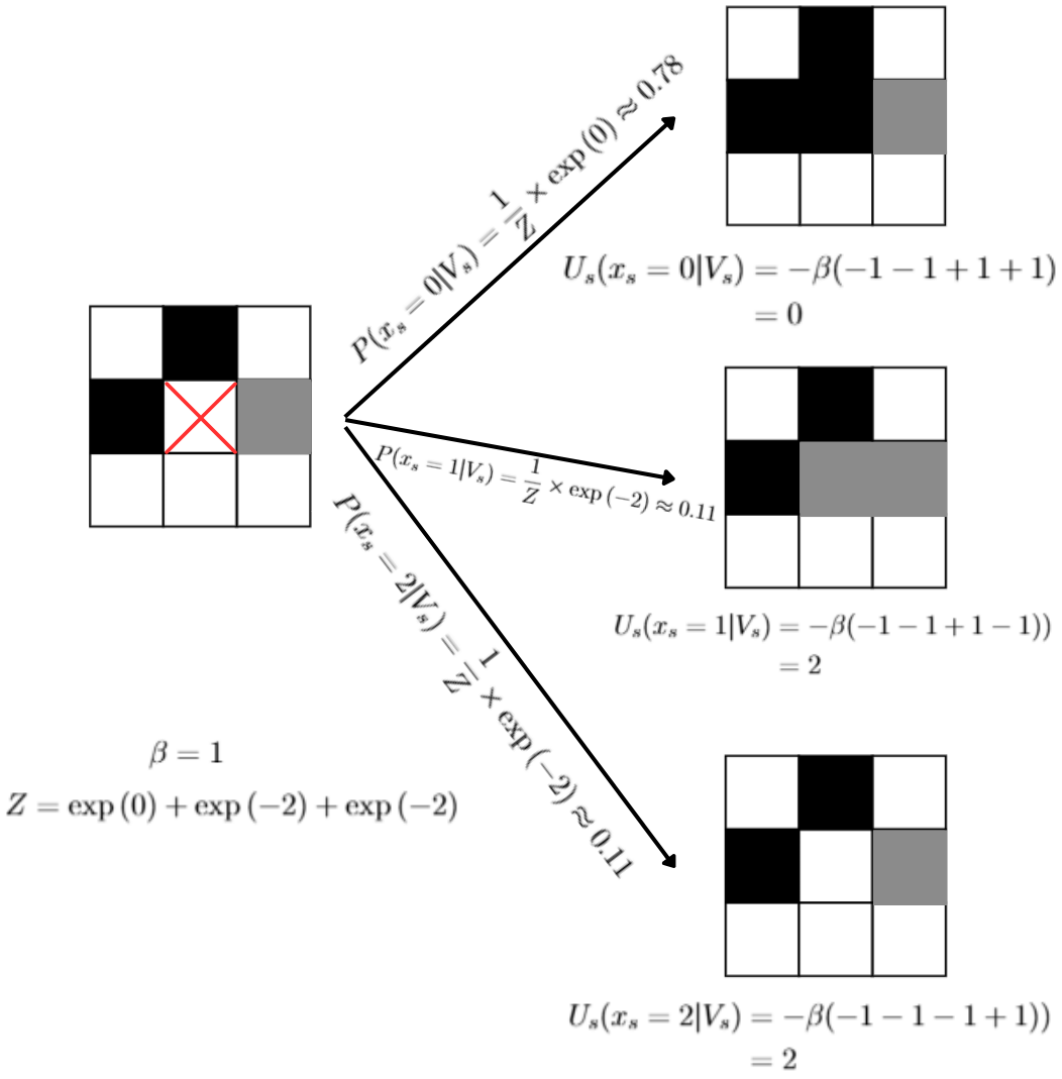


FIGURE 16 – Représentation d'un calcul d'énergie locale et de mesure de Gibbs dans le cadre d'un système de voisinage au site s et en 4-connexité.

Il est possible de parfaire le modèle de Potts en utilisant des valeurs de β différentes en fonction des directions des cliques (par exemple, une exploration verticale/horizontale en 4-connexité) et ainsi privilégier certaines directions. Les figures suivantes montrent des réalisations du modèle de Potts pour différents paramètres avec les algorithmes de Gibbs et de Metropolis-Hastings.

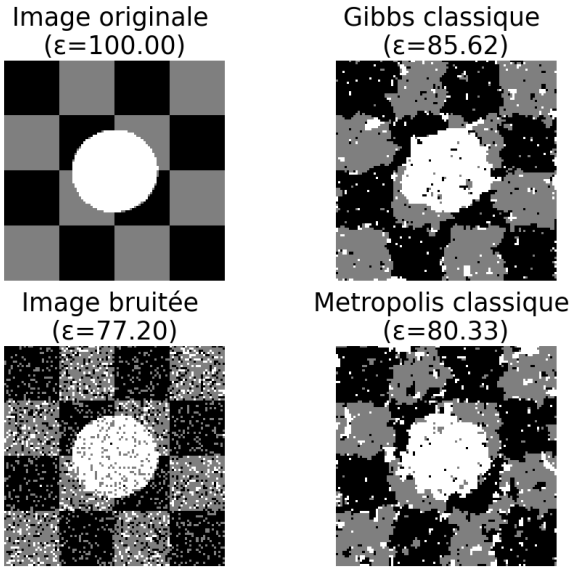


FIGURE 17 – Illustration d'une configuration comparant différentes simulations selon le modèle de Potts, bruit : ($\sigma_b^2 = 0.5$), nombre d'itérations : ($iter = 10^5$), nombre d'états : ($nbetats = 3$), valeur des paramètres : ($\beta = 0.6$)

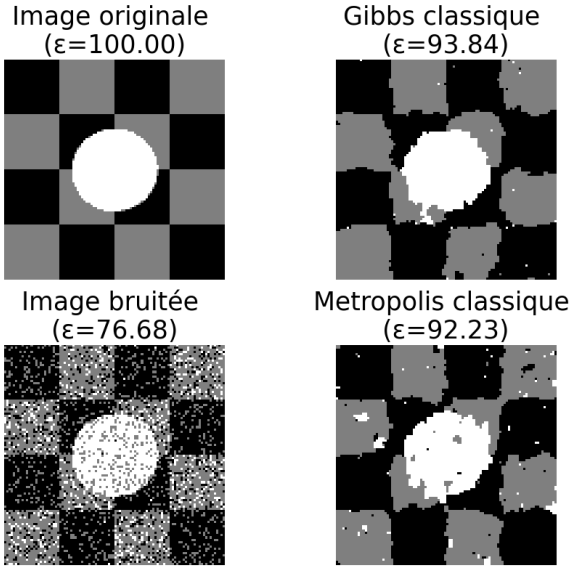


FIGURE 18 – Même configuration mais avec $\beta = 1$

5.3 Modèle Markovien Gaussien

Le modèle markovien gaussien convient à des images en deux dimensions et a pour espace d'états $E = \{0, 1, \ldots, N\}$, discret et de taille N. Plus particulièrement, dans le cas du traitement d'images, ce modèle est utilisé pour les images en niveaux de gris où $E = \{0, \ldots, 255\}$, et favorise les niveaux de gris proches pour des sites voisins. On parle de modèle markovien gaussien car l'image comporte un bruit gaussien, dont la densité de probabilité conditionnelle est une distribution gaussienne, de variance σ_b^2 et d'espérance nulle.

Plus précisément, on bruite l'image que l'on étudie : à chaque site s et son descripteur x_s, on tire une variable aléatoire b sur une loi normale de variance σ_b^2 et d'espérance nulle, on vient ensuite prendre l'arrondi de la somme de descripteur x_s et du bruit b. On remet alors nos valeurs dans E.

Les potentiels des cliques d'ordre 2 sont définis de la manière suivante :

$$U(x) = \beta \sum_{c=(s,t)} (x_s - x_t)^2 + \alpha \sum_{s \in S} (x_s - \mu_s)^2$$

où μ_s est une moyenne de niveaux de gris attendue pour le site s lorsque les descripteurs x_s sont les niveaux de gris. Le premier terme est un terme de régularisation, le second terme correspond à l'attache aux données. Si β est positif, les configurations qui sont favorisées sont celles où les différences entre les descripteurs des sites s et t, soit les niveaux de gris entre les sites voisins, sont faibles. Le rapport $\frac{\alpha}{\beta}$ pondère les influences respectives des termes de régularisation et d'attache aux données, les valeurs absolues de ces paramètres décrivant le caractère équiréparti ou localisé de la distribution.

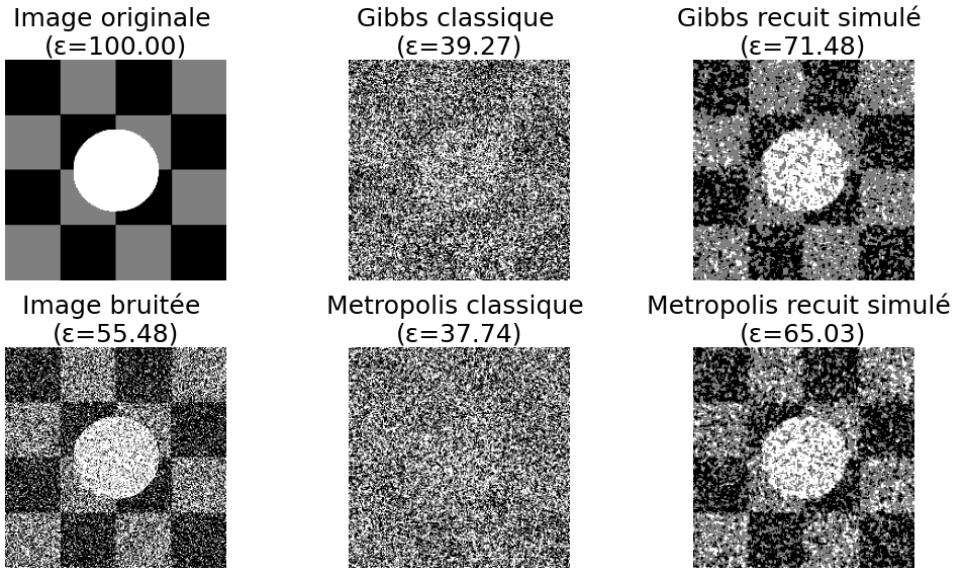


FIGURE 19 – Illustration d'une configuration comparant différentes simulations selon le modèle markovien gaussien avec et sans recuit simulé, bruit : ($\sigma_b^2 = 0.3$), nombre d'itérations : ($iter = 10^5$), nombre d'états : ($nbetats = 3$), valeur des paramètres : ($\beta = 0.1$, $\alpha = 0.01$)

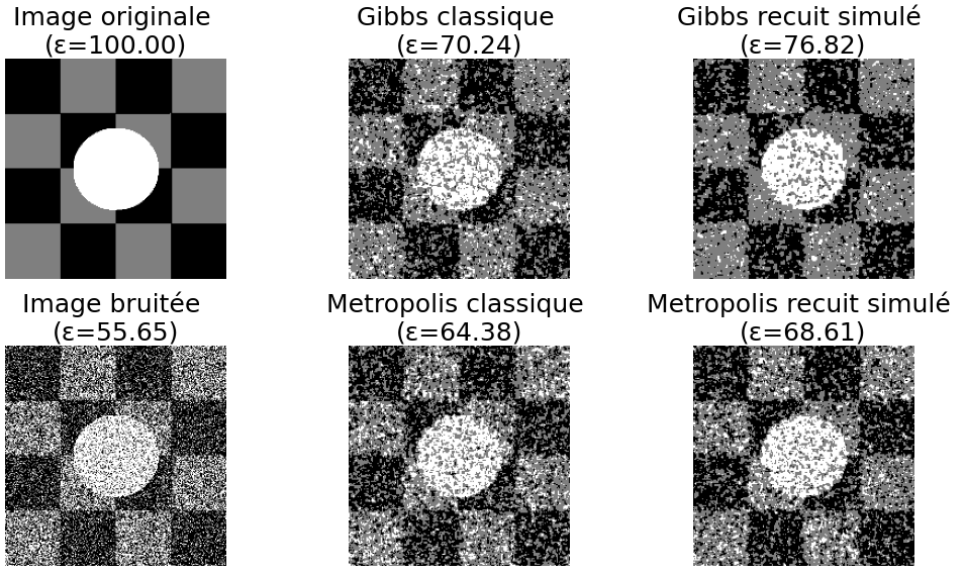


FIGURE 20 – Illustration d'une configuration comparant différentes simulations selon le modèle markovien gaussien avec et sans recuit simulé, bruit : ($\sigma_b^2 = 0.3$), nombre d'itérations : ($iter = 10^5$), nombre d'états : ($nbetats = 3$), valeur des paramètres : ($\beta = 1$, $\alpha = 0.01$)

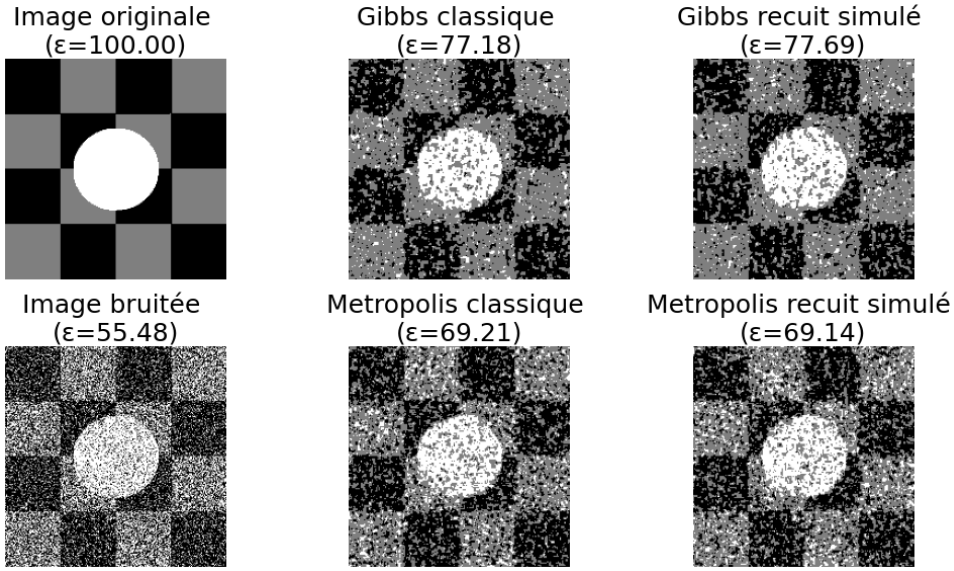


FIGURE 21 – Illustration d'une configuration comparant différentes simulations selon le modèle markovien gaussien avec et sans recuit simulé, bruit : ($\sigma_b^2 = 0.3$), nombre d'itérations : ($iter = 10^5$), nombre d'états : ($nbetats = 3$), valeur des paramètres : ($\beta = 10$, $\alpha = 0.01$)

On observe ci-dessous du flou, ce qui explique un taux de restauration faible. Mais l'objet de l'image est plus reconnaissable : un visage est identifié plus facilement avec un bruit gaussien qu'avec un bruit aléatoire. Par ailleurs, l'algorithme de Gibbs est beaucoup plus coûteux en temps que celui de Metropolis-Hastings, expliquant la variance du nombre d'itérations ci-dessous.

FIGURE 22 – Illustration d'une configuration comparant des simulations de l'algorithme de Gibbs selon le modèle markovien gaussien avec et sans recuit simulé, bruit : ($\sigma_b^2 = 30$), nombre d'itérations :($iter = 10^5$), nombre d'états : ($nbetats = 255$), valeur des paramètres : ($\beta = 20$, $\alpha = 0.01$)

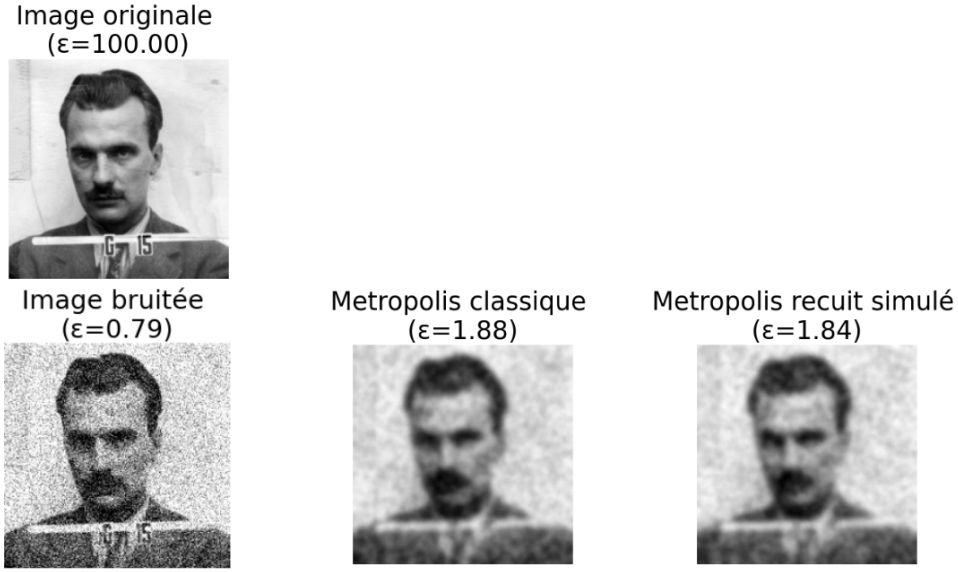


FIGURE 23 – Illustration d'une configuration comparant des simulations de l'algorithme de Metropolis-Hastings selon le modèle markovien gaussien avec et sans recuit simulé, bruit : ($\sigma_b^2 = 50$), nombre d'itérations :($iter = 10^7$), nombre d'états : ($nbetats = 255$), valeur des paramètres : ($\beta = 10$, $\alpha = 0.01$)

6 Estimateurs dans un cadre Markovien

Dans le cadre de la restauration d'images dégradées (par exemple bruitées), une approche probabiliste permet d'estimer l'image idéale à partir de l'image observée. On modélise l'image parfaite inconnue X comme un champ de Markov, et l'image bruitée Y comme une observation de ce champ à travers un processus bruité. L'objectif est de retrouver une

configuration $\hat{x}$ du champ X qui soit la plus plausible compte tenu de l'observation y. Comme nous sommes dans un cadre probabiliste, on peut chercher la configuration $\hat{x}$ qui maximise la probabilité $P(X = x \mid Y = y)$, et qui se note :

$$P(X = x \mid Y = y) = \frac{P(Y = y \mid X = x) \cdot P(X = x)}{P(Y = y)}$$

6.1 Estimateur MAP pour la restauration d'images

6.1.1 Définition de l'estimateur MAP

L'estimateur MAP (Maximum A Posteriori) consiste à rechercher la configuration la plus probable de l'image idéale x, compte tenu de l'image bruitée observée y. Il s'agit donc de maximiser la probabilité conditionnelle suivante :

$$\hat{x}_{MAP} = \arg\max_x \ P(X = x \mid Y = y) = \arg\max_x \ \exp(-U(x \mid y))$$

En appliquant la règle de Bayes, cette probabilité peut être réécrite comme :

$$P(X = x \mid Y = y) = \frac{P(Y = y \mid X = x) \cdot P(X = x)}{P(Y = y)}$$

Le dénominateur $P(Y = y)$ étant constant (indépendant de x), on peut se contenter de maximiser le numérateur :

$$\hat{x}_{MAP} = \arg\max_x \ P(Y = y \mid X = x) \cdot P(X = x)$$

Cette expression reflète un compromis entre deux aspects fondamentaux :

— L'attache aux données $P(Y = y \mid X = x)$: elle mesure à quel point une image candidate x est compatible avec les observations bruitées y. Sous l'hypothèse (fréquente mais simplificatrice) d'indépendance des pixels, c'est à dire à quel point x aurait pu produire y, selon le modèle de bruit.

— La régularisation $P(X = x)$: elle reflète les connaissances a priori sur les images « plausibles ». Ce terme est modélisé par un champ de Markov (Ising ou Potts par exemple) et favorise les états où les pixels voisins ont des états similaires.

6.1.2 Formulation énergétique

Comme travailler directement avec les probabilités peut s'avérer difficile, On préfère reformuler le problème en termes d'énergie, en prenant le négatif du logarithme de la densité de probabilité (ce qui revient à minimiser une énergie plutôt qu'à maximiser une probabilité).

En partant de :

$$\hat{x}_{MAP} = \arg\max_x \ \exp(-U(x \mid y))$$

Avec $U(x \mid y) telque$:

$$U(x \mid y) = -\ln P(Y = y \mid X = x) - \ln P(X = x)$$

L'estimateur MAP devient alors :

$$\hat{x}_{MAP} = \arg\min_{x} \; U(x \mid y)$$

Cette énergie $U(x \mid y)$ se décompose en deux termes :

$$U(x \mid y) = \underbrace{\sum_{s \in S} \frac{(x_s - y_s)^2}{2\sigma^2}}_{\text{Attache aux données}} + \underbrace{\beta \sum_{\{s,t\} \in \mathcal{C}_2} \phi(x_s, x_t)}_{\text{Régularisation}}$$

Plus précisément :

— Le premier terme mesure l'écart entre chaque pixel x_s de l'image reconstruite et l'observation bruitée y_s. Il repose sur l'hypothèse que l'image observée est le résultat de l'image idéale perturbée par un bruit gaussien de moyenne nulle et de variance σ_b^2.

Dans l'expression de l'énergie $U(x \mid y)$, la variance σ^2 intervient au dénominateur, cela signifie que plus elle est petite, plus l'attache aux données est forte, donc on considère que l'image restaurée x_s colle fortement à l'image bruitée y_s. Tandis qu'avec une plus grande valeur de σ^2, on donne plus d'importance à la régularité.

— Le second terme représente les interactions locales entre pixels voisins, c'est-à-dire la régularisation. Ce terme favorise les images où les pixels voisins prennent des valeurs similaires, ce qui permet d'éviter les solutions trop bruitées ou trop irrégulières.

Cette régularisation est modélisée à l'aide d'un champ de Markov, et plus précisément par une loi de Gibbs, dont la distribution a priori s'écrit formellement :

$$P(X = x) = \frac{1}{Z} \exp\left(-\beta \sum_{\{s,t\} \in \mathcal{C}_2} \phi(x_s, x_t) \right)$$

où Z est la constante de normalisation (fonction de partition), en général impossible à calculer car elle implique une somme sur toutes les configurations possibles. C'est pourquoi on se contente souvent d'exprimer cette distribution à une constante près, en écrivant :

$$P(X = x) \propto \exp\left(-\beta \sum_{\{s,t\} \in \mathcal{C}_2} \phi(x_s, x_t) \right)$$

Avec $\propto$ qui signifie propositionnel.

Le choix de la fonction ϕ détermine la nature des interactions entre pixels voisins. Par exemple, $\phi(x_s, x_t) = (x_s - x_t)^2$ favorise la continuité en pénalisant les grandes différences de valeur entre pixels. Dans notre cas, nous utilisons le modèle de Potts, où $\phi(x_s, x_t) = \mathbb{1}_{x_s \neq x_t}$, c'est-à-dire que seules les discontinuités entre pixels différents sont pénalisées, sans tenir compte de l'amplitude de la différence. Ce modèle encourage donc la formation de régions homogènes tout en permettant la préservation des frontières nettes entre zones distinctes.

Le paramètre $\beta > 0$ est un coefficient de pondération qui équilibre la fidélité aux données et la régularité. Une petite valeur de β laisse plus de liberté à l'image pour suivre les données, au risque de conserver le bruit. Une plus grande valeur de β force une image plus lisse, au risque de lisser les détails fins. De plus, choisir un $\beta > 0$ pénalise

les différences entre pixels (c'est ce que l'on veut faire), tandis qu'une valeur négative de β aura tendance à faire l'inverse, c'est-à-dire récompenser les différences, ce qui crée des motifs parfois intéressants (mais qui n'ont plus rien à voir avec l'image de base).

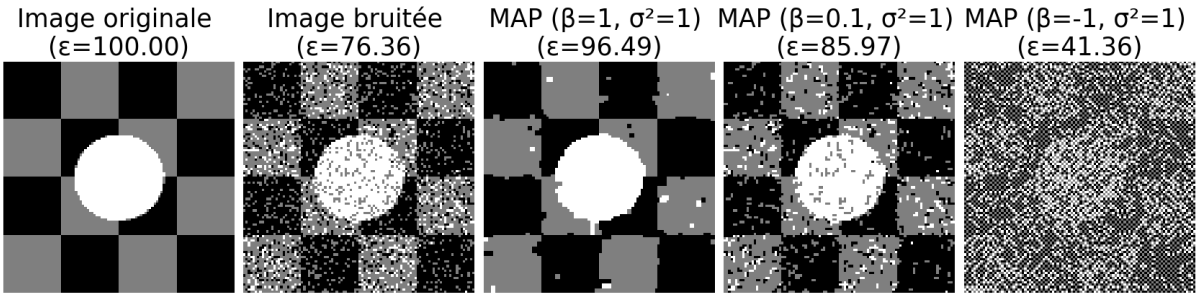


FIGURE 24 – Illustration d'une configuration comparant différentes valeurs de β pour un estimateur MAP, bruit : $(\sigma_b^2 = 0.5)$, nombre d'itérations : $(iter = 10^5)$ et nombre d'états : $(nbetats = 3)$

On voit bien que paramétrer une grande valeur de β a eu pour conséquence de favoriser la régularisation, donc pour 100 000 itérations on obtient une image presque uniforme qui ressemble à l'image initiale. Les carrés gris on été bien délimités par l'estimateur avec une petite valeur de β, mais il reste un nombre non négligeable de pixels qui n'ont pas été changés. L'algorithme fait confiance au bruit (il ignore presque le membre de droite), il préfère une solution fidèle à l'image bruitée, même si elle est incohérente spatiallement. Le résultat d'un estimateur lorsqu'on lui donne pour mission de créer le plus de différences entre les pixels, donc en mettant une valeur de beta négative tend à créer des bruits Gaussiens.

6.1.3 Intérêt de l'estimateur MAP

L'estimateur MAP est particulièrement adapté aux problèmes de restauration d'images car :

— Il offre une solution globale cohérente, en prenant en compte à la fois les données et des contraintes de régularité.

— Il est robuste au bruit, notamment dans les situations où les observations sont fortement dégradées.

— Il se prête bien à des techniques d'optimisation efficaces, qu'elles soient stochastiques (comme le recuit simulé) ou déterministes (comme l'algorithme ICM ou les méthodes de gradient).

Toutefois, comme la fonction d'énergie $U(x \mid y)$ est généralement non convexe, la recherche du minimum global est complexe. Des algorithmes comme le recuit simulé permettent de s'approcher de ce minimum en explorant intelligemment l'espace des solutions, tandis que des méthodes plus rapides comme l'agorithme ICM convergent vers un minimum local, ce qui peut suffire dans certains cas.

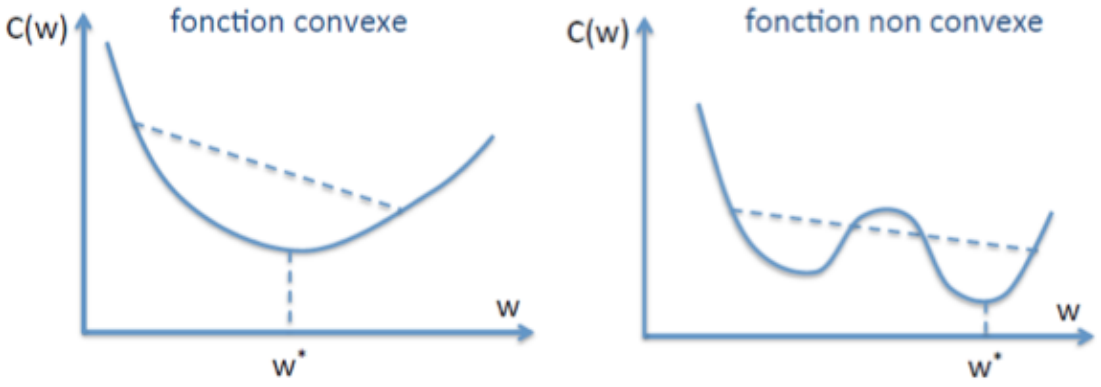


FIGURE 25 – Rappel : Une fonction convexe admet un unique minimum, il est donc global. Pour une fonction non convexe il est plus délicat de déterminer le minimum global, il faut éviter les minima locaux

6.1.4 Application de l'estimateur MAP

Nous présentons ici deux façons de simuler un estimateur MAP : une approche déterministe par descente locale (type ICM), et une approche stochastique par recuit simulé. Ils recherchent tous deux la configuration la plus probable (la moins énergétique).

L'algorithme ICM met à jour les pixels un par un. Dans notre cas, à chaque itération, un pixel s_{ij} est sélectionné aléatoirement (légère variation de la version canonique). Le reste de l'algorithme reste le même.

Le recuit simulé introduit une température décroissante dans le processus, ce qui permet d'accepter temporairement des solutions sous-optimales pour mieux explorer l'espace des configurations.

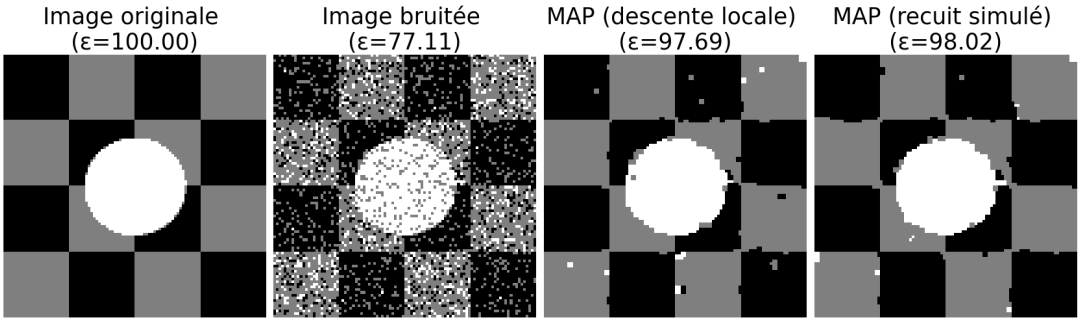


FIGURE 26 – Illustration de deux méthodes d'optimisation pour l'estimateur MAP, bruit : $(\sigma_b^2 = 0.5)$, nombre d'itérations : $(iter = 10^5)$, nombre d'états : $(nbetats = 3)$, valeur des paramètres : $(\sigma^2 = 2,\ \beta = 0.3)$

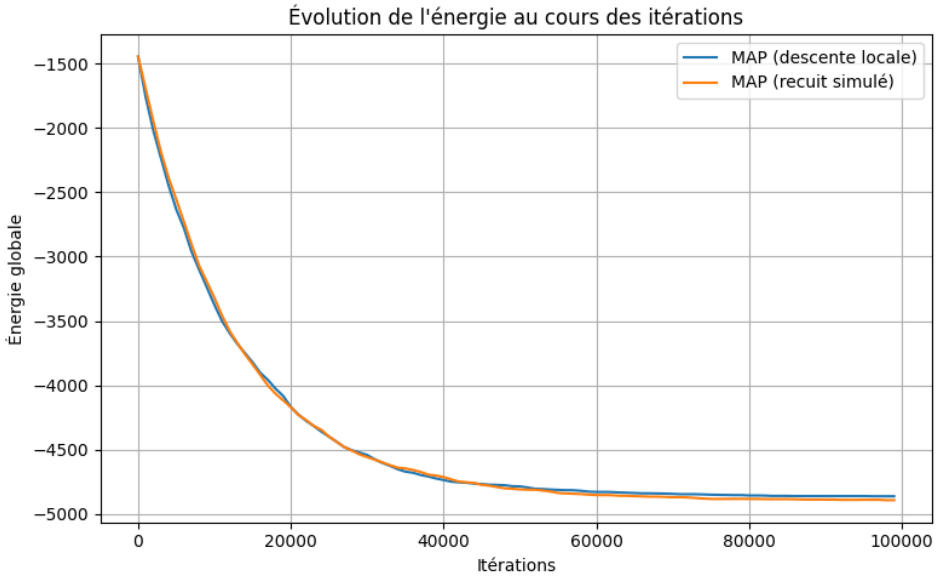


FIGURE 27 – On observe une évolution similaire de l'énergie pour les simulations d'un estimateur MAP par ICM et recuit simulé.

6.2 Estimateur MPM pour la restauration d'images

6.2.1 Définition de l'estimateur MPM

L'estimateur MPM (Maximum Posterior Marginal) adopte une stratégie locale : au lieu de chercher l'image entière la plus probable (comme le fait l'estimateur MAP), il cherche, pour chaque pixel s, la valeur de x_s la plus probable compte tenu de l'image observée y. Cela revient à maximiser la marginale de la loi a posteriori :

$$\hat{x}_{MPM}(s) = \arg\max_{x_s} \; P(X_s = x_s \mid Y = y)$$

Cet estimateur vise à minimiser le nombre moyen de pixels mal restaurés, ce qui le rend particulièrement pertinent lorsque l'on souhaite réduire localement les erreurs, même si la cohérence globale de l'image n'est pas garantie.

6.2.2 Méthode de calcul

Le calcul analytique exact de $P(X_s = x_s \mid Y = y)$ est en général inenvisageable, car il nécessiterait de sommer sur toutes les configurations possibles de l'image. En pratique, on utilise des techniques d'échantillonnage comme l'algorithme de Gibbs pour générer un ensemble de configurations $x^{(1)}, \ldots, x^{(N)}$ distribuées selon la loi a posteriori.

$$P(X_s = x_s \mid Y = y) \approx \frac{1}{N} \sum_{k=1}^{N} \mathbf{1}_{\{x_s^{(k)} = x_s\}}$$

On sélectionne ensuite, pour chaque pixel s, la valeur qui apparaît le plus souvent :

$$\hat{x}_{MPM}(s) = \arg\max_{x_s} \; \text{Fréquence}(x_s)$$

6.2.3 Intérêt de l'estimateur MPM

L'estimateur MPM présente plusieurs avantages :

— Il est moins sensible à l'initialisation que le MAP, car il s'appuie sur des moyennes statistiques issues de multiples échantillons.

— Il est robuste au bruit local et bien adapté lorsque plusieurs solutions globales sont envisageables (situation multimodale).

— Il fournit une estimation pixel par pixel, ce qui peut être utile dans certaines applications où la précision locale est primordiale.

En revanche, comme il ne tient pas compte explicitement des interactions entre pixels dans la décision finale, il peut produire une image moins cohérente globalement, avec un effet « flou » ou bruité sur certaines zones.

6.2.4 Application de l'estimateur MPM

Pour simuler un estimateur MPM, il nous faut tout d'abord générer une distribution de Gibbs. Chaque configuration représente un état possible du champ aléatoire, avec la valeur des pixels suivant la distribution de Gibbs. Voici un exemple de cette distribution avec les configurations $x^{(1)}, x^{(2)}, x^{(3)}$:

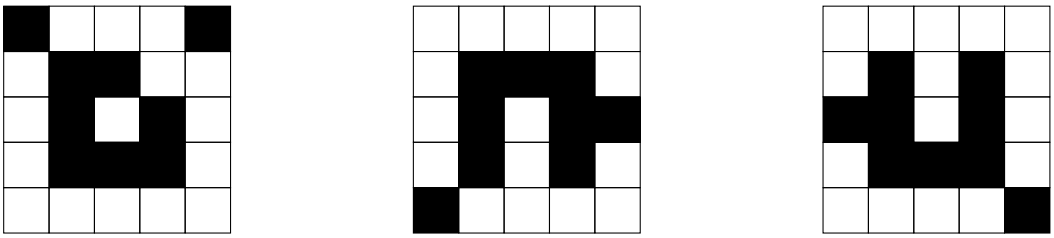


FIGURE 28 – Représentation de gauche à droite des configurations $x^{(1)}, x^{(2)}, x^{(3)}$ issues d'un échantillon de Gibbs

Nous nous intéressons maintenant au pixel s_{00} en haut à gauche. Les valeurs du pixel s_{00} dans ces configurations sont :

— $x_{00}^{(1)} = 1$

— $x_{00}^{(2)} = 1$

— $x_{00}^{(3)} = 0$

Nous devons à présent déterminer la fréquence de chaque état du pixel s_{00} dans les configurations $x^{(1)}, x^{(2)}, x^{(3)}$:

— Fréquence(1) = 2 (car 1 apparaît deux fois).

— Fréquence(0) = 1 (car 0 apparaît une fois).

L'estimateur MPM choisit l'état qui a la fréquence la plus élevée. Dans ce cas, l'état 1 apparaît le plus souvent, donc :

$$\hat{x}_{MPM}(s_{00}) = \arg\max_{x_s} \{\text{Fréquence}(1), \text{Fréquence}(0)\} = \arg\max\{2, 1\} = 1$$

Ainsi, le pixel en haut à gauche apparaît blanc dans l'image finale générée par notre estimateur. Ce processus est appliqué de manière similaire à chaque pixel, ce qui aboutit à une image d'un carré nettoyée du bruit.

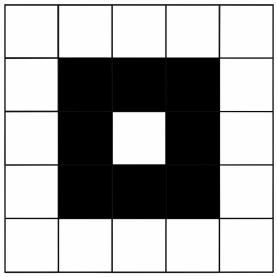


FIGURE 29 – Image restaurée par l'estimateur MPM

6.3 Estimateur TPM pour la restauration d'images

6.3.1 Définition de l'estimateur TPM

L'estimateur TPM (Thresholded Posterior Mean) est basé sur l'espérance conditionnelle de chaque pixel X_s connaissant l'image observée y. Contrairement aux estimateurs MAP et MPM qui choisissent un état précis, l'estimateur TPM calcule une moyenne pondérée des valeurs possibles :

$$\hat{x}_{TPM}(s) = \mathbb{E}[X_s \mid Y = y]$$

Lorsque les états sont discrets (par exemple $\{0, 1\}$ ou un nombre fini de classes), on applique souvent un seuillage (ou un arrondi) pour obtenir une image utilisable :

$$\hat{x}_{TPM}(s) = \lfloor \mathbb{E}[X_s \mid Y = y] + 0.5 \rfloor$$

6.3.2 Méthode de calcul

Comme pour l'estimateur MPM, cette espérance est difficile à calculer directement. On l'approche par une moyenne empirique sur des échantillons obtenus via un algorithme de Monte-Carlo (Gibbs ou Metropolis-Hastings) :

$$\mathbb{E}[X_s \mid Y = y] \approx \frac{1}{N} \sum_{k=1}^{N} x_s^{(k)}$$

Le seuillage final permet de transformer cette estimation continue en une image discrète (ou binaire), si nécessaire.

6.3.3 Intérêt de l'estimateur TPM

L'estimateur TPM est le plus adapté lorsque le critère d'évaluation est l'erreur quadratique moyenne. Il possède les qualités suivantes :

— Il lisse naturellement l'image : les zones bruitées sont moyennées, ce qui atténue le bruit aléatoire.

— Il est bien adapté aux images à niveaux de gris et aux problèmes où les transitions douces sont préférées.

— Il est facile à calculer une fois les échantillons générés.

En revanche, ce lissage peut être un inconvénient dans des contextes où les discontinuités (contours nets) sont importantes, comme en segmentation. L'estimateur TPM tend alors à produire des images floues ou intermédiaires, notamment lorsque les classes sont mal séparées.

6.3.4 Application de l'estimateur TPM

Comme pour MPM, pour simuler un estimateur TPM, il nous faut générer un échantillon de configurations à partir de la distribution a posteriori via un algorithme de Monte-Carlo, comme Gibbs ou Metropolis-Hastings. Mais cette foi-ci, les pixels sont estimés en calculant la moyenne de leurs valeurs dans toutes les configurations échantillonnées.

Prenons l'exemple du pixel s_{00} en haut à gauche, dont les valeurs dans les configurations $x^{(1)}, x^{(2)}, x^{(3)}$ sont :

— $x_{00}^{(1)} = 1$
— $x_{00}^{(2)} = 0$
— $x_{00}^{(3)} = 1$

Nous allons maintenant calculer l'espérance de s_{00} :

$$\mathbb{E}[X_{s_{00}} \mid Y = y] \approx \frac{1}{3}(1 + 0 + 1) = \frac{2}{3}$$

Cette moyenne nous donne une estimation continue du pixel. Cependant, pour générer une image binaire, nous appliquons un seuil pour obtenir une valeur discrète, par exemple :

$$\hat{x}_{TPM}(s_{00}) = \left\lfloor \frac{2}{3} + 0.5 \right\rfloor = 1$$

Ainsi, pour le pixel s_{00}, l'estimateur TPM choisit la valeur 1. Ce processus est répété pour chaque pixel de l'image. Dans le cas de face, et bien souvent on obtient des résultats similaires a ceux obtenus avec un estimateur MPM.

6.4 Pré-traitement de l'image

Avant d'appliquer les différents estimateurs, nous avons conçu une fonction pour gérer les niveaux de gris dans l'image. Chaque pixel de l'image peut prendre une valeur parmi un ensemble fini d'états λ_i, où $Card(E)$ représente le nombre d'états possibles. Pour une image en noir et blanc, il y a généralement 256 nuances de gris possibles (sur 8 bits), bien que l'œil humain ne soit capable d'en distinguer qu'environ une trentaine. En fonction des besoins de notre expérience, nous pouvons restreindre ce nombre d'états en réduisant la plage de valeurs possibles des pixels, passant ainsi de $Card(E) = 256$ à un $Card(E)$ plus petit, par exemple $Card(E) = 16$ ou $Card(E) = 2$, selon nos besoins spécifiques.

6.5 Conclusion sur les estimateurs

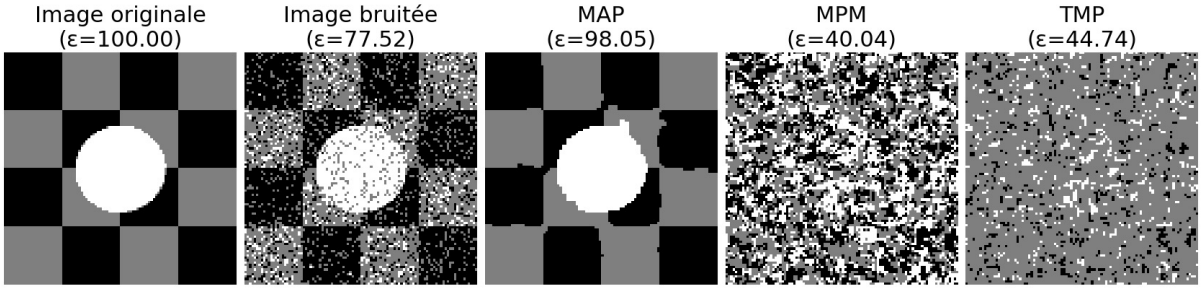


FIGURE 30 – Illustration des différents estimateurs vus dans cette section, bruit : ($\sigma_b^2 = 0.5$), nombre d'itérations : ($iter = 2 \times 10^5$) et nombre d'états : ($nbetats = 3$), valeur des paramètres : ($\sigma^2 = 4$, $\beta = 0.3$)

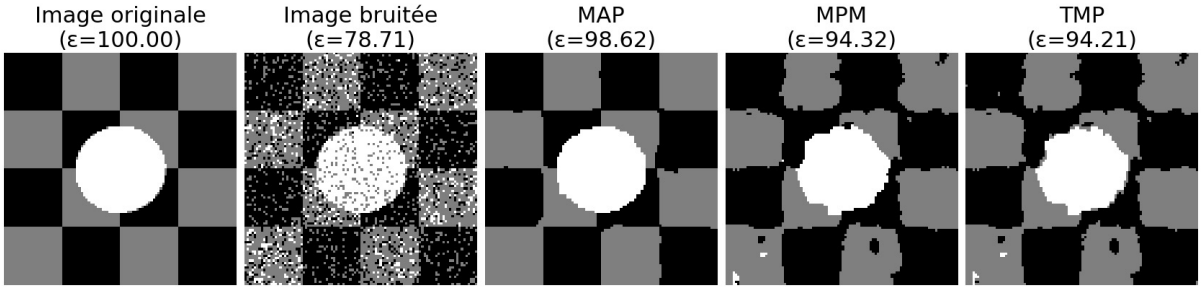


FIGURE 31 – Même configuration que la figure précédente, mais avec $\beta = 0.8$.

L'augmentation de β intensifie la régularisation spatiale, ce qui favorise la cohérence locale entre pixels voisins. Cela est particulièrement efficace pour des images comportant de grands aplats de couleur. Dans ce contexte, les estimateurs MPM et TPM montrent leur capacité à lisser efficacement l'image, conduisant à des résultats proches de ceux du MAP, avec un taux de restauration comparable.

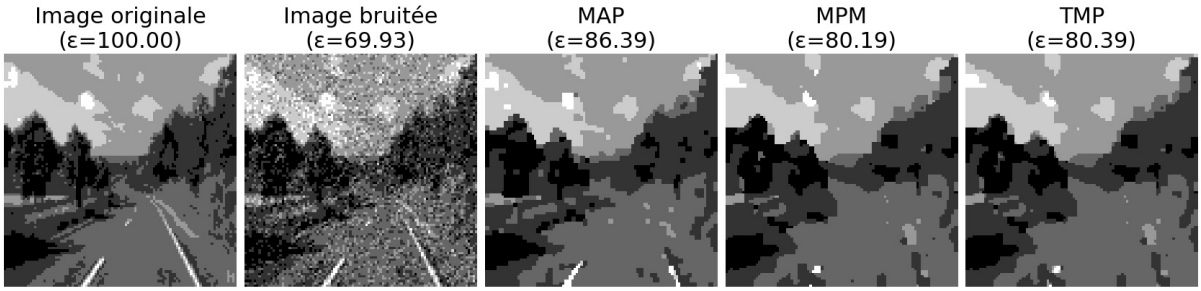


FIGURE 32 – Illustration des différents estimateurs vus dans cette section, bruit : ($\sigma_b^2 = 0.5$), nombre d'itérations : ($iter = 2 \times 10^5$) et nombre d'états : ($nbetats = 6$), valeur des paramètres : ($\sigma^2 = 0.8$, $\beta = 2$)

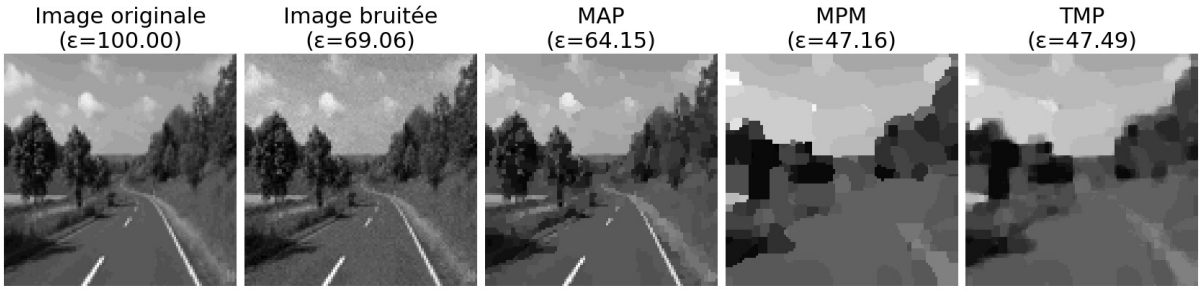


FIGURE 33 – Même image avec 6 fois plus d'itérations.

Dans ce second cas, l'image contient davantage de détails. Un σ^2 faible est donc préférable pour préserver la précision de l'information d'observation. Toutefois, pour les estimateurs MPM et TPM, basés sur des moyennes empiriques, un trop grand nombre d'itérations peut nuire à la qualité visuelle. En effet, ces moyennes ont tendance à lisser excessivement et peuvent faire disparaître des structures fines.

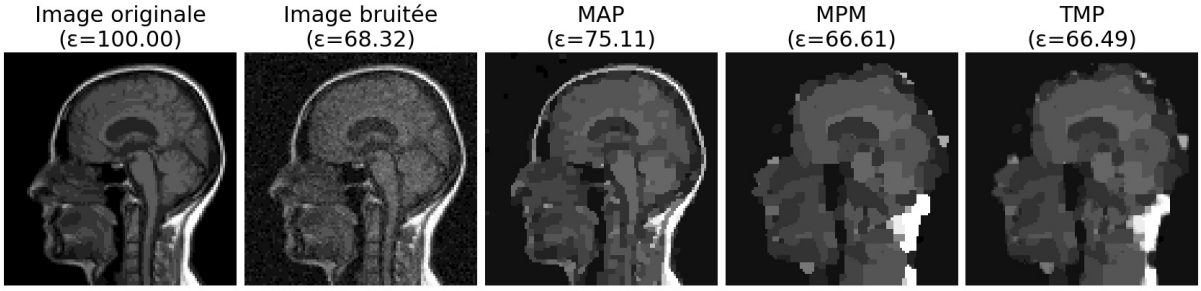


FIGURE 34 – Illustration des différents estimateurs vus dans cette section, bruit : ($\sigma_b^2 = 0.5$), nombre d'itérations : ($iter = 2 \times 10^5$) et nombre d'états : ($nbetats = 16$), valeur des paramètres : ($\sigma^2 = 2$, $\beta = 1$)

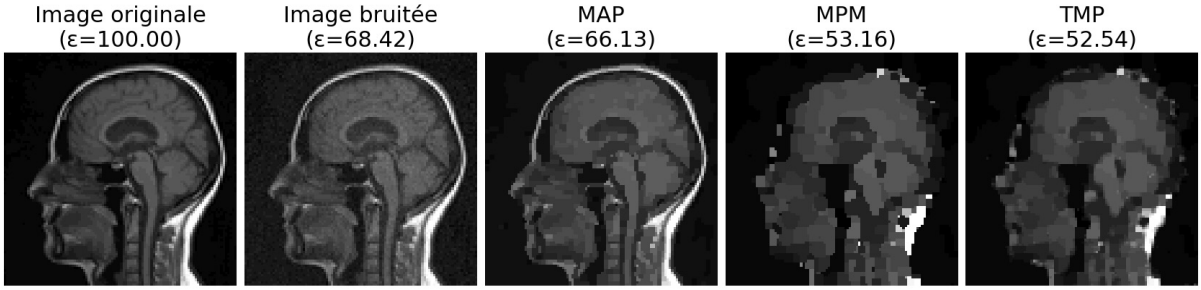


FIGURE 35 – Même configuration mais avec 32 états.

L'augmentation du nombre d'états possibles complexifie le problème. Pour MAP, si le nombre d'itérations est insuffisant, le modèle reste bloqué dans un minimum local, ce qui peut se traduire par des zones mal reconstruites (pixels blancs résiduels). En parallèle, les estimateurs MPM et surtout TPM produisent des images plus floues, du fait de l'effet moyen sur les nombreuses classes.

En résumé, les trois estimateurs sont performants, mais leurs comportements diffèrent selon le contexte :

— MAP privilégie la cohérence globale. Il offre un bon compromis biais/variance.

— MPM donne des décisions locales fiables, robustes au bruit, mais sans garantie de cohérence globale.

— TPM est très efficace pour lisser, mais peut dégrader les détails importants si le nombre d'états est grand, menant à des images floues.

En appliquant l'estimateur MPM à des configurations issues d'un estimateur MAP, on arrive a des restaurations presque parfaites.

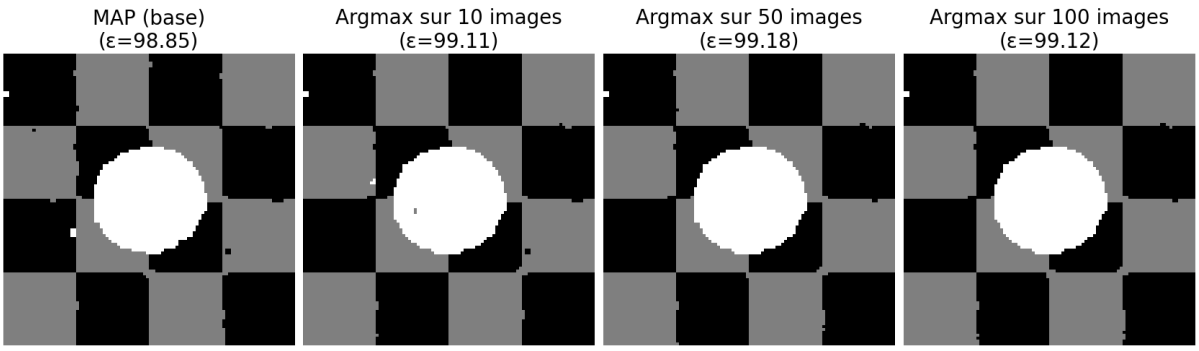


FIGURE 36 – Expérimentation sur les estimateurs

7 Conclusion générale

Ainsi, nous avons pu aborder dans ce travail le cadre théorique du formalisme markovien appliqué aux champs de Markov. Les outils de descriptions des voisinages aux sites d'une image nous ont permis d'introduire les champs de Gibbs et d'établir leur équivalence avec les champs de Markov que nous étudions. Nous avons appliqué les algorithmes issus des méthodes de Monte-Carlo par J. W. Gibbs et N. Metropolis au traitement d'images afin de restaurer des images bruitées. Plusieurs paramètres (β, T, etc.) et méthodes sont entrés en jeu, du modèle d'Ising pour les images en noir et blanc aux modèles de Potts et markovien gaussien pour les images en nuances de gris, afin de trouver une configuration d'énergie minimale du champ, et donc une bonne estimation de l'image originelle. Enfin, nous avons abordé différents estimateurs bayésiens (MAP, MPM, TPM).

L'étude théorique du modèle markovien souligne l'importance cruciale du choix de l'approche de minimisation de l'énergie ainsi que de l'algorithme d'échantillonnage. Dans le cadre de notre travail, nous avons pu identifier des limites à ces méthodes d'échantillonnage. Nous avons été confrontés à la problématique du calibrage de divers paramètres, notamment α pour l'attache aux données, et β qui agit sur la régularisation dans nos modèles. Notre démarche a consisté à rechercher les paramètres optimaux afin d'obtenir les meilleurs résultats visuels, ce qui a nécessité un grand nombre d'essais impliquant des modifications successives des paramètres. Certains essais reposant sur un très grand nombre d'itérations, le temps de calcul s'est avéré particulièrement long pour les algorithmes utilisant l'échantillonneur de Gibbs. Une fois les paramètres sélectionnés, les réalisations du champ de Markov obtenues, bien que visuellement satisfaisantes, ont pu présenter un taux de restauration insuffisant. Le cas markovien gaussien, avec 255 niveaux de gris, illustre parfaitement ce défi : il y a deux paramètres à estimer, et les taux de restauration sont inférieurs à ceux de la version bruitée. Ce modèle met en lumière la complexité du sujet. Concrètement, bien que l'image soit dégradée par la méthode, elle conserve visuellement son sens.

Par ailleurs, l'hypothèse gibbsienne présente également des limites. En ne considérant que le voisinage d'un site à chaque itération, il n'est pas possible de transmettre l'intégralité du sens de l'image ; le tout n'est pas simplement la somme des parties. Mais en considérant l'énergie globale de l'image, et donc en la calculant sur l'ensemble des sites, le temps de calcul explose. C'est dans ce sens qu'il convient d'interpréter les méthodes utilisées pour certains estimateurs, en maintenant une attache aux données initiales afin de ne pas trop s'éloigner de l'image de base. Des modèles plus complexes, tels que les Champs de Markov Cachés, semblent mieux adaptés pour capturer des motifs plus élaborés, comme des bords ou des textures particulières. C'est notamment le cas pour le motif zébré, que nous n'avons donc pas pu reproduire. Nous avons également rencontré des exemples impliquant des Chaînes de Markov Couple ou Triplet. Cependant, nous avons choisi de ne pas les aborder dans ce rapport en raison de leur complexité.

Comme mentionné précédemment, ce rapport présente des algorithmes moins « énergivores » que les modèles récents basés sur le machine learning. Il est apparu que, pour certains exemples, nous avons dû effectuer des calculs pendant plusieurs heures. Un objectif futur serait de comparer rigoureusement les apports de telles méthodes sur le plan écologique et de réfléchir à des algorithmes plus efficaces.

C'est dans cette perspective que nous avons également étudié le rôle des estimateurs statistiques dans un cadre markovien. En effet, l'obtention d'une configuration d'énergie minimale n'est pas toujours suffisante, il est parfois pertinent d'exploiter plusieurs réalisations ou plusieurs critères de décision pour affiner la restauration. Nous avons ainsi analysé trois estimateurs bayésiens majeurs : le MAP, le MPM et le TPM. Chacun propose une stratégie différente pour extraire une estimation à partir de la distribution a posteriori. Cela nous a donc ammené à combiner ces estimateurs, afin d'obtenir de meilleurs résultats, ce qui a fonctionné dans une certaine mesure. En appliquant l'estimateur MPM à des configurations issues de plusieurs estimateurs MAP (qui est le modèle le plus robuste parmi ceux que nous avons étudié), plutôt qu'à des configurations successives d'un même échantillon de Gibbs, nous sommes arrivés a obtenir les meilleures restaurations, avec un taux des restauratiton atteignant 99.18. Mais il faudrait approfondir d'avantage cette méthode pour qu'elle soit réellement efficace, ce que nous n'avons pas eu le temps de faire ici. On peut voir qu'en appliquant l'estimateur MPM à 50 images, on obtient un meilleur résultat que sur un plus grand nombre d'images ici 100, ce qui semble contre intuitif.

En accord avec la littérature, nous avons choisi de n'appliquer que des bruits gaussiens dans nos essais. Dans nos recherches, nous avons également appliqué nos algorithmes à des images dégradées par bruit uniforme. Autrement dit, pour bruiter l'image, nous tirons sur l'ensemble des états E, avec pour chaque états une probabilité $\frac{1}{Card(E)}$ d'être tirés, et on accepte le bruitage avec une probabilité choisie. Aussi, un bruit uniforme permettrait de mieux tester la qualité des méthodes et modèles. En guise d'exemple, voici un de nos essais avec un bruit uniforme :

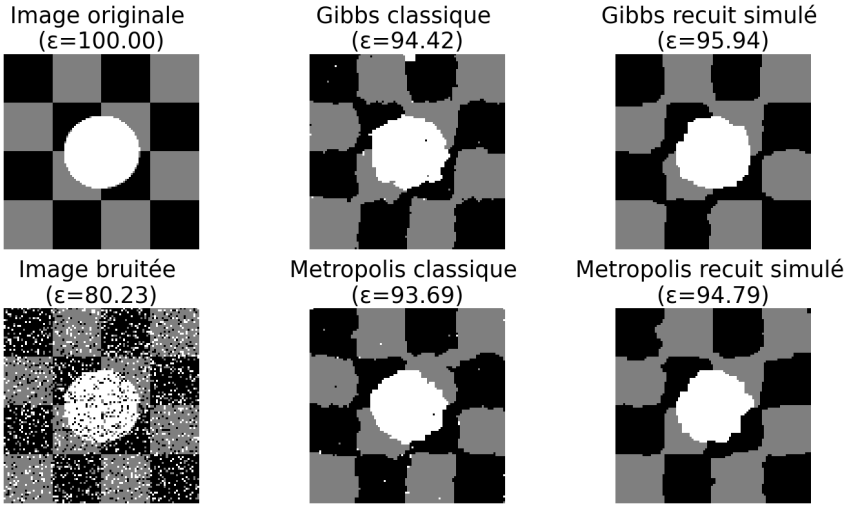


FIGURE 37 – Illustration d'une configuration comparant différentes simulations selon le modèle de Potts avec et sans recuit simulé, bruit : ($p = 0.3$), nombre d'itérations : ($iter = 2 \times 10^5$), nombre d'états : ($nbetats = 3$), valeur des paramètres : ($\beta = 1$)

Bibliographie

— Vincent Barra, *Modélisation markovienne en imagerie*, Institut Supérieur d'Informatique, de Modélisation et de leurs Applications, Campus des Cézeaux, Master MSI, Année universitaire 2005/2006

— Julian Besag, *Spatial Interaction and the Statistical Analysis of Lattice Systems*, Journal of the Royal Statistical Society. Series B (Methodological), Vol. 36, No. 2 (1974), pp. 192-236

— I. Bloch, Y. Gousseau, H. Maître, D. Matignon, B. Pesquet-Popescu, F. Schmitt, M. Sigelle, F. Tupin, *Le traitement des images*, Département TSI - Télécom Paris, 2004.

— Stuart Geman and Donald Geman, *Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 1984.

— Arnaud Guyader, *Simulations Monte-Carlo*, Université Pierre et Marie Curie, Master Mathématiques et Applications, Spécialité Statistique, Année 2016/2017

— W. K. Hastings, *Monte Carlo sampling methods using Markov chains and their applications*, University of Toronto, Biometrika, Vol. 57, No. 1 (Apr., 1970), pp. 97-109

— Patrick Héas, *Méthodes de Simulation de Monte-Carlo en Analyse d'Images*, Université de Rennes 1, Master SISEA, 2015

— M. Karpe, A. Sadaca, N. Soussi, C. Zeng, *Restauration Markovienne d'Images*, Projet d'Initiation à la Recherche sous la direction de A. Maruani, Ecole Nationale des Ponts et Chaussées, 2017

— S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, *Optimization by Simulated Annealing*, Science, New Series, Vol. 220, No. 4598, May 13, 1983, pp. 671–680.

— Junxiang Yang, Junseok Kim, *Computer simulation of the nonhomogeneous zebra pattern formation using a mathematical model with space-dependent parameters*, Department of Mathematics, Korea University, Seoul 02841, Republic of Korea, April 2023

— Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, Edward Teller, *Equation of State Calculations by Fast Computing Machines*, Journal of Chemical Physics, 1953.

— Wojciech Pieczynski, *Modèles de Markov en traitements d'images*, Institut national des télécommunications, Département CITI, CNRS, Traitement du Signal, Vol. 20, No. 3, pp. 255-278, 2003

— Jérôme Poix, *Processus : Notes de cours*, Licence MIASHS 3, 2025

— Marc Sigelle, Florence Turpin, *Champs de Markov en Traitement d'Image*, Module C3M, Département Traitement de Signal et des Images, 1999

— Alix Yan, Laurent Mugnier, Jean-François Giovannelli, Romain Fétick, Cyril Petit, *Restauration d'images astronomiques corrigées par optique adaptative : méthode marginale étendue par algorithme MCMC*, GRETSI 2022, Nancy, France. HAL : ffhal-03837136ff.

ANNEXE 1

```python
# === Application des algorithmes de Gibbs et Metropolis-Hastings pour le modèle d'Ising (champ à deux états, noir et blanc) ===

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from tqdm import tqdm

# === Chargement et quantification de l'image ===
def charger_image_grayscale(path, nb_etats):
    img = Image.open(path).convert("L").resize((100, 100))
    img_np = np.array(img)
    img_bin = np.floor(img_np / (256 / nb_etats)).astype(int)  # 0 ou 1
    return 2 * img_bin - 1  # transforme en -1 ou +1

# === Ajout de bruit : simulateur du modèle d'observation P(Y|X) ===
def ajouter_bruit_uniforme(champ, p):
    bruit = np.random.choice([-1, 1], size=champ.shape)
    masque = np.random.rand(*champ.shape) < p
    return np.where(masque, bruit, champ)

def ajouter_bruit_gaussien_discret(champ, sigma):
    bruit = np.random.normal(0, sigma, size=champ.shape)
    champ_bruite = np.sign(champ + bruit)  # Garde -1 ou +1
    champ_bruite[champ_bruite == 0] = 1    # Remplace 0 par +1 (par convention)
    return champ_bruite

# === Calcul de l'énergie locale ===
def energie_locale_ising(i, j, H, L, champ, etat, beta, B):
    voisins = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    somme_voisins = 0
    for dx, dy in voisins:
        ni, nj = i + dx, j + dy
        if 0 <= ni < H and 0 <= nj < L:
            somme_voisins += champ[ni, nj] * etat
    energie = -beta * somme_voisins - B * etat
    return energie

# === Algorithme de Gibbs pour le modèle d'Ising ===
def gibbs_ising(champ, nb_iter, beta, B):
    H, L = champ.shape
    for _ in tqdm(range(nb_iter), desc="Gibbs Ising"):
        i, j = np.random.randint(0, H), np.random.randint(0, L)
        energies = []
        for s in [-1, 1]:
            e = energie_locale_ising(i, j, H, L, champ, s, beta, B)
            energies.append(np.exp(-e))
        proba = energies / np.sum(energies)
        champ[i, j] = np.random.choice([-1, 1], p=proba)
    return champ
```

```python
# === Algorithme de Metropolis-Hastings pour le modèle d'Ising ===
def metropolis_ising(champ, nb_iter, beta, B):
    H, L = champ.shape
    for _ in tqdm(range(nb_iter), desc="Metropolis Ising"):
        i, j = np.random.randint(0, H), np.random.randint(0, L)
        etat_actuel = champ[i, j]
        etat_nouveau = -etat_actuel
        E_actuel = energie_locale_ising(i, j, H, L, champ, etat_actuel, beta, B)
        E_nouveau = energie_locale_ising(i, j, H, L, champ, etat_nouveau, beta, B)
        delta_E = E_nouveau - E_actuel
        if delta_E < 0 or np.random.rand() < np.exp(-delta_E):
            champ[i, j] = etat_nouveau
    return champ

# === Visualisation ===
def afficher_resultats(img_init, img_bruitee, gibbs, metro, nb_etats):
    def to_image(img):
        return (img * (255 / (nb_etats - 1))).astype(np.uint8)

    # Calcul du taux de restauration pour les deux estimateurs
    taux = taux_restauration(img_init, img_init)
    taux_base = taux_restauration(img_init, img_bruitee)
    taux_gibbs = taux_restauration(img_init, gibbs)
    taux_metro = taux_restauration(img_init, metro)

    imgs = [img_init, gibbs, img_bruitee, metro]
    titres = [
        f"Image originale \n(ε={taux:.2f})", f"Gibbs classique \n(ε={taux_gibbs:.2f})",
        f"Image bruitée \n(ε={taux_base:.2f})", f"Metropolis classique
\n(ε={taux_metro:.2f})"
    ]

    fig, axs = plt.subplots(2, 2, figsize=(10, 8))
    axs = axs.flatten()

    for i, (ax, titre, img) in enumerate(zip(axs, titres, imgs)):
        ax.imshow(to_image(img), cmap="gray")
        ax.set_title(titre, fontsize=25)
        ax.axis("off")

    plt.tight_layout()
    plt.show()

# === Calcul du taux de restauration de l'image ===
def taux_restauration(img_originale, img_restauree):
    pixels_corrects = np.sum(img_originale == img_restauree)
    total_pixels = img_originale.size
    taux = (pixels_corrects / total_pixels) * 100
    return taux

# === Paramètres ===
```

```
chemin_image = "images/souris.png"
sigma_bruit = 1
p_bruit = 0.3
nb_etats = 2
nb_iter = 100000
beta = 1
B = 0.0  # champ externe nul

# === Exécution ===
img = charger_image_grayscale(chemin_image, nb_etats)
img_bruitee = ajouter_bruit_gaussien_discret(img, sigma_bruit)
#img_bruitee = ajouter_bruit_uniforme(img, p_bruit)
champ_gibbs = img_bruitee.copy()
champ_metro = img_bruitee.copy()

champ_gibbs = gibbs_ising(champ_gibbs, nb_iter, beta, B)
champ_metro = metropolis_ising(champ_metro, nb_iter, beta, B)

# Affichage des résultats
afficher_resultats(img, img_bruitee, champ_gibbs, champ_metro, nb_etats)
```

ANNEXE 2

```
# === Application des algorithmes de Gibbs et Metropolis-Hastings pour le modèle de Potts
(nuances de gris) ===

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from tqdm import tqdm

# === Chargement et quantification de l'image ===
def charger_image_grayscale(path, nb_etats):
    img = Image.open(path).convert("L").resize((100, 100))
    img_np = np.array(img)
    return np.floor(img_np / (256 / nb_etats)).astype(int)

# === Ajout de bruit : simulateur du modèle d'observation P(Y|X) ===
def ajouter_bruit_uniforme(champ, p, nb_etats):
    bruit = np.random.choice(range(nb_etats), size=champ.shape)
    masque = np.random.rand(*champ.shape) < p
    return np.where(masque, bruit, champ)

def ajouter_bruit_gaussien_discret(champ, sigma, nb_etats):
    bruit = np.random.normal(0, sigma, size=champ.shape)
    champ_bruite = np.round(champ + bruit).astype(int)
    champ_bruite = np.clip(champ_bruite, 0, nb_etats - 1)
    return champ_bruite
```

```python
# === Calcul de l'énergie locale ===
def cdf_locale_4(i, j, H, L, champ, etat, poids_aretes, poids_sommets):
    voisins = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    energie = poids_sommets[etat]
    for dx, dy in voisins:
        ni, nj = i + dx, j + dy
        if 0 <= ni < H and 0 <= nj < L:
            energie += poids_aretes[(etat, champ[ni, nj])]
    return energie

# === Algorithme de Gibbs pour le modèle de Potts ===
def gibbs_classique(champ, nb_iter, modele):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="Gibbs classique (Potts)"):
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
        energies = np.array([
            cdf_locale_4(i, j, H, L, champ, s, modele['poids_aretes'],
modele['poids_sommets'])
            for s in range(modele['nb_etats'])
        ])
        proba = np.exp(-energies)
        proba /= np.sum(proba)
        champ[i, j] = np.random.choice(range(modele['nb_etats']), p=proba)
    return champ

# === Algorithme de Metropolis-Hastings pour le modèle de Potts ===
def metropolis_classique(champ, nb_iter, modele):
    H, L = champ.shape
    T = 1
    for k in tqdm(range(nb_iter), desc="Metropolis classique (Potts)"):
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
        etat_actuel = champ[i, j]
        nouvel_etat = np.random.choice([s for s in range(modele['nb_etats']) if s !=
etat_actuel])
        energie_actuelle = cdf_locale_4(i, j, H, L, champ, etat_actuel,
modele['poids_aretes'], modele['poids_sommets'])
        energie_nouvelle = cdf_locale_4(i, j, H, L, champ, nouvel_etat,
modele['poids_aretes'], modele['poids_sommets'])
        delta_E = energie_nouvelle - energie_actuelle
        if delta_E < 0 or np.random.rand() < np.exp(-delta_E / T):
            champ[i, j] = nouvel_etat
    return champ

# === Visualisation ===
def afficher_resultats(img_init, champ_gibbs, img_bruitee, champ_metro, nb_etats):
    def to_image(img):
        return (img * (255 / (nb_etats - 1))).astype(np.uint8)

    # Calcul du taux de restauration pour les deux estimateurs
    taux = taux_restauration(img_init, img_init)
```

```
    taux_base = taux_restauration(img_init, img_bruitee)
    taux_gibbs = taux_restauration(img_init, champ_gibbs)
    taux_metro = taux_restauration(img_init, champ_metro)

    imgs = [img_init, champ_gibbs, img_bruitee, champ_metro]
    titres = [
        f"Image originale \n(ε={taux:.2f})", f"Gibbs classique \n(ε={taux_gibbs:.2f})",
        f"Image bruitée \n(ε={taux_base:.2f})", f"Metropolis classique
\n(ε={taux_metro:.2f})"
    ]
    fig, axs = plt.subplots(2, 2, figsize=(10, 8))
    axs = axs.flatten()
    for ax, titre, img in zip(axs, titres, imgs):
        ax.imshow(to_image(img), cmap="gray")
        ax.set_title(titre, fontsize=25)
        ax.axis("off")
    plt.tight_layout()
    plt.show()

# === Calcul du taux de restauration de l'image ===
def taux_restauration(img_originale, img_restauree):
    pixels_corrects = np.sum(img_originale == img_restauree)
    total_pixels = img_originale.size
    taux = (pixels_corrects / total_pixels) * 100
    return taux

# === Paramètres ===
chemin_image = "images/test1.png"
sigma_bruit = 0.8
p_bruit = 0.5
nb_etats = 3
nb_iter = 200000
beta = 1

# === Exécution ===
img = charger_image_grayscale(chemin_image, nb_etats)
#img_bruitee = ajouter_bruit_uniforme(img, p_bruit, nb_etats)
img_bruitee = ajouter_bruit_gaussien_discret(img, sigma_bruit, nb_etats)
champ_gibbs = img_bruitee.copy()
champ_metro = img_bruitee.copy()

#poids_aretes = {(a, b): -1 if a == b else 1 for a in range(nb_etats) for b in
range(nb_etats)}
poids_aretes = {(a, b): beta * (-1 if a == b else 1) for a in range(nb_etats) for b in
range(nb_etats)}
poids_sommets = [0] * nb_etats
modele = {
    'nb_etats': nb_etats,
    'poids_aretes': poids_aretes,
    'poids_sommets': poids_sommets
}
```

```
champ_gibbs = gibbs_classique(champ_gibbs, nb_iter, modele)
champ_metro = metropolis_classique(champ_metro, nb_iter, modele)

# Affichage des résultats
afficher_resultats(img, champ_gibbs, img_bruitee, champ_metro, nb_etats)
```

ANNEXE 3

```
# === Comparaison des différents algorithmes pour le modèle d'Ising (champ à deux états,
noir et blanc) ===

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from tqdm import tqdm

# === Chargement et quantification de l'image ===
def charger_image_grayscale(path, nb_etats):
    img = Image.open(path).convert("L").resize((100, 100))
    img_np = np.array(img)
    return np.floor(img_np / (256 / nb_etats)).astype(int)

# === Ajout de bruit : simulateur du modèle d'observation P(Y|X) ===
def ajouter_bruit_uniforme(champ, p, nb_etats):
    bruit = np.random.choice(range(nb_etats), size=champ.shape)
    masque = np.random.rand(*champ.shape) < p
    return np.where(masque, bruit, champ)

def ajouter_bruit_gaussien_discret(champ, sigma, nb_etats):
    bruit = np.random.normal(0, sigma, size=champ.shape)
    champ_bruite = np.round(champ + bruit).astype(int)
    champ_bruite = np.clip(champ_bruite, 0, nb_etats - 1)
    return champ_bruite

# === Calcul de l'énergie locale ===
def cdf_locale_4(i, j, H, L, champ, etat, poids_aretes, poids_sommets):
    voisins = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    energie = poids_sommets[etat]
    for dx, dy in voisins:
        ni, nj = i + dx, j + dy
        if 0 <= ni < H and 0 <= nj < L:
            energie += poids_aretes[(etat, champ[ni, nj])]
    return energie

# === Algorithme de Gibbs pour le modèle d'Ising ===
def gibbs_classique(champ, nb_iter, modele):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="Gibbs classique"):
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
```

```
        energies = np.array([
            cdf_locale_4(i, j, H, L, champ, s, modele['poids_aretes'],
modele['poids_sommets'])
            for s in range(modele['nb_etats'])
        ])
        proba = np.exp(-energies)
        proba /= np.sum(proba)
        champ[i, j] = np.random.choice(range(modele['nb_etats']), p=proba)
    return champ

# === Algorithme de Gibbs pour le modèle d'Ising avec recuit simulé===
def gibbs_recuit(champ, nb_iter, modele):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="Gibbs recuit simulé"):
        T = 1 / np.log(2 + k)
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
        energies = np.array([
            cdf_locale_4(i, j, H, L, champ, s, modele['poids_aretes'],
modele['poids_sommets'])
            for s in range(modele['nb_etats'])
        ])
        proba = np.exp(-energies / T)
        proba /= np.sum(proba)
        champ[i, j] = np.random.choice(range(modele['nb_etats']), p=proba)
    return champ

# === Algorithme de Metropolis-Hastings pour le modèle d'Ising ===
def metropolis_classique(champ, nb_iter, modele):
    H, L = champ.shape
    T = 1  # température constante
    for k in tqdm(range(nb_iter), desc="Metropolis classique"):
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
        etat_actuel = champ[i, j]
        energie_actuelle = cdf_locale_4(i, j, H, L, champ, etat_actuel,
modele['poids_aretes'], modele['poids_sommets'])
        nouvel_etat = np.random.randint(0, modele['nb_etats'])
        if nouvel_etat == etat_actuel:
            continue
        energie_nouvelle = cdf_locale_4(i, j, H, L, champ, nouvel_etat,
modele['poids_aretes'], modele['poids_sommets'])
        delta_E = energie_nouvelle - energie_actuelle
        if delta_E < 0 or np.random.rand() < np.exp(-delta_E / T):
            champ[i, j] = nouvel_etat
    return champ

# === Algorithme de Metropolis-Hastings pour le modèle d'Ising avec recuit simulé ===
def metropolis_recuit(champ, nb_iter, modele):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="Metropolis recuit simulé"):
        T = 1 / np.log(2 + k)
```

```
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
        etat_actuel = champ[i, j]
        energie_actuelle = cdf_locale_4(i, j, H, L, champ, etat_actuel,
modele['poids_aretes'], modele['poids_sommets'])
        nouvel_etat = np.random.randint(0, modele['nb_etats'])
        if nouvel_etat == etat_actuel:
            continue
        energie_nouvelle = cdf_locale_4(i, j, H, L, champ, nouvel_etat,
modele['poids_aretes'], modele['poids_sommets'])
        delta_E = energie_nouvelle - energie_actuelle
        if delta_E < 0 or np.random.rand() < np.exp(-delta_E / T):
            champ[i, j] = nouvel_etat
    return champ

# === Visualisation ===
def afficher_resultats(img_init, img_bruitee, gibbs, gibbs_rec, metro, metro_rec, nb_etats):
    def to_image(img):
        return (img * (255 / (nb_etats - 1))).astype(np.uint8)

    # Calcul du taux de restauration pour les deux estimateurs
    taux = taux_restauration(img_init, img_init)
    taux_base = taux_restauration(img_init, img_bruitee)
    taux_gibbs = taux_restauration(img_init, champ_gibbs)
    taux_gibbs_rec = taux_restauration(img_init, champ_gibbs_rec)
    taux_metro = taux_restauration(img_init, champ_metro)
    taux_metro_rec = taux_restauration(img_init, champ_metro_rec)

    imgs = [img_init, gibbs, gibbs_rec, img_bruitee, metro, metro_rec]
    titres = [
        f"Image originale \n(ε={taux:.2f})", f"Gibbs classique \n(ε={taux_gibbs:.2f})",
        f"Gibbs recuit simulé \n(ε={taux_gibbs_rec:.2f})", f"Image bruitée
\n(ε={taux_base:.2f})",
        f"Metropolis classique \n(ε={taux_metro:.2f})", f"Metropolis recuit simulé
\n(ε={taux_metro_rec:.2f})"
    ]

    fig, axs = plt.subplots(2, 3, figsize=(15, 8))  # 2x3 = 6 images
    axs = axs.flatten()

    for i, (ax, titre, img) in enumerate(zip(axs, titres, imgs)):
        ax.imshow(to_image(img), cmap="gray")
        ax.set_title(titre, fontsize=25)
        ax.axis("off")

    plt.tight_layout()
    plt.show()

# === Calcul du taux de restauration de l'image ===
def taux_restauration(img_originale, img_restauree):
    pixels_corrects = np.sum(img_originale == img_restauree)
    total_pixels = img_originale.size
```

```
    taux = (pixels_corrects / total_pixels) * 100
    return taux

# === Paramètres ===
chemin_image = "images/souris.png"
#chemin_image = "images/chat.jpg"
sigma_bruit = 0.5
p_bruit = 0.3
nb_etats = 2
nb_iter = 200000
beta = 0.8

# === Exécution ===
img = charger_image_grayscale(chemin_image, nb_etats)
#img_bruitee = ajouter_bruit_uniforme(img, p_bruit, nb_etats)
img_bruitee = ajouter_bruit_gaussien_discret(img, sigma_bruit, nb_etats)
champ_gibbs = img_bruitee.copy()
champ_gibbs_rec = img_bruitee.copy()
champ_metro = img_bruitee.copy()
champ_metro_rec = img_bruitee.copy()

poids_aretes = {(a, b): -1 if a == b else 1 for a in range(nb_etats) for b in
range(nb_etats)}
poids_sommets = [0] * nb_etats
modele = {
    'nb_etats': nb_etats,
    'poids_aretes': poids_aretes,
    'poids_sommets': poids_sommets
}

champ_gibbs = gibbs_classique(champ_gibbs, nb_iter, modele)
champ_gibbs_rec = gibbs_recuit(champ_gibbs_rec, nb_iter, modele)
champ_metro = metropolis_classique(champ_metro, nb_iter, modele)
champ_metro_rec = metropolis_recuit(champ_metro_rec, nb_iter, modele)

# Affichage des résultats
afficher_resultats(img, img_bruitee, champ_gibbs, champ_gibbs_rec, champ_metro,
champ_metro_rec, nb_etats)
```

ANNEXE 4

```
# === Comparaison des différents algorithmes pour le modèle de Potts  ===

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from tqdm import tqdm

# === Chargement et quantification de l'image ===
def charger_image_grayscale(path, nb_etats):
```

```
    img = Image.open(path).convert("L").resize((100, 100))
    img_np = np.array(img)
    return np.floor(img_np / (256 / nb_etats)).astype(int)

# === Ajout de bruit : simulateur du modèle d'observation P(Y|X) ===
def ajouter_bruit_uniforme(champ, p, nb_etats):
    bruit = np.random.choice(range(nb_etats), size=champ.shape)
    masque = np.random.rand(*champ.shape) < p
    return np.where(masque, bruit, champ)

def ajouter_bruit_gaussien_discret(champ, sigma, nb_etats):
    bruit = np.random.normal(0, sigma, size=champ.shape)
    champ_bruite = np.round(champ + bruit).astype(int)
    champ_bruite = np.clip(champ_bruite, 0, nb_etats - 1)
    return champ_bruite

# === Calcul de l'énergie locale ===
def cdf_locale_4(i, j, H, L, champ, etat, poids_aretes, poids_sommets):
    voisins = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    energie = poids_sommets[etat]
    for dx, dy in voisins:
        ni, nj = i + dx, j + dy
        if 0 <= ni < H and 0 <= nj < L:
            energie += poids_aretes[(etat, champ[ni, nj])]
    return energie

# === Algorithme de Gibbs pour le modèle de Potts ===
def gibbs_classique(champ, nb_iter, modele):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="Gibbs classique (Potts)"):
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
        energies = np.array([
            cdf_locale_4(i, j, H, L, champ, s, modele['poids_aretes'],
modele['poids_sommets'])
            for s in range(modele['nb_etats'])
        ])
        proba = np.exp(-energies)
        proba /= np.sum(proba)
        champ[i, j] = np.random.choice(range(modele['nb_etats']), p=proba)
    return champ

# === Algorithme de Gibbs pour le modèle de Potts avec recuit simulé===
def gibbs_recuit(champ, nb_iter, modele, t):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="Gibbs recuit simulé (Potts)"):
        T = t / np.log(2 + k)
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
        energies = np.array([
            cdf_locale_4(i, j, H, L, champ, s, modele['poids_aretes'],
modele['poids_sommets'])
```

```
            for s in range(modele['nb_etats'])
        ])
        proba = np.exp(-energies / T)
        proba /= np.sum(proba)
        champ[i, j] = np.random.choice(range(modele['nb_etats']), p=proba)
    return champ

# === Algorithme de Metropolis-Hastings pour le modèle de Potts ===
def metropolis_classique(champ, nb_iter, modele):
    H, L = champ.shape
    T = 1
    for k in tqdm(range(nb_iter), desc="Metropolis classique (Potts)"):
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
        etat_actuel = champ[i, j]
        nouvel_etat = np.random.choice([s for s in range(modele['nb_etats']) if s !=
etat_actuel])
        energie_actuelle = cdf_locale_4(i, j, H, L, champ, etat_actuel,
modele['poids_aretes'], modele['poids_sommets'])
        energie_nouvelle = cdf_locale_4(i, j, H, L, champ, nouvel_etat,
modele['poids_aretes'], modele['poids_sommets'])
        delta_E = energie_nouvelle - energie_actuelle
        if delta_E < 0 or np.random.rand() < np.exp(-delta_E / T):
            champ[i, j] = nouvel_etat
    return champ

# === Algorithme de Metropolis-Hastings pour le modèle de Potts avec recuit simulé ===
def metropolis_recuit(champ, nb_iter, modele, t):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="Metropolis recuit simulé (Potts)"):
        T = t / np.log(2 + k)
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
        etat_actuel = champ[i, j]
        nouvel_etat = np.random.choice([s for s in range(modele['nb_etats']) if s !=
etat_actuel])
        energie_actuelle = cdf_locale_4(i, j, H, L, champ, etat_actuel,
modele['poids_aretes'], modele['poids_sommets'])
        energie_nouvelle = cdf_locale_4(i, j, H, L, champ, nouvel_etat,
modele['poids_aretes'], modele['poids_sommets'])
        delta_E = energie_nouvelle - energie_actuelle
        if delta_E < 0 or np.random.rand() < np.exp(-delta_E / T):
            champ[i, j] = nouvel_etat
    return champ

# === Visualisation ===
def afficher_resultats(img_init, img_bruitee, gibbs, gibbs_rec, metro, metro_rec, nb_etats):
    def to_image(img):
        return (img * (255 / (nb_etats - 1))).astype(np.uint8)

    # Calcul du taux de restauration pour les deux estimateurs
    taux = taux_restauration(img_init, img_init)
```

```
    taux_base = taux_restauration(img_init, img_bruitee)
    taux_gibbs = taux_restauration(img_init, champ_gibbs)
    taux_gibbs_rec = taux_restauration(img_init, champ_gibbs_rec)
    taux_metro = taux_restauration(img_init, champ_metro)
    taux_metro_rec = taux_restauration(img_init, champ_metro_rec)

    imgs = [img_init, gibbs, gibbs_rec, img_bruitee, metro, metro_rec]
    titres = [
        f"Image originale \n(ε={taux:.2f})", f"Gibbs classique \n(ε={taux_gibbs:.2f})",
        f"Gibbs recuit simulé \n(ε={taux_gibbs_rec:.2f})", f"Image bruitée
\n(ε={taux_base:.2f})",
        f"Metropolis classique \n(ε={taux_metro:.2f})", f"Metropolis recuit simulé
\n(ε={taux_metro_rec:.2f})"
    ]
    fig, axs = plt.subplots(2, 3, figsize=(15, 8))
    axs = axs.flatten()
    for ax, titre, img in zip(axs, titres, imgs):
        ax.imshow(to_image(img), cmap="gray")
        ax.set_title(titre, fontsize=25)
        ax.axis("off")
    plt.tight_layout()
    plt.show()

# === Calcul du taux de restauration de l'image ===
def taux_restauration(img_originale, img_restauree):
    pixels_corrects = np.sum(img_originale == img_restauree)
    total_pixels = img_originale.size
    taux = (pixels_corrects / total_pixels) * 100
    return taux

# === Paramètres ===
chemin_image = "images/test1.png"
sigma_bruit = 0.5
p_bruit = 0.3
nb_etats = 3
nb_iter = 200000
beta = 0.5
temp = 1

# === Exécution ===
img = charger_image_grayscale(chemin_image, nb_etats)
#img_bruitee = ajouter_bruit_uniforme(img, p_bruit, nb_etats)
img_bruitee = ajouter_bruit_gaussien_discret(img, sigma_bruit, nb_etats)
champ_gibbs = img_bruitee.copy()
champ_gibbs_rec = img_bruitee.copy()
champ_metro = img_bruitee.copy()
champ_metro_rec = img_bruitee.copy()

#poids_aretes = {(a, b): -1 if a == b else 1 for a in range(nb_etats) for b in
range(nb_etats)}
poids_aretes = {(a, b): beta * (-1 if a == b else 1) for a in range(nb_etats) for b in
range(nb_etats)}
```

```
poids_sommets = [0] * nb_etats
modele = {
    'nb_etats': nb_etats,
    'poids_aretes': poids_aretes,
    'poids_sommets': poids_sommets
}

champ_gibbs = gibbs_classique(champ_gibbs, nb_iter, modele)
champ_gibbs_rec = gibbs_recuit(champ_gibbs_rec, nb_iter, modele, temp)
champ_metro = metropolis_classique(champ_metro, nb_iter, modele)
champ_metro_rec = metropolis_recuit(champ_metro_rec, nb_iter, modele, temp)

# Affichage des résultats
afficher_resultats(img, img_bruitee, champ_gibbs, champ_gibbs_rec, champ_metro,
champ_metro_rec, nb_etats)
```

ANNEXE 5

```
# === Comparaison des différents algorithmes pour le modèle markovien gaussien  ===

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from tqdm import tqdm

# === Chargement et quantification de l'image ===
def charger_image_grayscale(path, nb_etats):
    #img = Image.open(path)
    img = Image.open(path).convert("L").resize((300, 300))
    img_np = np.array(img)
    return np.floor(img_np / (256 / nb_etats)).astype(int)

# === Ajout de bruit : simulateur du modèle d'observation P(Y|X) ===
def ajouter_bruit_uniforme(champ, p, nb_etats):
    bruit = np.random.choice(range(nb_etats), size=champ.shape)
    masque = np.random.rand(*champ.shape) < p
    return np.where(masque, bruit, champ)

def ajouter_bruit_gaussien_discret(champ, sigma, nb_etats):
    bruit = np.random.normal(0, sigma, size=champ.shape)
    champ_bruite = np.round(champ + bruit).astype(int)
    champ_bruite = np.clip(champ_bruite, 0, nb_etats - 1)
    return champ_bruite

# === Calcul de l'énergie locale ===
def cdf_locale_4(i, j, H, L, champ, etat):
    voisins = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    attache=0
    energie_locale= 0
    for dx, dy in voisins:
```

```
        ni, nj = i + dx, j + dy
        if 0 <= ni < H and 0 <= nj < L:
            energie_locale+=(etat-champ[ni,nj])**2
    attache=np.sum((champ - img_bruitee)**2)
    return beta*energie_locale + alpha* attache

# === Algorithme de Gibbs pour le modèle markovien gaussien ===
def gibbs_classique(champ, nb_iter, modele):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="Gibbs classique (Potts)"):
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
        energies = np.array([
            cdf_locale_4(i, j, H, L, champ, s)
            for s in range(modele['nb_etats'])
        ])
        energies -= np.min(energies)
        proba = np.exp(-energies)
        proba_sum = np.sum(proba)
        if proba_sum == 0 or np.isnan(proba_sum):
            proba = np.ones_like(proba) / len(proba)  # uniformiser en cas de défaillance
        else:
            proba /= proba_sum
        champ[i, j] = np.random.choice(range(modele['nb_etats']), p=proba)
    return champ

# === Algorithme de Gibbs pour le modèle markovien gaussien avec recuit simulé ===
def gibbs_recuit(champ, nb_iter, modele, t):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="Gibbs recuit simulé (Potts)"):
        T = t / np.log(2 + k)
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
        energies = np.array([
            cdf_locale_4(i, j, H, L, champ, s)
            for s in range(modele['nb_etats'])
        ])
        energies -= np.min(energies)
        proba = np.exp(-energies/T)
        proba_sum = np.sum(proba)
        if proba_sum == 0 or np.isnan(proba_sum):
            proba = np.ones_like(proba) / len(proba)  # uniformiser en cas de défaillance
        else:
            proba /= proba_sum
        champ[i, j] = np.random.choice(range(modele['nb_etats']), p=proba)
    return champ

# === Algorithme de Metropolis-Hastings pour le modèle markovien gaussien ===
def metropolis_classique(champ, nb_iter, modele):
    H, L = champ.shape
    T = 1
    for k in tqdm(range(nb_iter), desc="Metropolis classique (Potts)"):
```

```
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
        etat_actuel = champ[i, j]
        nouvel_etat = np.random.choice([s for s in range(modele['nb_etats']) if s !=
etat_actuel])
        energie_actuelle = cdf_locale_4(i, j, H, L, champ, etat_actuel)
        energie_nouvelle = cdf_locale_4(i, j, H, L, champ, nouvel_etat)
        delta_E = energie_nouvelle - energie_actuelle
        if delta_E < 0 or np.random.rand() < np.exp(-delta_E / T):
            champ[i, j] = nouvel_etat
    return champ

# === Algorithme de Metropolis-Hastings pour le modèle markovien gaussien avec recuit simulé
===
def metropolis_recuit(champ, nb_iter, modele, t):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="Metropolis recuit simulé (Potts)"):
        T = t / np.log(2 + k)
        i = np.random.randint(0, H)
        j = np.random.randint(0, L)
        etat_actuel = champ[i, j]
        nouvel_etat = np.random.choice([s for s in range(modele['nb_etats']) if s !=
etat_actuel])
        energie_actuelle = cdf_locale_4(i, j, H, L, champ, etat_actuel)
        energie_nouvelle = cdf_locale_4(i, j, H, L, champ, nouvel_etat)
        delta_E = energie_nouvelle - energie_actuelle
        if delta_E < 0 or np.random.rand() < np.exp(-delta_E / T):
            champ[i, j] = nouvel_etat
    return champ

# === Visualisation ===
def afficher_resultats(img_init, img_bruitee, gibbs, gibbs_rec, metro, metro_rec, nb_etats):
    def to_image(img):
        return (img * (255 / (nb_etats - 1))).astype(np.uint8)

    # Calcul du taux de restauration pour les deux estimateurs
    taux = taux_restauration(img_init, img_init)
    taux_base = taux_restauration(img_init, img_bruitee)
    taux_gibbs = taux_restauration(img_init, champ_gibbs)
    taux_gibbs_rec = taux_restauration(img_init, champ_gibbs_rec)
    taux_metro = taux_restauration(img_init, champ_metro)
    taux_metro_rec = taux_restauration(img_init, champ_metro_rec)

    imgs = [img_init, gibbs, gibbs_rec, img_bruitee, metro, metro_rec]
    titres = [
        f"Image originale \n(ε={taux:.2f})",
        f"Gibbs classique \n(ε={taux_gibbs:.2f})",
        f"Gibbs recuit simulé \n(ε={taux_gibbs_rec:.2f})",
        f"Image bruitée \n(ε={taux_base:.2f})",
        f"Metropolis classique \n(ε={taux_metro:.2f})",
        f"Metropolis recuit simulé \n(ε={taux_metro_rec:.2f})"
    ]
```

```
    fig, axs = plt.subplots(2, 3, figsize=(15, 8))
    axs = axs.flatten()
    for ax, titre, img in zip(axs, titres, imgs):
        ax.imshow(to_image(img), cmap="gray")
        ax.set_title(titre, fontsize=25)
        ax.axis("off")
    plt.tight_layout()
    plt.show()

# === Calcul du taux de restauration de l'image ===
def taux_restauration(img_originale, img_restauree):
    pixels_corrects = np.sum(img_originale == img_restauree)
    total_pixels = img_originale.size
    taux = (pixels_corrects / total_pixels) * 100
    return taux

# === Paramètres ===
chemin_image = "images/nb.png"
sigma_bruit = 60
p_bruit = 0.3
nb_etats = 255
nb_iter = 20000
beta = 5
alpha= 0.005
temp = 1

# === Exécution ===
img = charger_image_grayscale(chemin_image, nb_etats)
#img_bruitee = ajouter_bruit_uniforme(img, p_bruit, nb_etats)
img_bruitee = ajouter_bruit_gaussien_discret(img, sigma_bruit, nb_etats)
champ_gibbs = img_bruitee.copy()
champ_gibbs_rec = img_bruitee.copy()
champ_metro = img_bruitee.copy()
champ_metro_rec = img_bruitee.copy()

#poids_aretes = {(a, b): -1 if a == b else 1 for a in range(nb_etats) for b in
range(nb_etats)}
poids_aretes = {(a, b): beta * (-1 if a == b else 1) for a in range(nb_etats) for b in
range(nb_etats)}
poids_sommets = [0] * nb_etats
modele = {
    'nb_etats': nb_etats,
    'poids_aretes': poids_aretes,
    'poids_sommets': poids_sommets
}

champ_gibbs = gibbs_classique(champ_gibbs, nb_iter, modele)
champ_gibbs_rec = gibbs_recuit(champ_gibbs_rec, nb_iter, modele, temp)
champ_metro = metropolis_classique(champ_metro, nb_iter, modele)
champ_metro_rec = metropolis_recuit(champ_metro_rec, nb_iter, modele, temp)

afficher_resultats(img, img_bruitee, champ_gibbs, champ_gibbs_rec, champ_metro,
```

```
champ_metro_rec, nb_etats)
```

ANNEXE 6

```
# === Comparaison des algorithmes ICM et recuit simulé pour l'estimateur MAP ===

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from tqdm import tqdm

# === Chargement et quantification de l'image ===
def charger_image_grayscale(path, nb_etats):
    img = Image.open(path).convert("L").resize((100, 100))
    img_np = np.array(img)
    return np.floor(img_np / (256 / nb_etats)).astype(int)

# === Ajout de bruit : simulateur du modèle d'observation P(Y|X) ===
def ajouter_bruit_uniforme(champ, p, nb_etats):
    bruit = np.random.choice(range(nb_etats), size=champ.shape)
    masque = np.random.rand(*champ.shape) < p
    return np.where(masque, bruit, champ)

def ajouter_bruit_gaussien_discret(champ, sigma, nb_etats):
    bruit = np.random.normal(0, sigma, size=champ.shape)
    champ_bruite = np.round(champ + bruit).astype(int)
    champ_bruite = np.clip(champ_bruite, 0, nb_etats - 1)
    return champ_bruite

# === Calcul de l'énergie locale pour MAP (avec attache aux données) ===
def energie_locale_map(i, j, H, L, champ, etat, poids_aretes, observation, sigma2):
    voisins = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    energie = ((etat - observation[i, j]) ** 2) / (2 * sigma2)
    for dx, dy in voisins:
        ni, nj = i + dx, j + dy
        if 0 <= ni < H and 0 <= nj < L:
            energie += poids_aretes[(etat, champ[ni, nj])]
    return energie

# === Estimateur MAP via descente locale (type ICM) ===
def estimateur_map(champ, observation, nb_iter, modele, sigma2):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="MAP"):
        i, j = np.random.randint(H), np.random.randint(L)
        local_energies = np.array([
            energie_locale_map(i, j, H, L, champ, s, modele['poids_aretes'], observation,
sigma2)
            for s in range(modele['nb_etats'])
        ])
        champ[i, j] = np.argmin(local_energies)
```

```
    return champ

# === Estimateur MAP avec recuit simulé ===
def estimateur_map_recuit(champ, observation, nb_iter, modele, sigma2, t=1.0):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="MAP recuit simulé"):
        T = t / np.log(2 + k)
        i, j = np.random.randint(H), np.random.randint(L)
        local_energies = np.array([
            energie_locale_map(i, j, H, L, champ, s, modele['poids_aretes'], observation,
sigma2)
            for s in range(modele['nb_etats'])
        ])
        proba = np.exp(-local_energies / T)
        proba /= np.sum(proba)
        champ[i, j] = np.random.choice(range(modele['nb_etats']), p=proba)
    return champ

# === Visualisation ===
def afficher_resultats_multi_map(img_init, img_bruitee, map1, map2, map3, nb_etats):
    def to_image(img):
        return (img * (255 / (nb_etats - 1))).astype(np.uint8)

    # Calcul des taux de restauration
    taux = taux_restauration(img_init, img_init)
    taux_base = taux_restauration(img_init, img_bruitee)
    taux_map1 = taux_restauration(img_init, map1)
    taux_map2 = taux_restauration(img_init, map2)
    taux_map3 = taux_restauration(img_init, map3)

    titres = [f"Image originale \n(ε={taux:.2f})", f"Image bruitée \n(ε={taux_base:.2f})",
f"MAP (β={beta1}, σ²={sigma1}) \n(ε={taux_map1:.2f})", f"MAP (β={beta2}, σ²={sigma2})
\n(ε={taux_map2:.2f})", f"MAP (β={beta3}, σ²={sigma3}) \n(ε={taux_map3:.2f})"]
    images = [img_init, img_bruitee, map1, map2, map3]
    fig, axs = plt.subplots(1, 5, figsize=(20, 5))
    for ax, titre, img in zip(axs, titres, images):
        ax.imshow(to_image(img), cmap="gray")
        ax.set_title(titre, fontsize=25)
        ax.axis("off")
    plt.tight_layout()
    plt.show()

# === Calcul du taux de restauration de l'image ===
def taux_restauration(img_originale, img_restauree):
    pixels_corrects = np.sum(img_originale == img_restauree)
    total_pixels = img_originale.size
    taux = (pixels_corrects / total_pixels) * 100
    return taux

# === Paramètres ===
chemin_image = "images/test1.png"
sigma_bruit = 0.5
```

```
p_bruit = 0.3
nb_etats = 3
nb_iter = 300000

img = charger_image_grayscale(chemin_image, nb_etats)
#img_bruitee = ajouter_bruit_uniforme(img, p_bruit, nb_etats)
img_bruitee = ajouter_bruit_gaussien_discret(img, sigma_bruit, nb_etats)
champ_init = img_bruitee.copy()

# MAP 1 : beta = 0.3, sigma2 = 1
beta1 = 1
sigma1 = 1
poids_aretes1 = {(a, b): beta1 * (-1 if a == b else 1) for a in range(nb_etats) for b in
range(nb_etats)}
modele1 = {'nb_etats': nb_etats, 'poids_aretes': poids_aretes1, 'poids_sommets': [0] *
nb_etats}
map1 = estimateur_map(champ_init.copy(), img_bruitee, nb_iter, modele1, sigma1)

# MAP 2 : beta = 0.7, sigma2 = 1
beta2 = 0.1
sigma2 = 1
poids_aretes2 = {(a, b): beta2 * (-1 if a == b else 1) for a in range(nb_etats) for b in
range(nb_etats)}
modele2 = {'nb_etats': nb_etats, 'poids_aretes': poids_aretes2, 'poids_sommets': [0] *
nb_etats}
map2 = estimateur_map(champ_init.copy(), img_bruitee, nb_iter, modele2, sigma2)

# MAP 3 : beta = 1.0, sigma2 = 2
beta3 = -1
sigma3 = 1
poids_aretes3 = {(a, b): beta3 * (-1 if a == b else 1) for a in range(nb_etats) for b in
range(nb_etats)}
modele3 = {'nb_etats': nb_etats, 'poids_aretes': poids_aretes3, 'poids_sommets': [0] *
nb_etats}
map3 = estimateur_map(champ_init.copy(), img_bruitee, nb_iter, modele3, sigma3)

# Affichage final
afficher_resultats_multi_map(img, img_bruitee, map1, map2, map3, nb_etats)
```

ANNEXE 7

```
# === Comparaison des énergies par les algorithmes d'ICM et du recuit simulé pour
l'estimateur MAP ===

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from tqdm import tqdm

# === Chargement et quantification de l'image ===
```

```python
def charger_image_grayscale(path, nb_etats):
    img = Image.open(path).convert("L").resize((100, 100))
    img_np = np.array(img)
    return np.floor(img_np / (256 / nb_etats)).astype(int)

# === Ajout de bruit : simulateur du modèle d'observation P(Y|X) ===
def ajouter_bruit_uniforme(champ, p, nb_etats):
    bruit = np.random.choice(range(nb_etats), size=champ.shape)
    masque = np.random.rand(*champ.shape) < p
    return np.where(masque, bruit, champ)

def ajouter_bruit_gaussien_discret(champ, sigma, nb_etats):
    bruit = np.random.normal(0, sigma, size=champ.shape)
    champ_bruite = np.round(champ + bruit).astype(int)
    champ_bruite = np.clip(champ_bruite, 0, nb_etats - 1)
    return champ_bruite

# === Calcul de l'énergie locale pour MAP (avec attache aux données) ===
def energie_locale_map(i, j, H, L, champ, etat, poids_aretes, observation, sigma2):
    voisins = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    energie = ((etat - observation[i, j]) ** 2) / (2 * sigma2)
    for dx, dy in voisins:
        ni, nj = i + dx, j + dy
        if 0 <= ni < H and 0 <= nj < L:
            energie += poids_aretes[(etat, champ[ni, nj])]
    return energie

# === Énergie globale ===
def calcul_energie_globale(champ, observation, modele, sigma2):
    H, L = champ.shape
    energie = 0
    for i in range(H):
        for j in range(L):
            etat = champ[i, j]
            energie += ((etat - observation[i, j]) ** 2) / (2 * sigma2)
            for dx, dy in [(-1, 0), (0, -1)]:  # éviter double comptage
                ni, nj = i + dx, j + dy
                if 0 <= ni < H and 0 <= nj < L:
                    energie += modele['poids_aretes'][(etat, champ[ni, nj])]
    return energie

# === Estimateur MAP via descente locale ===
def estimateur_map(champ, observation, nb_iter, modele, sigma2):
    H, L = champ.shape
    energies = []
    for k in tqdm(range(nb_iter), desc="MAP"):
        i, j = np.random.randint(H), np.random.randint(L)
        local_energies = np.array([
            energie_locale_map(i, j, H, L, champ, s, modele['poids_aretes'], observation,
sigma2)
            for s in range(modele['nb_etats'])
        ])
```

```
        champ[i, j] = np.argmin(local_energies)
        if k % 1000 == 0:
            energies.append(calcul_energie_globale(champ, observation, modele, sigma2))
    return champ, energies

# === Estimateur MAP avec recuit simulé ===
def estimateur_map_recuit(champ, observation, nb_iter, modele, sigma2, t_init=1.0):
    H, L = champ.shape
    energies = []
    for k in tqdm(range(nb_iter), desc="MAP recuit simulé"):
        T = t_init / np.log(2 + k)
        i, j = np.random.randint(H), np.random.randint(L)
        local_energies = np.array([
            energie_locale_map(i, j, H, L, champ, s, modele['poids_aretes'], observation,
sigma2)
            for s in range(modele['nb_etats'])
        ])
        proba = np.exp(-local_energies / T)
        proba /= np.sum(proba)
        champ[i, j] = np.random.choice(range(modele['nb_etats']), p=proba)
        if k % 1000 == 0:
            energies.append(calcul_energie_globale(champ, observation, modele, sigma2))
    return champ, energies

# === Visualisation comparée MAP vs MAP recuit simulé ===
def afficher_map_vs_recuit(img_init, img_bruitee, map_est, map_recuit_est, nb_etats):
    def to_image(img):
        return (img * (255 / (nb_etats - 1))).astype(np.uint8)

    taux = taux_restauration(img_init, img_init)
    taux_base = taux_restauration(img_init, img_bruitee)
    taux_map = taux_restauration(img_init, map_est)
    taux_mapr = taux_restauration(img_init, map_recuit)

    images = [img_init, img_bruitee, map_est, map_recuit_est]
    #titres = ["Image originale", "Image bruitée", "MAP (descente locale)", "MAP (recuit
simulé)"]

    titres = [
        f"Image originale \n(ε={taux:.2f})", f"Image bruitée \n(ε={taux_base:.2f})",
        f"MAP (descente locale) \n(ε={taux_map:.2f})", f"MAP (recuit simulé)
\n(ε={taux_mapr:.2f})"
    ]

    fig, axs = plt.subplots(1, 4, figsize=(20, 5))
    for ax, titre, img in zip(axs, titres, images):
        ax.imshow(to_image(img), cmap="gray")
        ax.set_title(titre, fontsize=25)
        ax.axis("off")
    plt.tight_layout()
    plt.show()
```

```
# === Tracer l'évolution de l'énergie ===
def tracer_energie(energies_map, energies_recuit):
    plt.figure(figsize=(8, 5))
    plt.plot(np.arange(len(energies_map)) * 1000, energies_map, label="MAP (descente
locale)")
    plt.plot(np.arange(len(energies_recuit)) * 1000, energies_recuit, label="MAP (recuit
simulé)")
    plt.xlabel("Itérations")
    plt.ylabel("Énergie globale")
    plt.title("Évolution de l'énergie au cours des itérations")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# === Calcul du taux de restauration de l'image ===
def taux_restauration(img_originale, img_restauree):
    pixels_corrects = np.sum(img_originale == img_restauree)
    total_pixels = img_originale.size
    taux = (pixels_corrects / total_pixels) * 100
    return taux

# === Paramètres ===
chemin_image = "images/test1.png"
sigma_bruit = 0.5
p_bruit = 0.3
nb_etats = 3
nb_iter = 100000
beta = 0.3
sigma2 = 6
t_init = 1

# === Chargement et préparation des données ===
img = charger_image_grayscale(chemin_image, nb_etats)
#img_bruitee = ajouter_bruit_uniforme(img, p_bruit, nb_etats)
img_bruitee = ajouter_bruit_gaussien_discret(img, sigma_bruit, nb_etats)
champ_init = img_bruitee.copy()

# === Définition du modèle Potts ===
poids_aretes = {
    (a, b): beta * (-1 if a == b else 1)
    for a in range(nb_etats) for b in range(nb_etats)
}
poids_sommets = [0] * nb_etats
modele = {
    'nb_etats': nb_etats,
    'poids_aretes': poids_aretes,
    'poids_sommets': poids_sommets
}

# === Estimation MAP classique et recuit simulé ===
map_est, energie_map = estimateur_map(champ_init.copy(), img_bruitee, nb_iter, modele,
```

```
sigma2)
map_recuit, energie_recuit = estimateur_map_recuit(champ_init.copy(), img_bruitee, nb_iter,
modele, sigma2, t_init)

# === Affichage des résultats ===
afficher_map_vs_recuit(img, img_bruitee, map_est, map_recuit, nb_etats)
tracer_energie(energie_map, energie_recuit)
```

ANNEXE 8

```
# === Comparaison des différents estimateurs bayésiens ===

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from tqdm import tqdm

# === Chargement et quantification de l'image ===
def charger_image_grayscale(path, nb_etats):
    """
    Charge une image, la convertit en niveaux de gris, puis la quantifie sur 'nb_etats'
niveaux.
    Cela correspond à x ∈ {0, ..., nb_etats - 1}^S.
    """
    img = Image.open(path).convert("L").resize((100, 100))
    img_np = np.array(img)
    return np.floor(img_np / (256 / nb_etats)).astype(int)

# === Ajout de bruit : simulateur du modèle d'observation P(Y|X) ===
def ajouter_bruit(champ, p, nb_etats):
    """
    Simule une observation bruitée y ~ P(Y|X), avec un bruit uniforme ajouté avec
probabilité p.
    """
    bruit = np.random.choice(range(nb_etats), size=champ.shape)
    masque = np.random.rand(*champ.shape) < p
    return np.where(masque, bruit, champ)

def ajouter_bruit_uniforme(champ, p, nb_etats):
    bruit = np.random.choice(range(nb_etats), size=champ.shape)
    masque = np.random.rand(*champ.shape) < p
    return np.where(masque, bruit, champ)

def ajouter_bruit_gaussien_discret(champ, sigma, nb_etats):
    bruit = np.random.normal(0, sigma, size=champ.shape)
    champ_bruite = np.round(champ + bruit).astype(int)
    champ_bruite = np.clip(champ_bruite, 0, nb_etats - 1)
    return champ_bruite

# === Calcul de l'énergie locale (U_s) pour un pixel donné ===
```

```
def energie_locale(i, j, H, L, champ, etat, poids_aretes, poids_sommets):
    """
    Calcule l'énergie locale si on assigne 'etat' au pixel (i,j), en tenant compte des
voisins (cliques d'ordre 2).
    """
    voisins = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    energie = poids_sommets[etat]  # Potentiel de site (clique d'ordre 1)
    for dx, dy in voisins:
        ni, nj = i + dx, j + dy
        if 0 <= ni < H and 0 <= nj < L:
            energie += poids_aretes[(etat, champ[ni, nj])]
    return energie

# === Calcul de l'énergie locale pour MAP (avec attache aux données) ===
def energie_locale_map(i, j, H, L, champ, etat, poids_aretes, observation, sigma2):
    """
    même chose avec attache aux données
    """
    voisins = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    energie = ((etat - observation[i, j]) ** 2) / (2 * sigma2)
    for dx, dy in voisins:
        ni, nj = i + dx, j + dy
        if 0 <= ni < H and 0 <= nj < L:
            energie += poids_aretes[(etat, champ[ni, nj])]
    return energie

# === Estimateur MAP via recuit simulé (minimisation de U(x|y)) ===
def estimateur_map(champ, observation, nb_iter, modele, sigma2, t_init=1.0):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="MAP - recuit simulé"):
        T = t_init / np.log(2 + k)
        i, j = np.random.randint(H), np.random.randint(L)
        energies = np.array([
            energie_locale_map(i, j, H, L, champ, s, modele['poids_aretes'], observation,
sigma2)
            for s in range(modele['nb_etats'])
        ])
        proba = np.exp(-energies / T)
        proba /= np.sum(proba)
        champ[i, j] = np.random.choice(range(modele['nb_etats']), p=proba)
    return champ

# === Génération d'échantillons Gibbs pour MPM / TPM ===
def gibbs_mcmc(champ, nb_iter, modele, collect_start=50000, collect_every=1000):
    H, L = champ.shape
    samples = []
    for k in tqdm(range(nb_iter), desc="Gibbs MCMC"):
        i, j = np.random.randint(H), np.random.randint(L)
        energies = np.array([
            energie_locale(i, j, H, L, champ, s, modele['poids_aretes'],
modele['poids_sommets'])
            for s in range(modele['nb_etats'])
```

```
        ])
        proba = np.exp(-energies)
        proba /= np.sum(proba)
        champ[i, j] = np.random.choice(range(modele['nb_etats']), p=proba)
        if k >= collect_start and (k - collect_start) % collect_every == 0:
            samples.append(champ.copy())
    return samples

# === Estimateur MPM : argmax_s P(X_s | Y) ===
def estimateur_mpm(samples, nb_etats):
    H, L = samples[0].shape
    result = np.zeros((H, L), dtype=int)
    for i in range(H):
        for j in range(L):
            counts = np.bincount([s[i, j] for s in samples], minlength=nb_etats)
            result[i, j] = np.argmax(counts)
    return result

# === Estimateur TPM : E[X_s | Y] (moyenne empirique) ===
def estimateur_tpm(samples):
    return np.round(np.mean(np.array(samples), axis=0)).astype(int)

# === Visualisation ===
def afficher_resultats(img_init, img_bruitee, map_est, mpm_est, tpm_est, nb_etats, beta,
sigma2, p_bruit, nb_iter):
    def to_image(img):
        return (img * (255 / (nb_etats - 1))).astype(np.uint8)

    # Calcul du taux de restauration pour les deux estimateurs
    taux = taux_restauration(img_init, img_init)
    taux_base = taux_restauration(img_init, img_bruitee)
    taux_map = taux_restauration(img_init, map_est)
    taux_mpm = taux_restauration(img_init, mpm_est)
    taux_tpm = taux_restauration(img_init, tpm_est)

    titres = [
        f"Image originale\n(ε={taux:.2f})",
        f"Image bruitée\n(ε={taux_base:.2f})",
        f"MAP\n(ε={taux_map:.2f})",
        f"MPM\n(ε={taux_mpm:.2f})",
        f"TMP\n(ε={taux_tpm:.2f})"
    ]
    images = [img_init, img_bruitee, map_est, mpm_est, tpm_est]

    fig, axs = plt.subplots(1, 5, figsize=(22, 6))
    for ax, titre, img in zip(axs, titres, images):
        ax.imshow(to_image(img), cmap="gray")
        ax.set_title(titre, fontsize=22)
        ax.axis("off")

    param_text = f"Paramètres : nb_etats={nb_etats}, p_bruit={p_bruit}, nb_iter={nb_iter},
β={beta}, σ²={sigma2}"
```

```
    fig.text(0.5, 0.1, param_text, ha='center', fontsize=16)

    plt.tight_layout(rect=[0, 0.05, 1, 1])
    plt.show()

# === Calcul du taux de restauration de l'image ===
def taux_restauration(img_originale, img_restauree):
    pixels_corrects = np.sum(img_originale == img_restauree)
    total_pixels = img_originale.size
    taux = (pixels_corrects / total_pixels) * 100
    return taux

# === Paramètres ===
chemin_image = "images/test2.jpg"
sigma_bruit = 0.5
p_bruit = 0.3
nb_etats = 32
nb_iter = 60000
beta = 2
sigma2 = 0.8

# === Exécution ===
img = charger_image_grayscale(chemin_image, nb_etats)
#img_bruitee = ajouter_bruit_uniforme(img, p_bruit, nb_etats)
img_bruitee = ajouter_bruit_gaussien_discret(img, sigma_bruit, nb_etats)
champ_init = img_bruitee.copy()

# Modèle de Potts binaire
#poids_aretes = {(a, b): -1 if a == b else 1 for a in range(nb_etats) for b in
range(nb_etats)}
poids_aretes = {
    (a, b): beta * (-1 if a == b else 1)
    for a in range(nb_etats) for b in range(nb_etats)
}
poids_sommets = [0] * nb_etats
modele = {
    'nb_etats': nb_etats,
    'poids_aretes': poids_aretes,
    'poids_sommets': poids_sommets
}

# MAP
map_est = estimateur_map(champ_init.copy(), img_bruitee, nb_iter, modele, sigma2)

# MCMC
mcmc_samples = gibbs_mcmc(champ_init.copy(), nb_iter, modele)
mpm_est = estimateur_mpm(mcmc_samples, nb_etats)
tpm_est = estimateur_tpm(mcmc_samples)

# Affichage final
afficher_resultats(img, img_bruitee, map_est, mpm_est, tpm_est, nb_etats, beta, sigma2,
p_bruit, nb_iter)
```

ANNEXE 9

```
# === Comparaison entre différents paramètres pour l'estimateur MAP ===

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from tqdm import tqdm

# === Chargement et quantification de l'image ===
def charger_image_grayscale(path, nb_etats):
    img = Image.open(path).convert("L").resize((100, 100))
    img_np = np.array(img)
    return np.floor(img_np / (256 / nb_etats)).astype(int)

# === Ajout de bruit : simulateur du modèle d'observation P(Y|X) ===
def ajouter_bruit_uniforme(champ, p, nb_etats):
    bruit = np.random.choice(range(nb_etats), size=champ.shape)
    masque = np.random.rand(*champ.shape) < p
    return np.where(masque, bruit, champ)

def ajouter_bruit_gaussien_discret(champ, sigma, nb_etats):
    bruit = np.random.normal(0, sigma, size=champ.shape)
    champ_bruite = np.round(champ + bruit).astype(int)
    champ_bruite = np.clip(champ_bruite, 0, nb_etats - 1)
    return champ_bruite

# === Calcul de l'énergie locale pour MAP (avec attache aux données) ===
def energie_locale_map(i, j, H, L, champ, etat, poids_aretes, observation, sigma2):
    voisins = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    energie = ((etat - observation[i, j]) ** 2) / (2 * sigma2)
    for dx, dy in voisins:
        ni, nj = i + dx, j + dy
        if 0 <= ni < H and 0 <= nj < L:
            energie += poids_aretes[(etat, champ[ni, nj])]
    return energie

# === Estimateur MAP avec recuit simulé ===
def estimateur_map(champ, observation, nb_iter, modele, sigma2, t_init=1.0):
    H, L = champ.shape
    for k in tqdm(range(nb_iter), desc="MAP recuit simulé"):
        T = t_init / np.log(2 + k)
        i, j = np.random.randint(H), np.random.randint(L)
        local_energies = np.array([
            energie_locale_map(i, j, H, L, champ, s, modele['poids_aretes'], observation,
sigma2)
            for s in range(modele['nb_etats'])
        ])
        proba = np.exp(-local_energies / T)
        proba /= np.sum(proba)
        champ[i, j] = np.random.choice(range(modele['nb_etats']), p=proba)
```

```
    return champ

# === Visualisation comparée MAP vs MAP recuit simulé ===
def afficher_map_vs_recuit(img_init, img_bruitee, map_est, nb_etats, taux, taux_base,
taux_map, taux_mapr):
    def to_image(img):
        return (img * (255 / (nb_etats - 1))).astype(np.uint8)

    images = [img_init, img_bruitee, map_est]
    #titres = ["Image originale", "Image bruitée", "MAP (descente locale)", "MAP (recuit
simulé)"]

    titres = [
        f"Image originale \n(ε={taux:.2f})", f"Image bruitée \n(ε={taux_base:.2f})",
        f"MAP (descente locale) \n(ε={taux_map:.2f})"
    ]

    fig, axs = plt.subplots(1, 4, figsize=(18, 5))
    for ax, titre, img in zip(axs, titres, images):
        ax.imshow(to_image(img), cmap="gray")
        ax.set_title(titre, fontsize=25)
        ax.axis("off")
    plt.tight_layout()
    plt.show()

# === Calcul du taux de restauration de l'image ===
def taux_restauration(img_originale, img_restauree):
    pixels_corrects = np.sum(img_originale == img_restauree)
    total_pixels = img_originale.size
    taux = (pixels_corrects / total_pixels) * 100
    return taux

# === Fonction pour accumuler les restaurations ===
def generer_images_restaures(img_bruitee, nb_images, nb_iter, modele, sigma2):
    H, L = img_bruitee.shape
    toutes_images = np.zeros((nb_images, H, L), dtype=int)
    for idx in tqdm(range(nb_images), desc="Génération des images restaurées"):
        champ_init = img_bruitee.copy()
        champ_rest = estimateur_map(champ_init, img_bruitee, nb_iter, modele, sigma2)
        toutes_images[idx] = champ_rest
    return toutes_images

# === Fonction pour calculer l'argmax pixel par pixel ===
def argmax_pixel_par_pixel(samples, nb_etats):
    H, L = samples[0].shape
    result = np.zeros((H, L), dtype=int)
    for i in range(H):
        for j in range(L):
            counts = np.bincount([s[i, j] for s in samples], minlength=nb_etats)
            result[i, j] = np.argmax(counts)
    return result
```

```python
# === Affichage final comparatif ===
def afficher_comparaison(img_ref, img_100, img_500, img_1000, original, nb_etats):
    def to_image(img):
        return (img * (255 / (nb_etats - 1))).astype(np.uint8)

    taux_ref = taux_restauration(original, img_ref)
    taux_100 = taux_restauration(original, img_100)
    taux_500 = taux_restauration(original, img_500)
    taux_1000 = taux_restauration(original, img_1000)

    titres = [
        f"MAP (base)\n(\u03b5={taux_ref:.2f})",
        f"Argmax sur 10 images\n(ε={taux_100:.2f})",
        f"Argmax sur 50 images\n(ε={taux_500:.2f})",
        f"Argmax sur 100 images\n(ε={taux_1000:.2f})"
    ]
    images = [img_ref, img_100, img_500, img_1000]

    fig, axs = plt.subplots(1, 4, figsize=(20, 5))
    for ax, titre, img in zip(axs, titres, images):
        ax.imshow(to_image(img), cmap="gray")
        ax.set_title(titre, fontsize=20)
        ax.axis("off")
    plt.tight_layout()
    plt.show()

# === Paramètres ===
chemin_image = "images/test1.png"
sigma_bruit = 0.5
p_bruit = 0.3
nb_etats = 3
nb_iter = 60000
beta = 0.8
sigma2 = 5
t_init = 1

# === Chargement et préparation des données ===
img = charger_image_grayscale(chemin_image, nb_etats)
#img_bruitee = ajouter_bruit_uniforme(img, p_bruit, nb_etats)
img_bruitee = ajouter_bruit_gaussien_discret(img, sigma_bruit, nb_etats)
champ_init = img_bruitee.copy()

# === Définition du modèle Potts ===
poids_aretes = {
    (a, b): beta * (-1 if a == b else 1)
    for a in range(nb_etats) for b in range(nb_etats)
}
poids_sommets = [0] * nb_etats
modele = {
    'nb_etats': nb_etats,
    'poids_aretes': poids_aretes,
    'poids_sommets': poids_sommets
```

```
}

# === Estimation MAP classique et recuit simulé ===
map_est = estimateur_map(champ_init.copy(), img_bruitee, nb_iter, modele, sigma2)

# === Génération et traitement ===
toutes_images = generer_images_restaures(img_bruitee, 100, nb_iter, modele=modele,
sigma2=sigma2)

img_100 = argmax_pixel_par_pixel(toutes_images[:10], nb_etats)
img_500 = argmax_pixel_par_pixel(toutes_images[:50], nb_etats)
img_1000 = argmax_pixel_par_pixel(toutes_images, nb_etats)

# === Affichage ===
afficher_comparaison(map_est, img_100, img_500, img_1000, img, nb_etats)
```