Université de Technologie de Compiègne (UTC)

Printemps 2025

Stockage probabiliste et reconstruction stochastique d'images par modélisation gaussienne

Auteur :
Youssef KHEMIRA

Suiveur :
Salim BOUZEBDA

Un projet soumis pour l'UTC :

TZ | 2 ECTS CS

18 septembre 2025

Table des matières

1 Introduction

1.1 Présentation

De nos jours, l'informatique et les nouvelles technologies occupent une place centrale dans notre société. Alors que de nombreuses avancées sont faites chaque jour et que nos appareils ne cessent d'être améliorés, une problématique suit la tendance de cette évolution : le volume de données traitées. En effet, les échanges de données via Internet ou les réseaux locaux augmentent de manière impressionnante. La qualité croissante des jeux vidéos, films et musiques téléchargées les rend de plus en plus volumineux. Le stockage est une contrainte dont on se soucie moins aujourd'hui, notamment avec les avancées majeures en miniaturisation et l'évolution des supports de stockage. Une simple comparaison permet de constater ce changement : dans les années 1960, les bandes magnétiques permettaient de stocker une dizaine de mégaoctets, avec de faibles vitesses de lecture et écriture, alors que l'on retrouve maintenant des stockages de plusieurs téraoctets à des prix extrêmement faibles comparés à cette époque. Par exemple, un disque de 256 gigaoctets aurait coûté 20 milliards de dollars à produire en 1950, tandis qu'aujourd'hui, un SSD de cette capacité coûte une vingtaine de dollars, soit un milliard de fois moins cher [1].

Cependant, l'abondance de capacité de stockage ne doit pas être prise pour acquise, et l'optimisation des applications ne doit pas être délaissée. Malheureusement, c'est le cas avec de nombreux logiciels, qui sont de moins en moins optimisés pour réduire les coûts et la durée de développement. Mais pour les futurs ingénieurs, il est primordial de chercher à faire le travail le plus propre et élégant possible, en créant des logiciels capables de s'adapter au plus grand nombre d'environnements possibles, dont ceux très contraints ; ou simplement dans une démarche responsable d'éviter de consommer plus que nécessaire sous prétexte que la différence n'est pas assez importante.

Ce projet vise à étudier le fonctionnement des différentes méthodes de compressions, avec ou sans perte utilisées aujourd'hui et la théorie sur laquelle elles s'appuient. À partir de ces informations, on tentera de concevoir, implémenter et évaluer une méthode non conventionnelle de stockage de données, fondée non sur la conservation exacte de l'information, mais sur des représentations génératives ou prédictives permettant de reconstituer approximativement les données à partir d'une forme plus compacte. Pour cela, nous traiterons une méthode basée sur les probabilités, qui s'applique à des types de données spécifiques dont l'exactitude du signal n'est pas nécessaire, tels que des audios, des images ou encore des textes. Ici, nous étudierons spécifiquement le cas des images. Ce travail a une visée expérimentale et exploratoire, l'objectif n'est donc pas de réaliser un nouveau système de compression encore plus efficace que ceux déjà existants. Il s'agit de tester la faisabilité de cette approche, de découvrir des nouvelles notions mathématiques, de mesurer les performances obtenues, et d'évaluer le compromis entre compression et fidélité.

1.2 Structure du projet

Le projet est structuré en deux grandes parties, une théorique et une autre plus pratique. La partie théorique traitera des algorithmes et méthodes actuelles de compression ainsi que leur modélisation mathématique. La seconde partie sera consacrée à l'élaboration d'une méthode de compression innovante. Elle comportera un module de stockage probabiliste approximatif : au lieu d'être stockées individuellement, les valeurs sont découpées en blocs homogènes sur lesquels peuvent être appliquées des lois de probabilité. On stocke ensuite les paramètres de la loi de probabilité à la place de chaque valeur individuelle.

1.3 Méthodologie générale

Dans un premier temps, nous aborderons d'abord les fondements du stockage des données. Nous verrons ainsi les méthodes usuelles de compression, avec et sans perte. Aussi, cela nous permettra de mieux saisir les enjeux de la réduction de données à l'ère du Big Data, et le lien avec la notion d'entropie et de redondance.

Nous poursuivrons en élargissant notre analyse aux concepts connexes que nous allons étudier dans la suite du rapport. Cette section traitera principalement des enjeux de la tolérance à l'erreur et des différents types de compression, avec ou sans perte. Cela nous permettra d'avoir des bases théoriques solides pour passer à l'application pratique, et surtout de mieux saisir les limites de ce qu'il est possible de faire : il existe des limites mathématiques desquelles les algorithmes actuels s'approchent déjà beaucoup, il sera donc difficile de parvenir à réaliser un système de compression aussi efficace que ce qui existe déjà.

Une fois tous les concepts étudiés, nous passerons à l'application de la méthode expérimentale de compression de données, décrite dans la structure du projet. On présentera le principe général, l'éventuelle nature du problème que l'on cherche à résoudre, le type de données visé, puis une implémentation en Python.La partie sera conclue par une analyse des performances, un avis sur la qualité de reconstitution, et une discussion sur la faisabilité et l'utilité.

2 Fondements du stockage de données

2.1 Représentation binaire de l'information

Il convient de commencer par rappeler les bases de l'informatique afin de mieux saisir les enjeux, avant d'étudier les méthodes expérimentales de stockage.

Les ordinateurs modernes fonctionnent grâce à des transistors, un dispositif qui permet de contrôler d'intensité du courant. Ce sont des sortes d'interrupteurs laissant passer le courant ou non selon la tension qui lui est appliquée. Il n'y a donc que deux états possibles pour un transistor, conducteur ou isolant. Il est alors nécessaire d'établir un modèle de données pouvant être interprété par les ordinateurs, à savoir le langage binaire. On pose 0 qui représente l'état où le transistor ne laisse pas passer le courant, et 1 l'état où le courant passe. C'est la forme la plus simple de codage compréhensible par un ordinateur, dont l'unité minimale est le bit.

Pour que l'ordinateur puisse traiter les données qui lui sont fournies, il faut que toutes les informations numériques soient ramenées à une suite de bits. Pour distinguer les séparations entre les blocs d'informations, des conventions ont été établies sur la taille de ces blocs : dans la plupart des cas, l'unité minimale avec laquelle on travaille est l'octet (Byte) qui représente 8 bits. Cette unité permet de représenter les nombres entiers de 0 à 255, en effet, il existe $2^8 = 256$ représentations différentes de 0 et de 1 sur 8 caractères. Cela ne suffit évidemment pas à représenter toutes les informations que nous souhaitons, il faut donc parfois utiliser plusieurs octets pour représenter un seul bloc d'information, on parle alors de mots. On peut utiliser des mots de 2, 4 ou 8 octets pour représenter de très grands nombres entiers, ou des nombres flottants (à virgule), ou bien se contenter d'un octet avec une table de correspondance entre la valeur de l'octet et le caractère à afficher (ASCII). Par exemple, associer le caractère "A" au nombre 65 permet d'encoder chaque "A" par 65 puis de le décoder ; cela implique cependant d'établir en avance une correspondance exhaustive de tous les caractères à stocker, et d'utiliser la même table pour décoder. Enfin, selon la nature des données, on emploie une structure de stockage spécifique : par exemple, les images sont des matrices de pixels contenant chacun une composante rouge, verte et bleue et chacune de ses composantes a une valeur comprise entre 0 et 255.

2.2 Compression de données : définition et types

La compression des données est un aspect extrêmement important en informatique. Comme présenté dans l'introduction, le stockage n'a pas toujours été aussi abondant qu'aujourd'hui. Les ingénieurs de l'époque ont dû redoubler d'ingéniosité pour minimiser la taille de leurs programmes. Mais pour cela, un code optimisé ne suffit pas, il faut aussi se pencher sur la manière dont est stocké notre code une fois transcrit en binaire. L'objectif est de trouver des méthodes pour réduire la taille des données afin d'économiser du stockage et faciliter la transmission.

Il existe différentes méthodes, dont certaines sont plus efficaces sur certains types de données, tandis que d'autres ont un moins bon rendement mais s'applique à plus de données. Pour évaluer ce rendement, on se fie à un indicateur qui est le ratio de

compression. Ce dernier est simplement défini par :

$$\text{Ratio compression} = \frac{\text{Taille originale}}{\text{Taille compressée}} \tag{1}$$

Plus il est élevé, plus la compression est efficace.

Pour réduire la taille des données, il faut étudier la redondance d'informations, en cherchant notamment des motifs répétés ou prévisibles qui n'apportent pas d'information réelle, donc supprimables sans altérer l'intelligibilité de la donnée. Il faut supprimer l'inutile et garder l'essentiel. Cependant, selon l'application, la notion d'essentiel peut varier : parfois, il est primordial de restituer identiquement la donnée sans aucune variation tandis que d'autres peuvent tolérer une légère variation, une "perte".

2.2.1 Compression sans perte

La compression sans perte consiste à réduire la taille des données sans aucune altération : les informations originales peuvent être entièrement et exactement reconstituées après décompression. Quelques exemples sont le format ZIP pour les fichiers généraux, le PNG pour les images, ou encore le FLAC pour l'audio haute fidélité. Il existe pour cela plusieurs méthodes.

La compression sans perte est basée sur le codage entropique, qui lui-même repose sur le principe fondamental de la théorie de l'information : un message constitué de symboles aléatoires peut être représenté avec un nombre moyen de bits par symbole proche de son **entropie**. L'entropie est une notion centrale dans le traitement des données car c'est une indication sur la taille minimale de leur représentation binaire.

Soit une source aléatoire produisant une séquence de symboles

$$X = (x_1, x_2, \ldots, x_n), \tag{2}$$

où chaque symbole x_i appartient à un alphabet fini $\mathscr{A}$ et suit une loi de probabilité $P(x)$.

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i), \tag{3}$$

où $p(x_i)$ est la probabilité du symbole x_i. L'entropie s'exprime en bits/symbole. Plus la source est incertaine (c'est-à-dire proche d'une distribution uniforme), plus $H(X)$ est élevée. Concrètement, cette valeur exprime la quantité moyenne d'information contenue dans chaque symbole. C'est la valeur moyenne minimale de la taille des données sans perte.

Si à chaque symbole x_i on associe un mot de code de longueur l_i, alors :

$$L_{\text{moyen}} = \sum_{i=1}^{n} p(x_i) \cdot l_i. \tag{4}$$

C'est la moyenne des longueurs des codes pondérée par leurs probabilités d'apparition.

Une structure de stockage optimale de ces données devrait ainsi avoir une longueur moyenne de code qui tend vers l'entropie :

$$L_{\text{moyen}} \approx H(X). \tag{5}$$

Grâce à ce fonctionnement, les symboles ayant une forte probabilité d'apparition (les plus fréquents) sont représentés par des codes courts, et les symboles rares par des codes plus longs.

Enfin, le théorème de Shannon affirme :

$$H(X) \leq L_{\text{moyen}} < H(X) + 1. \tag{6}$$

La différence entre L_{moyen} et $H(X)$ est ce qu'on appelle la **redondance du code**. Cela veut dire que l'on ne peut jamais faire mieux que $H(X)$ bits par symbole en moyenne (c'est une borne inférieure). En pratique, on parvient à s'approcher de cette borne avec des méthodes de codage entropique optimales comme Huffman ou l'arithmétique, que nous verrons par la suite.

Codage de Huffman Le codage de Huffman repose sur la construction d'un code préfixe optimal, en choisissant un codage adapté à la fréquence d'apparition des syboles. Plus un symbole est probable, plus le ccode le représentant est court. On peut ainsi minimiser la longueur moyene du code tout en garantissant un décodage non ambigu.

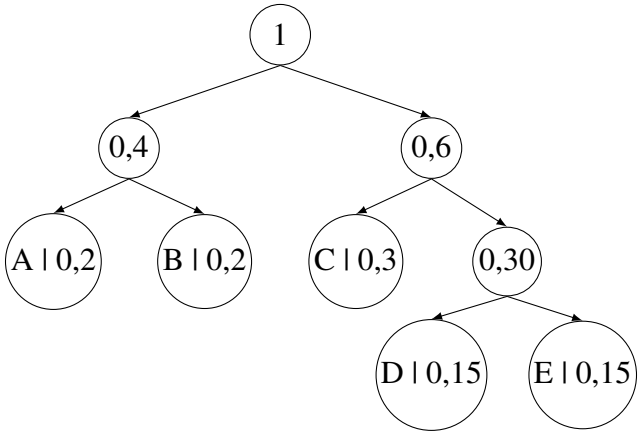
Soit un alphabet $\mathscr{A} = \{a_1, a_2, \ldots, a_m\}$ muni de probabilités $P(a_j)$, avec :

$$\sum_{j=1}^{m} P(a_j) = 1. \tag{7}$$

Le principe de Huffman se base sur une construction arborescente :

1. On place chaque symbole a_j dans une file de priorité, pondéré par sa probabilité $P(a_j)$.
2. On extrait les deux symboles (ou sous-arbres) de plus faible probabilité et on les fusionne en un nouveau nœud dont la probabilité est la somme des deux.
3. On répète le processus jusqu'à obtenir un arbre binaire unique.

Chaque symbole reçoit alors un code binaire correspondant au chemin parcouru dans l'arbre pour atteindre son noeud (par exemple 0 pour une branche gauche et 1 pour une branche droite).



Ci-dessus un exemple basique. On dispose de 5 symboles A, B, C, D et E dont les probabilités d'apparitions sont respectivement $0.2, 0.2, 0.3, 0.15, 0.15$.

1. On commence par faire un classement initial :

$$C : 0.3,\ A : 0.2,\ B : 0.2,\ D : 0.15,\ E : 0.15$$

2. On fusionne ensuite les deux nœuds les plus faibles :

$$(D, E) \to 0.15 + 0.15 = 0.3$$

   Nœuds restants : $C : 0.3, (D+E) : 0.3, A : 0.2, B : 0.2$

3. On fusionne A et B :

$$(A, B) \to 0.2 + 0.2 = 0.4$$

   Nœuds restants : $(A+B) : 0.4, C : 0.3, (D+E) : 0.3$

4. On fusionne C et $(D+E)$:

$$(C, D+E) \to 0.3 + 0.3 = 0.6$$

   Nœuds restants : $(A+B) : 0.4, (C+D+E) : 0.6$

5. Fusion finale :

$$(A+B) + (C+D+E) = 0.4 + 0.6 = 1.0$$

L'arbre est ainsi complet. En parcourant l'arbre de la racine aux feuilles (gauche = 0, droite = 1) :

$$\begin{aligned} A &= 00, \\ B &= 01, \\ C &= 10, \\ D &= 110, \\ E &= 111. \end{aligned} \tag{8}$$

La longueur moyenne pondérée des codes est :

$$L_{\text{moyen}} = 0.2 \cdot 2 + 0.2 \cdot 2 + 0.3 \cdot 2 + 0.15 \cdot 3 + 0.15 \cdot 3 = 2.25 \text{ bits/symbole.} \tag{9}$$

L'entropie de la source est :

$$H(X) = -\sum_i P(x_i) \log_2 P(x_i) \approx 2.23 \text{ bits/symbole.} \tag{10}$$

En conclusion, on voit que le code de Huffman obtenu est très proche de l'entropie, il sera donc difficile de mieux compresser sans perte la donnée. Aussi, le code obtenu est un code préfixe, ce qui signifie qu'aucun mot de code n'est préfixe d'un autre. On ne peut donc pas avoir d'ambigüité sur le symbole qui doit être représenté.

Codage arithmétique Le codage arithmétique consiste à représenter une séquence entière de symboles par un nombre réel unique compris entre 0 et 1. Chaque message correspond à un sous-intervalle unique dont la taille est proportionnelle à sa probabilité.

Soit un alphabet $\mathscr{A} = \{a_1, a_2, \ldots, a_m\}$ avec des probabilités associées $P(a_j)$, telles que :

$$\sum_{j=1}^{m} P(a_j) = 1. \tag{11}$$

On définit les bornes cumulatives :

$$C(a_j) = \sum_{k=1}^{j-1} P(a_k), \quad \text{avec } C(a_1) = 0. \tag{12}$$

Les bornes cumulatives indiquent le début de l'intervalle associé à un symbole. Chaque symbole a_j correspond alors à l'intervalle :

$$I(a_j) = [C(a_j), C(a_j) + P(a_j)). \tag{13}$$

Pour encoder une séquence $(x_1, x_2, \ldots, x_n)$, on effectue une subdivision successive de l'intervalle :

1. On commence avec l'intervalle initial $[0, 1)$.
2. Pour chaque symbole x_i, on divise l'intervalle courant proportionnellement aux probabilités, puis on sélectionne le sous-intervalle correspondant à x_i.
3. Après n symboles, on obtient un intervalle final $[L_n, U_n)$.

Après avoir subdivisé l'intervalle selon les probabilités, le message peut être représenté par n'importe quel nombre réel $z \in [L_n, U_n)$. La taille finale de cet intervalle est :

$$|I| = U_n - L_n = \prod_{i=1}^{n} P(x_i). \tag{14}$$

Pour représenter ce nombre réel en binaire, il faut environ :

$$\ell(x_1, \ldots, x_n) \approx -\log_2 |I| = -\log_2 \prod_{i=1}^{n} P(x_i) = \sum_{i=1}^{n} -\log_2 P(x_i) \tag{15}$$

bits. En effet, chaque symbole x_i contribue à approximativement $-\log_2 P(x_i)$ bits sur la séquence totale.

On peut alors établir la longueur moyenne pour n'importe quelle séquence. Si X est une variable aléatoire représentant les symboles de l'alphabet d'une séquence, la longueur moyenne par symbole est l'espérance :

$$\mathbb{E}[-\log_2 P(X)] = \sum_{j=1}^{m} P(a_j) \cdot \big(-\log_2 P(a_j)\big) = H(X), \tag{16}$$

où $H(X)$ est l'entropie de Shannon.

Ainsi, pour une séquence de n symboles indépendants :

$$\mathbb{E}[\ell] = \sum_{i=1}^{n} \mathbb{E}[-\log_2 P(X_i)] = n \cdot H(X), \tag{17}$$

ce qui montre que le codage arithmétique atteint une compression proche de la limite théorique donnée par Shannon.

Pour finir, voici un exemple qui applique le codage arithmétique :

$$\text{Alphabet : } \mathscr{A} = \{A, B, C\} \text{ avec } P(A) = 0.5, P(B) = 0.3, P(C) = 0.2$$
$$\text{Bornes cumulatives : } C(A) = 0, \quad C(B) = 0.5, \quad C(C) = 0.8$$

Intervalle initial : $[0,1)$. Séquence à encoder : `AB`

— Premier symbole A : intervalle $[0, 0.5)$
— Deuxième symbole B à l'intérieur de $[0, 0.5)$: longueur $=0.5 \times 0.3 = 0.15$
Position = $[0 + 0.5 \times 0, 0 + 0.5 \times 0.3) = [0, 0.15)$

Le message entier peut ainsi être représenté par n'importe quel nombre réel $z \in [0, 0.15)$.

2.2.2 Compression avec perte

La compression avec perte consiste à réduire la taille des données en tolérant une erreur contrôlée lors de la reconstruction des données. Cette erreur est modélisée par ce qu'on appelle la mesure de distorsion $d(x, \hat{x})$. C'est une fonction prenant deux paramètres, la donnée originale x et sa reconstruction $\hat{x}$, et qui renvoie une valeur de différence selon le modèle de fonction choisi. On travaille avec la distorsion moyenne, qui est établie à partir de la mesure de distorsion. Sur une séquence $x^n = (x_1, \ldots, x_n)$ et sa reconstruction $\hat{x}^n$, la distorsion moyenne est :

$$D = \frac{1}{n} \sum_{i=1}^{n} d(x_i, \hat{x}_i). \tag{18}$$

On choisit ensuite une fonction de distorsion adaptée au contexte des données traitées. Un choix fréquent, compatible avec beaucoup d'applications, est l'erreur quadratique moyenne $d(x, \hat{x}) = (x - \hat{x})^2$, mais il est possible d'utiliser d'autres fonctions si elles reflètent mieux la qualité perçue.

La question fondamentale est de savoir, pour une source X donnée et une contrainte D fixée, quel est le **débit minimal** (en bits par symbole) nécessaire pour représenter la source avec une distorsion moyenne $\leq D$. La réponse théorique est fournie par la **fonction taux–distorsion** de Shannon :

$$R(D) = \inf_{P_{\hat{X}|X} : \mathbb{E}[d(X,\hat{X})] \leq D} I(X; \hat{X}),$$

où $I(X; \hat{X})$ désigne l'information mutuelle sous la loi jointe $P_X P_{\hat{X}|X}$, c'est une mesure de la dépendance entre les deux variables qui peut s'écrire $I(X; \hat{X}) = H(X) - H(X|\hat{X})$. Cette fonction est décroissante et convexe en D : plus on autorise de distorsion, plus le débit minimal baisse. Il est aussi intéressant de noter que si l'on pose $D = 0$ (pas de perte), on obtient $R(0) = H(X)$, et la fonction taux-distorsion devient l'entropie. Cette dernière est un cas particulier de $R(D)$.

Prenons l'exemple d'une source gaussienne $X \sim \mathcal{N}(0, \sigma^2)$ avec une distorsion quadratique $d(x, \hat{x}) = (x - \hat{x})^2$. Dans ce cas, on obtient la formule suivante :

$$R(D) \;=\; \frac{1}{2}\log_2\Big(\frac{\sigma^2}{D}\Big) \quad \text{pour } 0 < D \leq \sigma^2, \qquad R(D) = 0 \text{ si } D \geq \sigma^2.$$

Cette formule montre que si l'on divise la distorsion D par 2, cela est compensé par environ $\frac{1}{2}$ bit par symbole en plus. Donc plus on cherche à obtenir une faible distorsion, plus la donnée sera volumineuse.

Un autre principe de base de la compression avec perte repose sur la **quantification**. Le fonctionnement est basé sur le fait que les données que l'on veut traiter sont souvent continues, elles peuvent donc prendre une infinité de valeurs. L'idée est alors de remplacer les valeurs par un nombre fini de valeurs possibles, des paliers appelés **niveaux de quantification**. Par exemple, considérons la variable continue X que l'on souhaite représenter par un nombre fini de niveaux. Si les niveaux de quantification sont régulièrement espacés, on remplace la valeur x de X par le niveau le plus proche, que l'on note $\hat{X}$.

Il est important de comprendre l'erreur introduite par les niveaux de quantification. On définit cette erreur comme la différence entre la valeur originale et la valeur quantifiée :

$$\varepsilon = X - \hat{X}.$$

Si le pas de quantification Δ est petit et que X est distribué de manière "lisse" sur chaque intervalle de quantification, on peut approximer l'erreur ε comme uniformément répartie entre $-\Delta/2$ et $+\Delta/2$. Autrement dit, l'erreur a une probabilité égale de prendre n'importe quelle valeur dans cet intervalle..

Pour une variable aléatoire uniforme $\varepsilon \sim \mathcal{U}(-\Delta/2, \Delta/2)$, l'espérance de ε est nulle et la variance, qui mesure la dispersion moyenne de l'erreur, est donnée par la formule classique de la variance d'une variable uniforme sur $[a, b]$:

$$\mathrm{Var}(\varepsilon) = \frac{(b-a)^2}{12}.$$

Ici, $a = -\Delta/2$ et $b = +\Delta/2$, donc :

$$\mathrm{Var}(\varepsilon) = \frac{(\Delta/2 - (-\Delta/2))^2}{12} = \frac{\Delta^2}{12}.$$

Dans le contexte de la compression avec perte, on s'intéresse à l'erreur quadratique moyenne (Mean Squared Error, MSE), qui s'exprime par la formule suivante :

$$\mathbb{E}[(X - \hat{X})^2] = \mathbb{E}[\varepsilon^2]$$

Or, la variance peut également s'écrire :

$$\mathrm{Var}(\varepsilon) = \mathbb{E}[(\varepsilon - \mathbb{E}[\varepsilon])^2] = \mathbb{E}[\varepsilon^2] - (\mathbb{E}[\varepsilon])^2 \tag{19}$$

Comme l'espérance est nulle, le second terme $(\mathbb{E}[\varepsilon])^2$ vaut 0. Ainsi, la distorsion moyenne correspond à l'erreur quadratique moyenne, qui vaut exactement la variance :

$$D = \mathbb{E}[(X - \hat{X})^2] = \mathbb{E}[\varepsilon^2] = \mathrm{Var}(\varepsilon) \approx \frac{\Delta^2}{12}.$$

Grâce à cette formule, on peut choisir le pas Δ pour atteindre la distorsion moyenne D ciblée. On constate aussi que réduire Δ réduit la distorsion mais augmente le nombre de niveaux à coder, donc le débit nécessaire.

Une autre technique essentielle est le **codage par transformation**. L'idée est d'effectuer une projection sur un autre espace avec une transformation pertinente qui concentrera l'information dans quelques coefficients. On note X le vecteur original et $Y = UX$ son image après transformation linéaire U, qui est souvent une matrice orthogonale. Par exemple, pour la compression du format JPEG, on utilise la DCT (Direct Cosine Transform). Un des intérêts est que la transformation conserve l'erreur quadratique totale :

$$\|X - \hat{X}\|_2^2 = \|Y - \hat{Y}\|_2^2.$$

Cela est possible car si la matrice U est orthogonale, la norme du vecteur X reste la même. On peut alors raisonner dans ce nouvel espace transformée Y comme si l'on était dans l'espace original X. Il reste alors à traiter l'importance des coefficients séparément : ceux qui portent beaucoup d'énergie sont ceux qui contiennent le plus d'informations sur l'image. Leur fréquence est plus basse donc on peut leur attribuer une quantification plus fine (un pas Δ plus petit, d'où une meilleure précision). Les coefficients faibles peuvent être quantifiés grossièrement ou ignorés pour économiser des bits.

Une fois que l'on a un vecteur représentant notre image transformée, on peut ensuite utiliser un codage entropique comme Huffman, pour compresser au maximum notre information, sans perte cette fois. La figure ci-dessous montre la suite du procédé, ainsi que les étapes de transformation de couleurs, sous-échantillonnage et découpage en blocs de pixel qui n'ont pas été évoquées.

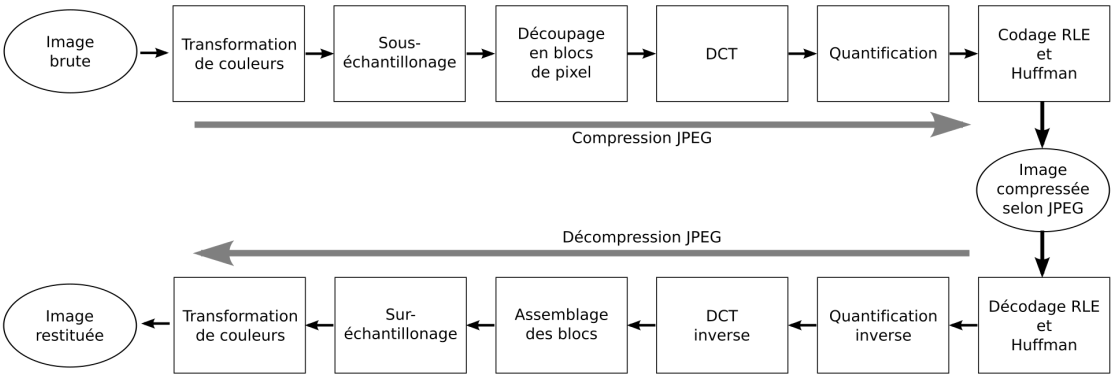


FIGURE 1 – Organigramme de compression JPEG (source : Wikipédia | Stéphane Brunner)

En résumé, la compression avec perte repose principalement sur la marge d'erreur que l'on tolère. Plus elle est grande, plus la compression sera efficace, au détriment de l'exactitude de l'information. En fonction des applications, on peut favoriser l'un ou l'autre, le tout étant de trouver un équilibre de sorte à ce que l'information reste intelligible.

2.3 Tolérance à l'erreur et robustesse de l'approximation

On comprend désormais qu'un des aspects les plus importants du stockage approximatif est sa tolérance à l'erreur. Contrairement au stockage exact où une altération des données est considérée comme une perte d'information, le stockage approximatif accepte une certaine marge d'erreur en se concentrant sur l'information dans son intégralité plutôt que localement.

L'idée est que l'information contenue dans les données n'a pas forcément besoin d'être reproduite à l'identique pour être utile. On peut notamment penser aux cas où la perception humaine est peu sensible à la distinction entre deux valeurs proches : par exemple, avec une image en niveaux de gris, l'œil humain est peu sensible aux légères variations locales d'intensité. Si deux blocs présentent des écarts faibles mais qu'ils conservent la même structure globale, ils seront alors perçus comme visuellement équivalents.

Dans ce contexte, l'erreur quadratique moyenne (présentée précédemment) est un critère pratique pour mesurer la qualité de notre approximation. Mais elle montre surtout une idée centrale : il faut faire un compromis entre la quantité d'information conservée et le niveau d'imperfection toléré. Plus l'erreur acceptable est grande, plus le stockage est compact. Plus la tolérance est faible, plus la précision se rapproche du stockage exact.

Ainsi, la tolérance à l'erreur n'est pas une faiblesse mais une propriété fondamentale du stockage approximatif. C'est un outil que l'on utilise pour réduire la taille de la donnée en fonction de l'importance de son intégrité. Elle permet d'adapter la précision des données à leur usage : rigoureuse lorsque l'on est dans le cas d'une application scientifique, ou souple lorsqu'une simple tendance ou une impression visuelle suffit.

2.4 Entropie et redondance

Pour rappel, l'entropie d'une source de données mesure son degré d'incertitude d'un point de vue probabiliste. Cela peut aussi être vu comme la quantité moyenne d'information contenue par symbole, car moins on peut prédire la valeur d'un symbole et donc sa fréquence d'apparition, plus la taille nécessaire pour le représenter sera grande. Une image aléatoire (un bruit pur) possède une entropie élevée, car chaque pixel est imprévisible, contrairement à une image qui contient de larges zones uniformes avec une entropie faible. Dans ce dernier cas, les pixels voisins sont redondants car ils n'apportent que peu d'information nouvelle.

C'est en exploitant cette redondance que le stockage approximatif est possible. Au lieu de conserver une donnée brute pour chaque pixel, il évalue la tendance globale des valeurs (par exemple la moyenne d'un bloc de pixels), et ignore les faibles variations. L'entropie du signal est alors réduite : les données deviennent plus compressibles car on peut mieux les prédire.

Ce point de vue montre la continuité entre approximation et compression. Ce sont effectivement deux notions liées : l'approximation est une méthode de compression qui vise également à réduire la redondance, mais en acceptant de perdre une partie de l'information pour être encore plus compacte.

2.5 Robustesse et applications pratiques

Un autre avantage du stockage approximatif, en plus d'être plus compact, est sa robustesse. Il est plus résilient face aux pertes de données et aux perturbations comme les corruptions car il ne cherche pas à restaurer la valeur exacte de l'information. C'est une propriété qui est logique car dans l'approximation, ce sont les valeurs dominantes et la structure générale qui subsistent.

Cette propriété explique pourquoi de nombreuses applications privilégient cette approche. Dans le domaine du traitement d'images, comme avec JPEG, elle permet de réduire considérablement la taille des fichiers tout en préservant l'essentiel de l'information visuelle. Ou bien en séries temporelles, elle facilite l'analyse de tendances sans avoir besoin de conserver l'intégralité des échantillons. De manière générale, dans tous les contextes où l'exactitude l'information n'est pas critique, l'approximation est un bon compromis entre efficacité de stockage et pertinence des données.

3 Application pratique : stockage probabiliste

Maintenant que les bases théoriques ont bien été posées, et que les contraintes et les limites sont connues, nous allons passer à la partie pratique de notre étude. Concrètement, nous allons réaliser une application en Python basée sur la méthode décrite dans l'introduction. L'objectif sera de constater la faisabilité, de comparer son efficacité et sa fidélité avec les méthodes classiques de compression.

3.1 Principe général

L'idée principale est de découper les données en petits blocs de taille pertinente, puis d'établir la loi statistique qui modélise au mieux les informations de ce bloc. On enregistrera les paramètres de cette loi plutôt que les valeurs brutes de la donnée. Nous appliquerons ce fonctionnement à des images : nous découperons donc chaque image en blocs de pixels, puis nous convertirons le système de couleurs RGB en YCbCr afin de décorréler les canaux.

3.2 Décorrélation

Dans une image RGB brute, non compressée et n'ayant subi aucun traitement, les canaux RGB sont fortement corrélés. C'est-à-dire que dans des zones aux couleurs relativement uniformes, les trois canaux ont des motifs similaires. On dit que les canaux sont corrélés et que leurs valeurs ne varient pas indépendamment. Par exemple, si on a un ciel bleu, on aura une forte composante B et une faible R, peu importe le pixel, les valeurs seront toujours liées. On peut donc décorréler les canaux en convertissant les canaux RGB en YCbCr (luminance et chrominance). Y portera les contrastes et la structure de l'image, tandis que Cb et Cr seront plus lisses et contiendront moins d'information.

$$\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix}.$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1.402 \\ 1 & -0.344136 & -0.714136 \\ 1 & 1.772 & 0 \end{pmatrix} \left(\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} - \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix} \right).$$

Voici le procédé de conversion des canaux RGB en YCbCr grâce à la matrice de transformation ci-dessus. Les coefficients utilisés sont ceux que l'on retrouve dans le standard ITU-R BT.601. [2]

3.3 Modèle statistique par bloc

L'image est d'abord découpée en blocs de taille $b \times b$. Cela permet d'avoir une modélisation plus précise et limite la complexité des opérations en travaillant sur des petits blocs à la fois. Ensuite, on définit pour chaque bloc un modèle probabiliste pour la luminance et un autre pour les chrominances.

Luminance (Y). C'est le canal le plus structuré, car il contient l'essentiel de l'information perçue par l'œil humain. Généralement, les valeurs de la luminance ne suivent pas simplement une loi gaussienne. En effet, il y a sur les images des zones claires et des zones d'ombres, que l'on peut assimiler à une distribution bimodale. Cela signifie que l'information globale, la "population", est composée de deux "sous-populations" qui sont les zones claires et sombres, il y aura alors un pic autour de la moyenne de valeurs de chaque zone.

Nous utiliserons un modèle adapté à notre situation qui est le modèle de mélange gaussien (*Gaussian Mixture Model*, GMM). C'est une distribution de probabilité définie comme une combinaison pondérée de plusieurs lois normales :

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x \mid \mu_k, \sigma_k^2),$$

où π_k sont les poids du mélange (tels que $\sum_k \pi_k = 1$), et où chaque composante $\mathcal{N}(x \mid \mu_k, \sigma_k^2)$ est une loi normale d'espérance μ_k et de variance σ_k^2.

Nous utiliserons un mélange à $K = 2$ composantes, ce qui est suffisant pour représenter les composantes claires et sombres de notre bloc. Il restera à estimer le poids π associé à la première composante (le second poids valant $1 - \pi$), les deux moyennes μ_1, μ_2 et les deux variances σ_1^2, σ_2^2. On aura donc 5 paramètres pour la luminance.

Chrominances (Cb et Cr). Pour les canaux de chrominance, on pourra simplement utiliser une modélisation par une loi gaussienne, étant donné que ces canaux ont généralement une variabilité bien plus faible que pour la luminance. Ainsi, il faudra estimer la moyenne et la variance de chaque canal, ce qui nous donne 4 paramètres pour la chrominance.

Limitation du sur-apprentissage. Il se peut que certains blocs de pixels soient suffisamment uniformes pour être correctement représentés par une seule composante dans la mixture gaussienne. Il existe un outil mathématique qui permet de pénaliser ce qu'on appelle le sur-apprentissage, c'est-à-dire le fait d'avoir trop de paramètres qui pourraient être retirés tout en collant aux données : c'est le **critère d'information bayésien** (BIC, *Bayesian Information Criterion*). La formule de calcul du BIC est la suivante :

$$BIC = -2\ln(L) + p\ln(n)$$

Avec L la vraisemblance du modèle (indique si le modèle restitue fidèlement l'information), p le nombre de paramètres libres du modèle et n le nombre d'observations (dans notre cas le nombre de pixels par bloc). Plus la valeur du BIC est faible, plus le modèle est optimal. Il suffit alors de calculer le BIC pour le modèle à 1 puis 2 composantes et de conserver celui dont le BIC est le plus faible. Nous verrons dans l'implémentation en Python que des librairies peuvent gérer ce calcul à notre place.

Bilan. Un bloc de taille $b \times b$ est ainsi représenté par environ **9 paramètres** (5 pour Y et 2 pour chaque canal chromatique), ou éventuellement 2 paramètres pour Y lorsque le mélange gaussien n'est pas nécessaire.

3.4 Méthode appliquée : estimation des paramètres

Nous allons maintenant voir comment sera structuré notre code pour implémenter les concepts vus précédemment. Tout d'abord, il faut commencer par découper en blocs l'image reçue. On commence par choisir une taille de bloc $b \times b$. Des petits blocs seront plus détaillés mais plus coûteux en stockage alors que des gros blocs prennent moins de place mais perdent des détails. Ensuite, on ajoute des marges à l'image pour remplir des bords de sorte à ce que les dimensions de l'image soient des multiples de b. Il y a plusieurs options pour cela, mais il existe des méthodes qui permettent d'ajouter des bords propres. Enfin, on extrait les vecteurs à 1 dimension des blocs $Y_{bloc} = \{y_n\}_{n=1}^{N}$ avec $N = b^2$, et de même Cb_{bloc}, Cr_{bloc}.

Modélisation de la luminance (Y). On peut commencer par calculer la moyenne empirique et l'écart-type : $\bar{y} = \frac{1}{N}\sum_n y_n$ et $\widehat{\sigma}_y = \sqrt{\frac{1}{N}\sum_n (y_n - \bar{y})^2}$. Si on a $\widehat{\sigma}_y$ qui est petit, par exemple inférieur à 1, cela signifie qu'il y a une très faible dispersion des valeurs et que la région est presque uniforme, on peut alors utiliser $K = 1$ sans passer par le GMM. Sinon, on essaie le mélange de deux gaussiennes ($K = 2$). Il faut commencer par obtenir deux centres initiaux pour avoir une initialisation des paramètres de la GMM. Il existe pour cela un algorithme appelé k-moyenne (k-means) [3] qui va établir ces valeurs. Nous ne rentrerons pas dans les détails car il y a des fonctions de bibliothèques Python qui effectueront ces calculs. Grâce à ces valeurs, on pourra ensuite effectuer l'algorithme EM (Espérance-Maximisation) ; cet algorithme a pour but d'estimer les paramètres de notre mélange gaussien. Il sera aussi calculé par une fonction directement en Python. On réitère ensuite un certain nombre de fois les algorithmes pour mieux estimer les valeurs, puis on récupère les paramètres.

Par la suite, on utilise le BIC pour choisir si l'on conserve $K = 1$ ou $K = 2$.

Modélisation des chrominances (Cb, Cr). Pour ces composantes, on simplifiera le calcul en estimant directement μ_{Cb}, σ_{Cb} et μ_{Cr}, σ_{Cr} avec leur moyenne empirique et leur écart-type.

Après cette étape, on dispose pour le bloc d'un vecteur de paramètres réels :

$$(K, \pi, \mu_1, \sigma_1, \mu_2, \sigma_2, \mu_{Cb}, \sigma_{Cb}, \mu_{Cr}, \sigma_{Cr}).$$

3.5 Stockage des paramètres (structure, quantification)

Il reste à établir une structure sur la manière dont seront stockés nos paramètres. On stockera chaque bloc avec un nombre fixe d'octets qui sera de 10. Afin de conserver une bonne précision, on quantifiera μ en linéaire (uint8) et σ en logarithmique. Aussi, nous utiliserons l'orientation petit-boutiste (little-endian) qui est souvent utilisée pour les fichiers sur PC, et pour expliciter le sens de lecture.

Pour la quantification des valeurs, on choisit ces types :

- μ valeurs en $[0, 255] \rightarrow$ `uint8` car une résolution de 1 niveau d'intensité suffira ;
- π poids du mélange $\in [0, 1] \rightarrow$ `uint8` en effectuant l'opération round$(255 \cdot \pi)$. Si $K = 1$ on peut stocker 255 pour que la taille puisse être réduite par la compression sans perte qui suivra ;
- σ (écart-type) $\rightarrow$ `uint8` grâce à une quantification logarithmique sur 1 octet. Pour la réaliser, on introduit $s_{\min}$ et $s_{\max}$ qui sont des bornes pour éviter que s soit nul ou ait une trop grande valeur. On prend $s_{\min} = 0.5$ et $s_{\max} = 128$, et la formule de quantification suivante : $q_\sigma = \left\lfloor 255 \cdot \frac{\log(s) - \log(s_{\min})}{\log(s_{\max}) - \log(s_{\min})} \right\rceil$ et décodage inverse $s = \exp\big(\log(s_{\min}) + \frac{q}{255}(\log(s_{\max}) - \log(s_{\min}))\big)$.
- **flags** $\rightarrow$ `uint8` indique K (1 ou 2), avec d'autres bits réservés inutilisés pour le moment.

L'ordre est le suivant, avec chaque crochet qui représente un octet :

$$\underbrace{[\text{flags}]}_{\text{K=1 ou 2}}\ [\pi]\ [\mu_1]\ [\sigma_1]\ [\mu_2]\ [\sigma_2]\ [\mu_{Cb}]\ [\sigma_{Cb}]\ [\mu_{Cr}]\ [\sigma_{Cr}]$$

Si $K = 1$, π est ignorée (on met sa valeur à 255), $\mu_2 = \mu_1, \sigma_2 = \sigma_1$. On a aussi un en-tête global au début du fichier qui contient : `width (W) uint16, height (H) uint16, block_size (B) uint8, seed (I) uint32, num_blocks (I) uint32` soit 13 octets et la déclaration du format little-endian.

La taille totale d'une image avant sa compression post-encodage pour une image $W \times H$ et bloc b peut être calculée simplement. Le nombre de blocs vaut $\approx \frac{W \times H}{b^2}$ et la taille d'un bloc est de 10 octets. On a donc $Taille(octets) = 13 + 10 \times \frac{W \times H}{b^2}$. On pourra ensuite compresser cette suite de bytes avec un compresseur générique sans pertes (gzip, lz4) pour réduire encore plus la taille.

Il est important de noter que l'on sait d'ores et déjà que notre codec sera moins efficace que JPEG. En effet, JPEG applique déjà la méthode optimale pour compresser

les données, et on peut calculer que pour une image de 1000×1000 pixels, notre codec sortira avant compression supplémentaire un fichier de taille $13 + 10 \times \frac{1000 \times 1000}{3^2} \approx 1,1$ Mo, ce qui est plus élevé qu'une image équivalente en JPEG.

3.6 Reconstruction de l'image

La reconstruction a pour objectif de reconstituer l'image originale à partir des paramètres quantifiés stockés pour chaque bloc. Cette étape est essentielle car elle permet de visualiser le résultat du codage probabiliste et d'évaluer la qualité de la compression.

Le décodage se fait bloc par bloc. Pour chaque bloc, on commence par lire les paramètres stockés : la proportion π associée à chaque composante de la luminance, les moyennes μ_1, μ_2 et écarts-types σ_1, σ_2 pour Y, ainsi que les moyennes et écarts-types des canaux de chrominance $\mu_{Cb}, \sigma_{Cb}, \mu_{Cr}, \sigma_{Cr}$.

Ensuite, on reconstruit les pixels du bloc de manière stochastique. Pour chaque pixel, on commence par s'occuper de la composante de luminance k : si le bloc possède deux composantes ($K = 2$), on effectue un tirage aléatoire selon une distribution de Bernoulli de paramètre π afin de choisir quelle composante sera utilisée pour ce pixel ; si la luminance n'a qu'une composante ($K = 1$), on utilise directement cette composante. Une fois la composante choisie, la valeur de luminance Y est tirée selon une loi normale de moyenne μ_k et d'écart-type σ_k. De la même manière, les canaux de chrominance Cb et Cr sont générés par tirage normal avec leurs paramètres respectifs. Chaque valeur obtenue est ensuite limitée à l'intervalle $[0, 255]$ et arrondie pour former un entier sur 8 bits.

Une fois toutes les valeurs de Y, Cb et Cr générées pour le bloc, elles sont réassemblées pour reformer le bloc en espace YCbCr. Le processus est répété bloc par bloc jusqu'à reconstituer l'image entière. Enfin, la dernière étape consiste à convertir l'image du format YCbCr en RGB afin de l'afficher ou de l'enregistrer.

Optionnellement, on pourra appliquer un filtre en post-traitement pour lisser l'image, par exemple avec un filtre bilatéral ou guidé.

Pour garantir la reproductibilité, une graine pseudo-aléatoire globale stockée dans l'entête est utilisée pour initialiser les tirages aléatoires. Cela permet d'obtenir exactement la même reconstruction si le fichier est décodé plusieurs fois avec la même graine. De cette manière, il nous suffit simplement de changer la graine pour obtenir une reconstruction différente de l'image, qui serait éventuellement plus fidèle.

3.7 Implémentation en Python

3.7.1 Bibliothèques utilisées

- **numpy** : manipulation efficace des tableaux, calculs statistiques vectorisés.
- **opencv-python (cv2)** : lecture/écriture d'images, conversion de l'espace colorimétrique (RGB $\leftrightarrow$ YCrCb), padding.
- **scikit-learn (GaussianMixture)** : implémentation robuste et optimisée d'EM pour les mélanges gaussiens, fournit aussi le BIC pour la sélection de modèle.

— **struct, io** : sérialisation binaire simple et portable.

3.7.2 Constantes et helpers de quantification

La première chose que l'on effectue est de définir des fonctions dites "helpers", qui sont des fonctions utilitaire utilisées dans le reste du code pour nous faciliter le codage. Ici, il s'agit des fonctions de quantification et de dé-quantification. Pour les paramètres μ et π, on applique une quantification linéaire, et pour l'écart-type σ on établit une quantification logarithmique comme décrit plus tôt, ce qui permet de mieux représenter les petites valeurs.

Listing 1 – Helpers de quantification

```
# quantification lineaire
def q_uint8(x, lo=0.0, hi=255.0):
    x = np.clip(x, lo, hi)
    return np.uint8(np.round((x - lo) / (hi - lo) * 255.0))

def dq_uint8(q, lo=0.0, hi=255.0):
    return (float(q) / 255.0) * (hi - lo) + lo

# quantification logarithmique
def q_sigma_log(sigma, lo=1e-3, hi=255.0):
    x = np.clip(sigma, lo, hi)
    logx = np.log(x)
    return np.uint8(np.round((logx - np.log(lo)) / (np.log(hi) -
        np.log(lo)) * 255.0))

def dq_sigma_log(q, lo=1e-3, hi=255.0):
    logx = (float(q) / 255.0) * (np.log(hi) - np.log(lo)) + np.log(lo)
    return np.exp(logx)
```

Ces fonctions sont l'application directe des formules de quantification établies dans la section 3.5.

3.7.3 Encodage par bloc

La fonction principale d'encodage, appelée `encode_image_to_bytes`, gère la transformation d'une image RGB en un flux binaire. Elle commence par effectuer un éventuel padding (ajouter des marges si nécessaire) et par convertir l'image de l'espace RGB vers l'espace YCrCb, en respectant l'ordre des canaux imposé par OpenCV, à savoir Y, Cr puis Cb.

Une fois cette préparation effectuée, le traitement se fait bloc par bloc. Pour chaque bloc, la fonction calcule les statistiques de luminance et décide du nombre de composantes K à utiliser. Si deux composantes semblent nécessaires, un modèle de mélange gaussien (GMM) est ajusté avec l'algorithme EM. Le choix entre $K = 1$ et $K = 2$ est ensuite validé en calculant le critère d'information bayésien (BIC), qui pénalise la complexité excessive et permet d'éviter le surapprentissage. Afin de rendre le traitement

plus stable, on introduit certaines régularisations, par exemple une variance minimale qui empêche d'obtenir un écart-type nul. Enfin, les paramètres estimés sont quantifiés à l'aide des helpers définis précédemment, puis rassemblés en un bloc de 10 octets qui est ajouté au flux binaire.

Listing 2 – Encodage par bloc (padding, RGB verts YCrCb, GMM, EM, BIC)

```
def encode_image_to_bytes(img_rgb, block_size=8, seed=1234):
    import io, struct, cv2, numpy as np
    from sklearn.mixture import GaussianMixture

    rng = np.random.default_rng(seed)

    # ajout du padding si necessaire
    h, w, _ = img_rgb.shape
    pad_h = (block_size - h % block_size) % block_size
    pad_w = (block_size - w % block_size) % block_size
    if pad_h > 0 or pad_w > 0:
        img_rgb = cv2.copyMakeBorder(img_rgb, 0, pad_h, 0, pad_w,
                                     cv2.BORDER_REPLICATE)

    # conversion RGB -> YCrCb (ordre OpenCV : Y,Cr,Cb)
    img_ycc = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2YCrCb)
    Yfull, Crfull, Cbfull = cv2.split(img_ycc)

    H, W = Yfull.shape
    blocks = []

    # constantes de regularisation
    MIN_STD = 0.5

    for by in range(0, H, block_size):
        for bx in range(0, W, block_size):
            Y = Yfull[by:by+block_size,
                bx:bx+block_size].astype(np.float32)
            Cb = Cbfull[by:by+block_size,
                bx:bx+block_size].astype(np.float32)
            Cr = Crfull[by:by+block_size,
                bx:bx+block_size].astype(np.float32)

            # statistiques simples
            muCb, sigCb = float(Cb.mean()), float(Cb.std())
            muCr, sigCr = float(Cr.mean()), float(Cr.std())

            # decision du nombre de composantes K
            yvec = Y.flatten().reshape(-1, 1)
            best_K, best_bic, best_gmm = 1, np.inf, None
            for K in [1, 2]:
                try:
                    gmm = GaussianMixture(n_components=K,
                        covariance_type='diag',
                                          random_state=seed)
```

```
                gmm.fit(yvec)
                bic = gmm.bic(yvec)
                if bic < best_bic:
                    best_K, best_bic, best_gmm = K, bic, gmm
            except Exception:
                continue

        if best_gmm is None:
            # choix d'une gaussienne unique
            muY, sigY = float(Y.mean()), max(float(Y.std()), MIN_STD)
            pis = [1.0]
            mus = [muY]
            sigs = [sigY]
        else:
            pis = best_gmm.weights_
            mus = best_gmm.means_.flatten()
            sigs = np.sqrt(best_gmm.covariances_.flatten())
            sigs = np.maximum(sigs, MIN_STD)

        # quantification
        if best_K == 1:
            q_flags = 0
            q_pi = q_uint8(1.0, 0.0, 1.0)
            q_mu1 = q_uint8(mus[0], 0.0, 255.0)
            q_sig1 = q_sigma_log(sigs[0])
            q_mu2 = q_uint8(0.0, 0.0, 255.0)
            q_sig2 = q_sigma_log(MIN_STD)
        else:
            q_flags = 1
            q_pi = q_uint8(pis[0], 0.0, 1.0)
            q_mu1 = q_uint8(mus[0], 0.0, 255.0)
            q_sig1 = q_sigma_log(sigs[0])
            q_mu2 = q_uint8(mus[1], 0.0, 255.0)
            q_sig2 = q_sigma_log(sigs[1])

        q_muCb = q_uint8(muCb, 0.0, 255.0)
        q_sigCb = q_sigma_log(sigCb)
        q_muCr = q_uint8(muCr, 0.0, 255.0)
        q_sigCr = q_sigma_log(sigCr)

        blocks.append(bytes([
            q_flags, q_pi, q_mu1, q_sig1, q_mu2, q_sig2,
            q_muCb, q_sigCb, q_muCr, q_sigCr
        ]))

# ecriture en flux binaire avec entete
buf = io.BytesIO()
buf.write(struct.pack('<IIi', W, H, seed))
for b in blocks:
    buf.write(b)
```

```
    return buf.getvalue()
```

3.7.4 Décodage et reconstruction

La phase de décodage repose sur la fonction `decode_image_from_bytes`, qui effectue le chemin inverse de l'encodage. Le programme commence par lire l'en-tête du fichier et alloue des tableaux destinés à stocker les canaux Y, Cb et Cr. Pour chaque bloc, il lit les dix octets codés, les dé-quantifie en utilisant les fonctions helpers, tire les valeurs aléatoires à partir des lois récupérées puis génère les pixels correspondants. Une fois tous les blocs traités, les trois canaux Y, Cr et Cb sont recombinés pour former une image complète au format YCrCb, qui est ensuite convertie en RGB pour obtenir l'image finale.

Listing 3 – Décodage et reconstruction (quantification, tirage, conversion)

```
def decode_image_from_bytes(buf):
    import io, struct, numpy as np, cv2

    # lecture de l'entete : largeur, hauteur, seed
    W, H, seed = struct.unpack('<IIi', buf.read(12))
    rng = np.random.default_rng(seed)

    block_size = 8
    # allocation des canaux
    Yfull = np.zeros((H, W), dtype=np.float32)
    Cbfull = np.zeros((H, W), dtype=np.float32)
    Crfull = np.zeros((H, W), dtype=np.float32)

    for by in range(0, H, block_size):
        for bx in range(0, W, block_size):
            # lecture des 10 octets du bloc
            q = buf.read(10)
            q_flags, q_pi, q_mu1, q_sig1, q_mu2, q_sig2, \
            q_muCb, q_sigCb, q_muCr, q_sigCr = struct.unpack('10B', q)

            # dequantification
            K = 2 if q_flags else 1
            pi = dq_uint8(q_pi, 0.0, 1.0)
            mu1 = dq_uint8(q_mu1, 0.0, 255.0)
            sig1 = dq_sigma_log(q_sig1)
            mu2 = dq_uint8(q_mu2, 0.0, 255.0)
            sig2 = dq_sigma_log(q_sig2)
            muCb = dq_uint8(q_muCb, 0.0, 255.0)
            sigCb = dq_sigma_log(q_sigCb)
            muCr = dq_uint8(q_muCr, 0.0, 255.0)
            sigCr = dq_sigma_log(q_sigCr)

            # generation stochastique de Y
            yblock = np.zeros((block_size, block_size), dtype=np.float32)
            if K == 1:
```

```
                yblock[:, :] = rng.normal(mu1, sig1, (block_size,
                    block_size))
            else:
                mask = rng.random((block_size, block_size)) < pi
                yblock[mask] = rng.normal(mu1, sig1,
                    np.count_nonzero(mask))
                yblock[~mask] = rng.normal(mu2, sig2,
                    np.count_nonzero(~mask))

            # generation stochastique Cb et Cr
            cbblock = rng.normal(muCb, sigCb, (block_size, block_size))
            crblock = rng.normal(muCr, sigCr, (block_size, block_size))

            # insertion dans les canaux
            Yfull[by:by+block_size, bx:bx+block_size] = yblock
            Cbfull[by:by+block_size, bx:bx+block_size] = cbblock
            Crfull[by:by+block_size, bx:bx+block_size] = crblock

    # recombinaison des canaux et conversion en RGB
    img_ycc = cv2.merge([Yfull, Crfull, Cbfull])
    img_rgb = cv2.cvtColor(img_ycc.astype(np.uint8),
        cv2.COLOR_YCrCb2RGB)
    return img_rgb
```

3.7.5 Entrées, sorties et reproductibilité

En plus de ces fonctions principales, nous avons quelques fonctions utilitaires de base, notamment pour gérer les entrées et sorties : chemin d'une image à convertir, emplacement de stockage du fichier encodé, chemin d'une image à décoder. Pour la reproductibilité, on ajoute la graine de tirage aléatoire à l'entête, ainsi, chaque tirage sera pseudo-aléatoire et donnera les mêmes résultats si la même graine est utilisée.

Listing 4 – Fonctions I/O de base, interface

```
import cv2
import io
import os

def main():
    print("----- Encodage d'image probabiliste -----")
    print("1 : Encoder une image")
    print("2 : Décoder un flux binaire")
    choice = input("Choisissez une option : ").strip()

    if choice == "1":
        input_path = input("Chemin de l'image a encoder : ").strip()
        output_path = input("Chemin de sortie pour le fichier encodé :
            ").strip()
        seed_input = input("Entrez une graine pour la reproductibilité
            (par défaut 12345) : ").strip()
```

```
        seed = int(seed_input) if seed_input else 12345

        img_rgb = cv2.imread(input_path)
        if img_rgb is None:
            print(f"Impossible de lire l'image : {input_path}")
            return

        encoded_bytes = encode_image_to_bytes(img_rgb, block_size=8,
            seed=seed)
        with open(output_path, "wb") as f:
            f.write(encoded_bytes)

        print(f"Image encodée et sauvegardée dans : {output_path}")

    elif choice == "2":
        input_path = input("Chemin du fichier binaire a décoder :
            ").strip()
        output_path = input("Chemin de sortie pour l'image reconstruite
            : ").strip()

        if not os.path.isfile(input_path):
            print(f"Fichier introuvable : {input_path}")
            return

        with open(input_path, "rb") as f:
            buf = io.BytesIO(f.read())

        img_reconstructed = decode_image_from_bytes(buf)
        cv2.imwrite(output_path, img_reconstructed)
        print(f"Image reconstruite enregistrée dans : {output_path}")

    else:
        print("Option invalide. Veuillez choisir 1 ou 2.")

if __name__ == "__main__":
    main()
```

3.7.6 Choix de conception

Voici quelques précisions sur certains choix effectués dans le code, basés sur des conseils trouvés sur Internet concernant les bibliothèques utilisées. La constante `MIN_STD = 0.5` assure le bon fonctionnement de l'algorithme EM et garantit que les échantillons générés restent cohérents. L'option `covariance_type='diag'` choisie pour le GMM devrait assurer une meilleure stabilité numérique.

3.8 Comparaison des résultats

Pour tester l'efficacité de l'application, nous utiliserons une image d'un paysage trouvée sur Internet. C'est une photo d'un lac avec un coucher de soleil, des montagnes en arrière-plan et des plantes au premier plan. Cette photo permettra donc d'évaluer la qualité selon différentes zones : un ciel plutôt homogène avec un dégradé lisse, la restitution du bruit visuel causé par les vagues et la finesse des détails des plantes au premier plan.

FIGURE 2 – Photographie d'un paysage (Auteur : Bruno Monginoux)

Cette photographie est au format JPEG. Sa taille est de 157 060 octets avec des dimensions de 375 par 500 pixels. Nous effectuerons deux tests, chacun avec une finesse différente. Le premier aura une taille de bloc de 8 pixels et le second une taille de 4 pixels.

3.8.1 Blocs de 8 pixels

FIGURE 3 – Photographie du paysage après encodage puis décodage

On constate que le résultat n'est pas très concluant. On discerne l'image mais elle est floue et les blocs sont trop marqués. Il y a également une forte impression de bruit. Cependant, les sources des problèmes sont connues :

- La génération est purement stochastique, on ne conserve donc pas assez d'informations spatiales et structurelles de l'image originale, c'est pour cela que les séparations entre les blocs sont marquées ;
- La taille des blocs peut être trop grande, mais diminuer la taille des blocs va fortement augmenter la taille du fichier ;
- La quantification peut être trop imprécise ;
- La modélisation peut être insuffisante et il faudrait plus de composantes.

Par ailleurs, la taille du fichier encodé est de 29 622 octets, ce qui représente une réduction de plus de 80% de la taille de l'image. Mais malheureusement, l'image est jugée de trop mauvaise qualité pour considérer que cette réduction soit acceptable.

Comme nous ne souhaitons pas augmenter la taille du fichier, nous allons mettre en place une fonction de post-traitement qui tentera d'améliorer l'image en appliquant des filtres. La fonction implémentée est la suivante :

Listing 5 – Fonction de post-traitement de l'image décodée

```
def postprocess(img_rgb):
    img_filtered = cv2.bilateralFilter(img_rgb, d=10, sigmaColor=75,
        sigmaSpace=75)

    img_median = cv2.medianBlur(img_filtered, 5)
```

```
    img_smooth = cv2.GaussianBlur(img_median, (3, 3), 0.5)

    gaussian_blur = cv2.GaussianBlur(img_smooth, (3, 3), 1.0)
    img_sharp = cv2.addWeighted(img_smooth, 1.3, gaussian_blur, -0.3, 0)

    return np.clip(img_sharp, 0, 255).astype(np.uint8)

# dans la fonction decode, on ajoute une variable pp qui vaut 0 par
    defaut
def decode_image_from_bytes(buf):
    pp = 0
    # ...

    if pp:
        return postprocess(img_rgb)
    else:
        return img_rgb
```

Après application du post-traitement, on obtient l'image ci-dessous, légèrement améliorée :

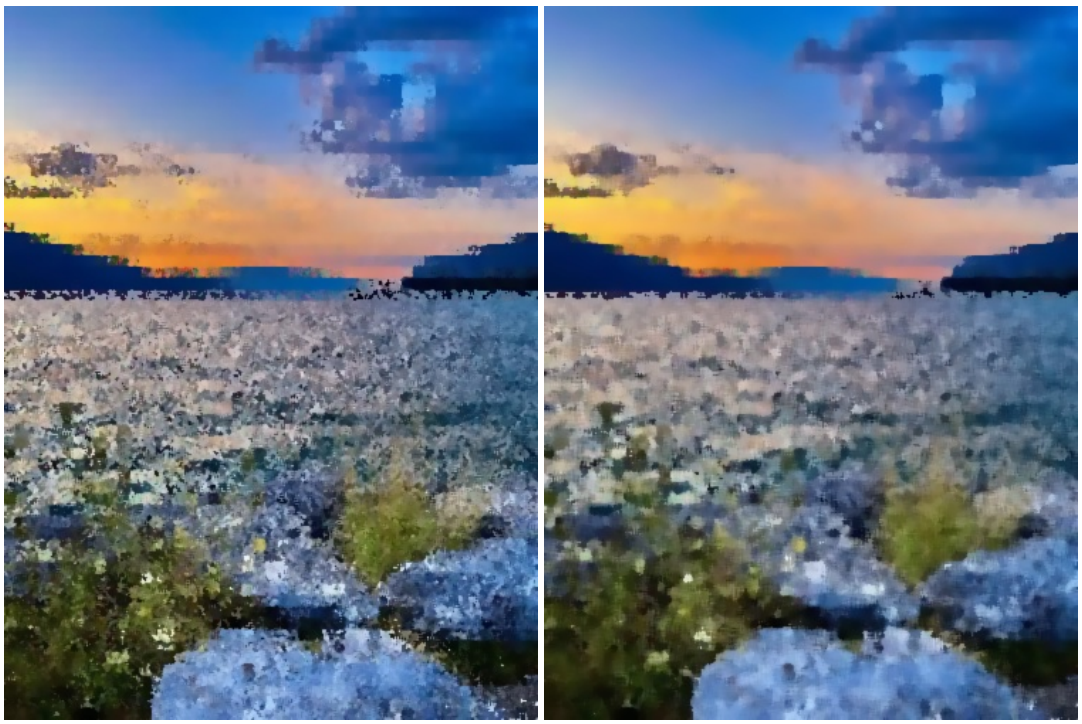


FIGURE 4 – Image avant et après post-traitement

L'image est plus lisse et de meilleure qualité, mais elle semble encore trop pixelisée et crénelée. C'est pourquoi nous allons réessayer l'encodage avec une taille de bloc plus petite.

3.8.2 Blocs de 4 pixels

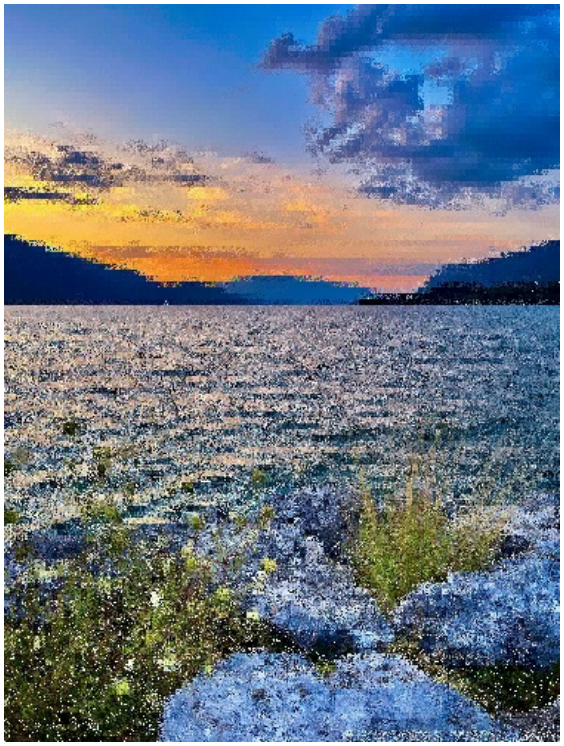


FIGURE 5 – Image avec une taille de blocs de 4x4 pixels

L'image semble bien plus détaillée et de meilleure qualité. Cependant, la taille a considérablement augmenté, car l'image encodée fait désormais 117 512 octets, ce qui correspond à une réduction de 25%. On peut maintenant appliquer le post-traitement et comparer avec l'image originale :

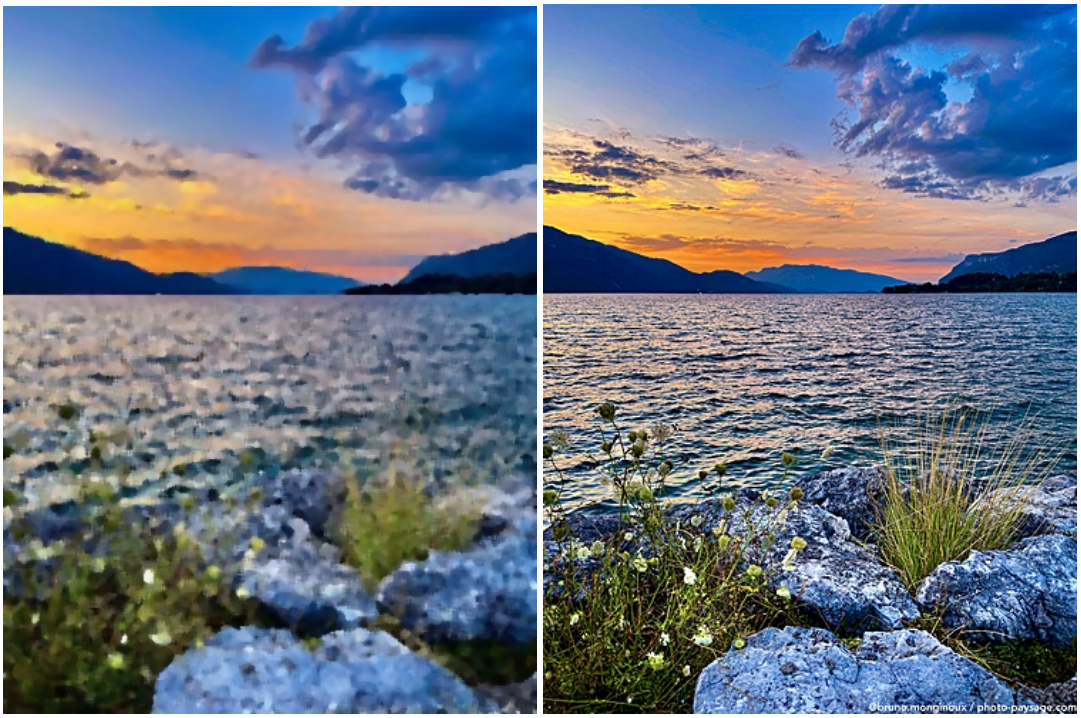


FIGURE 6 – Image décodée post-traitée comparée à l'originale

L'image obtenue est légèrement plus floue que l'image originale mais les aberrations ont disparu. Le ciel et les montagnes sont bien démarqués, l'eau est distinguable, mais les plantes ne sont pas bien discernables.

4 Conclusion et perspectives

L'objectif de ce projet était d'étudier les différentes méthodes de compression avec et sans pertes utilisées de nos jours, et de mieux comprendre les mathématiques et les statistiques qui se cachent derrière pour mieux cerner les enjeux et limites. Cette étude a permis de mieux appréhender la partie pratique du projet qui consistait à réaliser un codec, donc un programme qui permet l'encodage et le décodage, afin de compresser des données avec une approche atypique. Nous avons pu apprendre les moyens actuels d'encodage, notamment pour les images, ainsi que la différence et l'utilité de différents espaces colorimétriques.

Les résultats obtenus sont plutôt satisfaisants. L'image est largement reconnaissable et on constate bien une compression qui varie de 25% à plus de 80% dans le cas où un compromis est fait sur la netteté. Mais cette réduction est à nuancer : en effet, il y a de la perte de qualité non négligeable sur la photo, et il existe très probablement des codecs plus efficaces qui, pour la même qualité, fournissent une meilleure réduction.

Il y a plusieurs pistes d'améliorations pour le codec. D'abord, il est possible d'ajouter une compression sans perte une fois l'encodage réalisé, ce qui pourrait encore réduire la taille de 5 à 20% selon l'efficacité de la méthode et les données à compresser. On pourrait aussi explorer d'autres distributions statistiques plus complexes et plus représentatives, voire utiliser d'autres lois que la loi gaussienne en fonction des données

et de la loi qui colle le mieux. Enfin, on pourrait avoir une taille de blocs dynamique selon les régions de l'image, en fonction de la complexité locale des données. En termes de performance, il est possible d'optimiser les algorithmes d'encodage ou d'exploiter du calcul parallèle ou le GPU.

Ce projet était intéressant et a apporté de réelles connaissances scientifiques, particulièrement en mathématiques, statistiques, mais aussi en algorithmique et dans l'utilisation de certaines bibliothèques Python.Il a apporté de bonnes bases en ingénierie des données et en traitement de data. On pourrait par la suite tester le codec sur d'autres types de données, par exemple audio, et peaufiner le programme.

5 Code source

Listing 6 – Code source complet du programme

```
import cv2
import numpy as np
from sklearn.mixture import GaussianMixture
import struct
import io
import os

def q_uint8(x, lo=0.0, hi=255.0):
    x = np.clip(x, lo, hi)
    return np.uint8(np.round((x - lo) / (hi - lo) * 255.0))

def dq_uint8(q, lo=0.0, hi=255.0):
    return (float(q) / 255.0) * (hi - lo) + lo

def q_sigma_log(sigma, lo=1e-3, hi=255.0):
    x = np.clip(sigma, lo, hi)
    logx = np.log(x)
    return np.uint8(np.round((logx - np.log(lo)) / (np.log(hi) -
        np.log(lo)) * 255.0))

def dq_sigma_log(q, lo=1e-3, hi=255.0):
    logx = (float(q) / 255.0) * (np.log(hi) - np.log(lo)) + np.log(lo)
    return np.exp(logx)

def encode_image_to_bytes(img_rgb, block_size=8, seed=12345):

    # padding
    h, w, _ = img_rgb.shape
    pad_h = (block_size - h % block_size) % block_size
    pad_w = (block_size - w % block_size) % block_size
    if pad_h > 0 or pad_w > 0:
        img_rgb = cv2.copyMakeBorder(img_rgb, 0, pad_h, 0, pad_w,
            cv2.BORDER_REPLICATE)

    # conversion RGB -> YCrCb
```

```
img_ycc = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2YCrCb)
Yfull, Crfull, Cbfull = cv2.split(img_ycc)

H, W = Yfull.shape
blocks = []

MIN_STD = 0.5

for by in range(0, H, block_size):
    for bx in range(0, W, block_size):
        print("Traitement du bloc X =", bx, " Y =", by)
        Y = Yfull[by:by+block_size,
            bx:bx+block_size].astype(np.float32)
        Cb = Cbfull[by:by+block_size,
            bx:bx+block_size].astype(np.float32)
        Cr = Crfull[by:by+block_size,
            bx:bx+block_size].astype(np.float32)

        muCb, sigCb = float(Cb.mean()), float(Cb.std())
        muCr, sigCr = float(Cr.mean()), float(Cr.std())

        # Décision du nombre de composantes K
        yvec = Y.flatten().reshape(-1, 1)
        best_K, best_bic, best_gmm = 1, np.inf, None
        for K in [1, 2]:
            try:
                gmm = GaussianMixture(n_components=K,
                    covariance_type='diag', random_state=seed)
                gmm.fit(yvec)
                bic = gmm.bic(yvec)
                if bic < best_bic:
                    best_K, best_bic, best_gmm = K, bic, gmm
            except Exception:
                continue

        if best_gmm is None:
            muY, sigY = float(Y.mean()), max(float(Y.std()), MIN_STD)
            pis = [1.0]
            mus = [muY]
            sigs = [sigY]
        else:
            pis = best_gmm.weights_
            mus = best_gmm.means_.flatten()
            sigs = np.sqrt(best_gmm.covariances_.flatten())
            sigs = np.maximum(sigs, MIN_STD)

        # Quantification
        if best_K == 1:
            q_flags = 0
            q_pi = q_uint8(1.0, 0.0, 1.0)
            q_mu1 = q_uint8(mus[0], 0.0, 255.0)
```

```
                q_sig1 = q_sigma_log(sigs[0])
                q_mu2 = q_uint8(0.0, 0.0, 255.0)
                q_sig2 = q_sigma_log(MIN_STD)
            else:
                q_flags = 1
                q_pi = q_uint8(pis[0], 0.0, 1.0)
                q_mu1 = q_uint8(mus[0], 0.0, 255.0)
                q_sig1 = q_sigma_log(sigs[0])
                q_mu2 = q_uint8(mus[1], 0.0, 255.0)
                q_sig2 = q_sigma_log(sigs[1])

            q_muCb = q_uint8(muCb, 0.0, 255.0)
            q_sigCb = q_sigma_log(sigCb)
            q_muCr = q_uint8(muCr, 0.0, 255.0)
            q_sigCr = q_sigma_log(sigCr)

            blocks.append(bytes([
                q_flags, q_pi, q_mu1, q_sig1, q_mu2, q_sig2,
                q_muCb, q_sigCb, q_muCr, q_sigCr
            ]))

    # écriture en flux binaire
    buf = io.BytesIO()
    buf.write(struct.pack('<IIi', W, H, seed))
    for b in blocks:
        buf.write(b)

    return buf.getvalue()


def postprocess(img_rgb):
    img_filtered = cv2.bilateralFilter(img_rgb, d=10, sigmaColor=75,
        sigmaSpace=75)

    img_median = cv2.medianBlur(img_filtered, 5)

    img_smooth = cv2.GaussianBlur(img_median, (3, 3), 0.5)

    gaussian_blur = cv2.GaussianBlur(img_smooth, (3, 3), 1.0)
    img_sharp = cv2.addWeighted(img_smooth, 1.3, gaussian_blur, -0.3, 0)

    return np.clip(img_sharp, 0, 255).astype(np.uint8)


def decode_image_from_bytes(buf):
    W, H, seed = struct.unpack('<IIi', buf.read(12))
    rng = np.random.default_rng(seed)

    pp = 0
    block_size = 8
    Yfull = np.zeros((H, W), dtype=np.float32)
```

```
    Cbfull = np.zeros((H, W), dtype=np.float32)
    Crfull = np.zeros((H, W), dtype=np.float32)

    for by in range(0, H, block_size):
        for bx in range(0, W, block_size):
            q = buf.read(10)
            q_flags, q_pi, q_mu1, q_sig1, q_mu2, q_sig2, \
            q_muCb, q_sigCb, q_muCr, q_sigCr = struct.unpack('10B', q)

            K = 2 if q_flags else 1
            pi = dq_uint8(q_pi, 0.0, 1.0)
            mu1 = dq_uint8(q_mu1, 0.0, 255.0)
            sig1 = dq_sigma_log(q_sig1)
            mu2 = dq_uint8(q_mu2, 0.0, 255.0)
            sig2 = dq_sigma_log(q_sig2)
            muCb = dq_uint8(q_muCb, 0.0, 255.0)
            sigCb = dq_sigma_log(q_sigCb)
            muCr = dq_uint8(q_muCr, 0.0, 255.0)
            sigCr = dq_sigma_log(q_sigCr)

            # génération stochastique Y
            yblock = np.zeros((block_size, block_size), dtype=np.float32)
            if K == 1:
                yblock[:, :] = rng.normal(mu1, sig1, (block_size,
                    block_size))
            else:
                mask = rng.random((block_size, block_size)) < pi
                yblock[mask] = rng.normal(mu1, sig1,
                    np.count_nonzero(mask))
                yblock[~mask] = rng.normal(mu2, sig2,
                    np.count_nonzero(~mask))

            # génération stochastique Cb et Cr
            cbblock = rng.normal(muCb, sigCb, (block_size, block_size))
            crblock = rng.normal(muCr, sigCr, (block_size, block_size))

            Yfull[by:by+block_size, bx:bx+block_size] = yblock
            Cbfull[by:by+block_size, bx:bx+block_size] = cbblock
            Crfull[by:by+block_size, bx:bx+block_size] = crblock

    img_ycc = cv2.merge([Yfull, Crfull, Cbfull])
    img_rgb = cv2.cvtColor(img_ycc.astype(np.uint8),
        cv2.COLOR_YCrCb2RGB)

    if pp:
        return postprocess(img_rgb)
    else:
        return img_rgb


def main():
```

```
    print("--- Encodage d'image probabiliste ===")
    print("1 : Encoder une image")
    print("2 : Décoder un flux binaire")
    choice = input("Choisissez une option (1 ou 2) : ").strip()

    if choice == "1":
        input_path = input("Chemin de l'image àencoder : ").strip()
        output_path = input("Chemin de sortie pour le fichier encodé :
            ").strip()
        seed_input = input("Entrez une graine pour la reproductibilité
            (par défaut 12345) : ").strip()
        seed = int(seed_input) if seed_input else 12345

        img_rgb = cv2.imread(input_path)
        if img_rgb is None:
            print(f"Impossible de lire l'image : {input_path}")
            return

        encoded_bytes = encode_image_to_bytes(img_rgb, block_size=8,
            seed=seed)
        with open(output_path, "wb") as f:
            f.write(encoded_bytes)

        print(f"Image encodée et sauvegardée dans : {output_path}")

    elif choice == "2":
        input_path = input("Chemin du fichier binaire àdécoder :
            ").strip()
        output_path = input("Chemin de sortie pour l'image reconstruite
            : ").strip()

        if not os.path.isfile(input_path):
            print(f"Fichier introuvable : {input_path}")
            return

        with open(input_path, "rb") as f:
            buf = io.BytesIO(f.read())

        img_reconstructed = decode_image_from_bytes(buf)
        cv2.imwrite(output_path, img_reconstructed)
        print(f"Image reconstruite enregistrée dans : {output_path}")

    else:
        print("Option invalide. Veuillez choisir 1 ou 2.")


if __name__ == "__main__":
    main()
```

Références

[1] Edouard Mathieu. The price of computer storage has fallen exponentially since the 1950s. https://ourworldindata.org/data-insights/the-price-of-computer-storage-has-fallen-exponentially-since-the-1950s. Consulté le 3 août 2025.

[2] Wikipedia. Ycbcr. https://en.wikipedia.org/wiki/YCbCr. Consulté le 7 août 2025.

[3] Vanna Winland Eda Kavlakoglu. Qu'est-ce que le clustering k-means ? https://www.ibm.com/fr-fr/think/topics/k-means-clustering#:~:text=Le%20clustering%20k%2Dmeans%20est,utilis%C3%A9es%20dans%20le%20machine%20learning. Consulté le 14 août 2025.