Les courbes remplissant l'espace et leur applications dans le traitement d'images

Sami Ennedoui
Numéro SCEI: 15486

2024-2025

Sommaire

Sommaire

Histoire

- 1878 : George Cantor a découvert l'existence d'une bijection entre toutes deux variétés lisses de dimensions finies.
- 1879 : E. Netto a montré qu'une telle application ne peut pas être continue.
- 1890 : G. Peano a montré l'existence d'une telle application qui est continue mais qui n'est pas injective.

. Hans Sagan, *Space-filling Curves*, DOI :10.1007/978-1-4612-0871-6

Sommaire

Sommaire

Définition 1

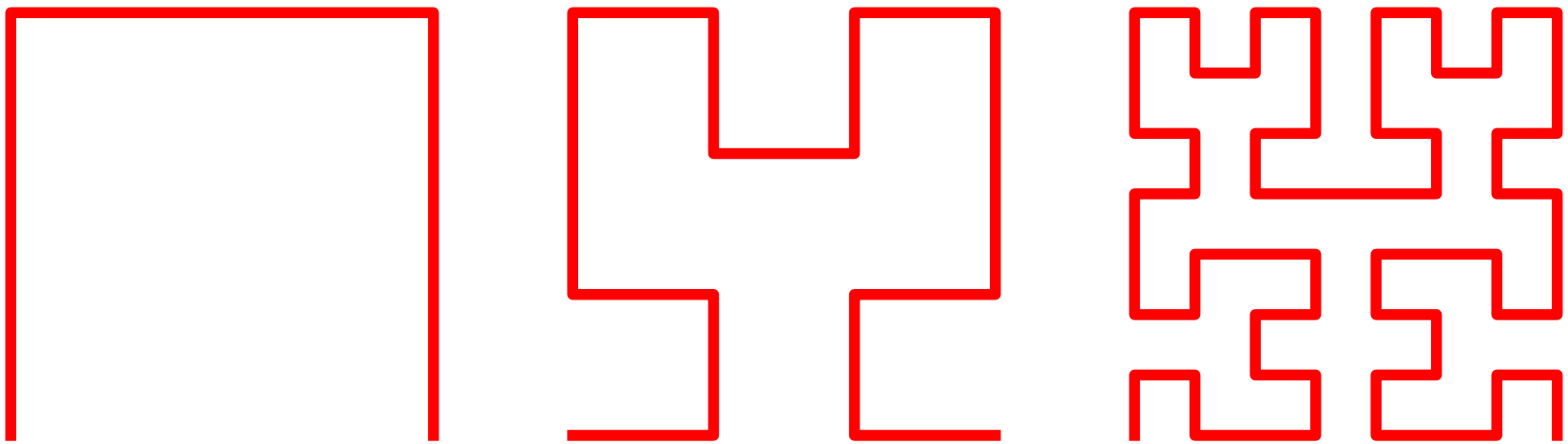
Soit $f : E \to F$ une fonction **continue**, où E et F sont des ensembles. $f(E)$ est alors dit une courbe.

Définition 2

Une courbe est dite **remplissant l'espace** si sa mesure de Peano-Jordan (ex. aire, volume, ...) est **non nulle**.

Sommaire

Courbes de Hilbert d'ordres 1, 2 et 3



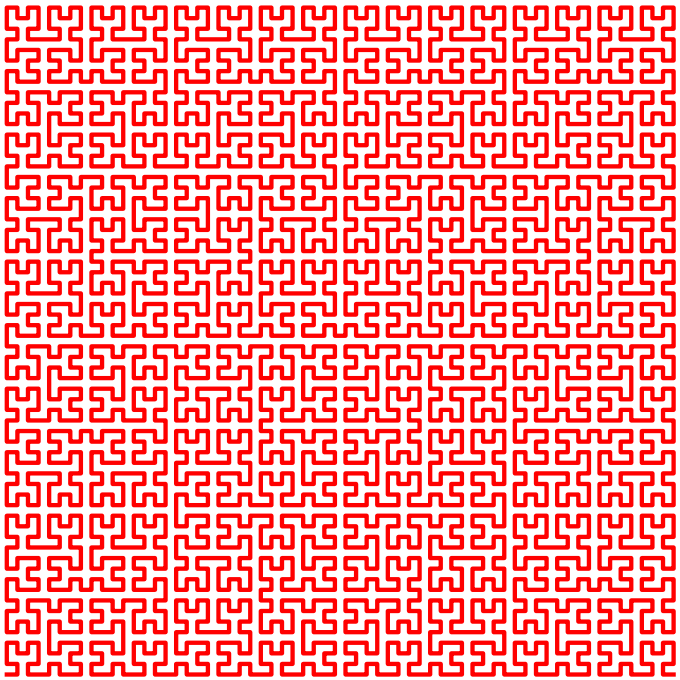
(a) Courbe de Hilbert d'ordre 1

(b) Courbe de Hilbert d'ordre 2

(c) Courbe de Hilbert d'ordre 3

Figure 1 – Évolution des courbes de Hilbert en fonction de leur ordre

Courbes de Hilbert pour des ordres plus élevés

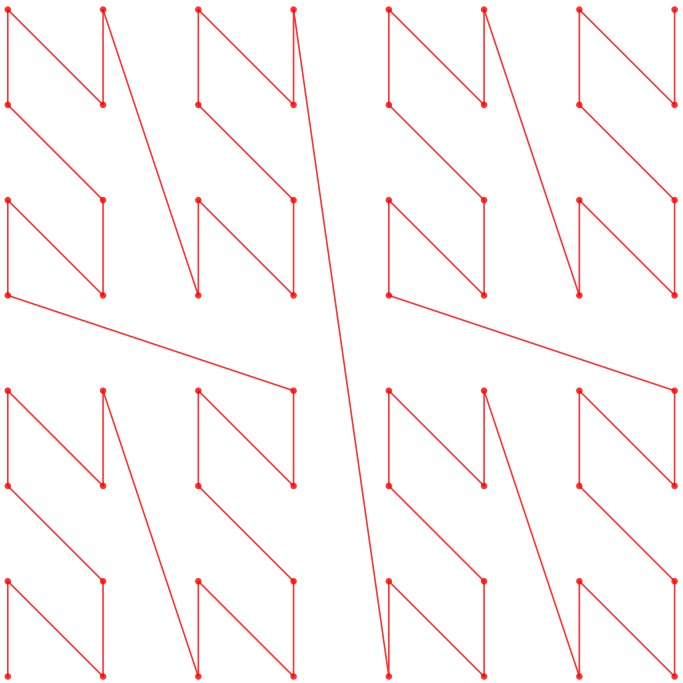


(a) Courbe de Hilbert d'ordre 6

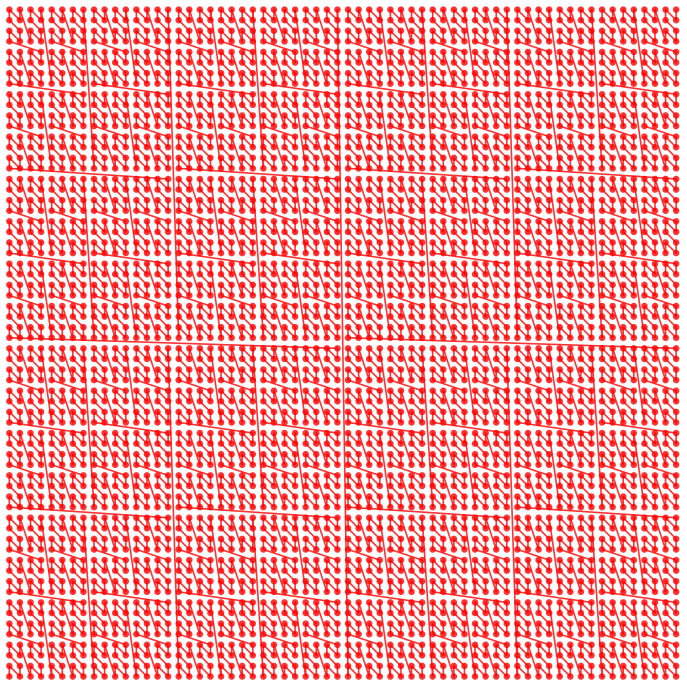
(b) Courbe de Hilbert d'ordre 8

Figure 2 – Des courbes de Hilbert pour des ordres élevés.

Des autres types des CRE

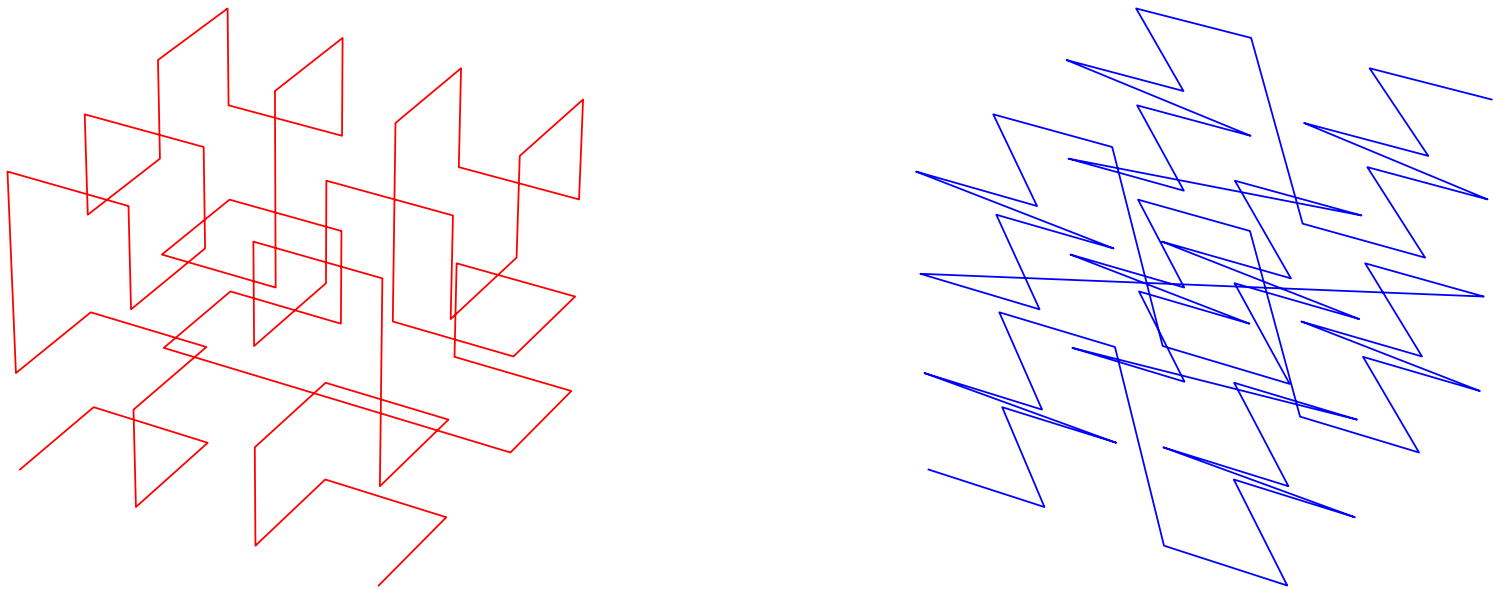


(a) Courbe de Lebesgue d'ordre 3



(b) Courbe de Lebesgue d'ordre 6

Figure 3 – Courbes de Lebesgue



(a) Courbe de Hilbert 3D

(b) Courbe de Lebesgue 3D

Figure 4 – Courbes de Hilbert et de Lebesgue tridimensionnelles

Sommaire

Théorème 3 (G.Cantor)

Les ensembles $[0,1]$ *et* $[0,1]^2$ *sont en bijection.*

Théorème 4 (E. Netto)

Il n'existe aucune bijection ***continue*** *entre* $[0,1]$ *et* $[0,1]^2$.

Mais est-il possible de trouver une surjection continue ?

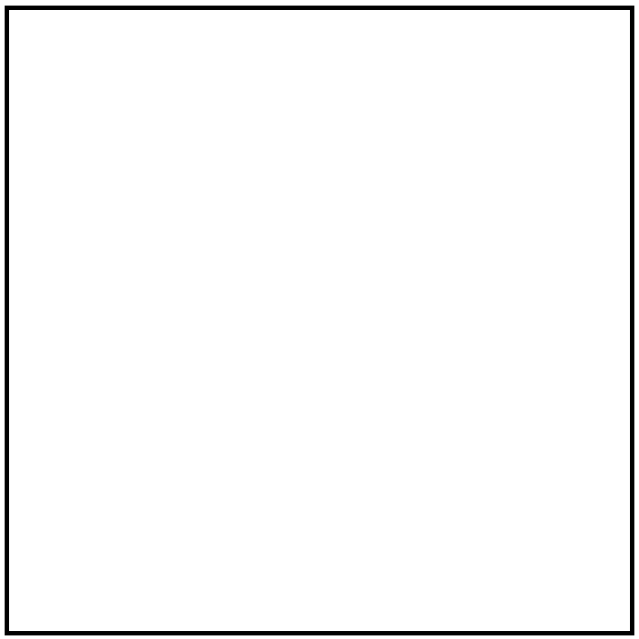
Théorème 4 (E. Netto)

Il n'existe aucune bijection ***continue*** *entre* $[0,1]$ *et* $[0,1]^2$.

Mais est-il possible de trouver une surjection continue ?

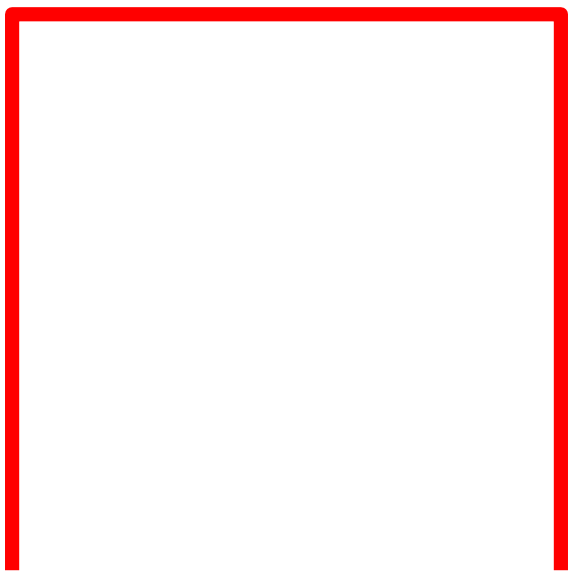
La réponse est **oui**.

Construction de la courbe de Hilbert



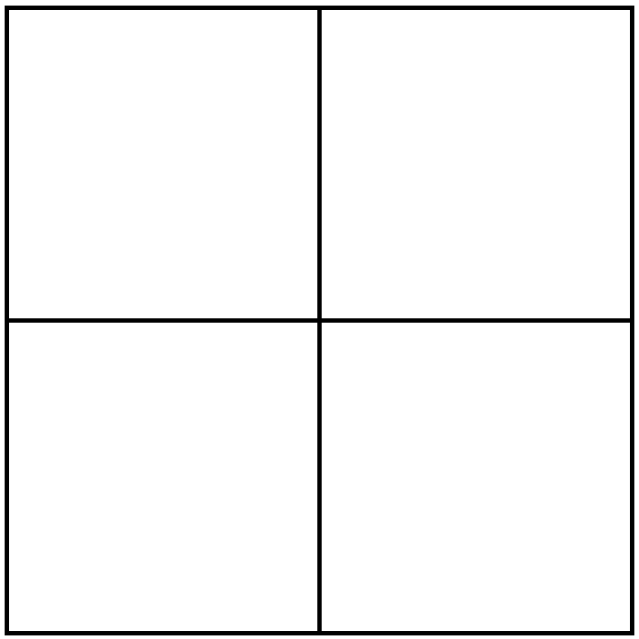
On commence par le carré $[0,1]^2$ tout entier

Construction de la courbe de Hilbert



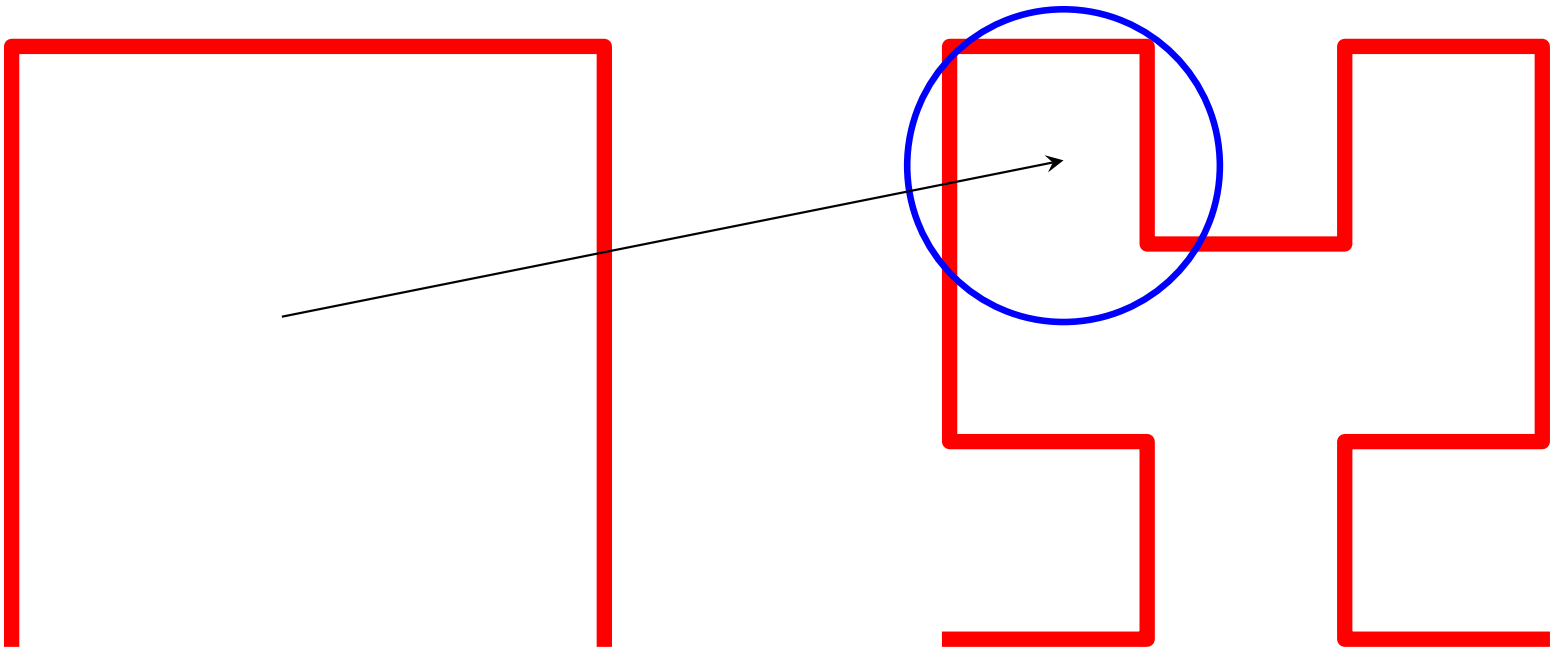
Puis on 'remplit' ce carré

Construction de la courbe de Hilbert



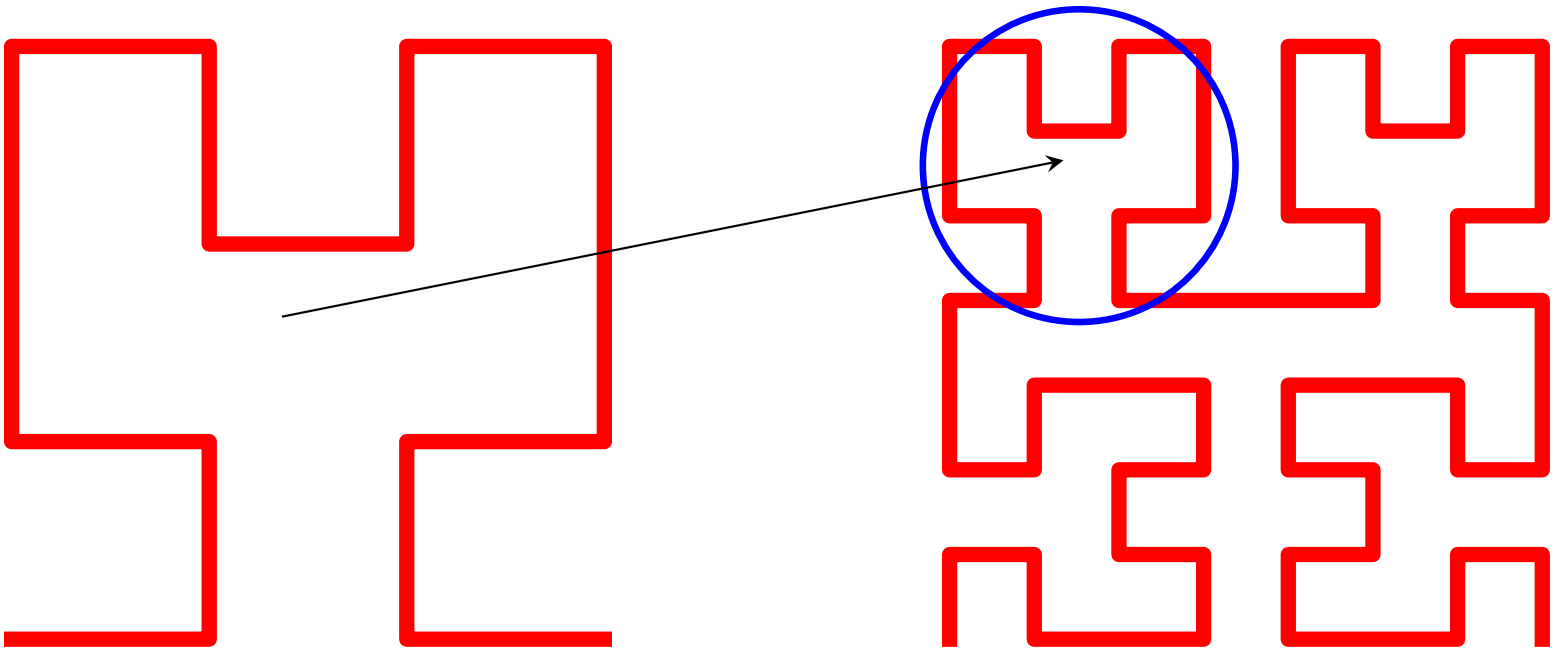
Pour la deuxième itération, on dévise le carré sur $2^2 = 4$ petits carrés d'arêtes $2^{-2} = \frac{1}{2}$

Construction de la courbe de Hilbert



Puis chaque petit carré est remplit par la courbe de l'iteration précendente

Construction de la courbe de Hilbert



Puis chaque petit carré est remplit par la courbe de l'iteration précendente

Remarque

On admet pour la suite que pour tout $d \in \mathbb{N}^*$, ce processus est généralisable au $[0,1]^d$ ou en général tout espace euclidien.

. KimDong Ryul,Existence of Space-Filling Curve in Euclidean Space, KPF Mathematics Seminar,March 20, 2016.

Sommaire

Sommaire

Rappel

On rappelle d'abord ce théorème du cours.

Théorème 5

Soit $d \in \mathbb{N}$ $(f_n)_{n\in\mathbb{N}}$ *une suite de fonctions continues sur* $[0,1]$ *à valeurs dans* $[0,1]^d$ *qui converge uniformément vers* f *sur* $[0,1]$, *alors* f *est continue sur* $[0,1]$.

Continuité de la courbe de Hilbert

Théorème 6

Soit $(f_n)_{n\in\mathbb{N}} \in (\mathcal{C}^0([0,1],[0,1]^2))^{\mathbb{N}}$
Si pour tout $\varepsilon > 0$ *, il existe* $n_0 \in \mathbb{N}$ *tel que* $\forall n, m \geq n_0$ *on a :*
$\sup_{x\in[0,1]} \|f_n(x) - f_m(x)\| < \varepsilon$
Alors $f(x) := \lim_{x\to\infty} f_n(x)$ *existe pour tout* $x \in [0,1]$*.Et* f *est continue*

Proposition 7

La suite des $(\mathcal{H}_n)_{n\in\mathbb{N}}$ *associés à la courbe de Hilbert d'ordre n converge uniformément vers* $\mathcal{H}$*.*

Corollaire 8

*La fonction $\mathcal{H}$ est **continue**.*

Théorème 9

*La fonction $\mathcal{H}$ est **surjective**.*

Définition 10

On définit ainsi la courbe de Hilbert par $\mathcal{H}([0,1]) = [0,1]^2$ puisque $\mathcal{H}$ est surjective.

Ainsi la courbe de Hilbert est une **CRE**.

□

Sommaire

Les courbes remplissant l'espace peuvent donc **transformer** d'une manière **continue** des donnés multidimensionnelles en une séquence 1D **plus facile à manipuler**.

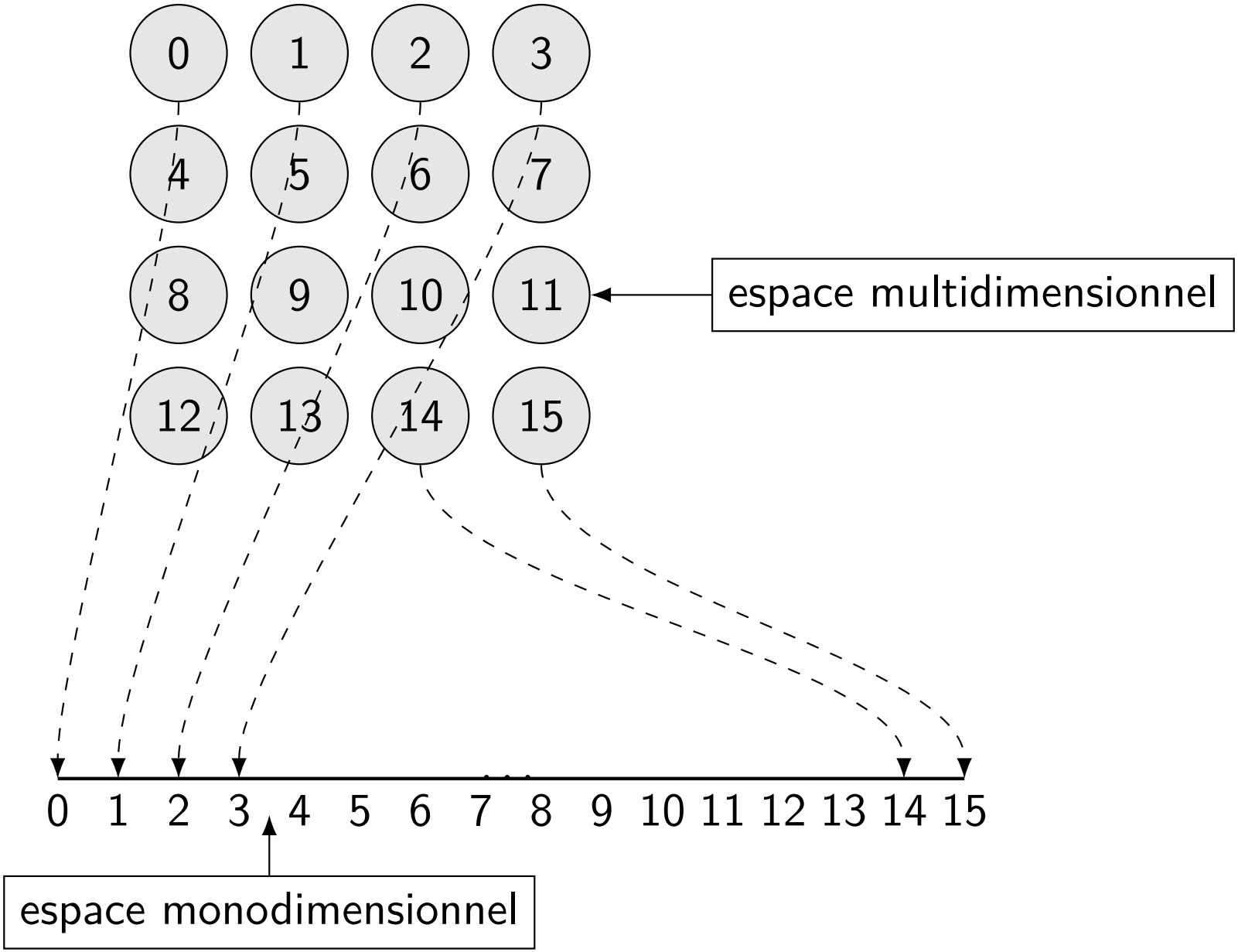


Figure 5 – Illustration de cette propriété.

Sommaire

Motivation derrière les CRE

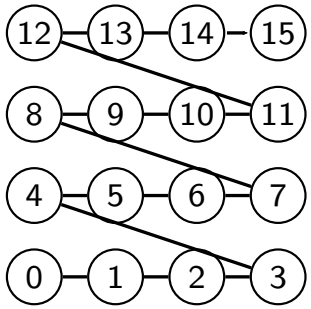
Cependant, les CRE résolvent ce problème en conservant mieux la localité des données spatiales : les points proches dans un espace multidimensionnel (par exemple, euclidien) restent probablement proches dans une représentation 1D comme l'intervalle $[0, 1]$.

Un exemple illustratif

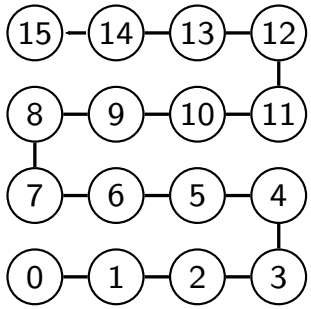
Considérons deux méthodes naïfs pour parcourir un carré de dimensions 4×4.
Puis on la compare avec la courbe de Hilbert et celle de Lebesgue.

. Inspiré par **Giap Nguyen**

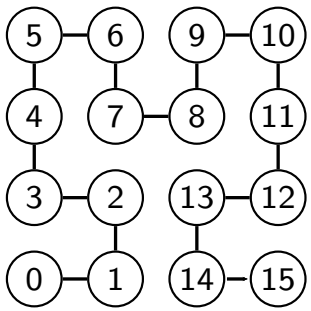
Visualisation des types de parcours selon la courbe



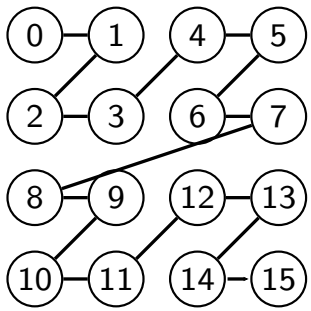
(a) Méthode Sweep



(b) Méthode Scan



(c) Méthode Hilbert



(d) Méthode Morton(ou Lebesgue)

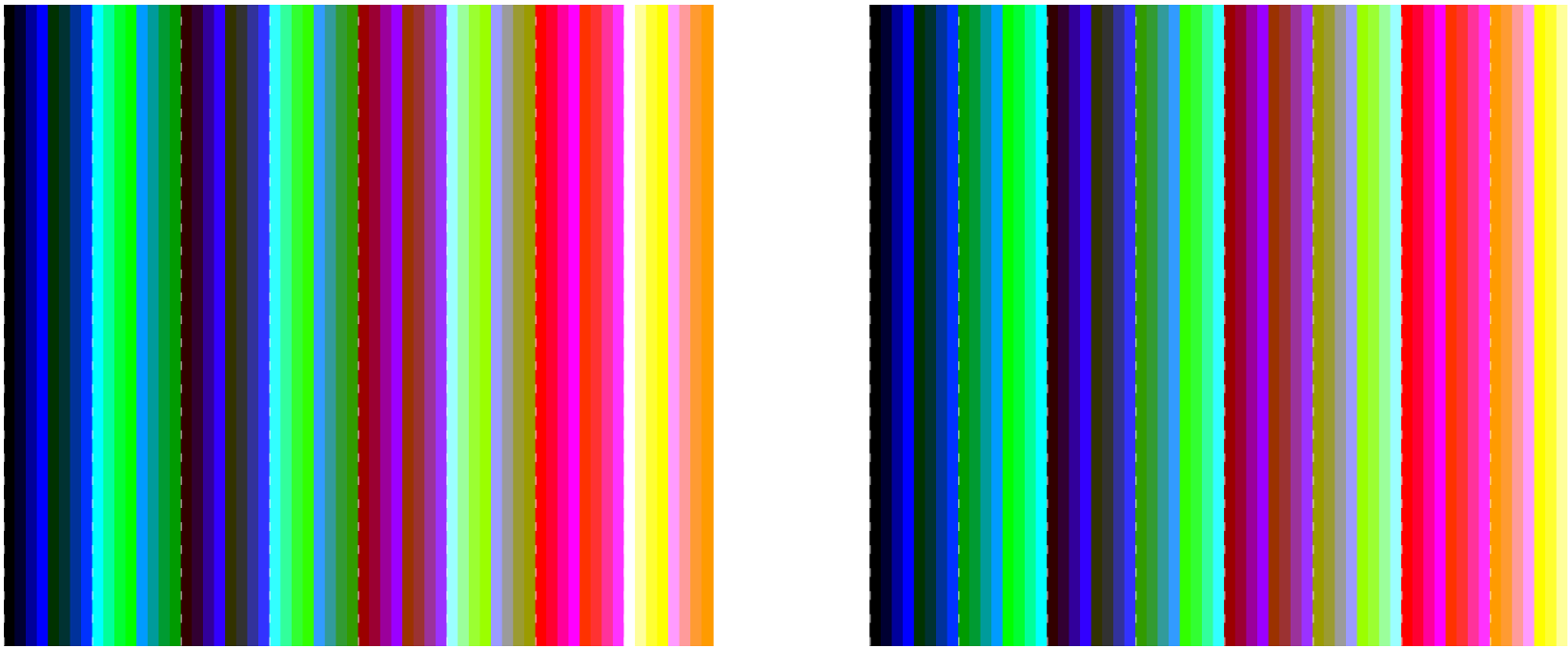
Figure 6 – Visualisation des différents types de parcours d'un carré 4×4 par de différents méthodes

Une petite expérience

On souhaite trier des différents couleurs sous la forme (r,v,b) où $r, v, b \in \{0, 50, 155, 255\}$

Il s'agit des donnés tri-dimensionnels donc il faut faire une réduction de dimension.

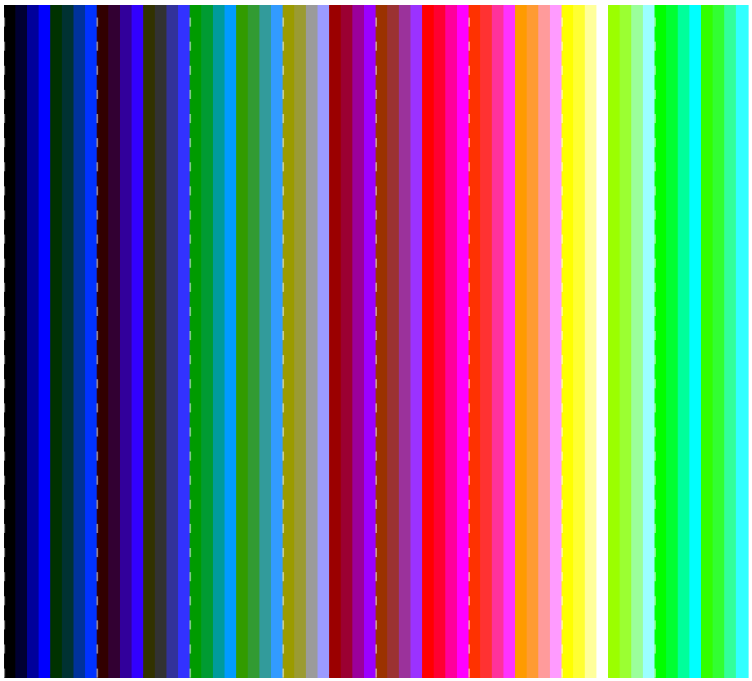
Pour cela on utilise les 4 méthodes de la figure 5.



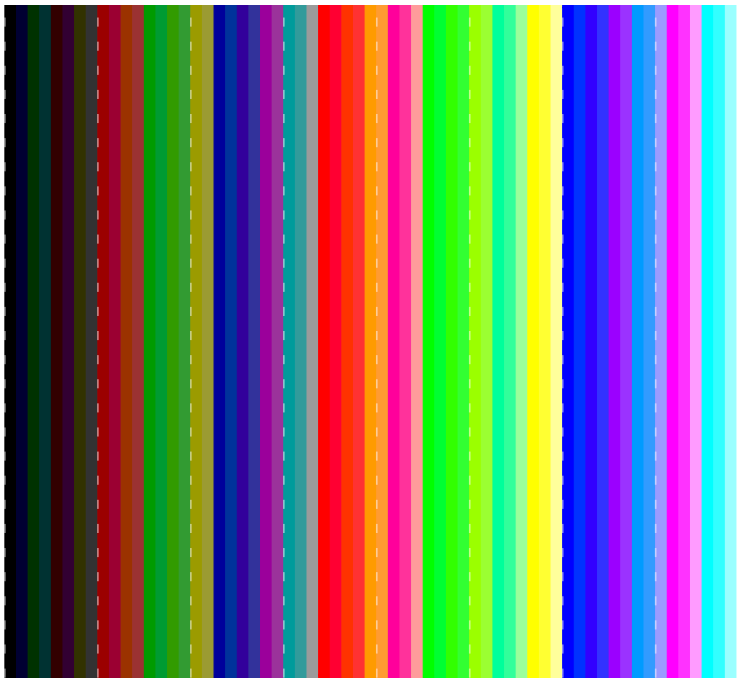
(a) Méthode Scan

(b) Méthode Sweep

Figure 7 – Triage des couleurs à l'aide des méthodes naïfs



(a) Méthode Hilbert



(b) Méthode Morton (Lebesgue)

Figure 8 – Triage des couleurs à l'aide des méthodes plus optimales

Conclusion

Les courbes de Lebesgue et de Hilbert conservent donc mieux la localité des donnés multidimensionelles.

. Linear Clustering of Objects with Multiple Attributes, H V Jagadish,AT&T Bell Laboratories , Murray Hill,New Jeresy 07974.

Sommaire

Sommaire

Formulation de la problématique

Comment exploiter les propriétés des courbes remplissant l'espace, notamment leur capacité à préserver la localité, pour résoudre des problèmes pratiques dans le domaine du traitement d'images ?

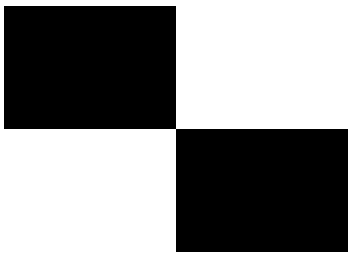
Sommaire

Sommaire

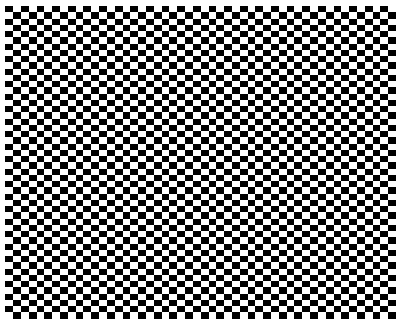
Généralités

La méthode du tramage consiste à créer l'illusion d'une couleur même qui ne peut pas être affichée à cause de la nature de l'afficheur carachtériseé par la profondeurs des bits(bit depth).

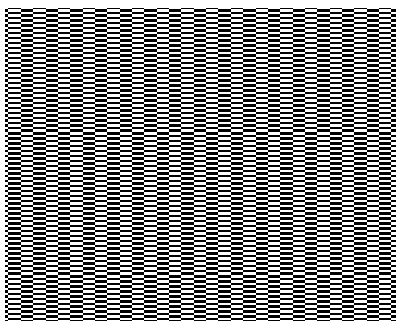
'Création' d'une troisième couleur (le gris) avec 2 couleurs (le noir et le blanc)



(a) 2×2



(b) 50×50



(c) 750×750

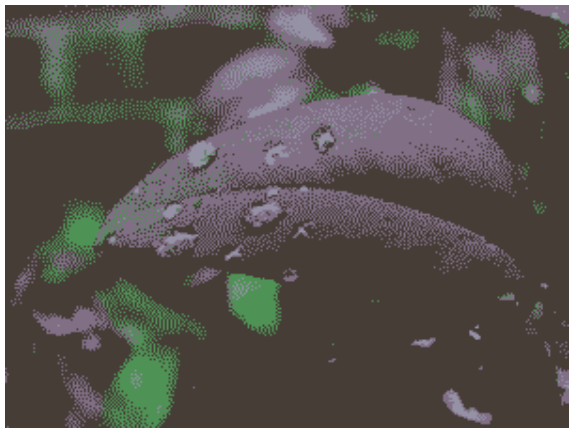
Figure 9 – nombre de carreaux et l'illusion qu'il crée

Ordres de grandeurs

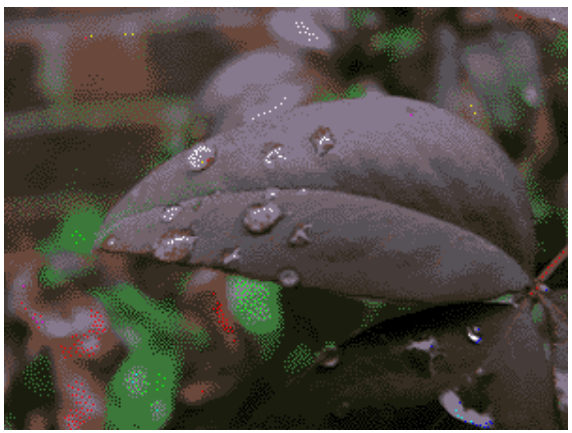
- **Les téléphones portables modernes** : 10 ou 8 bits à chaque couleur (de 10^6 à 10^9 couleurs différents)
- **Les anciens afficheurs** : 8 bits en total (seulement 256 couleurs)

Figure 10 – Évolution de la qualité de l'image (PNG) en fonction des bits

(a) 1 bit (2 couleurs différents,5Ko)



(b) 2 bits (8 couleurs différents,6Ko)



(c) 4 bits(4096 couleurs différents,13Ko)

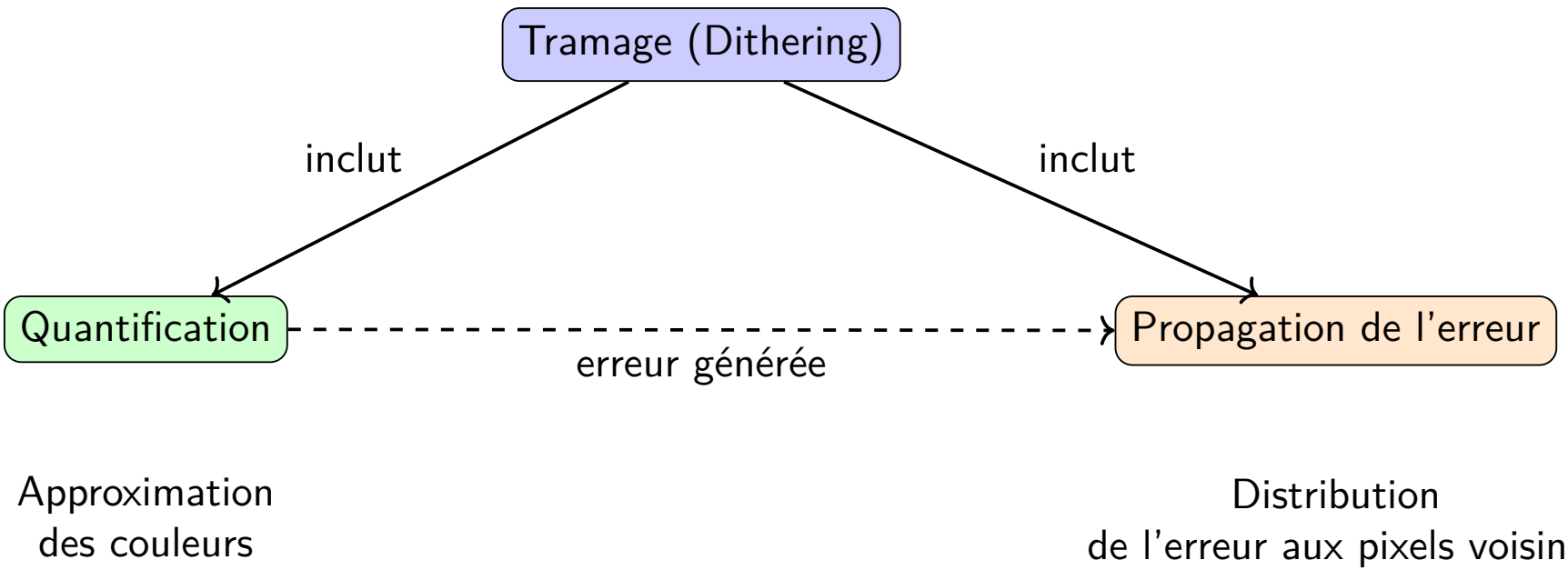
(d) 8 bits (16 777 216 couleurs différents,38Ko)

1ère Technique de compression

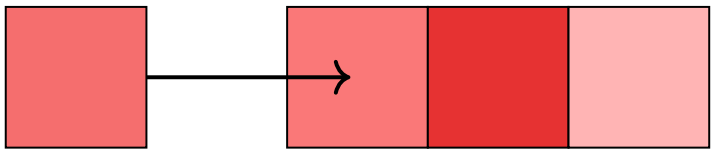
En réduisant le nombre de bits nécessaires à la création d'une image, on peut donc réduire sa taille en sacrifiant la qualité de l'image.

Mais il existe une $2^{\text{ème}}$:

Pricipe du tramage



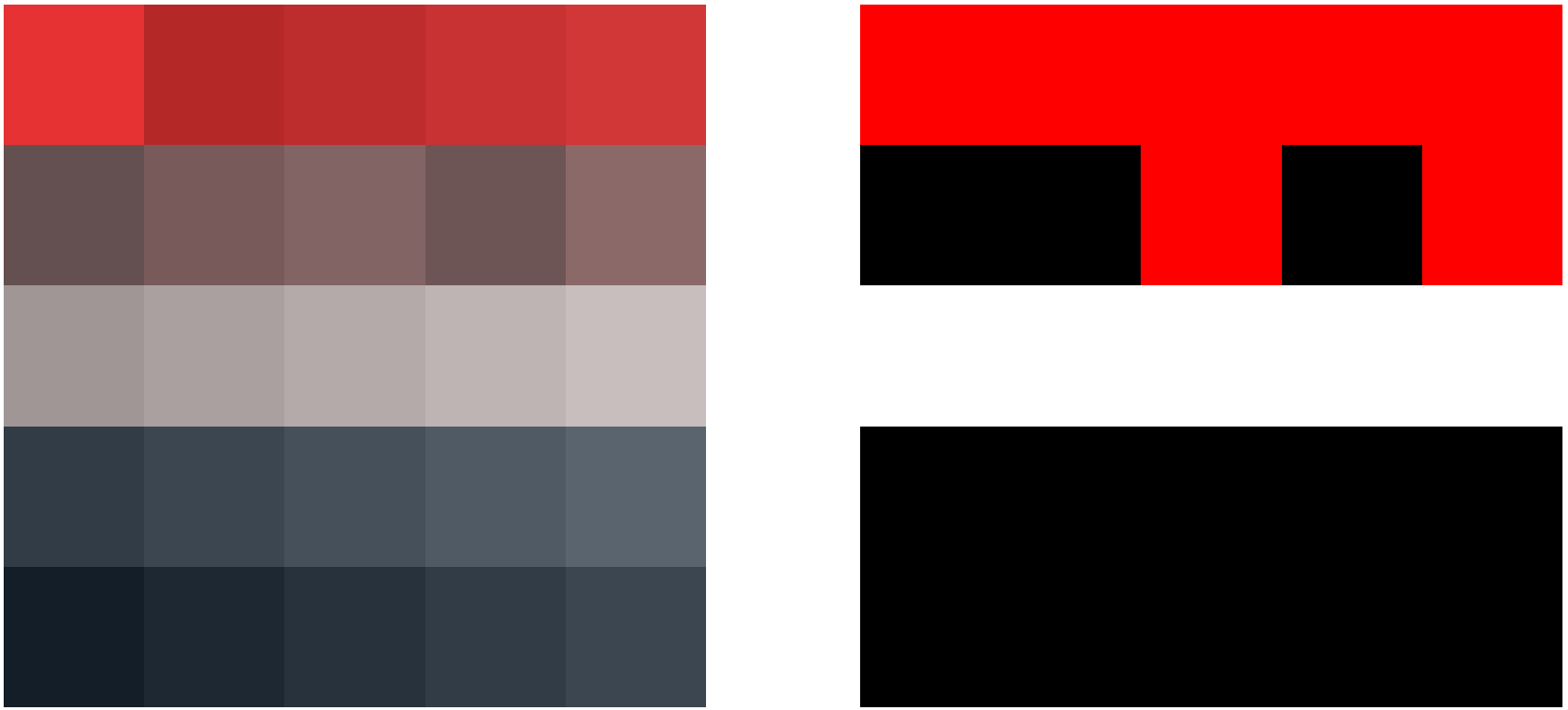
Quantification



$$\|\mathbf{x} - \mathbf{p}\|_2 = \sqrt{(x_r - p_r)^2 + (x_g - p_g)^2 + (x_b - p_b)^2}$$

Norme euclidienne (norme des couleurs)

Figure 11 – Exemple de quantification : un pixel est associé à la couleur la plus proche dans la palette selon la norme euclidienne.



(a) Image Originale

(b) Image quantifiée

Figure 12 – Autre exemple de quantification

Propagation de l'erreur

La quantification produit un certain erreur pour tout pixel.

Propagation de l'erreur

La quantification produit un certain erreur pour tout pixel.

La solution ?

Propagation de l'erreur

La quantification produit un certain erreur pour tout pixel.

La solution ?

Faire 'propager' cette erreur !

Exemple (Atkinson) I

A	B	C
D	0.7	E
F	G	H

Figure 13 – Grille 3×3 des pixels, le pixel central est subit à une quantification : $(1) \rightarrow (0.7)$

Exemple (Atkinson) — Propagation de l'erreur

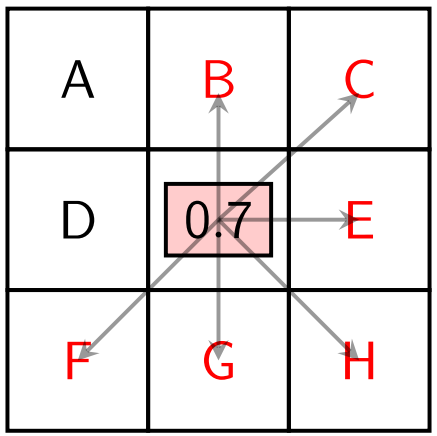


Figure 14 – Distribution de l'erreur selon le noyau d'Atkinson

$$\text{Erreur propagée} = -0.3 \times \begin{bmatrix} 0 & \frac{1}{8} & \frac{1}{8} \\ 0 & * & \frac{1}{8} \\ \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \end{bmatrix} = \begin{bmatrix} 0 & -0.0375 & -0.0375 \\ 0 & * & -0.0375 \\ -0.0375 & -0.0375 & -0.0375 \end{bmatrix}$$

Exemple (Floyd-Steinberg) — Propagation de l'erreur

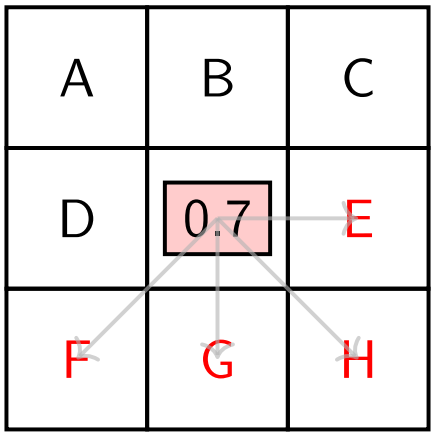


Figure 15 – Distribution de l'erreur selon le noyau de Floyd-Steinberg

$$\text{Erreur propagée} = -0.3 \times \begin{bmatrix} 0 & 0 & 0 \\ \frac{7}{16} & * & \frac{1}{16} \\ \frac{5}{16} & \frac{3}{16} & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ -0.13125 & * & -0.01875 \\ -0.09375 & -0.05625 & 0 \end{bmatrix}$$

Relation avec les CRE

Pour effectuer le tramage, il faut parcourir chaque pixel, l'expérience montre que l'ordre du parcours est important. Pour cela on compare trois methodes de parcours :

- Hilbert
- Morton
- Scan (Serpent)

Sommaire

Résultats et discussion

(a) Originale

(b) Hilbert

(c) morton

(d) scan

Figure 16 – Différents types du tramage(en couleurs)

(a) Originale

(b) Hilbert

(c) morton

(d) scan

Figure 17 – Différents types du tramage(noir et blanc)

Résultats BW

Méthode	**Entropie (bits/pixel)**	**Taille gzip (octets)**
Hilbert	0.9127	1901
Morton	0.9030	1948
Serpent	0.9037	1939

Tableau 1 – Entropie et Taille gzip pour chacun des méthodes (image en noir et blanc)

Résultats RGB

Méthode	**Entropie (bits/pixel)**	**Taille gzip (octets)**
Hilbert	0.9030	4251
Morton	0.9057	4802
Serpent	0.9066	4955

Tableau 2 – Entropie et Taille gzip pour chacun des méthodes (image en couleur)

Sommaire

Des autres algorithmes (toujours lossless)

Méthode	**gzip (octets)**	**bz2 (octets)**	**lzma (octets)**
Original	38 103	36 282	32 932
Hilbert Dither	4 245	4 172	4 024
Morton Dither	4 796	4 656	4 484
Scan Dither	4 946	4 759	4 600

Tableau 3 – Comparaison des tailles de compression lossless

Merci pour votre attention

Réalisation de la figure 1

```
import matplotlib.pyplot as plt
import numpy as np
from hilbertcurve.hilbertcurve import HilbertCurve
def courbe_hilbert(ordre):
        n, p = 2, ordre
        courbe = HilbertCurve(p, n)
        nb_points = 2 ** (p * n)
        coordonnees = []
        for i in range(nb_points):
                point = courbe.point_from_distance(i)
                coordonnees.append(point)
        return np.array(coordonnees)
for ordre in [1, 2, 3]:
        points = courbe_hilbert(ordre)
        plt.figure(figsize=(5, 5))
        plt.plot(points[:, 0], points[:, 1], 'r-', linewidth=8)
        plt.axis('off')
        plt.axis('equal')
        plt.tight_layout()
        plt.savefig(f"hilbert_{ordre}.pdf"
        , facecolor='white', bbox_inches="tight")
        plt.close()
```

Réalisation de la figure 2

```
import matplotlib.pyplot as plt
import numpy as np
from hilbertcurve.hilbertcurve import HilbertCurve
def courbe_hilbert(ordre):
        n,p=2,ordre
        courbe = HilbertCurve(p, n)
        nb_points = 2 ** (p * n)
        coordonnees = []
        for i in range(nb_points):
                point = courbe.point_from_distance(i)
                coordonnees.append(point)
        return np.array(coordonnees)
ordre = 6 # puis 8 et on réexécute pour obtenir (b) #
points = courbe_hilbert(ordre)
plt.figure(figsize=(5, 5))
plt.plot(points[:, 0], points[:, 1], 'r-', linewidth=2)
plt.axis('off')
plt.axis('equal')
plt.tight_layout()
plt.savefig(f"hilbert{ordre}.pdf",
facecolor='white',bbox_inches="tight")
plt.close()
```

Réalisation de 3 I

```
import matplotlib.pyplot as plt
import numpy as np
import pymorton
def generer_points_courbe_z_ordre(ordre_bits):
# ordre = ordre_bits = itération de la courbe , type = int
        max_coordonnee = 2**ordre_bits
        tous_les_points = []
        for y in range(max_coordonnee):
                for x in range(max_coordonnee):
                        tous_les_points.append((x, y))
        points_tries_par_z = []
        for x, y in tous_les_points:
                valeur_z = pymorton.interleave2(x, y)
                points_tries_par_z.append((valeur_z, (x, y)))
        points_tries_par_z.sort()
        coordonnees = np.array([point[1] for point in   points_t
        ries_par_z])
        return coordonnees
```

Réalisation de 3 II

```
ordre_courbe = 3
points_courbe = generer_points_courbe_z_ordre(ordre_courbe)
valeur_max = 2**ordre_courbe - 1
points_transformes = np.copy(points_courbe)
points_transformes[:, 0] = valeur_max - points_courbe[:, 1]
points_transformes[:, 1] = valeur_max - points_courbe[:, 0]
# mirroir par rapport à Y
points_courbe = points_transformes
plt.figure(figsize=(5, 5))
plt.plot(points_courbe[:, 0], points_courbe[:, 1],
'r-', linewidth=0.75, marker='o', markersize=2, alpha=0.8)
plt.axis('off')
plt.axis('equal')
plt.tight_layout()
plt.savefig(f"Lebesgue{ordre_courbe}.pdf",
facecolor='white', bbox_inches="tight")
plt.close()
```

Réalisatin de la figure 4 I

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from hilbertcurve.hilbertcurve import HilbertCurve
import pymorton

def courbe_hilbert_3d(order):
        n = 3
        hilbert = HilbertCurve(order, n)
        nb_points = 2 ** (order * n)
        return np.array([hilbert.point_from_distance(i) for i
        in range(nb_points)])

def courbe_zorder_3d(order):
        max_coord = 2 ** order
        coords = []
        nb_points = max_coord ** 3
        for i in range(nb_points):
```

Réalisatin de la figure 4 II

```
                try:
                        x, y, z = pymorton.deinterleave3(i)
                        if x < max_coord and y < max_coord and z < max_
                        coord:
                        coords.append([x, y, z])
        except Exception:
        continue
        return np.array(coords)
        ordre = 2
hilbert_points = courbe_hilbert_3d(ordre)
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot(hilbert_points[:,0], hilbert_points[:,1], hilbe
rt_points[:,2], 'r-', linewidth=1)
ax.axis('off')
plt.tight_layout()
plt.savefig(f"hilbert{ordre}.pdf", bbox_inches='tight', facec
olor='white')
plt.close()
```

Réalisatin de la figure 4 III

```
zorder_points = courbe_zorder_3d(ordre)
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot(zorder_points[:,0], zorder_points[:,1], zorder_points[
:,2], 'b-', linewidth=1)
ax.axis('off')
plt.tight_layout()
plt.savefig(f"morton{ordre}.pdf", bbox_inches='tight', facecolo
r='white')
plt.close()
```

Preuve de 3

Démonstration.

D'après le développement décimal de tout nombre réel x, on a

$$x = \sum_{n=1}^{\infty} \frac{x_n}{10^n}$$

où $x_n \in \{0, 1, \ldots, 9\}$ pour tout $n \in \mathbb{N}$. On définit alors la fonction suivante :

$$f : [0,1] \to [0,1]^2, \quad x \mapsto \left(\sum_{n=1}^{\infty} \frac{x_{2n-1}}{10^{2n-1}}, \sum_{n=1}^{\infty} \frac{x_{2n}}{10^{2n}} \right)$$

Pour les nombres ayant des développements décimaux non uniques (ceux se terminant par une infinité de 9), nous adoptons la convention d'utiliser la représentation finie (terminant par des 0). Ceci garantit que f est bien définie et bijective.

Preuve de 4 (Partie 1/2)

Démonstration.

Supposons qu'une telle bijection continue existe $f : [0,1] \to [0,1]^2$. Alors, on montre d'abord que $f^{-1} : [0,1]^2 \to [0,1]$ est continue. Soit $C \subset [0,1]$ un fermé. Puisque C est borné, alors C est un compact. Or f est bijective donc surjective, donc $(f^{-1})^{-1}(C) = f(C)$ et l'image d'un compact par une fonction continue est un compact. Ainsi, $(f^{-1})^{-1}(F)$ pour tout F fermé de $[0,1]$ est un fermé, donc f^{-1} est continue.
Maintenant on considère la fonction, en fixant $(p,q) \in]0,1[^2 \times]0,1[$:

$$g : [0,1]^2 \setminus \{p\} \to [0,1] \setminus \{q\}, \quad x \mapsto f^{-1}(x)$$

□

Preuve de 4 (Partie 2/2)

Suite.

D'après ce qui précède, g est continue et bijective. Or $[0,1]^2 \setminus \{p\}$ est connexe par arcs tandis que $g([0,1]^2 \setminus \{p\}) = [0,1] \setminus \{q\}$ ne l'est pas, donc g ne peut pas être continue, d'où f n'est pas continue.

Ainsi, une bijection continue entre $[0,1]$ et $[0,1]^2$ ne peut pas exister.

□

Démonstration du Théorème 6

Démonstration.

Soit $n, m \in \mathbb{N}$ et $(x, y) \in [0,1]^2$:
$\|f(x) - f(y)\| \leq \|f(x) - f_m(x)\| + \|f_m(x) - f_n(x)\| + \|f(y) - f_n(x)\|$
Pour tout $\varepsilon \geq 0$ en prenant $N \in \mathbb{N}$ assez grand alors pour $n \geq N$:
$\|f_n(x) - f(x)\| \leq \frac{\varepsilon}{3}$
Par la continuité des f_n, on a :
$\exists \delta \geq 0 \forall x, y \in [0,1] |x - y| \leq \delta \implies |f_n(x) - f_n(y)\| \leq \frac{\varepsilon}{3}$
Donc, pour $x, y \in [0,1]$ tels que $|x - y| \leq \delta$, on a :
$\|f(x) - f(y)\| \leq \varepsilon$

D'où la continuité de f et son existance.

□

Preuve de la propositon 7

Démonstration.

Pour $n > m$, la construction de h_n modifie $\mathcal{H}_m$ uniquement à l'intérieur de chaque carré de côté 2^{-m}.
Dans un tel carré, la distance maximale entre deux points est $\sqrt{2} \cdot 2^{-m}$.
Donc :
$\sup_{x \in [0,1]} \|\mathcal{H}_n(x) - \mathcal{H}_m(x)\| \leq \sqrt{2} \cdot 2^{-m} = \sqrt{2} \cdot 2^{-\min(n,m)}$

La suite est donc uniformément convergente.

□

Preuve du théorème 9

Démonstration.

La courbe de la fonction $\mathcal{H}_n$ passe par chacun des 4^n petits carrés de côtés 2^{-n}.
Ainsi tout point $(x, y) \in [0, 1]^2$ est à une distance au plus $\frac{2^{-n}}{\sqrt{2}}$ du tracé de $\mathcal{H}_n$
Pour $n \to \infty$, (x, y) appartient à l'adhérence de l'image de $\mathcal{H}$.
Or l'image continue d'un compact est un compact, donc (x, y) est atteint par $\mathcal{H}$.

$\mathcal{H}$ **est donc surjective**

□

Réalisation des figures 7 et 8 I

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
from hilbertcurve.hilbertcurve import HilbertCurve
import pymorton

composantes = [0, 50, 155, 255]

couleurs = [(r, g, b) for r in composantes
                                                for g in composantes
                                                for b in composantes]

def ordre_balayage(couleurs):
        return sorted(couleurs, key=lambda c: (c[0], c[1], c[2]))

def ordre_zigzag(couleurs):
        couleurs_triees = sorted(couleurs, key=lambda c: (c[0],
        c[1], c[2]))
```

Réalisation des figures 7 et 8 II

```
        resultat = []
        for i in range(0, len(couleurs_triees), 8):
                if (i//8) % 2 == 0:
                        resultat.extend(couleurs_triees[i:i+8])
                else:
                        resultat.extend(couleurs_triees[i+7:i-1:-1])
        return resultat

def ordre_hilbert(couleurs):
        courbe_hilbert = HilbertCurve(3, 2)
        index_couleurs = {}
        for couleur in couleurs:
                valeur_reduite = tuple(c//85 for c in couleur)
                index = courbe_hilbert.distance_from_point(valeur_red
                uite)
                index_couleurs[couleur] = index
        return sorted(couleurs, key=lambda c: index_couleurs[c])

def ordre_morton(couleurs):
```

Réalisation des figures 7 et 8 III

```
        def cle_morton(couleur):
                r, g, b = (c//85 for c in couleur)
                        return pymorton.interleave3(r, g, b)
        return sorted(couleurs, key=cle_morton)

methodes = {
        "Balayage": ordre_balayage(couleurs),
        "Zigzag": ordre_zigzag(couleurs),
        "Hilbert": ordre_hilbert(couleurs),
        "Morton": ordre_morton(couleurs)
}

for nom_methode, couleurs_ordonnees in methodes.items():
        with PdfPages(f'{nom_methode}couleur.pdf') as pdf:
                fig, ax = plt.subplots(figsize=(12, 2))

                for i, couleur in enumerate(couleurs_ordonnees):
                        ax.add_patch(plt.Rectangle((i, 0), 1, 1, color=np\
                        .array(couleur)
```

Réalisation des figures 7 et 8 IV

```
                        /255))
                        if i % 8 == 0:
                                ax.axvline(i, color='white', linestyle=':'\
                                , alpha=0.5)

                ax.set_xlim(0, len(couleurs_ordonnees))
                ax.set_ylim(0, 1)
                ax.axis('off')
                pdf.savefig(fig, bbox_inches='tight')
                plt.close()
```

Réalisation de la figure 9 I

```
from reportlab.lib.pagesizes import letter
from reportlab.lib.units import inch
from reportlab.pdfgen import canvas

def creer_noiretblanc1():
        dessin = canvas.Canvas("blacknwhite1.pdf", pagesize=let
        ter)
        largeur, hauteur = letter
        taille_case = inch * 2
        for i in range(2):
                for j in range(2):
                        x = i * taille_case
                        y = hauteur - (j + 1) * taille_case
                        dessin.setFillColorRGB(0, 0, 0) if i == j else
                        dessin.setFillCo
                        lorRGB(1, 1, 1)
                        dessin.rect(x, y, taille_case, taille_case, fil
                        l=1, stroke=0)
```

Réalisation de la figure 9 II

```
        dessin.save()

def creer_noiretblanc2():
        dessin = canvas.Canvas("blacknwhite2.pdf", pagesize=let
        ter)
        largeur, hauteur = letter
        lignes, colonnes = 50, 50
        largeur_case = largeur / colonnes
        hauteur_case = hauteur / lignes
        for i in range(colonnes):
                for j in range(lignes):
                        x = i * largeur_case
                        y = hauteur - (j + 1) * hauteur_case
                        dessin.setFillColorRGB(0, 0, 0) if (i + j)
                         % 2 ==
                         0 else
                                dessin.setFillColorRGB(1, 1, 1)
                        dessin.rect(x, y, largeur_case, hauteur_case,
                        fill
```

Réalisation de la figure 9 III

```
                        =1, stroke=0)
        dessin.save()

def creer_noiretblanc3():
        dessin = canvas.Canvas("blacknwhite3.pdf", pagesize=letter)
        largeur, hauteur = letter
        lignes, colonnes = 750,750
        largeur_case = largeur / colonnes
        hauteur_case = hauteur / lignes
        for i in range(colonnes):
                for j in range(lignes):
                        x = i * largeur_case
                        y = hauteur - (j + 1) * hauteur_case
                        dessin.setFillColorRGB(0, 0, 0) if (i + j)
                        % 2 == 0
                         else dessin.setFillColorRGB(1, 1, 1)
                        dessin.rect(x, y, largeur_case, hauteur_case
                        , fill=1,
                         stroke=0)
```

Réalisation de la figure 9 IV

```
        dessin.save()

creer_noiretblanc1()
creer_noiretblanc2()
creer_noiretblanc3()
```

Réalisation de la figure 12 I

```
import numpy as np
from PIL import Image

def quantifier(pixel):
return 255 if pixel > 127 else 0

def quantifier_image(image):
        h, l, c = image.shape
        resultat = np.copy(image)
        for canal in range(c):
                for y in range(h):
                        for x in range(l):
                                resultat[y, x, canal] = quantifier(resultat\
                                [y, x, canal])
        return resultat.astype(np.uint8)
def creer_image_test():
        return np.array([
                [[230, 50, 50], [180, 40, 40], [190, 45, 45],
```

Réalisation de la figure 12 II

```
                [200, 50, 50], [210, 55, 55]],
[[100, 80, 80], [120, 90, 90], [130, 100, 100],
[110, 85, 85], [140, 105, 105]],
[[160, 150, 150], [170, 160, 160], [180, 170, 170], [190,
 180, 180], [200, 190, 190]],
[[50, 60, 70], [60, 70, 80], [70, 80, 90], [80, 90, 100],
[90, 100, 110]],
[[20, 30, 40], [30, 40, 50], [40, 50, 60], [50, 60, 70], [6
0, 70, 80]]
], dtype=np.uint8)

def sauvegarder_image(nom, matrice):
        img = Image.fromarray(matrice, mode='RGB')
        img.save(nom)
image_originale = creer_image_test()
image_quantifiee = quantifier_image(image_originale)
sauvegarder_image('originale.png', image_originale)
sauvegarder_image('quantifiee.png', image_quantifiee)
```

Réalisation des figures 16 et 17 I

```
import numpy as np
from PIL import Image
from pymorton import interleave2
from hilbertcurve.hilbertcurve import HilbertCurve
def charger_image_rgb():
img = Image.open("hopper.png").convert("RGB")
return img_np

# Courbes
def coords_hilbert(w, h):
        n = max(w, h)
        p = 1
        ordre = 0
        while p < n:
                p <<= 1
                ordre += 1
        courbe = HilbertCurve(ordre, 2)
        coords = []
```

Réalisation des figures 16 et 17 II

```
        for y in range(p):
                for x in range(p):
                        d = courbe.distance_from_point([x, y])
                        if 0 <= x < w and 0 <= y < h:
                                coords.append((d, x, y))
        coords.sort()
        return [(x, y) for _, x, y in coords]

def coords_morton(w, h):
        n = max(w, h)
        p = 1
        while p < n:
                p <<= 1
        coords = []
        for y in range(p):
                for x in range(p):
                        code = interleave2(x, y)
                        if 0 <= x < w and 0 <= y < h:
                        coords.append((code, x, y))
```

Réalisation des figures 16 et 17 III

```
        coords.sort()
        return [(x, y) for _, x, y in coords]

def coords_serpent(w, h):
        coords = []
        for y in range(h):
                for x in range(w):
                        coords.append((x, y))
        return coords

# Atkinson (canal unique)
def atkinson_dither(image_channel, coords):
        h, w = image_channel.shape
        img = image_channel.copy()
        sortie = np.zeros_like(img)
        voisins = [(1, 0), (2, 0), (-1, 1), (0, 1), (1, 1), (0, 2)]
        for x, y in coords:
                ancien_pixel = img[y, x]
                nouveau_pixel = float(ancien_pixel > 0.5)
```

Réalisation des figures 16 et 17 IV

```
                sortie[y, x] = nouveau_pixel
                erreur = (ancien_pixel - nouveau_pixel) / 8
                for dx, dy in voisins:
                        nx, ny = x + dx, y + dy
                        if 0 <= nx < w and 0 <= ny < h:
                                img[ny, nx] += erreur
        return sortie
# Dither RGB par canal
def dither_rgb_1bit(image_rgb, coords):
        R = atkinson_dither(image_rgb[:, :, 0], coords)
        G = atkinson_dither(image_rgb[:, :, 1], coords)
        B = atkinson_dither(image_rgb[:, :, 2], coords)
        image_rgb_dithered = np.stack([R, G, B], axis=2)
        return image_rgb_dithered

# Export PNG 1-bit (grayscale)
def exporter_png_1bit(image_np, nom_fichier):
        img = Image.fromarray((image_np > 0.5).astype(np.uint8) * 25
        5, mode="L")
```

Réalisation des figures 16 et 17 V

```
        img_1bit = img.convert("1")  # vrai PNG 1 bit
        img_1bit.save(nom_fichier, compress_level=0)

# Export PNG palette (RGB compressé)
def exporter_png_palette(image_np, nom_fichier):
        img = Image.fromarray((image_np * 255).astype(np.uint8), mode
        ="RGB")
        img_palette = img.convert("P", palette=Image.ADAPTIVE, colors=
        8)
        img_palette.save(nom_fichier, compress_level=0)

# Export PNG RGB (plein)
def exporter_png_rgb(image_np, nom_fichier):
        img = Image.fromarray((image_np * 255).astype(np.uint8), mode=
        "RGB")
        img.save(nom_fichier, compress_level=0)

# Main
image_orig_rgb = charger_image_rgb()
```

Réalisation des figures 16 et 17 VI

```
hauteur, largeur = image_orig_rgb.shape[0], image_orig_rgb.shape[1]

# Grayscale image pour export 1 bit
image_orig_gray = np.mean(image_orig_rgb, axis=2)

# Courbes
coords_h = coords_hilbert(largeur, hauteur)
coords_m = coords_morton(largeur, hauteur)
coords_s = coords_serpent(largeur, hauteur)

# Dither Grayscale
image_h_bw = atkinson_dither(image_orig_gray, coords_h)
image_m_bw = atkinson_dither(image_orig_gray, coords_m)
image_s_bw = atkinson_dither(image_orig_gray, coords_s)

# Dither RGB + quantization
image_h_rgb = dither_rgb_1bit(image_orig_rgb, coords_h)
image_m_rgb = dither_rgb_1bit(image_orig_rgb, coords_m)
image_s_rgb = dither_rgb_1bit(image_orig_rgb, coords_s)
```

Réalisation des figures 16 et 17 VII

```
# Sauvegardes
exporter_png_rgb(image_orig_rgb, "doriginal.png")

exporter_png_1bit(image_h_bw, "dhilbert_bw.png")
exporter_png_1bit(image_m_bw, "dmorton_bw.png")
exporter_png_1bit(image_s_bw, "dsnake_bw.png")

exporter_png_palette(image_h_rgb, "dhilbert_palette.png")
exporter_png_palette(image_m_rgb, "dmorton_palette.png")
exporter_png_palette(image_s_rgb, "dsnake_palette.png")
```

Réalisation des tableaux 1 et 2 I

```
import numpy as np
from PIL import Image
from pymorton import interleave2
from hilbertcurve.hilbertcurve import HilbertCurve
import os
import gzip
import shutil
from scipy.stats import entropy
def calculer_entropy(image_np):
        hist, _ = np.histogram(image_np.flatten(), bins=256, range=(0,1), densi
        hist = hist[hist > 0]  # éviter log(0)
        return entropy(hist, base=2)

# === Fonction pour sauvegarder RAW + compresser GZIP ===
def exporter_raw_gzip(image_np, nom_base):
        raw_name = nom_base + ".raw"
        gzip_name = nom_base + ".raw.gz"
```

Réalisation des tableaux 1 et 2 II

```
        image_bytes = (image_np * 255).astype(np.uint8).tobytes()
        with open(raw_name, "wb") as f:
                f.write(image_bytes)

        with open(raw_name, "rb") as f_in:
        with gzip.open(gzip_name, "wb") as f_out:
                shutil.copyfileobj(f_in, f_out)
        os.remove(raw_name)
        return os.path.getsize(gzip_name)

def charger_image_rgb():
        img = Image.open("hopper.png").convert("RGB")
        img_np = np.array(img) / 255.0
        return img_np
# Courbes
def coords_hilbert(w, h):
        n = max(w, h)
        p = 1
        ordre = 0
```

Réalisation des tableaux 1 et 2 III

```
        while p < n:
                p <<= 1
                ordre += 1
        courbe = HilbertCurve(ordre, 2)
        coords = []
        for y in range(p):
                for x in range(p):
                        d = courbe.distance_from_point([x, y])
                        if 0 <= x < w and 0 <= y < h:
                                coords.append((d, x, y))
        coords.sort()
        return [(x, y) for _, x, y in coords]

def coords_morton(w, h):
        n = max(w, h)
        p = 1
        while p < n:
                p <<= 1
        coords = []
```

Réalisation des tableaux 1 et 2 IV

```
        for y in range(p):
                for x in range(p):
                        code = interleave2(x, y)
                        if 0 <= x < w and 0 <= y < h:
                                coords.append((code, x, y))
        coords.sort()
return [(x, y) for _, x, y in coords]

def coords_serpent(w, h):
        coords = []
        for y in range(h):
                for x in range(w):
                        coords.append((x, y))
        return coords

# Atkinson (canal unique)
def atkinson_dither(image_channel, coords):
        h, w = image_channel.shape
        img = image_channel.copy()
```

Réalisation des tableaux 1 et 2 V

```
        sortie = np.zeros_like(img)
        voisins = [(1, 0), (2, 0), (-1, 1), (0, 1), (1, 1), (0, 2)]
        for x, y in coords:
                ancien_pixel = img[y, x]
                nouveau_pixel = float(ancien_pixel > 0.5)
                sortie[y, x] = nouveau_pixel
                erreur = (ancien_pixel - nouveau_pixel) / 8
        for dx, dy in voisins:
                nx, ny = x + dx, y + dy
                if 0 <= nx < w and 0 <= ny < h:
                        img[ny, nx] += erreur
        return sortie

def dither_rgb_1bit(image_rgb, coords):
        R = atkinson_dither(image_rgb[:, :, 0], coords)
        G = atkinson_dither(image_rgb[:, :, 1], coords)
        B = atkinson_dither(image_rgb[:, :, 2], coords)
        image_rgb_dithered = np.stack([R, G, B], axis=2)
        return image_rgb_dithered
```

Réalisation des tableaux 1 et 2 VI

```
def exporter_png_1bit(image_np, nom_fichier):
        img = Image.fromarray((image_np > 0.5).astype(np.uint8) * 255
        , mode="L")
        img_1bit = img.convert("1")  # vrai PNG 1 bit
        img_1bit.save(nom_fichier, compress_level=0)

def exporter_png_palette(image_np, nom_fichier):
        img = Image.fromarray((image_np * 255).astype(np.uint8), mode
        ="RGB")
        img_palette = img.convert("P", palette=Image.ADAPTIVE, colors
        =8)
        img_palette.save(nom_fichier, compress_level=0)

def exporter_png_rgb(image_np, nom_fichier):
        img = Image.fromarray((image_np * 255).astype(np.uint8), mode
        ="RGB")
        img.save(nom_fichier, compress_level=0)
```

Réalisation des tableaux 1 et 2 VII

```
image_orig_rgb = charger_image_rgb()
hauteur, largeur = image_orig_rgb.shape[0], image_orig_rgb.shape[1]

image_orig_gray = np.mean(image_orig_rgb, axis=2)

coords_h = coords_hilbert(largeur, hauteur)
coords_m = coords_morton(largeur, hauteur)
coords_s = coords_serpent(largeur, hauteur)

# Perform dithering on grayscale
image_h_bw = atkinson_dither(image_orig_gray, coords_h)
image_m_bw = atkinson_dither(image_orig_gray, coords_m)
image_s_bw = atkinson_dither(image_orig_gray, coords_s)

image_h_rgb = dither_rgb_1bit(image_orig_rgb, coords_h)
image_m_rgb = dither_rgb_1bit(image_orig_rgb, coords_m)
image_s_rgb = dither_rgb_1bit(image_orig_rgb, coords_s)

images_bw = [
```

Réalisation des tableaux 1 et 2 VIII

```
        ("Hilbert", image_h_bw),
        ("Morton", image_m_bw),
        ("Serpent", image_s_bw),
]

images_rgb = [
        ("Hilbert", image_h_rgb),
        ("Morton", image_m_rgb),
        ("Serpent", image_s_rgb),
]

# Save output files
exporter_png_rgb(image_orig_rgb, "doriginal.png")

exporter_png_1bit(image_h_bw, "dhilbert_bw.png")
exporter_png_1bit(image_m_bw, "dmorton_bw.png")
exporter_png_1bit(image_s_bw, "dsnake_bw.png")

exporter_png_palette(image_h_rgb, "dhilbert_palette.png")
```

Réalisation des tableaux 1 et 2 IX

```
exporter_png_palette(image_m_rgb, "dmorton_palette.png")
exporter_png_palette(image_s_rgb, "dsnake_palette.png")

# === Analyze file sizes ===
fichiers = [
"doriginal.png",
"dhilbert_bw.png",
"dmorton_bw.png",
"dsnake_bw.png",
"dhilbert_palette.png",
"dmorton_palette.png",
"dsnake_palette.png"
]

print("\nComparaison des tailles de fichiers (en octets) :")
for f in fichiers:
        taille = os.path.getsize(f)
        print(f"{f} : {taille} octets")
```

Réalisation des tableaux 1 et 2 X

```
# === Analyse BW ===
print("\n== Résultats BW ==\n")
print(f"{'Méthode':<10} | Entropie (bits/pixel) | Taille gzip \
(octets)")
print("-" * 50)
for nom, img in images_bw:
        ent = calculer_entropy(img)
        size_gz = exporter_raw_gzip(img, f"export_{nom}_bw")
        print(f"{nom:<10} | {ent:>8.4f}                | {size_gz:>10\
        } ")

# === Analyse RGB ===
print("\n== Résultats RGB ==\n")
print(f"{'Méthode':<10} | Entropie (bits/pixel) | Taille gzip (o\
ctets)")
print("-" * 50)
for nom, img in images_rgb:
        ent = calculer_entropy(img)
        size_gz = exporter_raw_gzip(img, f"export_{nom}_rgb")
```

Réalisation des tableaux 1 et 2 XI

```
        print(f"{nom:<10} | {ent:>8.4f}                    | {size_gz:>10}\
         ")
```

Réalisation du tableau 3 I

```
import numpy as np
from PIL import Image
from pymorton import interleave2
from hilbertcurve.hilbertcurve import HilbertCurve
import os
import gzip
import bz2
import lzma
import shutil

def charger_image_rgb():
    if os.path.exists("hopper.png"):
        img = Image.open("hopper.png").convert("RGB")
    else:
        largeur, hauteur = 256, 256
        gradient = np.tile(np.linspace(0, 255, largeur, dtype=np.uint8),
            (hauteur, 1))
        img = np.stack([gradient, gradient, gradient], axis=2)
```

Réalisation du tableau 3 II

```
        img = Image.fromarray(img, mode="RGB")
    img_np = np.array(img) / 255.0
    return img_np

def coords_hilbert(w, h):
    n = max(w, h)
    p = 1
    ordre = 0
    while p < n:
        p <<= 1
        ordre += 1
    courbe = HilbertCurve(ordre, 2)
    coords = []
    for y in range(p):
        for x in range(p):
            d = courbe.distance_from_point([x, y])
            if 0 <= x < w and 0 <= y < h:
                coords.append((d, x, y))
    coords.sort()
```

Réalisation du tableau 3 III

```
    return [(x, y) for _, x, y in coords]

def coords_morton(w, h):
    n = max(w, h)
    p = 1
    while p < n:
        p <<= 1
    coords = []
    for y in range(p):
        for x in range(p):
            code = interleave2(x, y)
            if 0 <= x < w and 0 <= y < h:
                coords.append((code, x, y))
    coords.sort()
    return [(x, y) for _, x, y in coords]

def coords_serpent(w, h):
    coords = []
    for y in range(h):
```

Réalisation du tableau 3 IV

```
        for x in range(w):
            coords.append((x, y))
    return coords

def atkinson_dither(image_channel, coords):
    h, w = image_channel.shape
    img = image_channel.copy()
    sortie = np.zeros_like(img)
    voisins = [(1, 0), (2, 0), (-1, 1), (0, 1), (1, 1), (0, 2)]
    for x, y in coords:
        ancien_pixel = img[y, x]
        nouveau_pixel = float(ancien_pixel > 0.5)
        sortie[y, x] = nouveau_pixel
        erreur = (ancien_pixel - nouveau_pixel) / 8
        for dx, dy in voisins:
            nx, ny = x + dx, y + dy
            if 0 <= nx < w and 0 <= ny < h:
                img[ny, nx] += erreur
    return sortie
```

Réalisation du tableau 3 V

```
def dither_rgb_1bit(image_rgb, coords):
    R = atkinson_dither(image_rgb[:, :, 0], coords)
    G = atkinson_dither(image_rgb[:, :, 1], coords)
    B = atkinson_dither(image_rgb[:, :, 2], coords)
    image_rgb_dithered = np.stack([R, G, B], axis=2)
    return image_rgb_dithered

def exporter_png_rgb(image_np, nom_fichier):
    img = Image.fromarray((image_np * 255).astype(np.uint8), mode="RGB")
    img.save(nom_fichier, compress_level=0)

def exporter_png_1bit(image_np, nom_fichier):
    img = Image.fromarray((image_np > 0.5).astype(np.uint8) * 255,
        mode="L")
    img_1bit = img.convert("1")
    img_1bit.save(nom_fichier, compress_level=0)

def compresser_gzip(nom_fichier):
```

Réalisation du tableau 3 VI

```
    with open(nom_fichier, "rb") as f_in:
        with gzip.open(nom_fichier + ".gz", "wb") as f_out:
            shutil.copyfileobj(f_in, f_out)
    return os.path.getsize(nom_fichier + ".gz")

def compresser_bz2(nom_fichier):
    with open(nom_fichier, "rb") as f_in:
        with bz2.open(nom_fichier + ".bz2", "wb") as f_out:
            shutil.copyfileobj(f_in, f_out)
    return os.path.getsize(nom_fichier + ".bz2")

def compresser_lzma(nom_fichier):
    with open(nom_fichier, "rb") as f_in:
        with lzma.open(nom_fichier + ".xz", "wb") as f_out:
            shutil.copyfileobj(f_in, f_out)
    return os.path.getsize(nom_fichier + ".xz")

image_orig_rgb = charger_image_rgb()
hauteur, largeur = image_orig_rgb.shape[0], image_orig_rgb.shape[1]
```

Réalisation du tableau 3 VII

```
image_orig_gray = np.mean(image_orig_rgb, axis=2)

coords_h = coords_hilbert(largeur, hauteur)
coords_m = coords_morton(largeur, hauteur)
coords_s = coords_serpent(largeur, hauteur)

image_h_rgb = dither_rgb_1bit(image_orig_rgb, coords_h)
image_m_rgb = dither_rgb_1bit(image_orig_rgb, coords_m)
image_s_rgb = dither_rgb_1bit(image_orig_rgb, coords_s)

exporter_png_rgb(image_orig_rgb, "original.png")
exporter_png_rgb(image_h_rgb, "hilbert.png")
exporter_png_rgb(image_m_rgb, "morton.png")
exporter_png_rgb(image_s_rgb, "scan.png")

taille_gzip_original = compresser_gzip("original.png")
taille_bz2_original = compresser_bz2("original.png")
taille_lzma_original = compresser_lzma("original.png")
```

Réalisation du tableau 3 VIII

```
taille_gzip_hilbert = compresser_gzip("hilbert.png")
taille_bz2_hilbert = compresser_bz2("hilbert.png")
taille_lzma_hilbert = compresser_lzma("hilbert.png")

taille_gzip_morton = compresser_gzip("morton.png")
taille_bz2_morton = compresser_bz2("morton.png")
taille_lzma_morton = compresser_lzma("morton.png")

taille_gzip_scan = compresser_gzip("scan.png")
taille_bz2_scan = compresser_bz2("scan.png")
taille_lzma_scan = compresser_lzma("scan.png")

print("Original :", taille_gzip_original, taille_bz2_original,
    taille_lzma_original)
print("Hilbert :", taille_gzip_hilbert, taille_bz2_hilbert,
    taille_lzma_hilbert)
print("Morton :", taille_gzip_morton, taille_bz2_morton,
    taille_lzma_morton)
```

Réalisation du tableau 3 IX

```
print("Scan :", taille_gzip_scan, taille_bz2_scan, taille_lzma_scan)
```