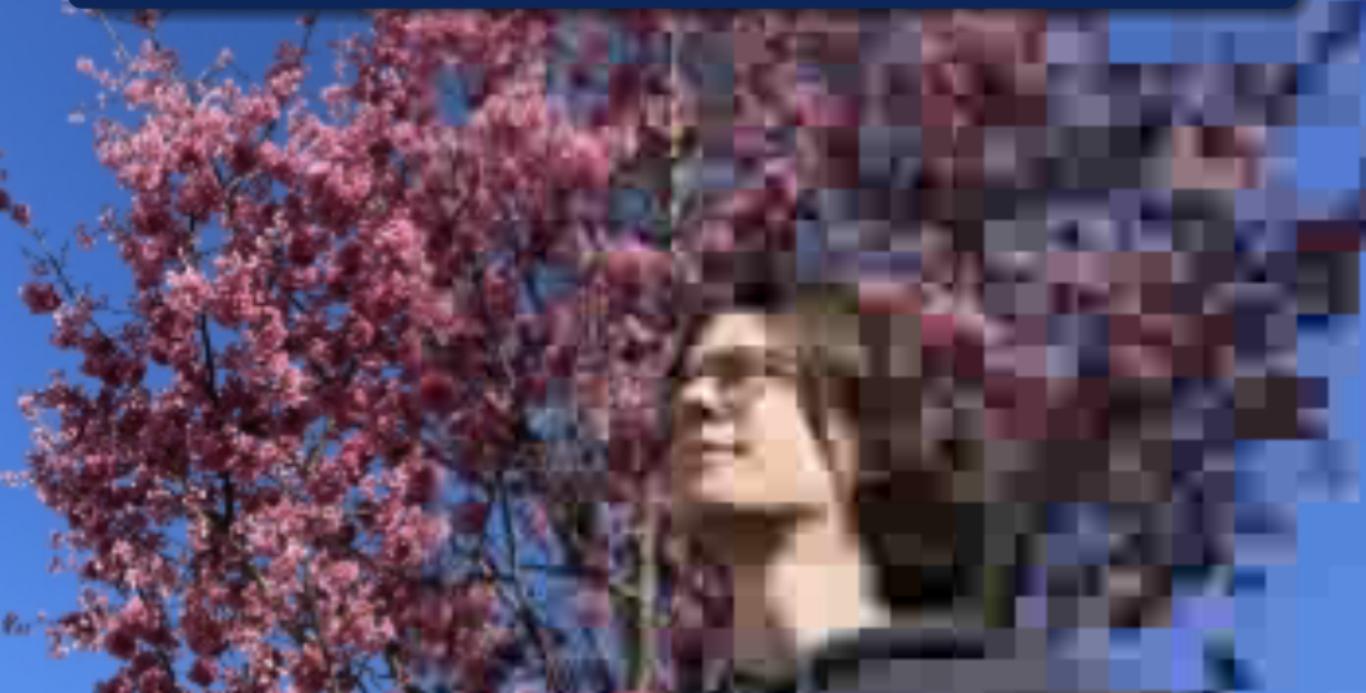


Compression d'images avec pertes

Maximilien Dubus . MPI . Candidat 16078



Introduction

100 000 000 de photos sont publiées chaque jour sur Instagram.

$$\underbrace{100\,000\,000}_{\text{nombre de photos}} \times \underbrace{2\,000 \times 1\,000}_{\text{nombre de pixels par photo}} \times \underbrace{3}_{\text{nombre d'octets par pixel}} = 600 \text{ To}$$



source de l'image (CC0 1.0) : <https://pixabay.com/photos/smartphone-notebook-social-media-1701096/>

Problématique

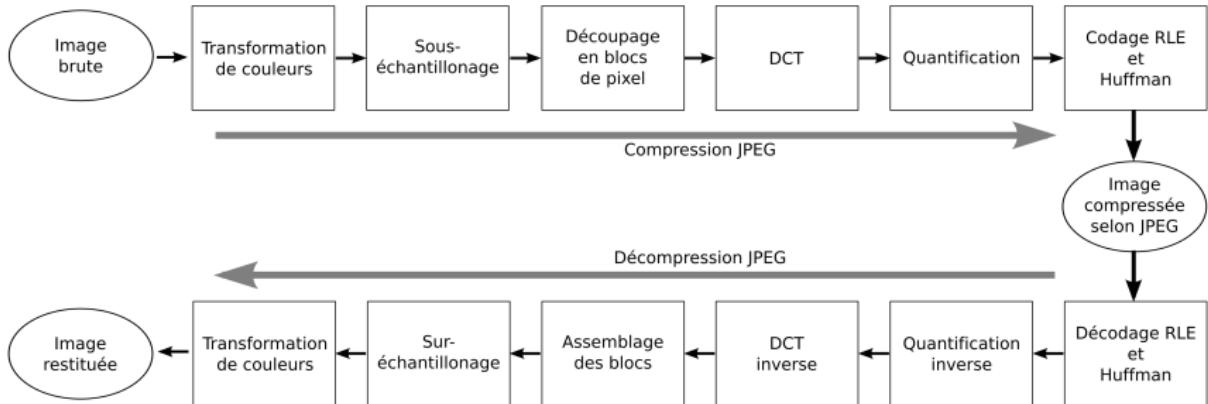
Problématique

En exploitant la redondance de l'information et en prenant en compte les limites physiologiques de l'œil humain, **comment optimiser les techniques de compression d'images avec pertes ?**

Sommaire

1. La compression JPEG
2. Analyse des performances en taille et qualité

Principe de la compression JPEG



Étapes de la compression/décompression JPEG

source de l'image (CC BY-SA 3.0) : wikimedia.org/wiki/File:Compression_JPEG.svg

Transformation des couleurs



Transformation des couleurs

La transformation du modèle RVB au modèle YCbCr est affine

$$Y = 0,299 R + 0,587 V + 0,114 B$$

$$Cb = -0,1687 R - 0,3313 V + 0,5 B + 128$$

$$Cr = 0,5 R - 0,4187 V - 0,0813 B + 128$$



Figure 1: Image Originale



Figure 2: Décomposée en R V B



Figure 3: Décomposée en Y Cb Cr

Sous-échantillonnage de la chrominance



Sous-échantillonnage de la chrominance



Figure 4: les deux matrices de chrominance



on garde **toute** l'information



on garde **la moitié** de l'information



on garde **un quart** de l'information

Découpage en matrices 8x8



Découpage en matrices 8×8



Figure 5: la luminance

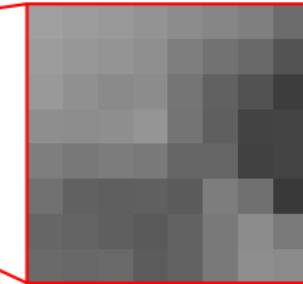


Figure 6: une matrice 8×8 extraite

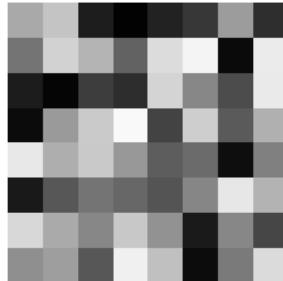


Figure 7: une matrice générée aléatoirement

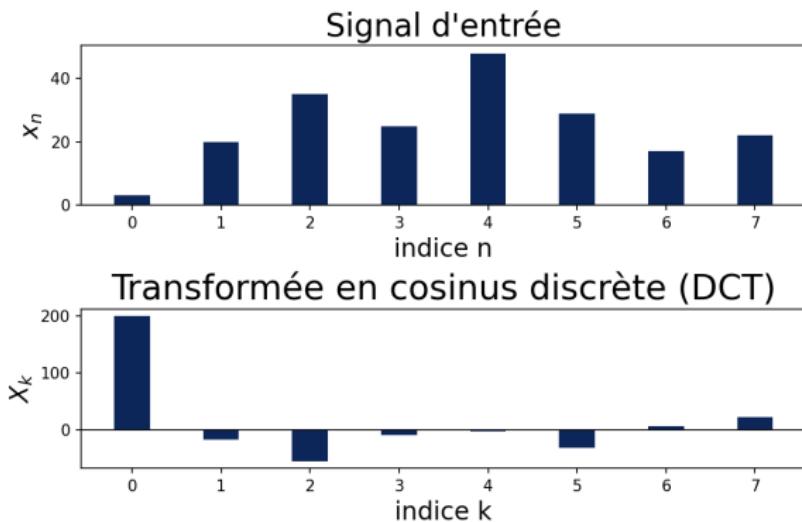
Transformée en cosinus discrète



Transformée en cosinus discrète unidimensionnelle

DCT unidimensionnelle

$$X_k = \sum_{n=0}^{N-1} x_n \cos \left(\frac{k\pi}{N} \left(n + \frac{1}{2} \right) \right)$$



Transformée en cosinus discrète bidimensionnelle

DCT bidimensionnelle

$$DCT(i,j) = \frac{2}{N} c(i)c(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x,y) \cos\left(\frac{\pi}{N} i \left(x + \frac{1}{2}\right)\right) \cos\left(\frac{\pi}{N} j \left(y + \frac{1}{2}\right)\right)$$

avec $c(\alpha) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } \alpha = 0 \\ 1 & \text{sinon} \end{cases}$

Les deux bases

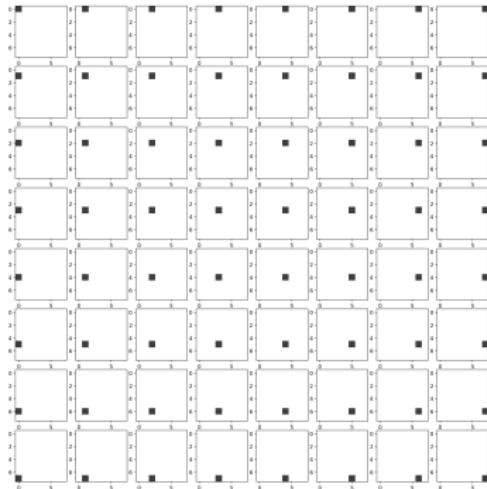


Figure 8: Base canonique

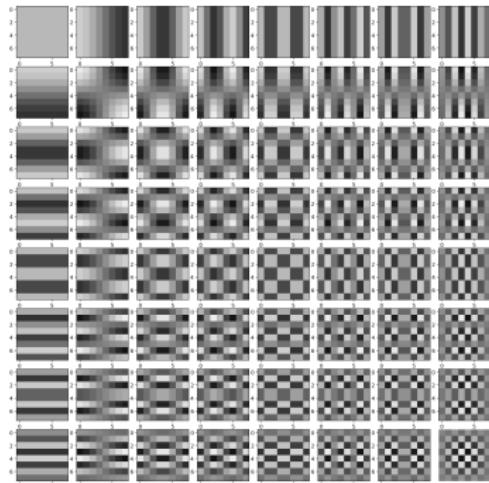


Figure 9: Base de la DCT

Un exemple

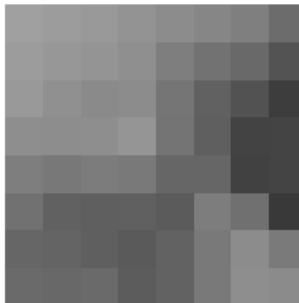


Figure 10: une matrice 8×8

160	156	153	147	140	134	127	108
156	151	148	143	126	114	105	83
153	144	138	140	117	97	82	61
142	141	142	149	116	95	67	68
126	120	124	121	102	65	65	68
113	97	95	96	91	125	112	57
102	100	95	90	98	121	140	122
106	104	106	92	98	121	142	139

Table 1: Dans la base canonique

-99	98	-20	20	-16	20	2	-6
78	103	-38	-8	17	-3	-3	-1
58	-75	40	0	-13	1	-8	13
-7	-18	-4	23	-14	5	4	-2
21	7	-2	-20	12	-12	9	-3
12	-10	1	-2	2	0	-6	0
-9	4	-3	15	-7	8	0	0
-2	-2	6	-5	5	-7	2	1

Table 2: Dans la base DCT

Quantification



Quantification

La matrice de fréquence est divisée par la matrice de quantification.

- L'objectif est de réduire la taille de l'information en approximant les basses fréquences et en éliminant les hautes fréquences.
- Plus les valeurs de quantification sont grandes, plus la perte est importante, mais la compression est meilleure.
- **k** est le facteur de quantification.

-99	98	-20	20	-16	20	2	-6
78	103	-38	-8	17	-3	-3	-1
58	-75	40	0	-13	1	-8	13
-7	-18	-4	23	-14	5	4	-2
21	7	-2	-20	12	-12	9	-3
12	-10	1	-2	2	0	-6	0
-9	4	-3	15	-7	8	0	0
-2	-2	6	-5	5	-7	2	1

k ÷

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

-6	9	-2	1	-1	0	0	0
6	9	-3	0	1	0	0	0
4	-6	2	0	0	0	0	0
0	-1	0	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

matrice après DCT

matrice de quantification choisie
empiriquement par JPEG

Matrice quantifiée

Codage entropique



Parcours en zigzag

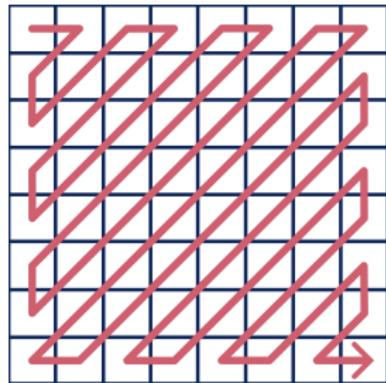


Figure 11: Parcours en zigzag

-6	9	-2	1	-1	0	0	0
6	9	-3	0	1	0	0	0
4	-6	2	0	0	0	0	0
0	-1	0	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Table 3: Matrice après quantification

Après zigzag :

-6, 9, 6, 4, 9, -2, 1, -3, -6, 0, 1, -1, 2, 0,
-1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, EOB

Codage entropique

Pour le codage RLE on code chaque coefficient non nul sous la forme :
(0-tête / catégorie) [position]

- 0-tête : nombre des zéros qui le séparent de son prédecesseur non-nul
- catégorie : catégorie du coefficient
- position : position dans la catégorie

1	-1	1
2	-3, -2	2, 3
3	-7, -6, -5, -4	4, 5, 6, 7
4	-15, -14, ..., -9, -8	8, 9, ..., 14, 15
5	-31, -30, ..., -17, -16	16, 17, ..., 30, 31
6	-63, -62, ..., -33, -32	32, 33, ..., 62, 63
7	-127, -126, ..., -65, -64	64, 65, ..., 126, 127
8	-255, -254, ..., -129, -128	128, 129, ..., 254, 255
9	-511, -510, ..., -257, -256	256, 257, ..., 510, 511
10	-1023, -1022, ..., -513, -512	512, 513, ..., 1022, 1023

Figure 12: Table des catégories

Après zigzag : -6, 9, 6, 4, 9, -2, 1, -3, -6, 0, 1, -1, 2, 0, -1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, EOB

après codage RLE : (0,3) 1, (0,4) 9, (0,3) 6, (0,3) 4, (0,4) 9, (0,2) 1, (0,1) 1, (0,2) 0,
(0,3) 1, (1,1) 1, (0,1) 0, (0,2) 2, (1,1) 0, (1,1) 1, (7,1) 1, (0,0)

après Huffman : 100 001 1011 1001 100 110 100 100 1011 1001 01 01 00 1 01 00 100
001 1100 1 00 0 01 11 1100 0 1100 1 11111010 1 1010

Décompression

Chaque étape est inversible :

- Décompression de Huffman
- Multiplication par la matrice de quantification
- Utilisation de la DCT-inverse
- Transformation des couleurs de YCbCr vers RVB

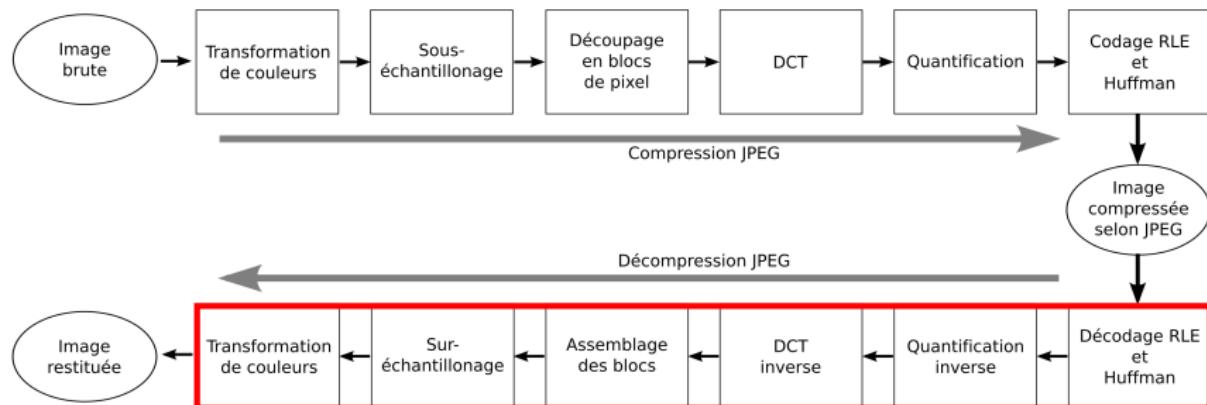


Figure 13: Étapes de la compression JPEG

Section 2

Analyse des performances en taille et qualité

Premier indice de qualité : le PSSB

H : Hauteur de l'image

L : Largeur de l'image

img : Image originale

$imgd$: Image compressée

Racine de l'Erreur Quadratique Moyenne : REQM

$$\text{REQM} = \sqrt{\frac{1}{3LH} \sum_{i=0}^{H-1} \sum_{j=0}^{L-1} \sum_{k=0}^2 (img(i,j,k) - imgd(i,j,k))^2}$$

Rapport du Pic du Signal Sur Bruit : PSSB

$$\text{PSSB} = 20 \cdot \log_{10} \left(\frac{255}{\text{REQM}} \right)$$

Second indice de qualité : le SSIM

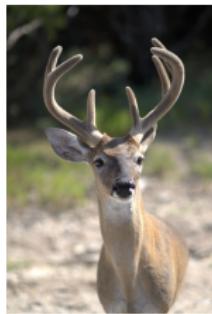
Structural SIMilarity : SSIM

$$\text{SSIM} = \frac{(2\mu_x\mu_y + c_1)(2\sigma_x\sigma_y + c_2)(\text{cov}_{xy} + c_3)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)(\sigma_x\sigma_y + c_3)}$$

Avec :

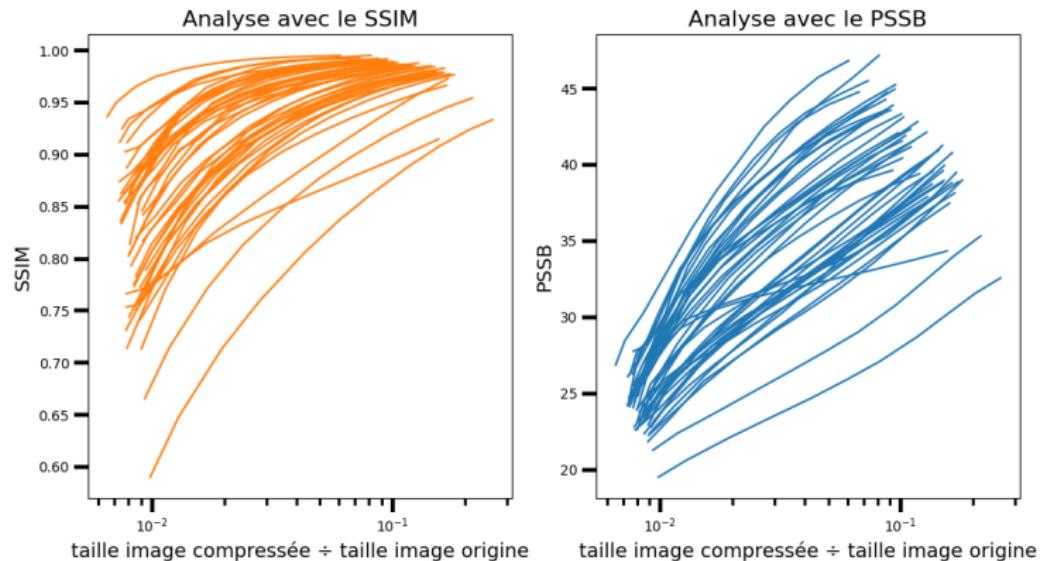
- μ_x la moyenne de x , μ_y la moyenne de y
- σ_x^2 la variance de x , σ_y^2 la variance de y
 - cov_{xy} la covariance de x et y
 - $c_1 = 6.5$, $c_2 = 58.5$ et $c_3 = 29.2$

Échantillon de 12 des 46 photos utilisées

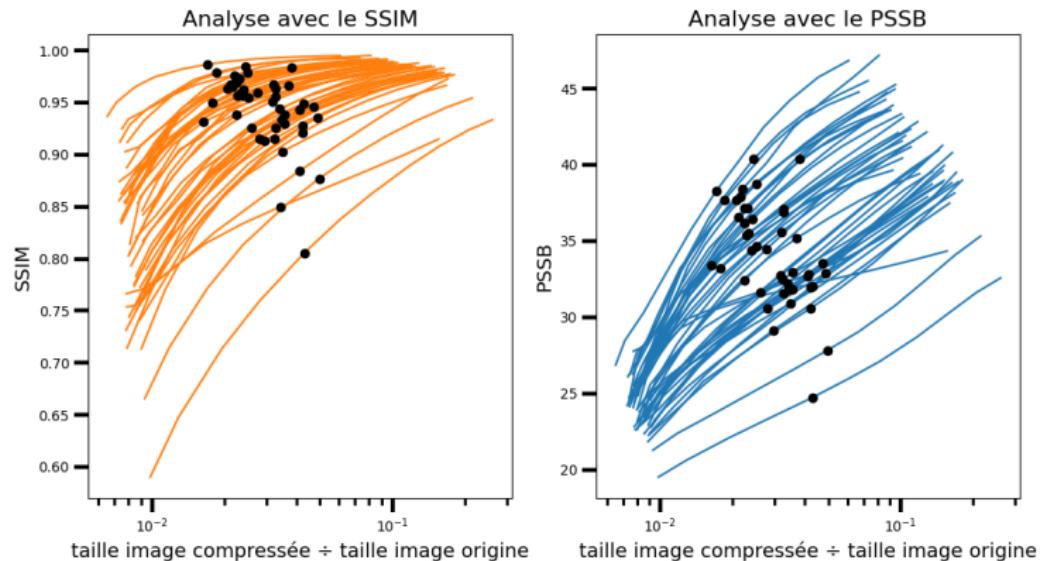


source des images (CC0) (CC0 1.0) : <https://www.signature edits.com/free-raw-photos/>

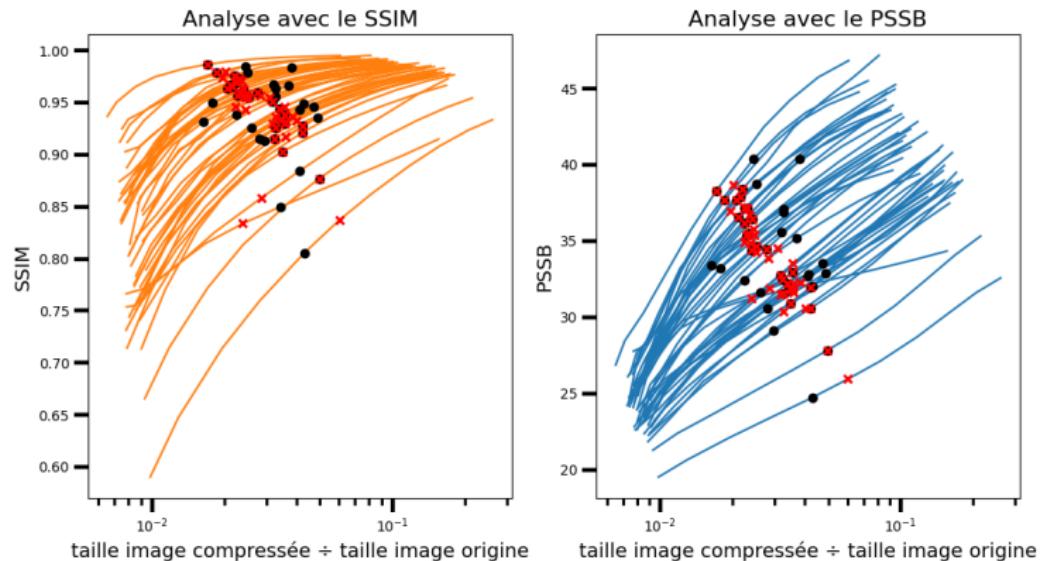
Résultats



Résultats



Résultats



Conclusion

- Les indices de qualité permettent de comparer la qualité d'une même image.
- Il est facile de réduire la taille d'une image d'un facteur 10.
- Le meilleur facteur de quantification à choisir est 1 dans mon étude.

Annexe

Annexe

Annexe : formalisation du sous-échantillonnage de la chrominance

J:a:b $\begin{cases} \mathbf{J} = \text{nombre d'échantillons de luminance Y par ligne} \\ \mathbf{a} = \text{nombre d'échantillons de chrominance (Cb, Cr) sur la première ligne} \\ \mathbf{b} = \text{nombre d'échantillons de chrominance (Cb, Cr) sur la deuxième ligne} \end{cases}$

4:4:4

on garde toute l'information

1	2	3	4
5	6	7	8

→

1	2	3	4
5	6	7	8

4:2:2

on garde 1 information
pour 2 pixels

1	2	3	4
5	6	7	8

→

1.5	3.5
5.5	7.5

4:2:0

on garde 1 information
pour 4 pixels

1	2	3	4
5	6	7	8

→

3.5	5.5
3.5	5.5

Annexe : la transformée de fourrier discrète

Transformation de Fourier discrète

$$S(k) = \sum_{n=0}^{N-1} s(n) e^{-2i\pi k \frac{n}{N}}$$

Transformation de Fourier discrète inverse

$$s(n) = \frac{1}{N} \sum_{k=0}^{N-1} S(k) e^{2i\pi n \frac{k}{N}}$$

Transformation de Fourier discrète bidimensionnelle

$$S(u, v) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} s(m, n) e^{-2i\pi \left(\frac{um}{M} + \frac{vn}{N} \right)}$$

Annexe : comparaison entre TFD et DCT

Transformation de Fourier discrète

$$S(k) = \sum_{n=0}^{N-1} s(n) e^{-2i\pi k \frac{n}{N}} = \sum_{n=0}^{N-1} s(n) \cos\left(2\pi k \frac{n}{N}\right) - i \sum_{n=0}^{N-1} s(n) \sin\left(2\pi k \frac{n}{N}\right)$$

DCT

$$S(k) = \sum_{n=0}^{N-1} s(n) \cos\left(\frac{k\pi}{N} \left(n + \frac{1}{2}\right)\right)$$

Annexe : comparaison entre TFD et DCT bidimensionnelle

Transformation de Fourier discrète bidimensionnelle

$$\begin{aligned} S(u, v) &= \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} s(m, n) e^{-2i\pi(\frac{um}{M} + \frac{vn}{N})} \\ &= \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} s(m, n) \cos\left(2\pi \frac{um}{M}\right) \cos\left(2\pi \frac{vn}{N}\right) \\ &\quad - \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} s(m, n) \sin\left(2\pi \frac{um}{M}\right) \sin\left(2\pi \frac{vn}{N}\right) \\ &\quad - i \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} s(m, n) \sin\left(2\pi \left(\frac{um}{M} + \frac{vn}{N}\right)\right) \end{aligned}$$

DCT

$$S(u, v) = \frac{2}{N} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} s(m, n) \cos\left(\frac{\pi}{N} u \left(m + \frac{1}{2}\right)\right) \cos\left(\frac{\pi}{N} v \left(n + \frac{1}{2}\right)\right)$$

Annexe : DCT inverse

DCT bidimensionnelle

$$DCT(i,j) = \frac{2}{N} c(i)c(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x,y) \cos\left(\frac{\pi}{N} i \left(x + \frac{1}{2}\right)\right) \cos\left(\frac{\pi}{N} j \left(y + \frac{1}{2}\right)\right)$$

avec $c(\alpha) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } \alpha = 0 \\ 1 & \text{sinon} \end{cases}$

DCT bidimensionnelle inverse

$$pixel(x,y) = \frac{2}{N} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} c(i)c(j) DCT(i,j) \cos\left(\frac{\pi}{N} i \left(x + \frac{1}{2}\right)\right) \cos\left(\frac{\pi}{N} j \left(y + \frac{1}{2}\right)\right)$$

Annexe : compression de Huffman

texte à encoder : "le tipe valorise la curiosite"

Comptage du nombre d'occurrences : $[(u, 1), (v, 1), (c, 1), (p, 1), (a, 2), (o, 2), (r, 2), (s, 2), (t, 3), (l, 3), (e, 4), (i, 4), (_, 4)]$

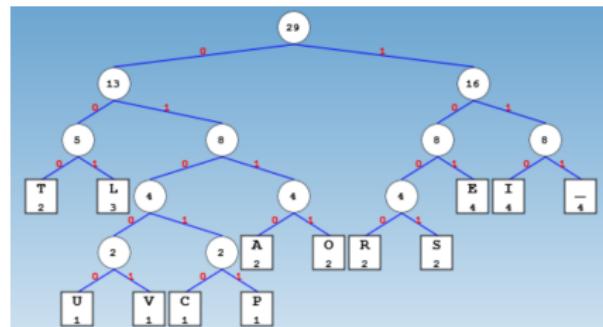


Figure 14: Arbre de Huffman

source de l'image : <http://lwh.free.fr/pages/algo/compression/huffman.html>

résultat : 001 101 111 000 110 01011 101 111 01001 0110 001 0111 1000 110
1001 101 111 001 0110 111 01010 01000 1000 110 0111 1001 110 000 101

Annexe : extrait de la table de Huffman

Table K.5 – Table for luminance AC coefficients (sheet 1 of 4)

Run/Size	Code length	Code word
0/0 (EOB)	4	1010
0/1	2	00
0/2	2	01
0/3	3	100
0/4	4	1011
0/5	5	11010
0/6	7	1111000
0/7	8	11111000
0/8	10	1111110110
0/9	16	111111110000010
0/A	16	111111110000011
1/1	4	1100
1/2	5	11011
1/3	7	1111001
1/4	9	111110110
1/5	11	11111101110
1/6	16	111111110000100
1/7	16	111111110000101
1/8	16	111111110000110
1/9	16	111111110000111
1/A	16	111111110001000
2/1	5	11100
2/2	8	1111001
2/3	10	111110111
2/4	12	111111101100
2/5	16	111111110001001
2/6	16	111111110001010
2/7	16	111111110001011
2/8	16	111111110001100
2/9	16	111111110001101
2/A	16	111111110001110
3/1	6	111010
3/2	9	111110111
3/3	12	111111110101
3/4	16	111111110001111
3/5	16	111111110010000
3/6	16	111111110010001
3/7	16	111111110010010
3/8	16	111111110010011
3/9	16	111111110010100
3/A	16	111111110010101

Table K.5 (sheet 2 of 4)

Run/Size	Code length	Code word
4/1	6	111011
4/2	10	1111111100
4/3	16	11111111110010110
4/4	16	11111111110010111
4/5	16	11111111110011000
4/6	16	11111111110011001
4/7	16	11111111110011010
4/8	16	11111111110011011
4/9	16	11111111110011100
4/A	16	11111111110011101
5/1	7	1111010
5/2	11	11111110111
5/3	16	11111111110011110
5/4	16	11111111110011111
5/5	16	11111111110100000
5/6	16	11111111110100001
5/7	16	11111111110100010
5/8	16	11111111110100011
5/9	16	11111111110100100
5/A	16	11111111110100101
6/1	7	1111011
6/2	12	1111111110110
6/3	16	11111111110110010
6/4	16	11111111110110011
6/5	16	111111111101101000
6/6	16	111111111101101001
6/7	16	111111111101101010
6/8	16	111111111101101011
6/9	16	111111111101101100
6/A	16	111111111101101101
7/1	8	11111010
7/2	12	1111111110111
7/3	16	1111111111011110
7/4	16	1111111111011111
7/5	16	11111111110110000
7/6	16	11111111110110001
7/7	16	11111111110110010
7/8	16	11111111110110011
7/9	16	11111111110110100
7/A	16	11111111110110101
8/1	9	111111000
8/2	15	111111111100000

source : <https://www.w3.org/Graphics/JPEG/itu-t81.pdf>

Annexe : SSIM

Origine du SSIM :

- **Créateurs** : Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, Eero P. Simoncelli
- **Année** : 2004
- **Article** :
"Image Quality Assessment: From Error Visibility to Structural Similarity"
IEEE Transactions on Image Processing, vol. 13, no. 4, Apr. 2004

L'objectif est de remplacer les métriques traditionnelles comme le PSSB qui sont peu corrélées à la perception visuelle.

L'œil humain est particulièrement sensible à la **structure**, au **contraste** et à la **luminance**, plutôt qu'aux erreurs locales en intensité.

$$\text{SSIM} = \underbrace{\frac{(2\mu_x\mu_y + c_1)}{(\mu_x^2 + \mu_y^2 + c_1)}}_{\text{luminance}} \times \underbrace{\frac{(2\sigma_x\sigma_y + c_2)}{(\sigma_x^2 + \sigma_y^2 + c_2)}}_{\text{contraste}} \times \underbrace{\frac{(\text{cov}_{xy} + c_3)}{(\sigma_x\sigma_y + c_3)}}_{\text{structure}}$$

Annexe : SSIM

$$\text{SSIM} = \frac{(2\mu_x\mu_y + c_1)}{(\mu_x^2 + \mu_y^2 + c_1)} \frac{(2\sigma_x\sigma_y + c_2)}{(\sigma_x^2 + \sigma_y^2 + c_2)} \frac{(\text{cov}_{xy} + c_3)}{(\sigma_x\sigma_y + c_3)}$$

- $c_1 = (k_1 L)^2$, $c_2 = (k_2 L)^2$ et $c_3 = \frac{c_2}{2}$: pour stabiliser la division quand le dénominateur est très faible.
- L : la dynamique des valeurs des pixels (255).
- $k_1 = 0,01$ et $k_2 = 0,03$: des petites valeurs.

Annexe : transformation YCbCr RVB

$$Y = 0,299 R + 0,587 V + 0,114 B$$

$$Cb = -0,1687 R - 0,3313 V + 0,5 B + 128$$

$$Cr = 0,5 R - 0,4187 V - 0,0813 B + 128$$

- Les coefficients ont été défini en 1982 par la norme ITU-R BT.601 de l'Union Internationale des Télécommunications (ITU).
- Les coefficients sont basés sur la sensibilité de l'œil aux couleurs.
- La luminance est l'intensité lumineuse perçue par l'œil qui est très sensible au vert, un peu moins au rouge, et beaucoup moins au bleu.

Les composantes **Cb** et **Cr** approximent $B - Y$ et $R - Y$ (différences de couleur).

- **Cb** : axe bleu ↔ jaune
- **Cr** : axe rouge ↔ cyan

Annexe : différentes matrices de quantification

3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3

Table 4: Quantification
constante $Q_{ij} = K$
(ici $K = 3$)

3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

Table 5: Quantification
 $Q_{ij} = 1 + K(1 + i + j)$
(ici $K = 2$)

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Table 6: matrice de
quantification choisie
empiriquement par
JPEG

Annexe : inconvénients de JPEG

- Apparition de blocs de 8×8 pixels : effet de mosaïque.
- Des dégradés avec paliers sont visibles.
- Non adapté pour les dessins, les textes ou les logos car les artefacts y sont fortement visibles (en raison des contrastes importants).
- Non adapté pour les modifications successives, la retouche photo n'est pas possible.

JPEG2000 règle ces problèmes avec l'utilisation d'ondelettes.

Code python : imports et variable globale

```
1 # Les modules utilisés
2 from random import randrange
3 import numpy as np
4 import math
5 import cv2 as cv
6 import matplotlib.pyplot as plt
7 import statistics
8 import copy
9 import os
10
11 bit_dict = {
12     "0/0": "1010", "0/1": "00", "0/2": "01", "0/3": "100", "0/4": "1011", "0/5": "11010",
13     "0/6": "1111000", "0/7": "11111000", "0/8": "1111110110", "0/9": "111111110000010",
14     "1/A": "111111110000011",
15     "1/1": "1100", "1/2": "11011", "1/3": "1111001", "1/4": "111110110", "1/5": "11111110110",
16     "1/6": "111111110000100", "1/7": "111111110000101", "1/8": "111111110000110",
17     "1/9": "111111110000111", "1/A": "111111110001000",
18     "2/1": "11100", "2/2": "11111001", "2/3": "1111110111", "2/4": "111111110100",
19     "2/5": "111111110001001", "2/6": "111111110001010", "2/7": "111111110001011", "2/8": "111111110001100", "2/9": "111111110001101", "2/A": "111111110001110", ...  
# dictionnaire trop long pour tout afficher
...  
"F/0": "11111111001", "F/1": "111111111110101", "F/2": "1111111111110110", "F/3": "1111111111110111", "F/4": "1111111111111000", "F/5": "11111111111111001", "F/6": "11111111111111010", "F/7": "11111111111111011", "F/8": "11111111111111100", "F/9": "11111111111111101", "F/A": "1111111111111110", }
```

Code python : rvb to ycbcr

```
1 def rvb_to_ycbcr(img):
2     """ renvoie 3 nouvelles matrices contenant les valeur Y Cb Cr """
3     hauteur = len(img)
4     largeur = len(img[0])
5     mat_y = [[0 for _ in range(largeur)] for _ in range(hauteur)]
6     mat_cb = [[0 for _ in range(largeur)] for _ in range(hauteur)]
7     mat_cr = [[0 for _ in range(largeur)] for _ in range(hauteur)]
8     for i in range(0, hauteur):
9         for j in range(0, largeur):
10            mat_y[i][j] = round(0.299 * img[i][j][0] + 0.587 * img[i][j][1] +
11                               0.114 * img[i][j][2])
12            mat_cb[i][j] = round(-0.1687 * img[i][j][0] - 0.3313 * img[i][j][1] +
13                                  0.5 * img[i][j][2] + 128)
14            mat_cr[i][j] = round(0.5 * img[i][j][0] - 0.4187 * img[i][j][1] -
15                               0.0813 * img[i][j][2] + 128)
16    return mat_y, mat_cb, mat_cr
```

Code python : YCbCr to RVB

```
1 def ycbcr_to_rvb(mat_y, mat_cb, mat_cr):
2     """ renvoie l'image en RVB """
3     hauteur = len(mat_y)
4     largeur = len(mat_y[0])
5     img = [[[0, 0, 0] for _ in range(largeur)] for _ in range(hauteur)]
6     for i in range(0, hauteur):
7         for j in range(0, largeur):
8             img[i][j] = [round(mat_y[i][j] + 1.402 * (mat_cr[i][j] - 128)),
9                         round(mat_y[i][j] - 0.34414 * (mat_cb[i][j] - 128) -
10                             0.71414 * (mat_cr[i][j] - 128)),
11                         round(mat_y[i][j] + 1.772 * (mat_cb[i][j] - 128))]
12     for k in range(0, 3):
13         if img[i][j][k] < 0:
14             img[i][j][k] = 0
15         elif img[i][j][k] > 255:
16             img[i][j][k] = 255
17
18     return img
```

Code python : sous échantillonnage 420

```
1 def sous_échantillonnage_420(img):
2     """renvoie une matrice sous-échantillonnée en 420"""
3     res_hauteur = int(len(img) / 2)
4     res_largeur = int(len(img[0]) / 2)
5     res = [[0 for _ in range(res_largeur)] for _ in range(res_hauteur)]
6     for i in range(0, res_hauteur):
7         for j in range(0, res_largeur):
8             res[i][j] = round(
9                 (img[2 * i][2 * j] + img[2 * i][2 * j + 1] + img[2 * i + 1][2 * j]
10                + img[2 * i + 1][2 * j + 1]) / 4)
11
12
13 def sous_échantillonnage_420_inverse(mat, hauteur, largeur):
14     """renvoie une matrice sur-échantillonnée en 420 de taille hauteur x largeur"""
15     mat_res = [[0 for _ in range(largeur)] for _ in range(hauteur)]
16     for i in range(0, hauteur):
17         for j in range(0, largeur):
18             mat_res[i][j] = mat[int(i / 2)][int(j / 2)]
19
return mat_res
```

Code python : sous échantillonnage 422

```
1 def sous_échantillonnage_422(img):
2     """renvoie une matrice sous-échantillonnée en 422"""
3     res_hauteur = len(img)
4     res_largeur = int(len(img[0]) / 2)
5     res = [[0 for _ in range(res_largeur)] for _ in range(res_hauteur)]
6
7     for i in range(0, res_hauteur):
8         for j in range(0, res_largeur):
9             res[i][j] = round((img[i][2 * j] + img[i][2 * j + 1]) / 2)
10    return res
11
12
13 def sous_échantillonnage_422_inverse(mat, hauteur, largeur):
14     """renvoie une matrice sur-échantillonnée en 422 de taille hauteur x largeur"""
15     mat_res = [[0 for _ in range(largeur)] for _ in range(hauteur)]
16     for i in range(0, hauteur):
17         for j in range(0, largeur):
18             mat_res[i][j] = mat[i][int(j / 2)]
19     return mat_res
```

Code python : trois fonctions utiles

```
1 def mettre_a_dimension(mat, hauteur, largeur):
2     """ :param mat: une matrice
3     :param hauteur: entier inférieur à la hauteur de la matrice
4     :param largeur: entier inférieur à la largeur de la matrice
5     :return: la matrice extraite en haut à gauche de taille hauteur x largeur """
6     mat_res = [[0 for _ in range(largeur)] for _ in range(hauteur)]
7     for i in range(0, hauteur):
8         for j in range(0, largeur):
9             mat_res[i][j] = mat[i][j]
10    return mat_res
11
12 def une_to_trois(mat):
13     """renvoie la matrice sous forme d'image en nuance de gris"""
14     hauteur = len(mat)
15     largeur = len(mat[0])
16     new = [[[0, 0, 0] for _ in range(largeur)] for _ in range(hauteur)]
17     for i in range(0, hauteur):
18         for j in range(0, largeur):
19             new[i][j] = [mat[i][j], mat[i][j], mat[i][j]]
20     return new
21
22 def ajouter_k_chaque_pixel(mat, k):
23     """ applique un ajout de k à tous les coefficients de la matrice """
24     hauteur = len(mat)
25     largeur = len(mat[0])
26     for i in range(0, hauteur):
27         for j in range(0, largeur):
28             mat[i][j] = mat[i][j] + k
```

Code python : la DCT

```
1 def dct_bidimensionnelle(mat):
2     """ renvoie la dct d'une matrice 8x8 avec la méthode naïve"""
3     res = [[0 for _ in range(8)] for _ in range(8)]
4     c = [math.sqrt(2) / 2, 1, 1, 1, 1, 1, 1, 1]
5     for i in range(0, 8):
6         for j in range(0, 8):
7             somme = 0
8             for x in range(0, 8):
9                 for y in range(0, 8):
10                    somme = somme + mat[x][y] * math.cos(((2 * x + 1) * math.pi *
11                                   i) / 16) * math.cos(
12                                   ((2 * y + 1) * math.pi * j) / 16)
13             res[i][j] = 0.25 * c[i] * c[j] * somme
14     return res
15
16 def dct_bidimensionnelle_inverse(mat_dct):
17     """ renvoie la dct inverse à une matrice 8x8 avec la méthode naïve"""
18     pixel = [[0 for _ in range(8)] for _ in range(8)]
19     c = [math.sqrt(2) / 2, 1, 1, 1, 1, 1, 1, 1]
20     for x in range(0, 8):
21         for y in range(0, 8):
22             somme = 0
23             for i in range(0, 8):
24                 for j in range(0, 8):
25                     somme = somme + c[i] * c[j] * mat_dct[i][j] * math.cos(((2 * x
26                                   + 1) * math.pi * i) / 16) * math.cos(
27                                   ((2 * y + 1) * math.pi * j) / 16)
28             pixel[x][y] = round(0.25 * somme)
29     return pixel
```

Code python : des quantifications

```
1 def quantification_constante(mat_dct, k):
2     """ divise toutes les valeurs d'une matrice 8x8 par k"""
3     for i in range(0, 8):
4         for j in range(0, 8):
5             mat_dct[i][j] = int(mat_dct[i][j] / k)
6
7
8 def quantification_croissante(mat_dct, k):
9     """ divise toutes les valeurs d'une matrice 8x8 par 1 + k * (1+i+j)"""
10    for i in range(0, 8):
11        for j in range(0, 8):
12            mat_dct[i][j] = int(mat_dct[i][j] / (1 + k * (1 + i + j)))
13
14
15 def quantification_croissante_inverse(mat_dct, k):
16     """ multiplie toutes les valeurs d'une matrice 8x8 par 1 + k * (1+i+j)"""
17     for i in range(0, 8):
18         for j in range(0, 8):
19             mat_dct[i][j] = mat_dct[i][j] * (1 + k * (1 + i + j))
```

Code python : la quantification fournie par JPEG

```
1  mat_q_commune = [
2      [16, 11, 10, 16, 24, 40, 51, 61],
3      [12, 12, 14, 19, 26, 58, 60, 55],
4      [14, 13, 16, 24, 40, 57, 69, 56],
5      [14, 17, 22, 29, 51, 87, 80, 62],
6      [18, 22, 37, 56, 68, 109, 103, 77],
7      [24, 35, 55, 64, 81, 104, 113, 92],
8      [49, 64, 78, 87, 103, 121, 120, 101],
9      [72, 92, 95, 98, 112, 100, 103, 99]
10 ]
11
12 def quantification_originale(mat_dct, k):
13     """ applique la quantification avec la matrice établie empiriquement par
14         JPEG """
15     for i in range(0, 8):
16         for j in range(0, 8):
17             mat_dct[i][j] = round((mat_dct[i][j] * k) / (mat_q_commune[i][j]))
18
19 def quantification_originale_inverse(mat_dct, k):
20     """ applique la quantification inverse avec la matrice établie empiriquement
21         par JPEG """
22     for i in range(0, 8):
23         for j in range(0, 8):
24             mat_dct[i][j] = (mat_dct[i][j] * mat_q_commune[i][j]) / k
```

Code python : parcours en zig zag

```
1 def codage_zig_zag(mat):
2     """ renvoie la liste du parcours en zig zag d'une matrice 8x8 """
3     res = [mat[0][0], mat[0][1], mat[1][1], mat[2][0], mat[1][1], mat[0][2],
4            mat[0][3], mat[1][2],
5            mat[2][1], mat[3][0], mat[4][0], mat[3][1], mat[2][2], mat[1][3],
6            mat[0][4], mat[0][5],
7            mat[1][4], mat[2][3], mat[3][2], mat[4][1], mat[5][0], mat[6][0],
8            mat[5][1], mat[4][2],
9            mat[3][3], mat[2][4], mat[1][5], mat[0][6], mat[0][7], mat[1][6],
10           mat[2][5], mat[3][4],
11           mat[4][3], mat[5][2], mat[6][1], mat[7][0], mat[7][1], mat[6][2],
12           mat[5][3], mat[4][4],
13           mat[3][5], mat[2][6], mat[1][7], mat[2][7], mat[3][6], mat[4][5],
14           mat[5][4], mat[6][3],
15           mat[7][2], mat[7][3], mat[6][4], mat[5][5], mat[4][6], mat[3][7],
16           mat[4][7], mat[5][6],
17           mat[6][5], mat[7][4], mat[7][5], mat[6][6], mat[5][7], mat[6][7],
18           mat[7][6], mat[7][7]]
19
20     i = 63
21     while res[i] == 0 and i > 0:
22         del res[i]
23         i -= 1
24
25     return res
```

Code python : deux fonctions auxiliaires de l'encodage

```
1 def decimal_to_binary(x, n):
2     """ renvoie la représentation de x (un nombre en base 10) en binaire sur n
3         bits sous la forme d'une chaîne de caractères"""
4     # Convertir le nombre en binaire et enlever le préfixe '0b'
5     binary = bin(x)[2:]
6     # Si le nombre de bits est plus grand que n on ne peut pas coder x
7     if len(binary) > n:
8         raise ValueError("Le nombre " + str(x) + " ne peut pas être représenté sur
9                         " + str(n) + " bits.")
10    # Compléter avec des zéros à gauche pour atteindre n bits
11    binary = binary.zfill(n)
12    return binary
13
14 def categorie_et_position(x):
15     """ renvoie la catégorie et la position de x"""
16     if not -1023 <= x <= 1023:
17         raise ValueError(str(x) + " doit être entre -1023 et 1023 inclus.")
18     for n in range(1, 11):
19         neg_start = -2**n + 1
20         neg_end = -2**n
21         pos_start = 2**n
22         pos_end = 2**n - 1
23         if neg_start <= x <= neg_end:
24             position = x - neg_start
25             return n, position
26         elif pos_start <= x <= pos_end:
27             position = x
28             return n, position
29     raise ValueError("x ne correspond à aucune catégorie.") # impossible
```

Code python : encodage RLE et Huffman

```
1 def rle_huffman(liste):
2     """ renvoie la taille de la chaine de bits obtenue en appliquant le codage RLE
3         puis Huffman à liste"""
4     transfert = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C',
5                   'D', 'E', 'F']
6     resultat = ""
7     nb_zero = 0
8     for val in liste:
9         if val == 0:
10            nb_zero += 1
11            if nb_zero == 16:
12                resultat = resultat + "1111111001" # ZRL
13                nb_zero = 0
14            else:
15                if val < -1023 or val > 1023: # dans ce cas extrêmement rare
16                    # on rajoute 16 bits pour coder val
17                    resultat = resultat + "0000000000000000"
18                else:
19                    cat, pos = categorie_et_position(val)
20                    resultat = resultat + bit_dict[transfert[nb_zero] + "/" +
21                                         transfert[cat]] + decimal_to_binary(pos, cat)
22                    nb_zero = 0
23    resultat = resultat + "1010" # EOB
24    return len(resultat)
```

Code python : application de JPEG sur chaque bloc 8x8

```
1  def jpeg88(mat, fact_quant):
2      """ :param mat: une matrice dont les dimensions sont des multiples de 8 à
3          valeur dans [-128, 127]
4      :param fact_quant: le facteur de quantification
5      :return: la matrice compressé puis décompressé et la taille de la matrice
6          compressée """
7      mat_resultat = []
8      taille = np.int64(0)
9      h = int(len(mat) / 8)
10     for i in range(h):
11         for _ in range(8):
12             mat_resultat.append([])
13         for j in range(int(len(mat[0]) / 8)):
14             bloc = [mat[k][j * 8:j * 8 + 8] for k in range(i * 8, i * 8 + 8)] # #
15                 extraction du bloc 8x8
16             bloc_dct = dct_bidimensionnelle(bloc) # la dct
17             quantification_originale(bloc_dct, fact_quant) # la quantification
18             taille = taille + rle_huffman(codage_zig_zag(bloc_dct))
19             quantification_originale_inverse(bloc_dct, fact_quant) # la
20                 quantification inverse
21             bloc_dct_inv = dct_bidimensionnelle_inverse(bloc_dct) # la dct inverse
22             for c in range(8):
23                 mat_resultat[c + i * 8] = mat_resultat[c + i * 8] + bloc_dct_inv[c]
24
25     return mat_resultat, taille
```

Code python : application de JPEG sur une matrice

```
1 def jpeg_mat(mat, fact_quant):
2     """ :param mat: une matrice à valeur dans [0, 255]
3     :param fact_quant: le facteur de quantification
4     :return: la matrice compressé puis décompressé et la taille de la matrice
5             compressée"""
6     # Lorsque les dimension ne sont pas un multiple de 8
7     hauteur = len(mat)
8     largeur = len(mat[0])
9
10    rl = largeur % 8
11    if rl != 0: # si il manque des pixels à droite on repete la dernière colonne
12        for i in range(hauteur):
13            mat[i] = mat[i] + [mat[i][largeur - 1]] * (8 - rl)
14    rh = hauteur % 8
15    if rh != 0: # si il manque des pixels en bas on repete la dernière ligne
16        for i in range(8 - rh):
17            mat.append(mat[hauteur - 1])
18
19    ajouter_k_chaque_pixel(mat, -128) # on enleve 128 à chaque coefficient
20    mat_res, taille = jpeg88(mat, fact_quant) # on applique : dct,
21          quantification, huffman
22    ajouter_k_chaque_pixel(mat_res, 128) # on rajoute 128 à chaque coefficient
23
24    return mat_res, taille
```

Code python : Une fonction utile

```
1 def img_int32_to_uint8(img):
2     """converti une image de int32 à uint8"""
3     hauteur = len(img)
4     largeur = len(img[0])
5     for i in range(0, hauteur):
6         for j in range(0, largeur):
7             for k in range(0, 3):
8                 img[i][j][k] = np.uint8(img[i][j][k])
```

Code python : trois fonctions auxiliaires au calcul du SSIM

```
1 def mean(matrix):
2     """renvoie la moyenne d'une matrice"""
3     flat = [val for row in matrix for val in row]
4     return statistics.mean(flat)
5
6 def variance(matrix, mu=None):
7     """renvoie la variance (avec moyenne donnée pour éviter double calcul)"""
8     flat = [val for row in matrix for val in row]
9     if mu is None:
10         mu = statistics.mean(flat)
11     return statistics.variance(flat, xbar=mu)
12
13 def covariance(block1, block2, mu1, mu2):
14     """renvoie la covariance entre deux blocs de mêmes dimensions """
15     flat1 = [val for row in block1 for val in row]
16     flat2 = [val for row in block2 for val in row]
17     n = len(flat1)
18     return sum((x - mu1) * (y - mu2) for x, y in zip(flat1, flat2)) / (n - 1)
```

Code python : calcul du SSIM

```
1 def calcul_ssim(img1, img2, window_size=8):
2     """renvoie le calcul du SSIM"""
3     hauteur = len(img1)
4     largeur = len(img1[0])
5     L = 255
6     k1 = 0.01
7     k2 = 0.03
8     c1 = (k1 * L) ** 2 # = 6.5025
9     c2 = (k2 * L) ** 2 # = 58.5225
10    c3 = c2 / 2 # = 29.26125
11    total_ssim = 0
12    cpt = 0
```

Code python : SUITE du calcul du SSIM

```
1  for i in range(0, hauteur - window_size - 9, window_size):
2      for j in range(0, largeur - window_size - 9, window_size):
3          block1 = extract_block(img1, i, j, window_size)
4          block2 = extract_block(img2, i, j, window_size)
5
6          mu1 = mean(block1)
7          mu2 = mean(block2)
8          sigma1_sq = variance(block1, mu1)
9          sigma2_sq = variance(block2, mu2)
10         sigma1 = sigma1_sq ** 0.5
11         sigma2 = sigma2_sq ** 0.5
12         cov12 = covariance(block1, block2, mu1, mu2)
13
14         numerateur = (2 * mu1 * mu2 + c1) * (2 * sigma1 * sigma2 + c2) *
15             (cov12 + c3)
16         denominateur = (mu1**2 + mu2**2 + c1) * (sigma1_sq + sigma2_sq + c2) *
17             (sigma1 * sigma2 + c3)
18         ssim_val = numerateur / denominateur
19
20         total_ssimm += ssim_val
21         cpt += 1
22
23     return total_ssimm / cpt if cpt > 0 else 0
```

Code python : application de JPEG

```
1 def jpeg(img_rvb, fact_quant=1): # facteur de quantification
2     """ applique la compression et la decompression jpeg à la matrice img_rvb.
3     renvoie le SSIM, le PSSB et la taille de l'image compressée / taille de
4         l'image originale
5     """
6
7     hauteur = len(img_rvb)
8     largeur = len(img_rvb[0])
9
10    mat_y, mat_cb, mat_cr = rvb_to_ycbcr(img_rvb) # RVB -> YCbCr
11
12    mat_y_ech = copy.deepcopy(mat_y) # sous-echantillonnage
13    mat_cb_ech = sous_echantillonnage_420(mat_cb)
14    mat_cr_ech = sous_echantillonnage_420(mat_cr)
15
16    mat_y_decomp, taille_y = jpeg_mat(mat_y_ech, fact_quant) # on applique jpeg à
17        chacune des matrices
18    mat_cb_decomp, taille_cb = jpeg_mat(mat_cb_ech, fact_quant)
19    mat_cr_decomp, taille_cr = jpeg_mat(mat_cr_ech, fact_quant)
20
21    mat_y_decomp_desech = mettre_a_dimension(mat_y_decomp, hauteur-8, largeur-8)
22        # sous echantillonnage inverse
23    mat_cb_decomp_desech = sous_echantillonnage_420_inverse(mat_cb_decomp,
24        hauteur-8, largeur-8)
25    mat_cr_decomp_desech = sous_echantillonnage_420_inverse(mat_cr_decomp,
26        hauteur-8, largeur-8)
27
28    img_decomp = ycbcr_to_rvb(mat_y_decomp_desech, mat_cb_decomp_desech,
29        mat_cr_decomp_desech)
```

Code python : SUITE de l'application de JPEG

```
1     ssim_y = calcul_ssim(mat_y, mat_y_decomp_desech)
2     ssim_cb = calcul_ssim(mat_cb, mat_cb_decomp_desech)
3     ssim_cr = calcul_ssim(mat_cr, mat_cr_decomp_desech)
4     ssim = (2 * ssim_y + ssim_cb + ssim_cr) / 4
5
6     reqm = np.int64(0) # calcul de la REQM
7     for i in range(0, hauteur-8):
8         for j in range(0, largeur-8):
9             for k in range(0, 3):
10                 reqm = reqm + (img_decomp[i][j][k] -
11                               img_rvb[i][j][k])*(img_decomp[i][j][k] - img_rvb[i][j][k])
12
13     reqm = math.sqrt(reqm / (3*(hauteur-8)*(largeur-8)))
14     pssb = 20*math.log10(255/reqm)
15
16     return ssim, pssb, (taille_y + taille_cb +
17                           taille_cr)/(3*8*(hauteur-8)*(largeur-8))
```