COMPRESSION D’IMAGE PAR MÉTHODE JPEG

TIPE 2025 Mathématiques - Informatique

SOMMAIRE

1. La compression JPEG et ses outils

a. Principe

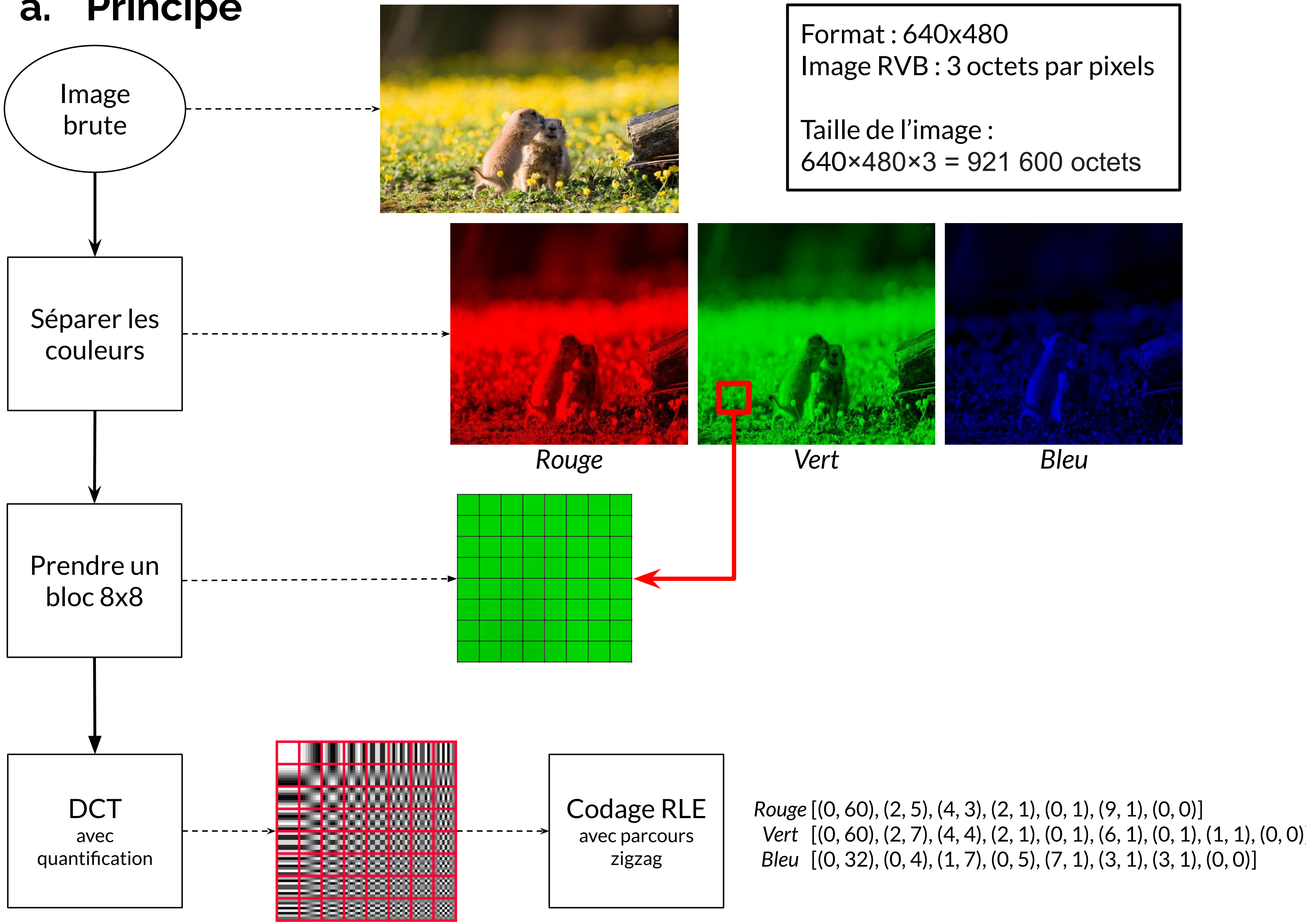


Image brute
Format : 640x480
Image RVB : 3 octets par pixels
Taille de l'image :
640×480×3 = 921 600 octets
Séparer les couleurs
Rouge
Vert
Bleu
Prendre un bloc 8x8
DCT avec quantification
Codage RLE avec parcours zigzag
Rouge [(0, 60), (2, 5), (4, 3), (2, 1), (0, 1), (9, 1), (0, 0)]
Vert [(0, 60), (2, 7), (4, 4), (2, 1), (0, 1), (6, 1), (0, 1), (1, 1), (0, 0)]
Bleu [(0, 32), (0, 4), (1, 7), (0, 5), (7, 1), (3, 1), (3, 1), (0, 0)]

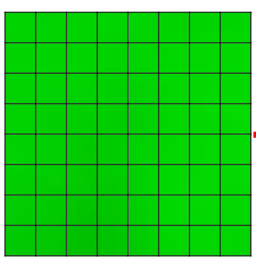
b. Transformée en Cosinus Discrète (DCT)

Pour chaque bloc $N \times N$ avec $(u,v) \in [|0, N-1|]^2$:

$$DCT(u,v) = \frac{2}{N} C(u)C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(u,v) \cos\left[\frac{(2x+1)u\pi}{2N}\right] \cos\left[\frac{(2y+1)v\pi}{2N}\right]$$

$$\text{avec } C(x) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } x = 0 \\ 1 & \text{si } x > 0 \end{cases}$$

Application DCT



$$m = \begin{bmatrix} 121 & 80 & 21 & 12 & 124 & 30 & 51 & 73 \\ 12 & 32 & 41 & 127 & 26 & 48 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 32 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix} \xrightarrow{\text{DCT}} M = \begin{bmatrix} 517 & -130 & -21 & 33 & -2 & 7 & -51 & 0 \\ -123 & 45 & 30 & 9 & 42 & -4 & -16 & 1 \\ 86 & 77 & 8 & -12 & 41 & -28 & -24 & 18 \\ 32 & -4 & 11 & 23 & 14 & -18 & -6 & 21 \\ 22 & 29 & 29 & 40 & 16 & -34 & 0 & 31 \\ 5 & 5 & 36 & 44 & 5 & -33 & 5 & 38 \\ -16 & 5 & 30 & 23 & 5 & -29 & -2 & 37 \\ 1 & -7 & 9 & 16 & -2 & -14 & 0 & 15 \end{bmatrix}$$

_ D'où vient la Transformée en Cosinus Discrète?

Pour chaque bloc $N \times N$ avec $k \in [|0, N-1|]$, et pour un signal $s = (s_0, ..., s_{N-1}) \in \mathbb{R}^N$: $\quad DFT(k) = \sum_{n=0}^{N-1} s_n e^{\frac{-2\pi i k n}{N}}$

$$DCT_{1D}(k) = \Re\mathfrak{e}(DFT(k)) = \sum_{n=0}^{N-1} s_n \cos\left[\frac{\pi}{N}(n + \frac{1}{2})k\right] = \sum_{n=0}^{N-1} s_n \cos\left[\frac{(2n+1)k\pi}{2N}\right]$$

$(DCT_{1D}(0), ..., DCT_{1D}(N-1))$ est l'écriture de $(pixel(0), ..., pixel(N-1))$ sur une base orthonormée $(e_0, ..., e_{N-1})$, où :

$$\forall k \in [|0, N-1|],\ e_i = \left(\sqrt{\frac{2}{N}} C(k) \cos\left[\frac{(2i+1)k\pi}{2N}\right]\right)_{i \in [|0, N-1|]}$$

Où le facteur de normalisation C est défini par : $\forall k \in [|0, N-1|],\ C(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } k = 0 \\ 1 & \text{si } k > 0 \end{cases}$

Notons $P = [(e_0, ..., e_{N-1})]_{(\varepsilon)}$ (avec (ε) la base canonique de $\mathbb{R}^N$: $P \in \mathcal{O}(\mathbb{R})$)

$$\begin{pmatrix} DCT_{1D}(0) \\ \vdots \\ DCT_{1D}(N-1) \end{pmatrix} = P^T \cdot \begin{pmatrix} pixel(0) \\ \vdots \\ pixel(N-1) \end{pmatrix}$$

Ainsi, $\forall k \in [|0, N-1|],\ DCT_{1D}(k) = \sqrt{\frac{2}{N}} C(k) \sum_{n=0}^{N-1} pixel(n) \cos\left[\frac{(2n+1)k\pi}{2N}\right]$ $\quad$ avec $C(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } k = 0 \\ 1 & \text{si } k > 0 \end{cases}$

_ Quantification par matrice standard de luminosité

Matrice obtenue après DCT : $M = \begin{bmatrix} 517 & -130 & -21 & 33 & -2 & 7 & -51 & 0 \\ -123 & 45 & 30 & 9 & 42 & -4 & -16 & 1 \\ 86 & 77 & 8 & -12 & 41 & -28 & -24 & 18 \\ 32 & -4 & 11 & 23 & 14 & -18 & -6 & 21 \\ 22 & 29 & 29 & 40 & 16 & -34 & 0 & 31 \\ 5 & 5 & 36 & 44 & 5 & -33 & 5 & 38 \\ -16 & 5 & 30 & 23 & 5 & -29 & -2 & 37 \\ 1 & -7 & 9 & 16 & -2 & -14 & 0 & 15 \end{bmatrix}$

$$p \times Q = p \times \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

*M / (p*Q))*

terme à terme

Quantification

$$M' = \begin{bmatrix} 32 & -12 & -2 & 2 & 0 & 0 & 0 & 0 \\ -10 & 4 & 2 & 0 & 2 & 0 & 0 & 0 \\ -1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 6 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Q : Matrice de quantification (compression ≈ 50%)
p : facteur de compression/de qualité (réel ≥ 1)

c. Codage par plages (RLE)

Parcours en zigzag

$$M' = \begin{bmatrix} 32 & -12 & -2 & 2 & 0 & 0 & 0 & 0 \\ -10 & 4 & 2 & 0 & 2 & 0 & 0 & 0 \\ -1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 6 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Zigzag sur *M'* :

L = [32, -12, -10, -1, 4, -2, 2, 2, -1, 6, 2, 6, 0, 0, 0, 0, 2, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

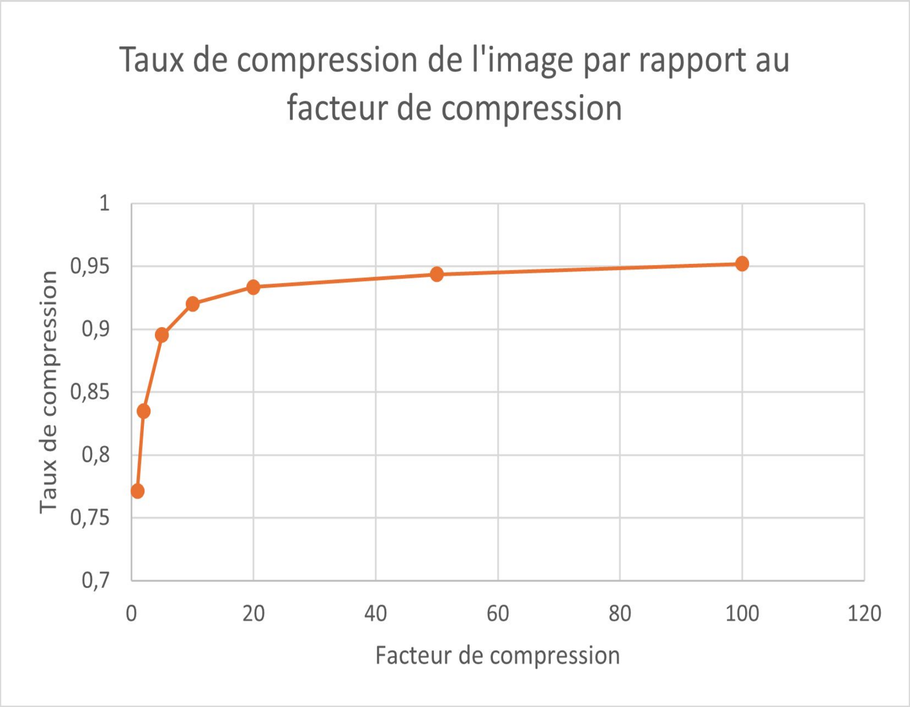
Application RLE

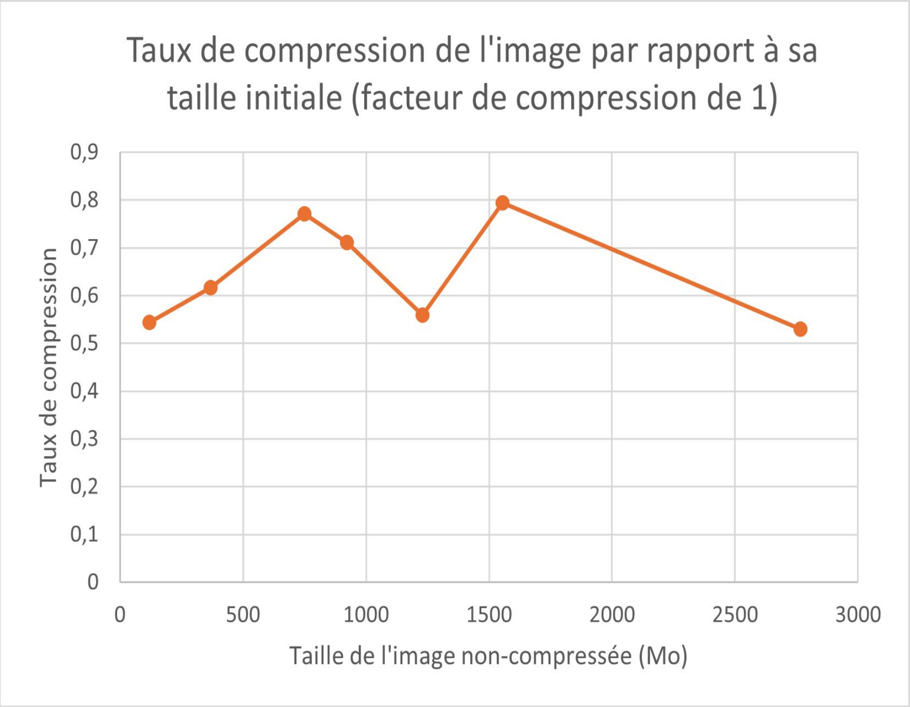
[(0,-1),(0,6)]   [(4,2)]

RLE sur *L* :

L' = [(0, 32), (0, -12), (0, -10), (0, -1), (0, 4), (0, -2), (0, 2), (0, 2), (0, -1), (0, 6), (0, 2), (0, 6), (4, 2), (0, 1), (0, 0)]

Résultats et problèmes rencontrés





Sans np.round() , image plus lourde :

[(0, 32.296875000000001), (0, -11.836334668845568), (0, -10.285761344047865), (0, 6.1081237788658695), (0, 3.789461803479678), (0, -2.1449019825281326), (0, 2.04983591802656), (0, 2.12412873605273), (0, 5.952375721654753), ...)

Facteur trop grand :

p = 22 renvoie [(0, 1), (0, -1), (0, 0)]

p = 65 renvoie [(0,0)]

2. La décompression et ses outils

a) Principe

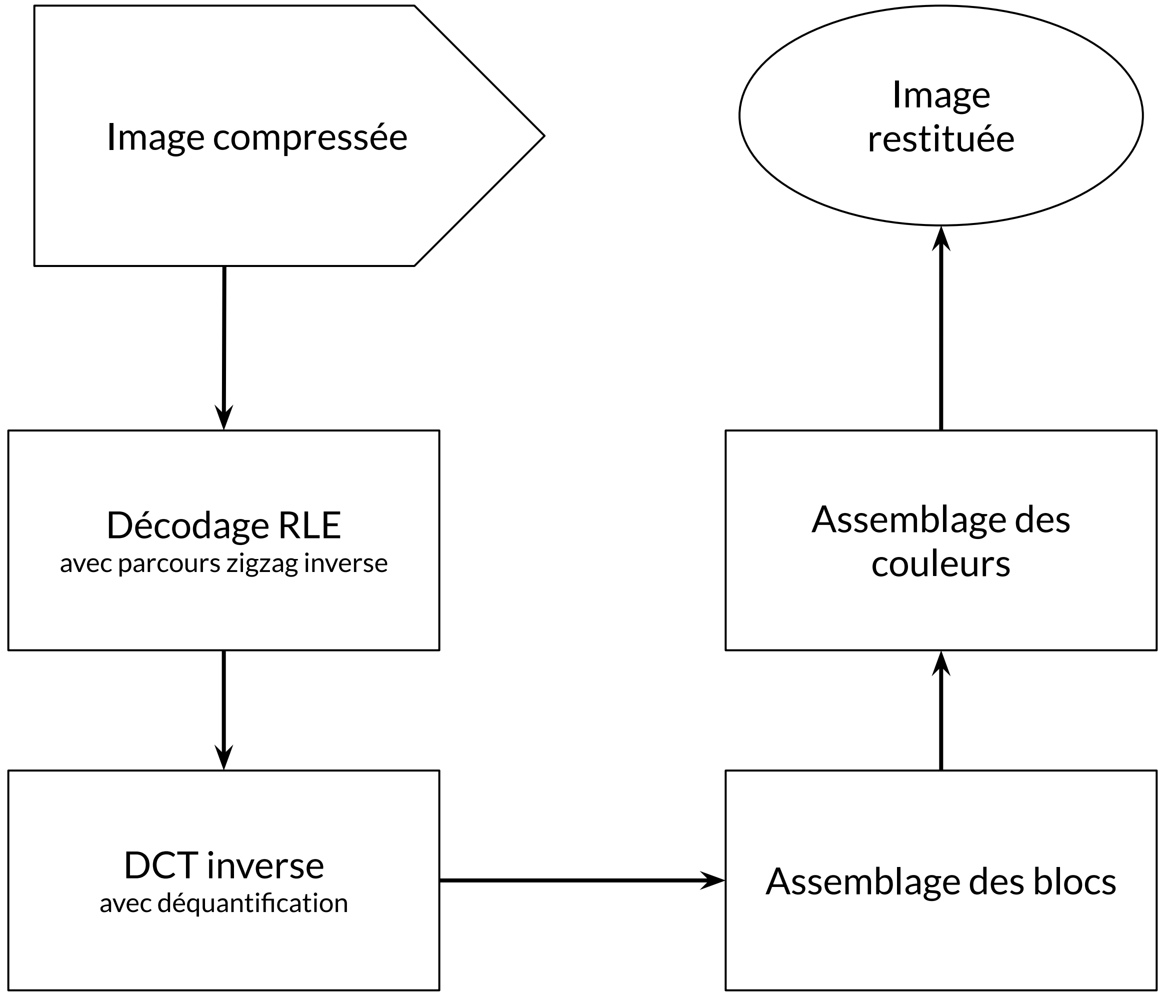


Image compressée
Décodage RLE
avec parcours zigzag inverse
DCT inverse
avec déquantification
Assemblage des blocs
Assemblage des
couleurs
Image
restituée

b) Codage par plages inversé (iRLE)

Application iRLE

L' = [(0, 32), (0, -12), (0, -10), (0, -1), (0, 4), (0, -2), (0, 2), (0, 2), (0, -1), (0, 6), (0, 2), (0, 6), (4, 2), (0, 1), (0, 0)]

L = [32, -12, -10, -1, 4, -2, 2, 2, -1, 6, 2, 6, 0, 0, 0, 0, 2, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Parcours en zigzag inversé

L = [32, -12, -10, -1, 4, -2, 2, 2, -1, 6, 2, 6, 0, 0, 0, 0, 2, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

$$M' = \begin{bmatrix} 32 & -12 & -2 & 2 & 0 & 0 & 0 & 0 \\ -10 & 4 & 2 & 0 & 2 & 0 & 0 & 0 \\ -1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 6 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

c) Transformée en Cosinus Discrète inverse (iDCT)

Pour chaque bloc $N \times N$ avec $(x, v) \in [|0, N-1|]^2$:

$$pixel(x,y) = \frac{2}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} C(u)C(v)DCT(u,v) \cos\left[\frac{(2x+1)u\pi}{2N}\right] \cos\left[\frac{(2y+1)v\pi}{2N}\right]$$

$$\text{avec } C(x) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } x = 0 \\ 1 & \text{si } x > 0 \end{cases}$$

En utilisant le même énoncé que la diapositivie 6 :
Soit $P = [(e_0, ..., e_{N-1})]_{(\varepsilon)}$ (avec (ε) la base canonique de $\mathbb{R}^N$).

$$\begin{pmatrix} DCT_{1D}(0) \\ \vdots \\ DCT_{1D}(N-1) \end{pmatrix} = P^T \cdot \begin{pmatrix} pixel(0) \\ \vdots \\ pixel(N-1) \end{pmatrix}$$

$$\text{Transformation inverse : } \begin{pmatrix} pixel(0) \\ \vdots \\ pixel(N-1) \end{pmatrix} = P \cdot \begin{pmatrix} DCT_{1D}(0) \\ \vdots \\ DCT_{1D}(N-1) \end{pmatrix}$$

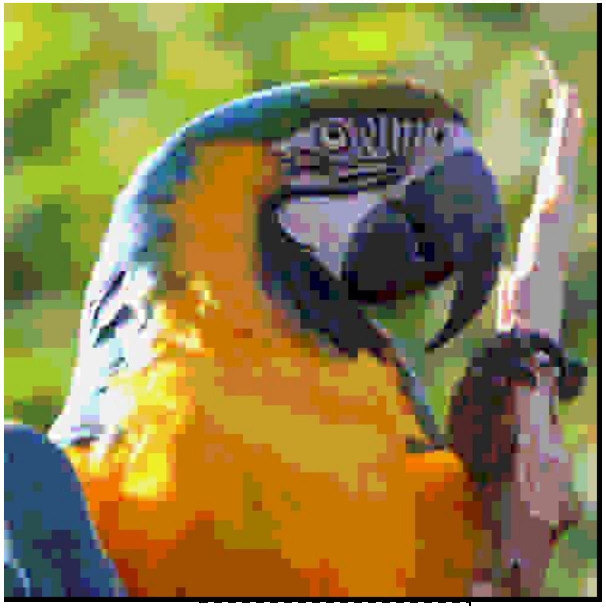
c'est à dire, $\forall k \in [|0, N-1|]$, $pixel(k) = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} C(n) DCT_{1D}(n) \cos\left[\frac{(2k+1)n\pi}{2N}\right]$

avec $C(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } k = 0 \\ 1 & \text{si } k > 0 \end{cases}$

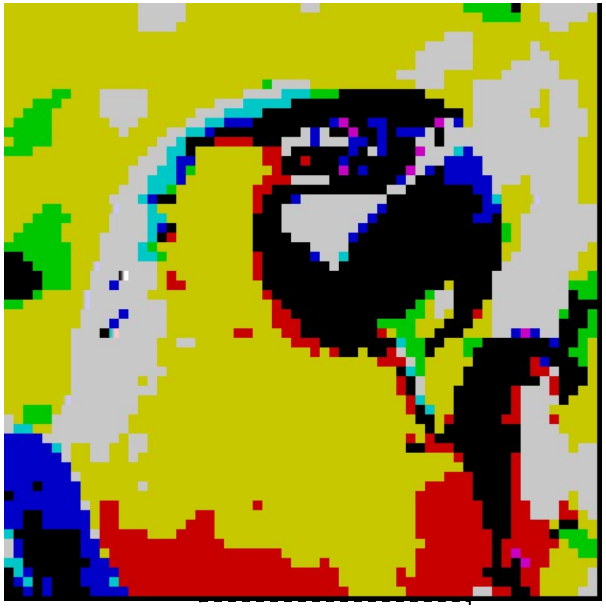
_ Résultats

Image d'origine

Facteur p=1

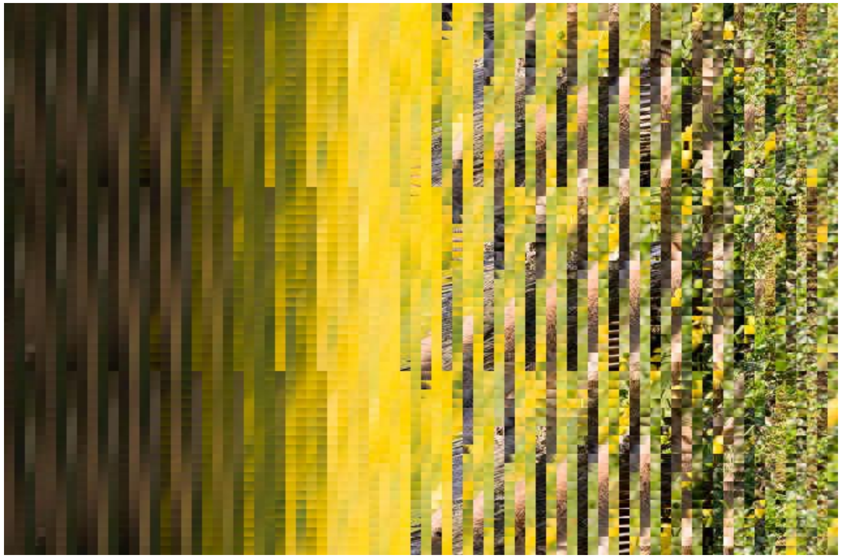


Facteur p=20



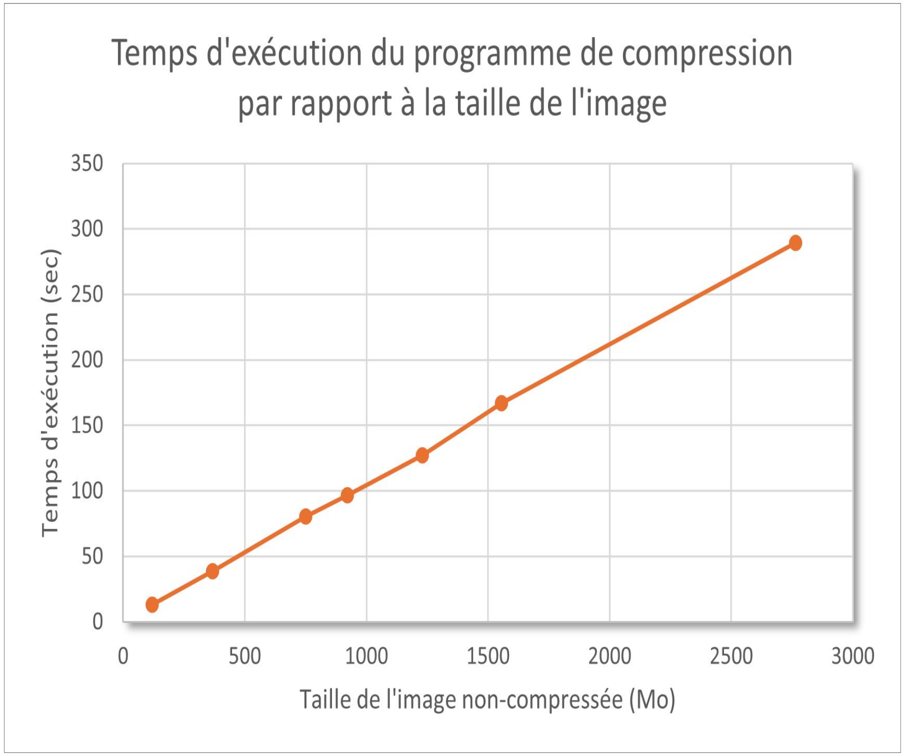
Facteur p=100

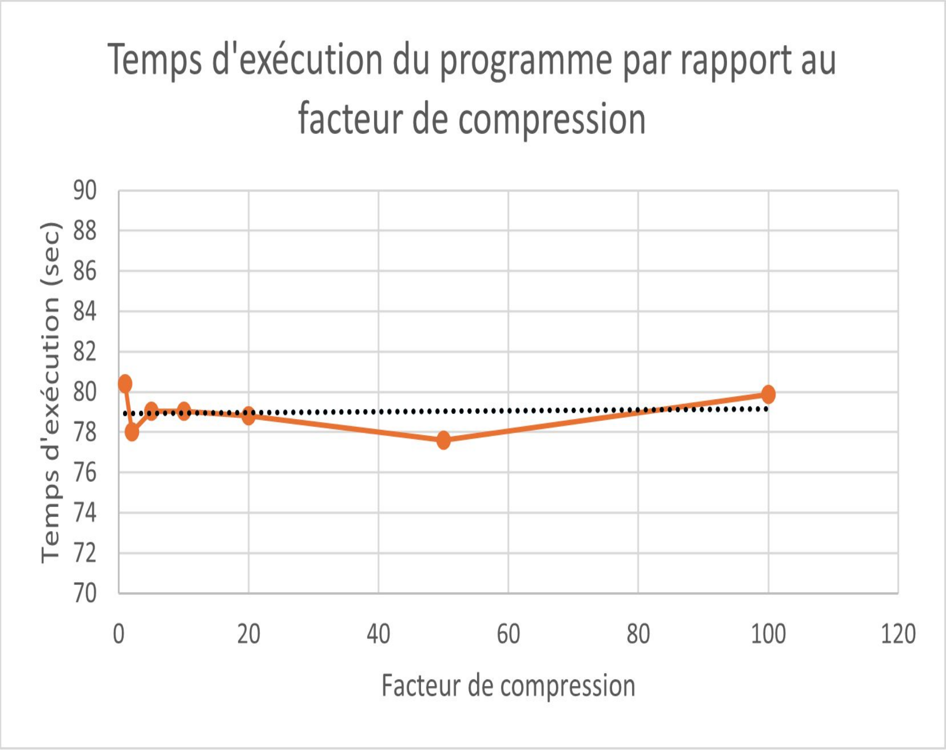
Problèmes rencontrés



3. Limites de la compression JPEG

a) Temps et complexité





— Complexité

Nombre de blocs 8x8 d'une image de taille HxL : $N = \frac{H \times L}{8^2}$

```python
for u in range(8):
  for v in range(8):
    for x in range(8):
      for y in range(8):
```

DCT :

par bloc : $\mathcal{O}(8^4) = \mathcal{O}(4096) = \mathcal{O}(1)$
au total : $\mathcal{O}(N) \cdot \mathcal{O}(1) = \mathcal{O}(N)$

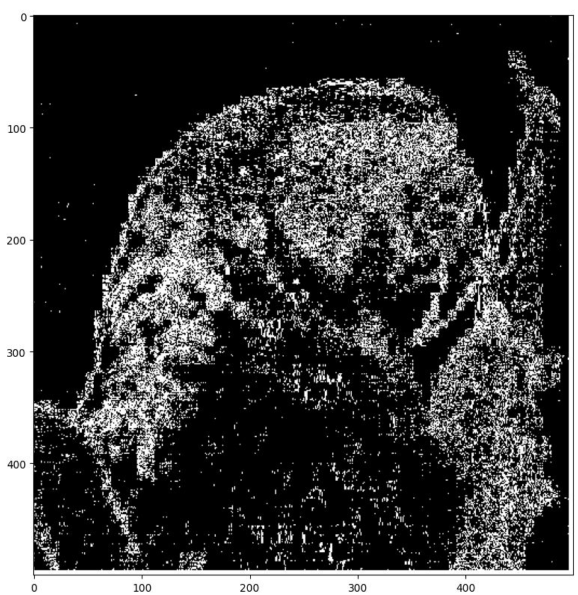
De même pour la quantification, Zigzag et RLE : $\mathcal{O}(N)$
$\longrightarrow$ Complexité compression : $4 \times \mathcal{O}(N) = \mathcal{O}(N)$

De même pour la décompression : $\mathcal{O}(N)$

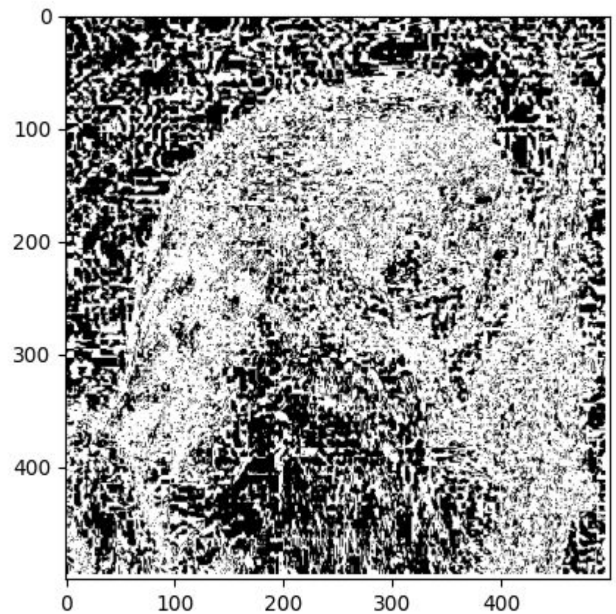
Complexité total : $2 \times \mathcal{O}(N) = \mathcal{O}(N)$

b) Les pertes d'informations

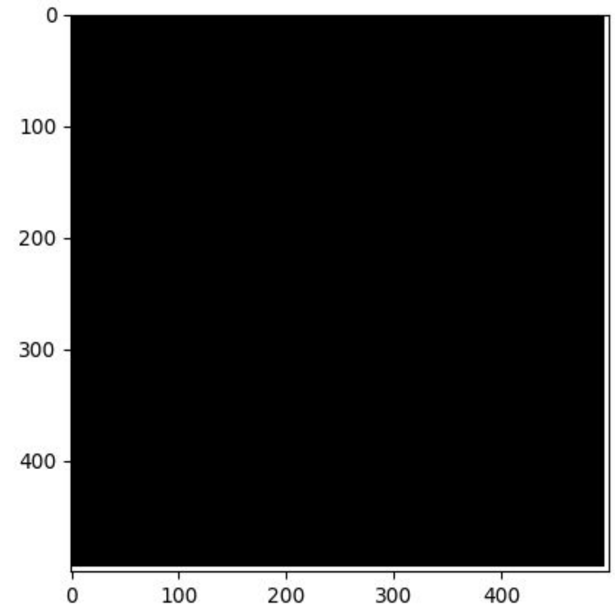
Via Matplotlib :



Facteur p=1



Facteur p=5



Sans quantification

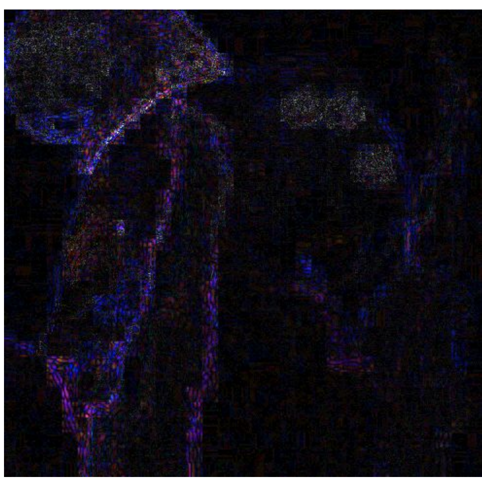
c) Extension sur la détection de photomontages

Comme la compression laisse des traces, des trucages sont alors détectables

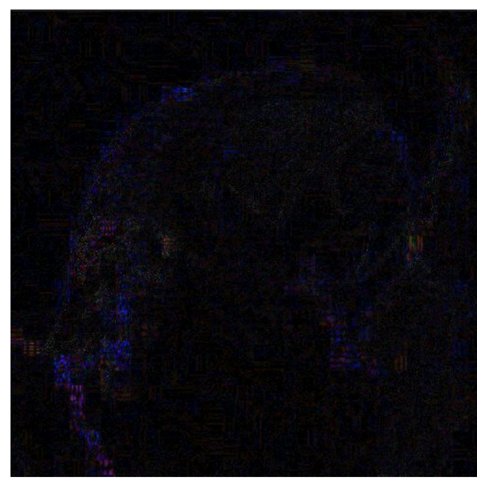
Méthode Error Level Analysis (ELA) :

Image truquée

Image originale



Après ELA



Après ELA

Conclusion

Annexe

Annexe 1 : Transformée en cosinus discrète en Python

```
import numpy as np
from PIL import Image


# Fonction pour la DCT 2D sur un bloc 8x8
def dct_2d(bloc):
    N = 8
    dct_bloc = np.zeros((N, N))

    for u in range(N):
        for v in range(N):
            somme = 0

            for x in range(N):
                for y in range(N):

                    somme += bloc[x, y] * np.cos(((2*x+1)*u*np.pi)/(2*N)) * np.cos(((2*y+1)*v*np.pi)/(2*N))

            C_u = np.sqrt(1/N) if u == 0 else np.sqrt(2/N)
            C_v = np.sqrt(1/N) if v == 0 else np.sqrt(2/N)
            dct_bloc[u][v] = round(C_u * C_v * somme)

    return dct_bloc
```

```python
def zigzag(bloc):
    zigzag_ordre = []
    N = 8

    for i in range(2*N - 1):

        if i % 2 == 0:

            for j in range(i + 1):
                if i - j < N and j < N:
                    zigzag_ordre.append(bloc[i - j, j])

        else:
            for j in range(i + 1):
                if j < N and i - j < N:
                    zigzag_ordre.append(bloc[j, i - j])

    return zigzag_ordre
```

Annexe 2 : Parcours en zigzag en Python

```python
def rle(bloc_zigzaggé):
    resultat = []
    zero_compteur = 0

    for num in bloc_zigzaggé:
        if num == 0:
            zero_compteur += 1

        else:
            resultat.append((int(zero_compteur), int(num)))
            zero_compteur = 0

    resultat.append((0, 0))  # Fin de bloc
    return resultat
```

Annexe 3 : Codage par plage sur une matrice en Python

Annexe 4 : Codage par plage inversé en Python

```
def irle(rle_compressé):
    resultat = []

    for (zeros, num) in rle_compressé:
        resultat.extend([0] * zeros)

        if num != 0:
            resultat.append(num)

    return resultat
```

Annexe 5 : Parcours en zigzag inversé en Python

```
def inverse_zigzag(zigzag_liste):
    N = 8
    index = 0
    bloc = np.zeros((N, N))

    for i in range(2 * N - 1):

        if index >= len(zigzag_liste):  # Vérifie que l'on ne dépasse pas la longueur de la liste
            break

        if i % 2 == 0:
            for j in range(i + 1):
                if i - j < N and j < N and index < len(zigzag_liste):
                    bloc[i - j, j] = zigzag_liste[index]
                    index += 1
        else:
            for j in range(i + 1):
                if j < N and i - j < N and index < len(zigzag_liste):
                    bloc[j, i - j] = zigzag_liste[index]
                    index += 1

    return bloc
```

Annexe 6 : Transformée en cosinus discrète inverse en Python

```
def idct_2d(bloc):
    N = 8
    idct_bloc = np.zeros((N, N))

    for x in range(N):
        for y in range(N):
            somme = 0

            for u in range(N):
                for v in range(N):

                    C_u = np.sqrt(1/N) if u == 0 else np.sqrt(2/N)
                    C_v = np.sqrt(1/N) if v == 0 else np.sqrt(2/N)
                    somme += C_u * C_v * bloc[u][v] * np.cos(((2*x+1)*u*np.pi)/(2*N)) * np.cos(((2*y+1)*v*np.pi)/(2*N))

            idct_bloc[x, y] = round(somme)

    return idct_bloc
```

Annexe 7 : Fonction de compression en Python

```
def compresser_canal(canal):
    global facteur_compression
    canal_compressé = []

    for i in range(hauteur_blocs):
        for j in range(largeur_blocs):

            # Extraire le bloc 8x8
            bloc = canal[i*8:(i+1)*8, j*8:(j+1)*8]

            # Appliquer la DCT manuellement
            dct_bloc = dct_2d(bloc)

            #Quantifier les coeffs
            bloc_quantifié = np.round(dct_bloc / (facteur_compression * matrice_quantification))

            # Zigzag
            zigzaggé = zigzag(bloc_quantifié)

            # RLE
            rle_liste = rle(zigzaggé)

            canal_compressé.append(rle_liste)

    return canal_compressé
```

Annexe 8 : Fonction de décompression en Python

```
def decompresser_canal(données_compressées,canal_decompressé):
    global facteur_compression
    données_decompressées=[]

    #IRLE + Inverse zigzag
    for i in range(len(données_compressées)):

        #IRLE
        irle_matrice = irle(données_compressées[i])

        #Inverse zigzag
        inv_zigzag = inverse_zigzag(irle_matrice)

        données_decompressées.append(inv_zigzag)

    #IDCT et déquantification
    bloc_indice=0
    for i in range(hauteur_blocs):
        for j in range(largeur_blocs):

            #Déquantification
            bloc = données_decompressées[bloc_indice] * facteur_compression * matrice_quantification

            #IDCT
            idct_bloc = idct_2d(bloc)

            canal_decompressé[i*8:(i+1)*8, j*8:(j+1)*8] = np.clip(idct_bloc, 0, 255)
            bloc_indice += 1

    return np.round(canal_decompressé)
```

Annexe 9 : La partie du script qui appelle les fonctions et manipule l'image sur Python

```
# Charger l'image (en RVB)
image = np.array(Image.open('E:\\TIPE Maths COMPRESSION MP\\perroquet.jpg'))
facteur_compression = 1

# Séparer l'image en trois canaux (R, V, B)
R, V, B = image[:, :, 0], image[:, :, 1], image[:, :, 2]

# Diviser l'image en blocs 8x8
hauteur, largeur = image.shape[0], image.shape[1]
hauteur_blocs = hauteur // 8
largeur_blocs = largeur // 8

# Initialiser les matrices pour stocker les canaux compressés
R_zeros = np.zeros((hauteur, largeur))
V_zeros = np.zeros((hauteur, largeur))
B_zeros = np.zeros((hauteur, largeur))


# Compresser et décompresser les trois canaux séparément
R_compressé = compresser_canal(R)
V_compressé = compresser_canal(V)
B_compressé = compresser_canal(B)

R_decompressé = decompresser_canal(R_compressé,R_zeros)
V_decompressé = decompresser_canal(V_compressé,V_zeros)
B_decompressé = decompresser_canal(B_compressé,B_zeros)


# Fusionner les trois canaux compressés
image_decompressée = np.stack([R_decompressé, V_decompressé, B_decompressé], axis=2)

# S'assurer que les valeurs sont dans l'intervalle [0, 255]
image_decompressée = np.clip(image_decompressée, 0, 255).astype(np.uint8)

# Sauvegarder l'image compressée
img_decompressée = Image.fromarray(image_decompressée)
img_decompressée.save('decompressed_image_perroquet.jpg')
img_decompressée.show()
```

Annexe 10 : Visualisation des pertes par Matplotlib sur Python

```
import matplotlib.pyplot as plt

diff = np.abs(image.astype(np.int16) - image_decompressée.astype(np.int16))

masque = np.sum(diff, axis=2) > 10
image_diff_binaire = np.zeros_like(image, dtype=np.uint8)
image_diff_binaire[masque] = [255, 255, 255]

plt.imshow(image_diff_binaire)
plt.show()
```

Annexe 11 : Méthode Error Level Analysis en Python

```
image = Image.open('E:\\TIPE Maths COMPRESSION MP\\perroquet.jpg')
img_decompressée = Image.open("decompressed_image_perroquet.jpg")

# ELA
ela_image = ImageChops.difference(image, img_decompressée)
ela_image = ela_image.point(lambda x: x * 5)  # Amplifie les différences
ela_image = ImageOps.invert(ela_image)

ela_image.show()
```