Présentation TIPE

Vidéosurveillance et compression de données

Ethan Bandasack – Candidat n°11973

Session 2023 – Thème : la ville

Sommaire

Des caméras dans la ville

Caméras de surveillance (« vidéo-protection »)

Ville	**Nombre de caméras**	**(pour 100.000 habitants)**
Paris	44 962	404
Nice	2 666	771
Londres	127 373	1 335
(Chine)	——	37 280

Estimations de 2022. Source : *https ://www.comparitech.com/vpn-privacy/the-worlds-most-surveilled-cities/*

Big data

Estimation naïve :

— Résolution 2560×1920 pixels « 5 MP »

— 15 images par seconde

— Profondeur de couleur « 8-bit color », 8 bits utilisés par pixel

$\longrightarrow$ $1{,}9 \times 10^{13}$ octets/jour/caméra

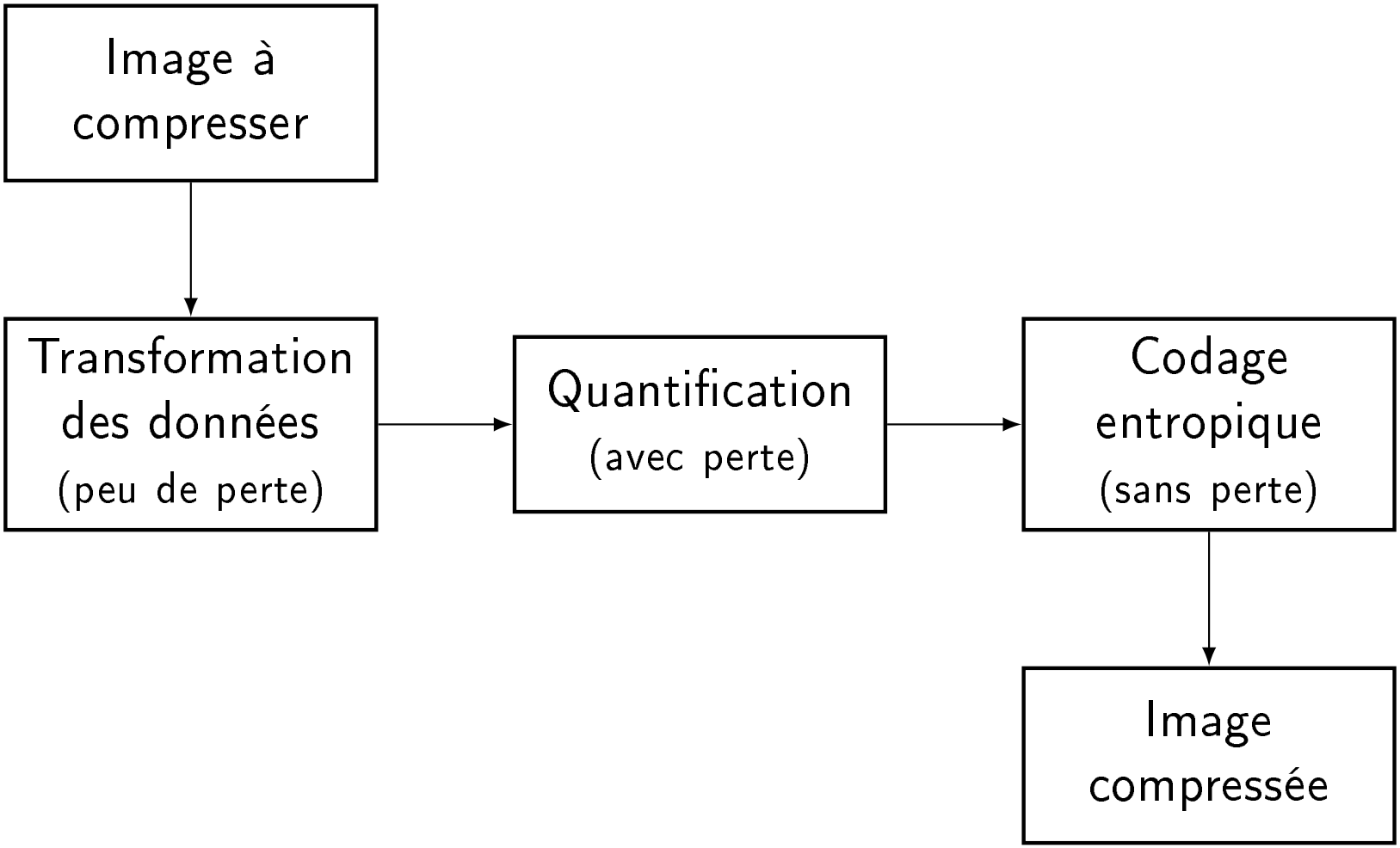
$\longrightarrow$ $8{,}6 \times 10^{17}$ octets/jour pour Paris

(Capacité du plus gros *data center* (Utah, États-Unis) : 10^{18} octets...)

Problématique retenue

Quelles sont les différentes méthodes à disposition pour compresser un flux vidéo ?

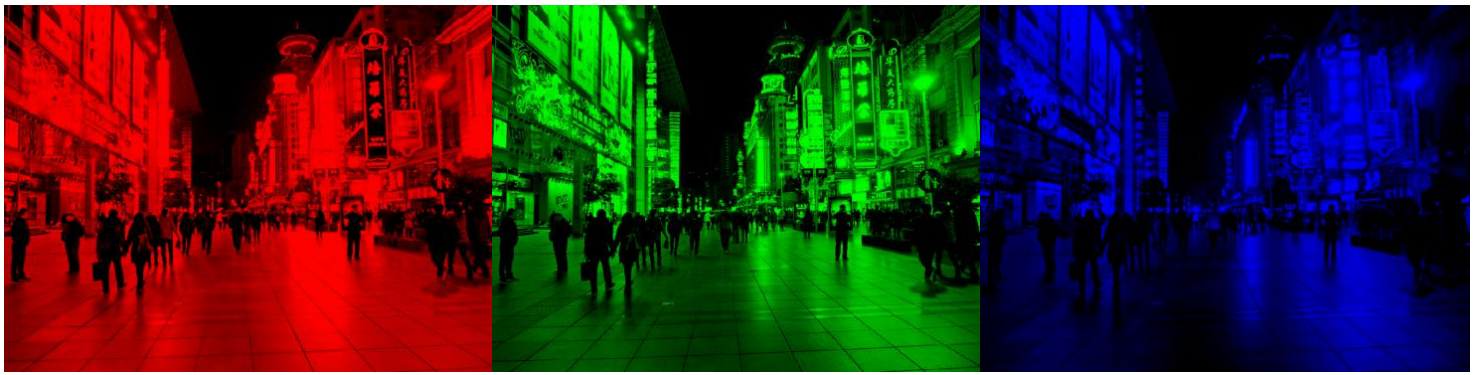
Principe de la compression



Un bloc

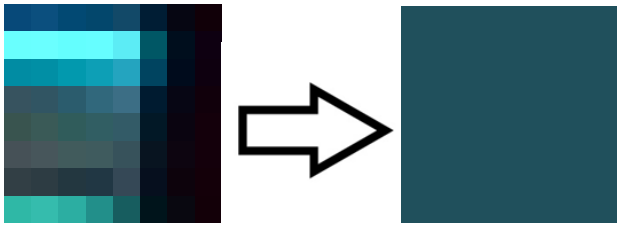
1 pixel

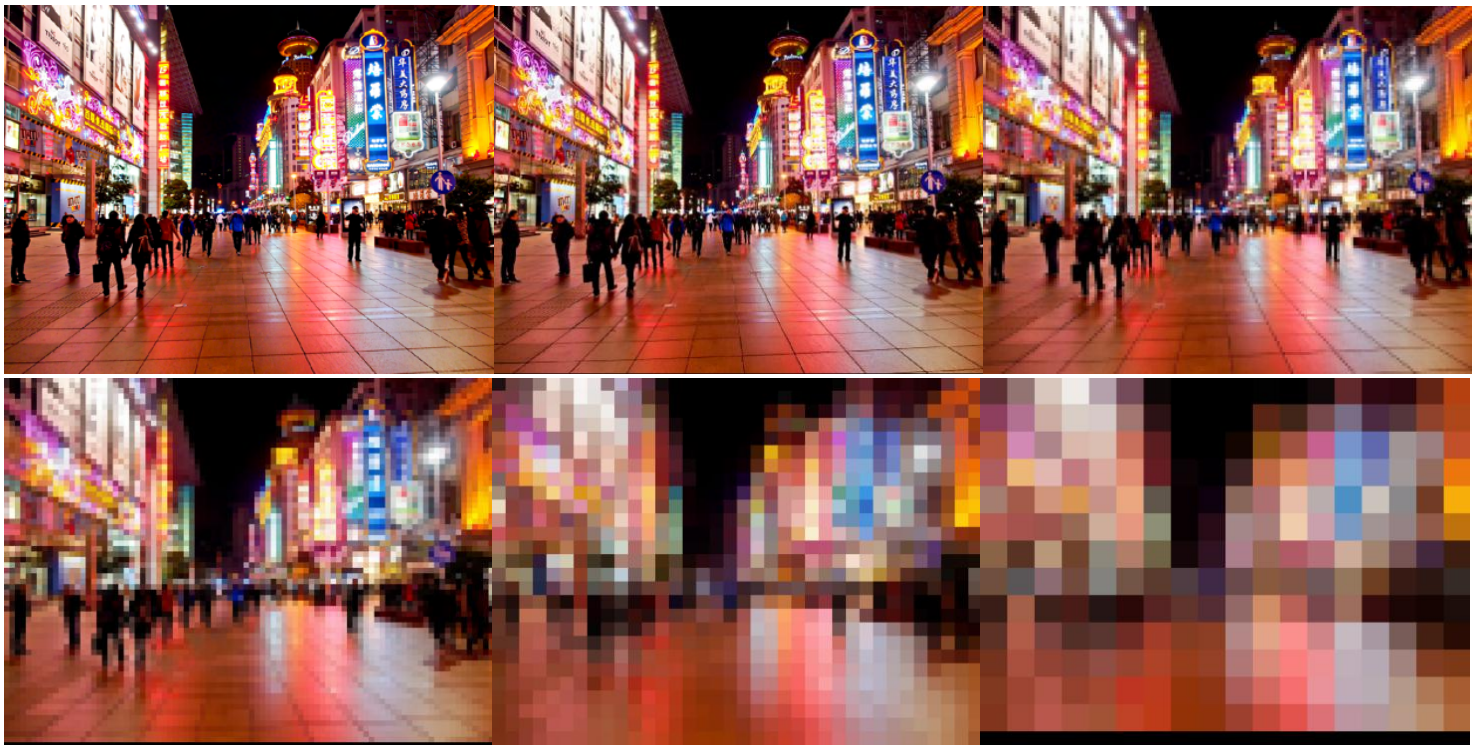
RGB : $(r, g, b) \in [\![0, 255]\!]^3$



Luminance / Chrominances : $(Y, C_r, C_b) \in [\![0; 255]\!] \times [\![-128; 127]\!]^2$

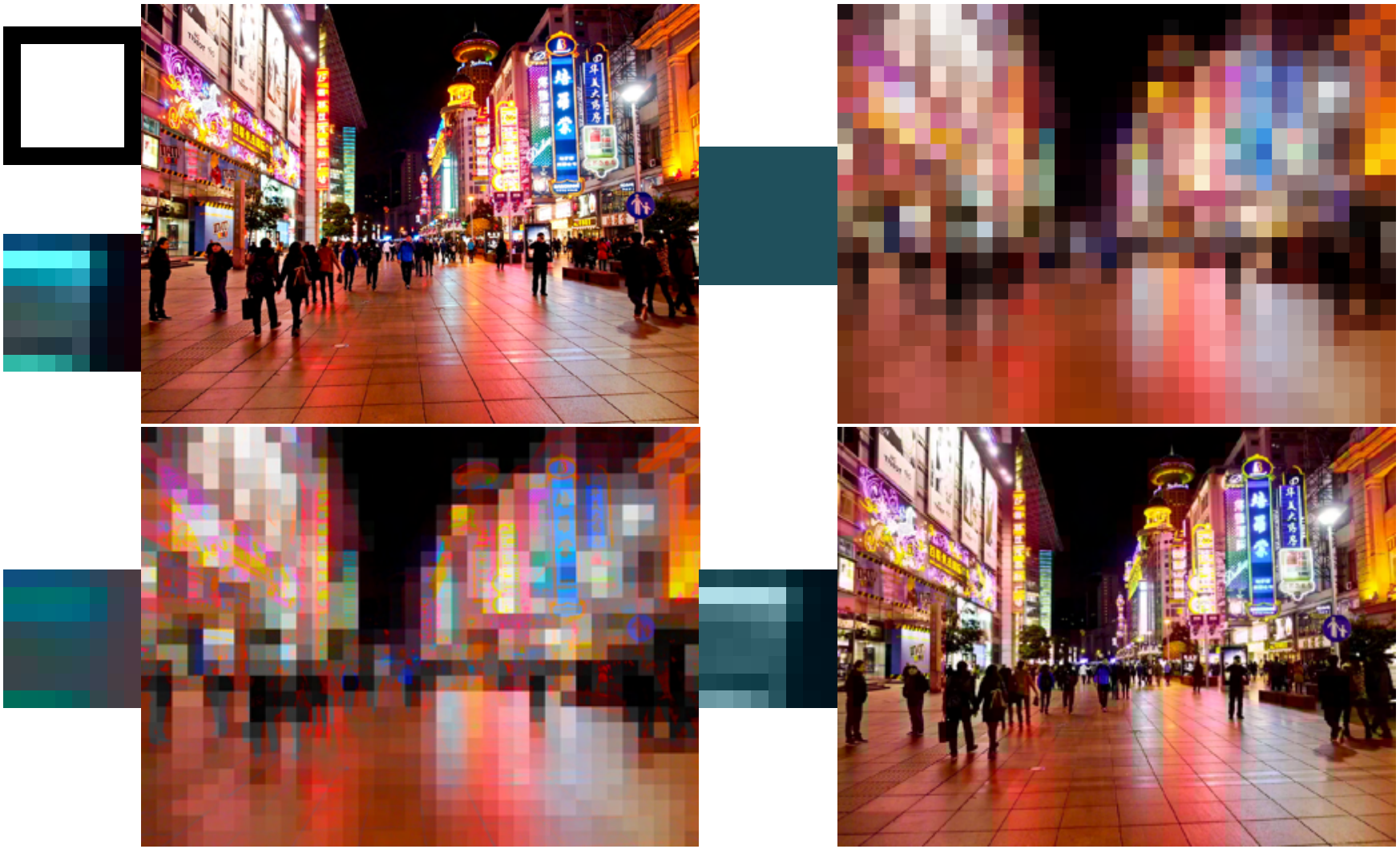
Compression naïve





Compression par composante RGB

Compression par composante YCrCb



Transformée de Fourier

Transformée de Fourier à 1 dimension

Soit $f : \mathbb{R} \to \mathbb{C}$ intégrable.

La transformée de Fourier de f est $\hat{f} : \left\{ \begin{array}{ccc} \mathbb{R} & \longrightarrow & \mathbb{C} \\ \nu & \longmapsto & \int_{\mathbb{R}} f(t) \mathrm{e}^{-\mathrm{i}2\pi\nu t} \mathrm{d}t \end{array} \right.$.

Si $\hat{f}$ est intégrable, on a : $\forall x \in \mathbb{R}, f(x) = \int_{\mathbb{R}}, \hat{f}(\nu) \mathrm{e}^{\mathrm{i}2\pi\nu x} \mathrm{d}\nu$.

$\longrightarrow$ Transformée de Fourier à 2 dimensions
$\longrightarrow$ Transformée de Fourier discrète
$\longrightarrow$ Transformée de Fourier discrète à 2 dimensions
$\longrightarrow$ Transformée en cosinus discrète

Définitions : *cf Annexe*

Transformée en cosinus discrète (DCT)

Transformée en cos discrète à 2 dimensions (2D DCT-II/III)

La transformée en cosinus discrète du signal s est :

$$S(k,l)=\frac{2}{\sqrt{NM}}\sum_{(n,m)\in\mathcal{N}}\alpha_k\alpha_l s(n,m)\cos\left(\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right)\cos\left(\frac{\pi}{M}\left(m+\frac{1}{2}\right)l\right).$$

On a alors, pour $(n,m)\in\mathcal{N}$: ($\alpha_x=\frac{1}{\sqrt{2}}$ si x=0, 1 sinon)

$$s(n,m)=\frac{2}{\sqrt{NM}}\sum_{(k,l)\in\mathcal{N}}\alpha_k\alpha_l S(k,l)\cos\left(\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right)\cos\left(\frac{\pi}{M}\left(m+\frac{1}{2}\right)l\right).$$

En matriciel :

$$\mathfrak{s} = \begin{pmatrix} s(0,0) & \dots & s(0,M-1) \\ \vdots & \ddots & \vdots \\ s(N-1,0) & \dots & s(N-1,M-1) \end{pmatrix}, \quad \mathscr{S} = \begin{pmatrix} S(0,0) & \dots & S(0,M-1) \\ \vdots & \ddots & \vdots \\ S(N-1,0) & \dots & S(N-1,M-1) \end{pmatrix}$$

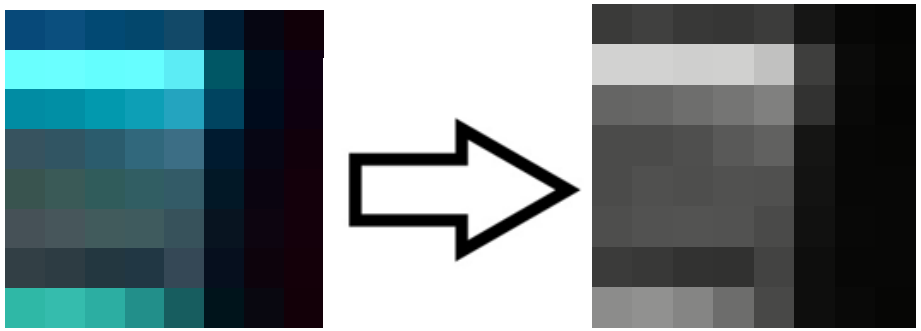
$$C = \left(\alpha_a \cos\left(\frac{\pi}{N} \left(b + \frac{1}{2} \right) a \right) \right)_{\substack{0 \leq a \leq N-1 \\ 0 \leq b \leq M-1}}$$

$$\mathscr{S} = C \mathfrak{s} D^T$$

$$D = \left(\alpha_a \cos\left(\frac{\pi}{M} \left(b + \frac{1}{2} \right) a \right) \right)_{\substack{0 \leq a \leq N-1 \\ 0 \leq b \leq M-1}}$$

$$\mathfrak{s} = C^T \mathscr{S} D$$

Mise en pratique (luminance)

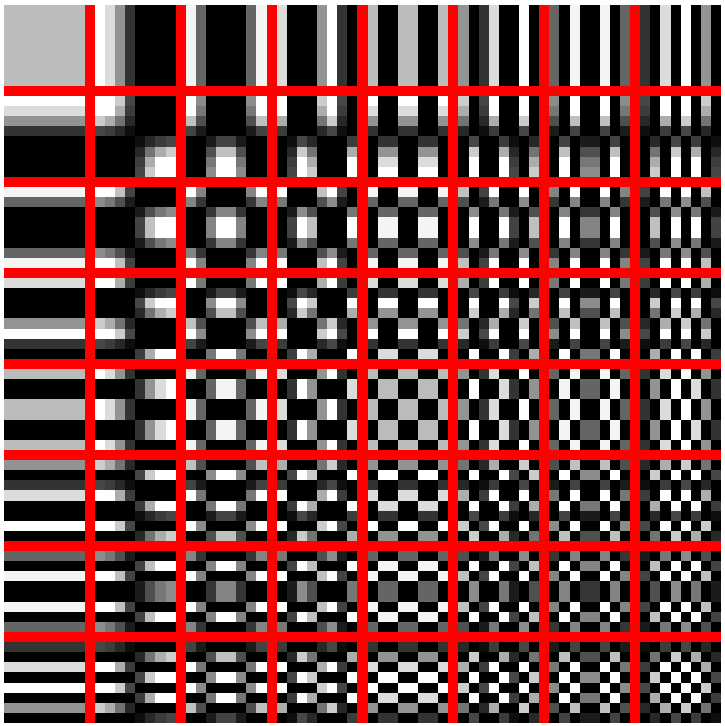


-70	-64	-72	-73	-69	-106	-121	-121
80	83	79	78	65	-65	-118	-119
-28	-26	-18	-11	0	-78	-120	-120
-53	-53	-49	-39	-30	-107	-121	-121
-53	-48	-49	-48	-49	-110	-122	-121
-50	-46	-46	-45	-54	-110	-120	-121
-68	-73	-78	-79	-61	-114	-122	-121
13	17	5	-18	-56	-115	-119	-121

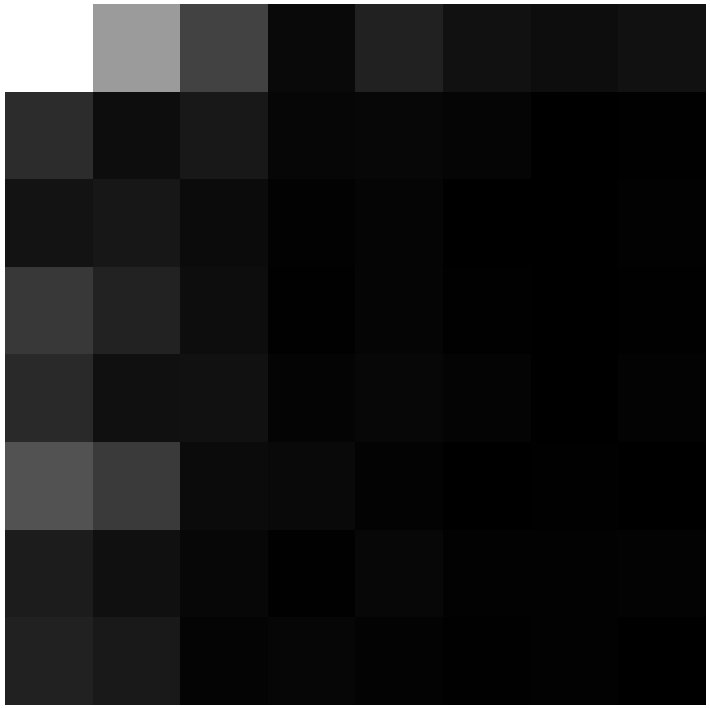
Coefficients de la transformée en cosinus discrète (DCT-II)

-486	296	-125	-20	65	-36	-25	33
85	24	-46	11	14	-13	2	2
38	45	22	-6	-11	2	1	-5
-108	-65	25	4	-11	3	0	-4
-80	-30	34	-9	-14	9	-2	-5
-158	-111	22	19	-6	0	3	1
-53	-32	14	-4	-14	7	5	-8
-65	-48	10	11	-7	-3	5	-1

Mise en pratique

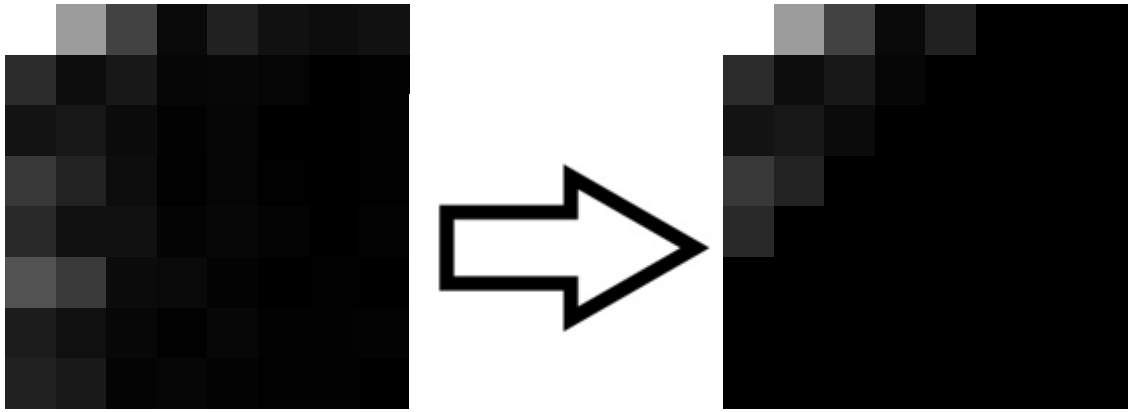


Nouvelle base fréquentielle



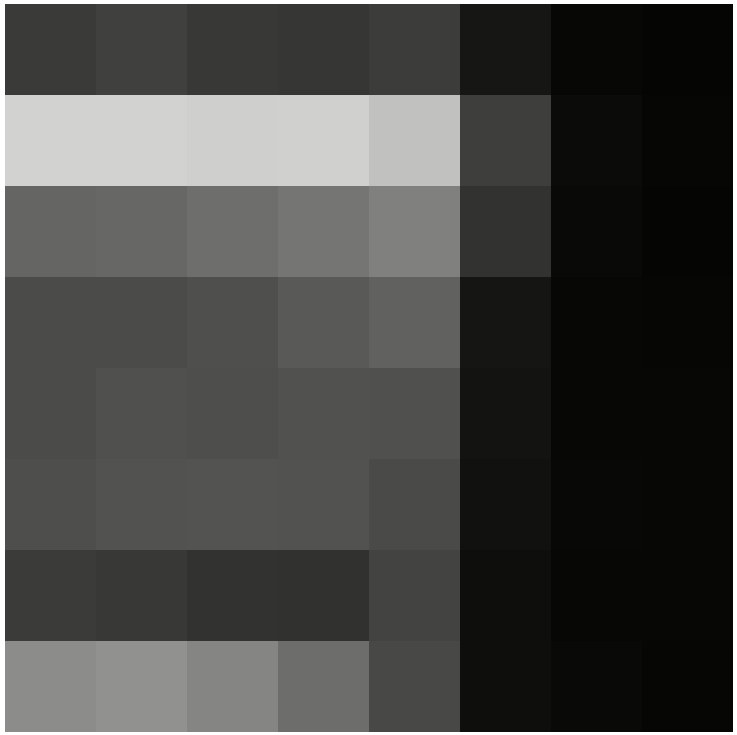
Représentation du bloc de DCT

"Filtre passe-bas"

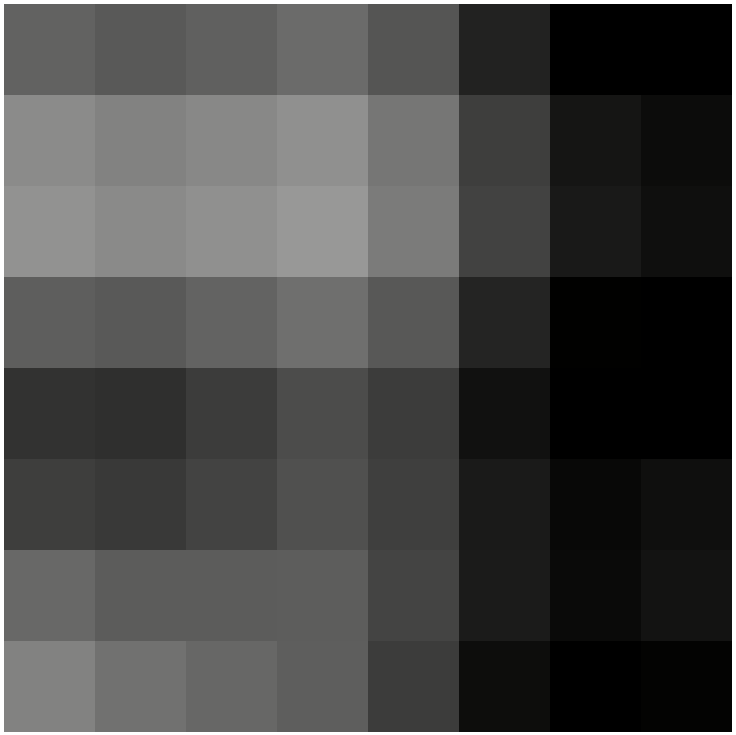


255	155	66	9	33	0	0	0
44	13	24	6	0	0	0	0
19	23	11	0	0	0	0	0
56	34	0	0	0	0	0	0
41	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

"Filtre passe-bas"



Bloc de base



Bloc dégradé "passe-bas"

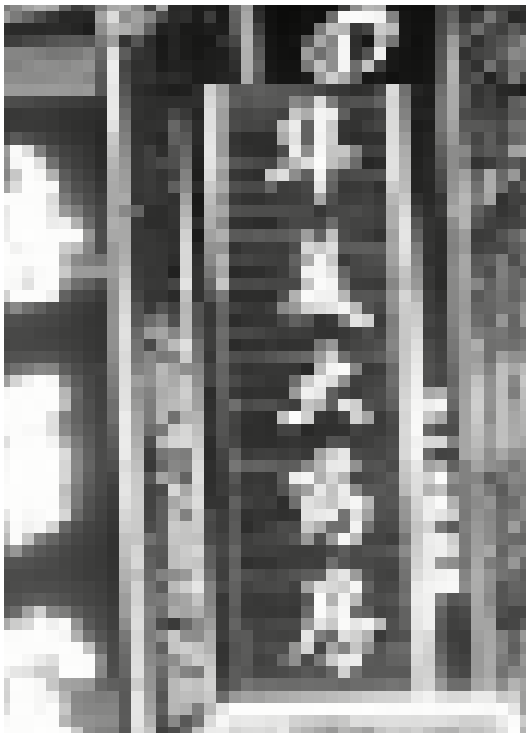
Comparaison avec la méthode naïve : réduction d'un facteur 4

Compression naïve (blocs 2×2)

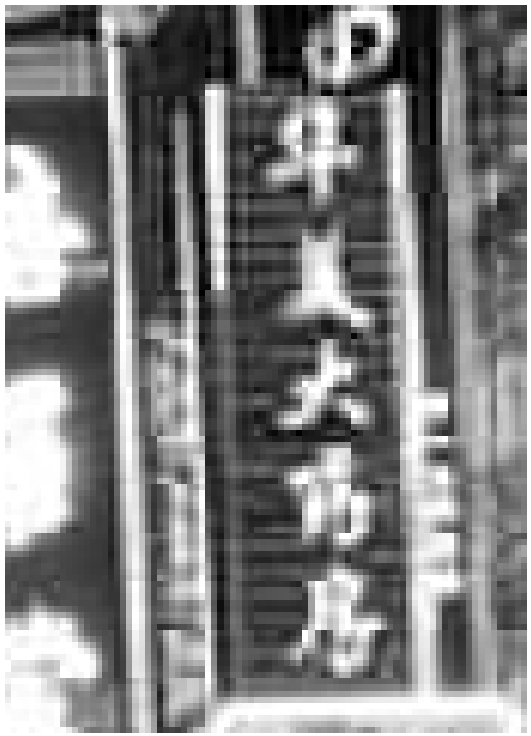
Compression "passe-bas"

Comparaison avec la méthode naïve : réduction d'un facteur 4

Image de base



Compression naïve



"Passe-bas"

Tables de quantification (JPEG)

Luminance

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

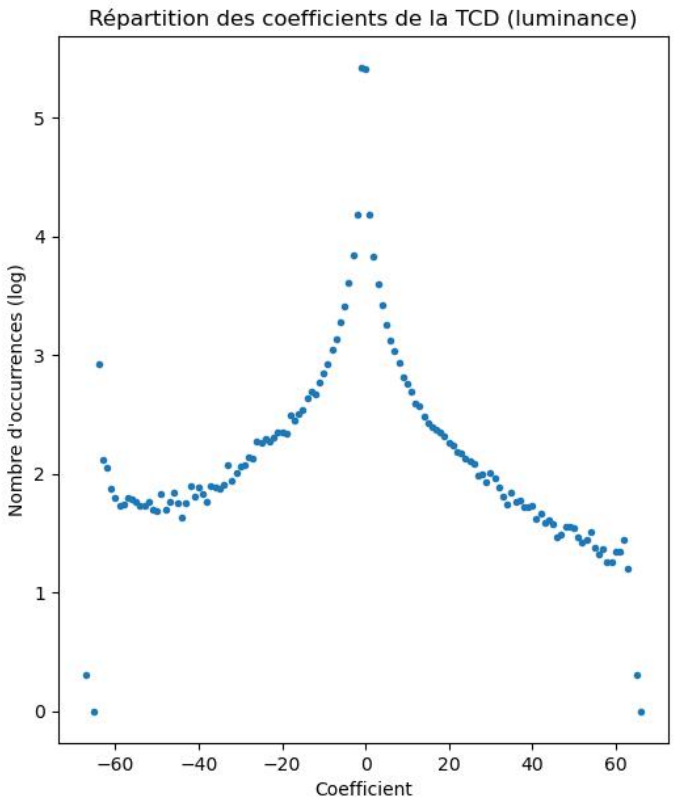
Chrominance

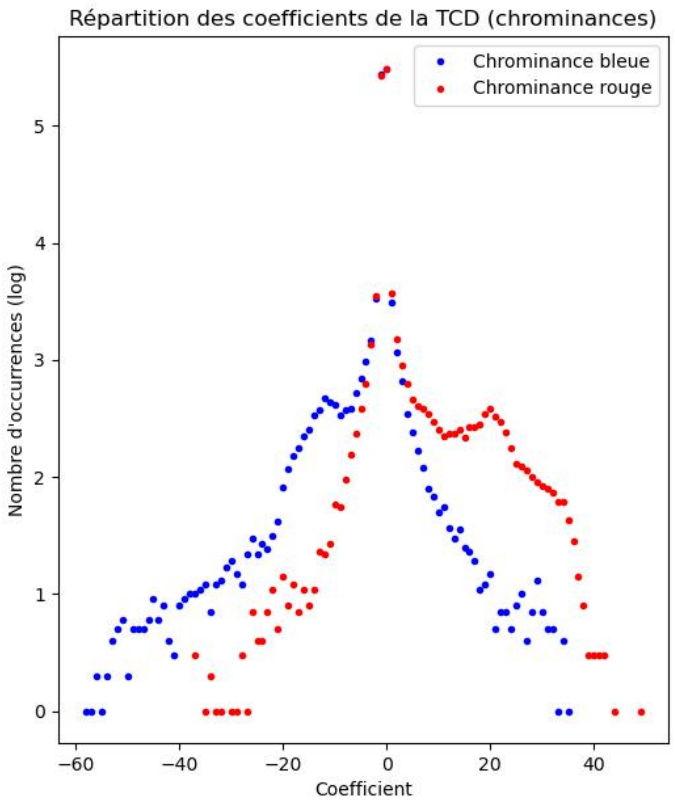
17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Exemples

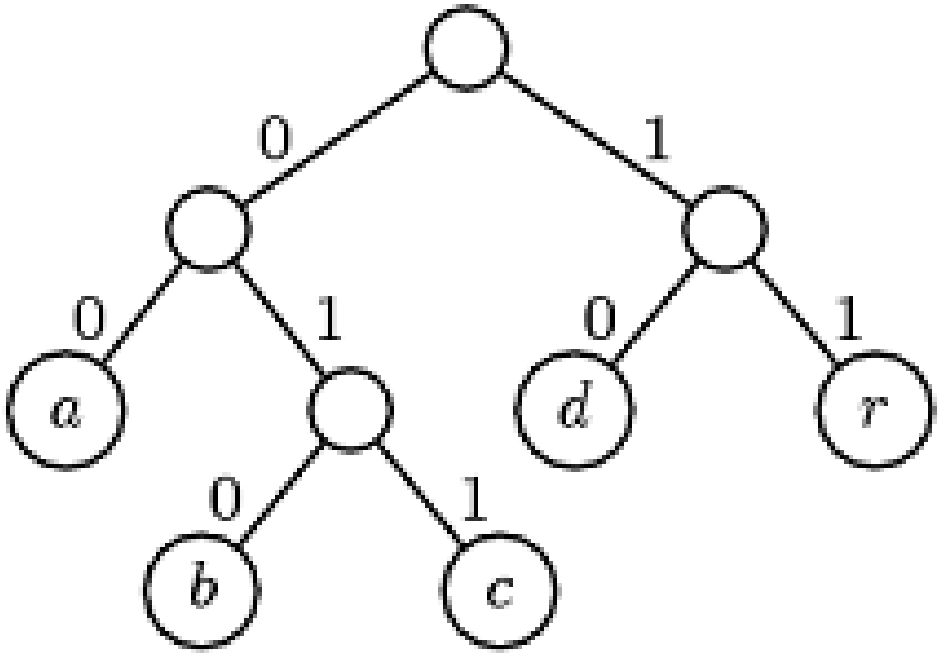
Exemples

Redondance


Répartition des coefficients de la TCD (luminance)
Nombre d'occurrences (log)
Coefficient
0
1
2
3
4
5
−60
−40
−20
0
20
40
60


Répartition des coefficients de la TCD (chrominances)
Chrominance bleue
Chrominance rouge
Nombre d'occurrences (log)
Coefficient
0
1
2
3
4
5
−60
−40
−20
0
20
40

Arbre binaire, code préfixe



Le code des feuilles est {00, 010, 011, 10, 11}.

Source : *blog.savoirfairelinux.com*

Exemple : "*bcarr*" $\rightarrow$ [010011001111], 12 bits

Entropie d'un code

Longueur d'un code

On définit la longueur d'un code préfixe c d'une suite de caractères X d'un alphabet $\mathcal{A} = (a_i)_{i \in I}$ par :

$$L(c) = \sum_{i \in I} p_i \ell_c(a_i)$$

avec $\ell_c(a_i)$ la longueur du caractère a_i codé par c,
et $p_i \in [0, 1]$ la fréquence d'apparition du caractère a_i dans X.

Entropie d'un code

Entropie de Shannon

On conserve les notations précédentes. On définit l'entropie de X par :

$$H(X) = -\sum_{i \in I} p_i \log_2(p_i).$$

Borne de Shannon

Pour tout code préfixe c : $L(c) \geq H(X)$.

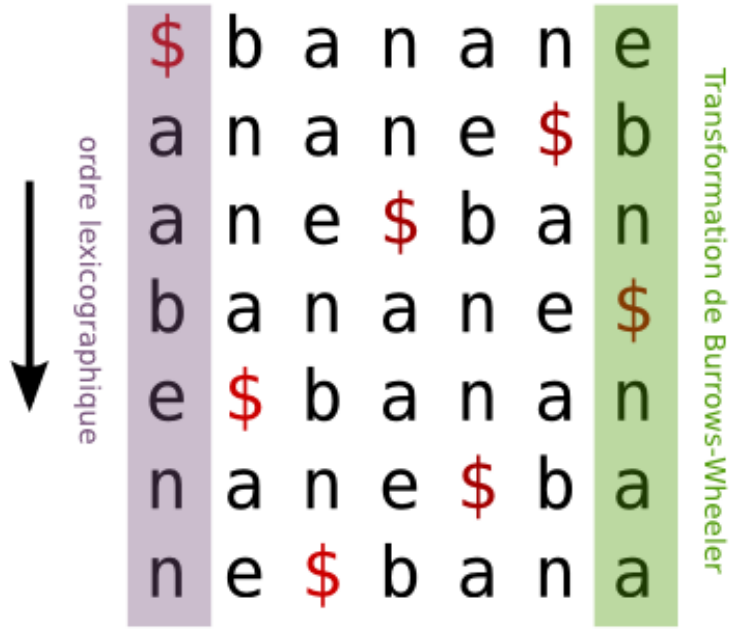
Algorithmes de compression sans perte

Codage par plages (RLE – Run-Length Encoding)

Algorithme MTF (Move-to-front)

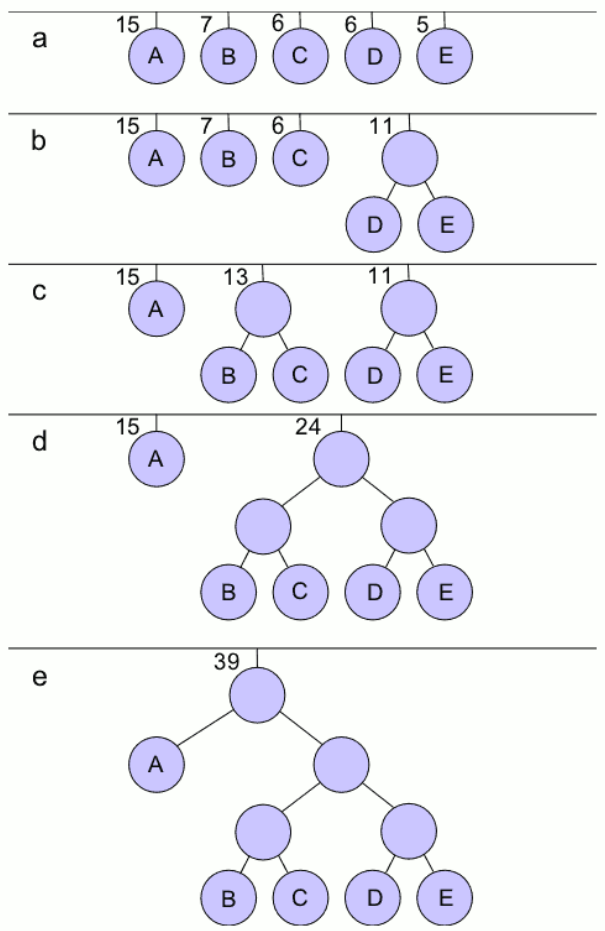
Codage de Huffman

Transformée de Burrows-Wheeler (BWT)



Exemple canonique de la BWT

Codage de Huffman : construction



Source : *https ://upload.wikimedia.org/wikipedia/commons/d/d8/HuffmanCodeAlg.png*

Quelques considérations numériques

Une distance entre $(M_{i,j,c})_{\substack{1\leq j\leq N\\1\leq j\leq M\\1\leq c\leq 3}}$ et $(M^{(ref)}_{i,j,c})_{\substack{1\leq j\leq N\\1\leq j\leq M\\1\leq c\leq 3}}$ $(\in \mathcal{M}_{i,j}(\mathbb{R}^3))$:

$$d(M, M^{(ref)}) = \frac{1}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M}\sum_{c=1}^{3}\left|M_{i,j,c} - M^{(ref)}_{N,M,c}\right|$$

Transformation	**Shanghaï**	**New York**
TCD (luminance seule)	76	90
TCD	131	153
Naïve (facteur 2)	29	34
Naïve (facteur 8)	64	65
Naïve (facteur 25)	95	87

Taux de compression de la BWT

Méthode	Octets occupés	Taux de compression
Avant codage entropique	1,8 million	1
Huffman seul	0,317 million	5,70
Huffman seul par blocs	0,365 million	4,96
Huffman + MTF	0,392 million	4,61
Huffman + MTF par blocs	0,386 million	4,69
Huffman + BWT	0,317 million	5,70
Huffman + BWT par blocs	0,372 million	4,86
Huffman + BWT + MTF	0,348 million	5,19
" + " + " par blocs	0,411 million	4,40

Changement de base

Une relation linéaire entre les deux triplets

$$\begin{pmatrix} 0,299 & 0,587 & 0,114 \\ -0,1687 & -0,3313 & 0,5 \\ 0,5 & -0,4187 & -0,0813 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix}$$

Transformée de Fourier

Transformée de Fourier à 1 dimension

Soit $f : \mathbb{R} \to \mathbb{C}$ intégrable.

La transformée de Fourier de f est $\hat{f} : \left\{ \begin{array}{ccc} \mathbb{R} & \longrightarrow & \mathbb{C} \\ \nu & \longmapsto & \int_{\mathbb{R}} f(t) \mathrm{e}^{-\mathrm{i}2\pi\nu t} \mathrm{d}t \end{array} \right.$.

On a alors : $\forall x \in \mathbb{R}, f(x) = \int_{\mathbb{R}}, \hat{f}(\nu) \mathrm{e}^{\mathrm{i}2\pi\nu x} \mathrm{d}\nu$.

Transformée de Fourier à 2 dimensions

Soit $f : \mathbb{R}^2 \longrightarrow \mathbb{C}$ intégrable.

La transformée de Fourier de f est :

$$\hat{f} : \left\{ \begin{array}{ccc} \mathbb{R}^2 & \longrightarrow & \mathbb{C} \\ (\nu, \xi) & \longmapsto & \int_{\mathbb{R}} \int_{\mathbb{R}} f(x, y) \mathrm{e}^{-\mathrm{i}2\pi(\nu x + \xi y)} \mathrm{d}x\mathrm{d}y \end{array} \right.$$

On a alors : $\forall (x, y) \in \mathbb{R}^2, f(x, y) = \int_{\mathbb{R}} \int_{\mathbb{R}} \hat{f}(\nu, \xi) \mathrm{e}^{\mathrm{i}2\pi(\nu x + \xi y)} \mathrm{d}\nu\mathrm{d}\xi$.

Transformée de Fourier discrète (DFT)

Échantillonage d'un signal continu

Soit $f : \mathbb{R} \to \mathbb{C}$. On considère un échantillion de $N \in \mathbb{N}$ valeurs aux points $(x_0, \dots x_{N-1})$ et

$$s : \left\{ \begin{array}{ccc} [\![0, N-1]\!] & \longrightarrow & \mathbb{C} \\ k & \longmapsto & f(x_k) \end{array} \right. .$$

Transformée de Fourier discrète à 1 dimension

La transformée de Fourier discrète du signal s est :

$$S : \left\{ \begin{array}{ccc} [\![0, N-1]\!] & \longrightarrow & \mathbb{C} \\ k & \longmapsto & \dfrac{1}{\sqrt{N}} \displaystyle\sum_{n=0}^{N-1} s(n) \mathrm{e}^{-2\mathrm{i}\pi k \frac{n}{N}} \end{array} \right. .$$

On a alors : $\forall n \in [\![0, N-1]\!], s(n) = \dfrac{1}{\sqrt{N}} \displaystyle\sum_{k=0}^{N-1} S(k) \mathrm{e}^{2\mathrm{i}\pi n \frac{k}{N}}$.

Transformée de Fourier discrète (DFT)

Transformée de Fourier discrète à 2 dimensions

Soit $f : \mathbb{R}^2 \longrightarrow \mathbb{C}$. On considère un échantillon de NM $(N, M \in \mathbb{N})$ valeurs aux points $(x_{k,l})_{\substack{0 \leq k \leq N-1 \\ 0 \leq l \leq M-1}}$ et

$$s : \left\{ \begin{array}{ccc} [\![0, N-1]\!] \times [\![0, M-1]\!] = \mathcal{N} & \longrightarrow & \mathbb{C} \\ (k, l) & \longmapsto & f(x_{k,l}) \end{array} \right. .$$

Transformée de Fourier discrète à 2 dimensions

La transformée de Fourier discrète du signal s est :

$$S : \left\{ \begin{array}{ccc} \mathcal{N} & \to \mathbb{C} & \\ (k, l) & \longmapsto & \dfrac{1}{\sqrt{MN}} \displaystyle\sum_{n=0}^{N-1} \sum_{m=0}^{M-1} s(k, l) \mathrm{e}^{-2\mathrm{i}\pi\left(k\frac{n}{N} + l\frac{m}{M}\right)} \end{array} \right. .$$

On a alors : $\forall (n, m) \in \mathcal{N}, s(n, m) = \dfrac{1}{\sqrt{NM}} \displaystyle\sum_{n=0}^{N-1} \sum_{m=0}^{M-1} S(k, l) \mathrm{e}^{2\mathrm{i}\pi\left(k\frac{n}{N} + l\frac{m}{M}\right)}$.

Transformée en cosinus discrète (DCT)

Transformée en cosinus discrète à 1 dimension (DCT-II/III)

La transformée en cosinus discrète du signal s est :

$$S : \begin{cases} [\![0, N-1]\!] & \longrightarrow & \mathbb{R} \\ k & \longmapsto & \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} s(n) \cos\left(\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right) \end{cases} .$$

On a alors, pour $n \in [\![0, N-1]\!]$:

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

Préambule

```
from PIL import Image
import numpy as np
from time import time
from matplotlib import pyplot as plt
im = Image.open([path])
plt.ioff()
```

Afficher

```
def afficher(img, dpi):

    h, l = len(img), len(img[0])
    fig = plt.figure(figsize = (l/dpi,h/dpi))
    ax = plt.Axes(fig, [0., 0., 1., 1.])
    ax.set_axis_off()
    fig.add_axes(ax)
    ax.imshow(img)
    plt.savefig(f"[path]/Image{time()}.png"\
        , bbox_inches="tight",\
        pad_inches = 0, dpi=dpi)
    plt.show()
```

Conversion image à tableau

```
def imgtoarray(img, largeur=None, hauteur = None):

    l, h = img.size

    if (largeur, hauteur) == (None, None):
        largeur, hauteur = l, h
    M = np.zeros((hauteur, largeur, 3))
    for x in range(min(l, largeur)):
        for y in range(min(h, hauteur)):
            (a,b,c) = img.getpixel((x, y))
            M[y][x] = np.array([a, b, c])
    return M
```

Manipulation de blocs

```
def divise(M, taille = 8):
    h,l = len(M), len(M[0])
    hr, lr = h//taille, l//taille
    blocs = [[] for _ in range(hr)]
    for y in range(hr):
        for x in range(lr):
            blocs[y].append(M[taille*y:taille*(y+1), taille*x:
                taille*(x+1)])
    return blocs

def rassemble(blocs):
    taille = len(blocs[0][0])
    hr, lr = len(blocs), len(blocs[0])
    h, l = hr*taille, lr*taille
    M = np.full((h,l), blocs[0][0][0][0])
    for y in range(h):
        for x in range(l):
            M[y][x] = blocs[y//taille][x//taille][y%taille][x%
                taille]
    return M
```

Composantes rgb

```
def rouge(M):
    h,l = len(M), len(M[0])
    L = M.copy()
    for i in range(h):
        for j in range(l):
            L[i][j][1] = L[i][j][2] = 0
    return L

def vert(M):
    h,l = len(M), len(M[0])
    L = M.copy()
    for i in range(h):
        for j in range(l):
            L[i][j][0] = L[i][j][2] = 0
    return L
```

Composantes rgb

```
def bleu(M):
    h,l = len(M), len(M[0])
    L = M.copy()
    for i in range(h):
        for j in range(l):
            L[i][j][1] = L[i][j][0] = 0
    return L

# R, G, B = rouge(A), vert(A), bleu(A)
# sauv(R), sauv(G), sauv(B)
# T = naif(R)
# A2 = T+G+B
# sauv(A2)
# T = naif(G)
# A2 = T+R+B
# sauv(A2)
# T = naif(B)
# A2 = T+R+G
# sauv(A2)
```

Composantes luminance / chrominances

```
def matY(M):
    h,l = len(M), len(M[0])
    L = np.zeros((h, l),dtype=int)
    coef = np.array([[.299], [.587], [.114]])
    for i in range(h):
        for j in range(l):
            L[i][j] = int(np.dot(M[i][j],coef))
    return L

def matCb(M):
    h,l = len(M), len(M[0])
    L = np.zeros((h, l),dtype=int)
    coef = np.array([[-.1687], [-.3313], [.5]])
    for i in range(h):
        for j in range(l):
            L[i][j] = int(np.dot(M[i][j],coef))
    return L
```

Changement de base

```
def matCr(M):
    h,l = len(M), len(M[0])
    L = np.zeros((h, l),dtype=int)
    coef = np.array([[.5], [-.4187], [-.0813]])
    for i in range(h):
        for j in range(l):
            L[i][j] = int(np.dot(M[i][j],coef))
    return L

def rgb(matY,matCb,matCr):
    h,l = min([len(matY), len(matCb), len(matCr)]), min([len(matY
        [0]), len(matCb[0]), len(matCr[0])])

    chang = np.linalg.inv(np.array
        ([[.299,-.1687,.5],[.587,-.3312,-.4187],[.114,.5,-.0813]]))

    M = np.zeros((h,l,3),dtype=int)
    for y in range(h):
        for x in range(l):
```

Changement de base

```
    YCC = np.array([matY[y][x], matCb[y][x], matCr[y][x]])
                couleurs = np.dot(YCC, chang)
                M[y][x][0] = max(min(int(couleurs[0]),255),0)
                M[y][x][1] = max(min(int(couleurs[1]),255),0)
                M[y][x][2] = max(min(int(couleurs[2]),255),0)
        return M

# Y, Cb, Cr = matY(A),matCb(A),matCr(A)
# YY = rgb(Y,np.zeros((h,l)),np.zeros((h,l)))
# BB = rgb(np.zeros((h,l)),Cb,np.zeros((h,l)))
# RR = rgb(np.zeros((h,l)),np.zeros((h,l)),Cr)
# sauv(YY)
# sauv(BB)
# sauv(RR)
# YYY = naif(YY)
# BBB = naif(BB)
# RRR = naif(RR)
# sauv(rgb(matY(YYY), Cb, Cr))
# sauv(rgb(Y, matCb(BBB), matCr(RRR)))
# R, G, B = rouge(A), vert(A), bleu(A)
```

Compression naïve

```
def naif(img, taux):

    l, h = img.size
    lc, hc = (l//taux+2)*taux, (h//taux+2)*taux
    M = imgtoarray(img, lc, hc)
    S = np.zeros((h, l, 3), dtype = int)
    for i in range(taux):
        for j in range(taux):
            S += M[i:h+i,j:l+j]
    S /= taux**2

    comp = np.zeros((h, l, 3), dtype = int)
    for x in range(h//taux):
        for i in range(taux):
            for y in range(l//taux):
                for j in range(taux):
                    comp[x*taux+i][y*taux+j] =\
                    S[x*taux][y*taux]
    return comp
```

Transformée en cosinus discrète (DCT)

```
def dct(s):

    N, M = len(s), len(s[0])
    C = np.zeros((N,N))
    D = np.zeros((M,M))

    for j in range(N):
        C[0][j]=1/2**.5
        for i in range(1,N):
            C[i][j]=np.cos(np.pi/N*(j+.5)*i)

    for j in range(M):
        D[0][j]=1/2**.5
        for i in range(1,M):
            D[i][j]=np.cos(np.pi/M*(j+.5)*i)

    return np.dot(C, np.dot(s,D.T))*2/(N*M)**.5
```

Transformée en cosinus discrète (DCT)

```
def idct(S):

    N, M = len(S), len(S[0])
    C = np.zeros((N,N))
    D = np.zeros((M,M))

    for j in range(N):
        C[0][j]=1/2**.5
        for i in range(1,N):
            C[i][j]=np.cos(np.pi/N*(j+.5)*i)

    for j in range(M):
        D[0][j]=1/2**.5
        for i in range(1,M):
            D[i][j]=np.cos(np.pi/M*(j+.5)*i)

    return np.dot(C.T, np.dot(S,D))*2/(N*M)**.5
```

TCD luminance

```
def tcdY(M):
    blocs = divise(M)
    hc, lc = len(blocs), len(blocs[0])
    transforme = [[] for _ in range(hc)]

    F = np.array(
[ 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],dtype=int).reshape((8,8))

    for y in range(hc):
        for x in range(lc):
            transforme[y].append(dct(blocs[y][x]-128)//F)
    return transforme
```

TCD chrominance

```
def tcdC(M):
    blocs = divise(M)
    hc, lc = len(blocs), len(blocs[0])
    transforme = [[] for _ in range(hc)]

    F = np.array(
[ 17,  18,  24,  47,  99,  99,  99,  99,
  18,  21,  26,  66,  99,  99,  99,  99,
  24,  26,  56,  99,  99,  99,  99,  99,
  47,  66,  99,  99,  99,  99,  99,  99,
  99,  99,  99,  99,  99,  99,  99,  99,
  99,  99,  99,  99,  99,  99,  99,  99,
  99,  99,  99,  99,  99,  99,  99,  99,
  99,  99,  99,  99,  99,  99,  99,  99],dtype=int).reshape((8,8))

    for y in range(hc):
        for x in range(lc):
            transforme[y].append(dct(blocs[y][x])//F)
    return transforme
```

TCD inverse luminance

```
def tcdiY(transforme):
    hc, lc = len(transforme), len(transforme[0])
    blocs = [[] for _ in range(hc)]

    F = np.array(
    [ 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],dtype=int).reshape
          ((8,8))

      for y in range(hc):
        for x in range(lc):
            blocs[y].append(idct(transforme[y][x]*F)+128)
    return blocs
```

TCD inverse chrominance

```
def tcdiC(transforme):
    hc, lc = len(transforme), len(transforme[0])
    blocs = [[] for _ in range(hc)]

    F = np.array(
[ 17,  18,  24,  47,  99,  99,  99,  99,
  18,  21,  26,  66,  99,  99,  99,  99,
  24,  26,  56,  99,  99,  99,  99,  99,
  47,  66,  99,  99,  99,  99,  99,  99,
  99,  99,  99,  99,  99,  99,  99,  99,
  99,  99,  99,  99,  99,  99,  99,  99,
  99,  99,  99,  99,  99,  99,  99,  99,
  99,  99,  99,  99,  99,  99,  99,  99],dtype=int).reshape((8,8))

    for y in range(hc):
        for x in range(lc):
            blocs[y].append(idct(transforme[y][x]*F))
    return blocs
```

Calcul de la transformée en cosinus discrète

```
# Z = np.zeros((h,l), dtype = int)
# Y = matY(A)
# M = rgb(Y, Z, Z)
# D = tcdY(Y)
# YY = rassemble(tcdiY(D))
# T = naif(M,1)
# sauv(T)


def tcos(M):

    Y, Cb, Cr = matY(M), matCb(M), matCr(M)
    Y1, Cb1, Cr1 = tcdiY(tcdY(Y)), tcdiC(tcdC(Cb)), tcdiC(tcdC(Cr))
    # sauv(rgb(rassemble(Y1), Cb, Cr))

    return rgb(rassemble(Y1), rassemble(Cb1), rassemble(Cr1))
```

Redondance dans les coefficients de la DCT (luminance)

```
# Y, Cb, Cr = (tcdY(matY(A))),(tcdC(matCb(A))),tcdC(matCr(A))
# uniqu, counts = np.unique(Y, return_counts=True)
# fig1 = plt.figure(figsize = (6, 7))
# plt.ylabel("Nombre d'occurrences (log)")
# plt.title("Repartition des coefficients de la TCD (luminance)")
# plt.plot(uniqu, np.log10(counts), '.')
# plt.savefig(f"[path]/Graphe Y {time()}.jpg")
```

Redondance dans les coefficients de la DCT (chrominances)

```
# uniqu, counts = np.unique(Cb, return_counts=True)
# fig2 = plt.figure(figsize = (6, 7))
# plt.xlabel('Coefficient')
# plt.ylabel("Nombre d'occurrences (log)")
# plt.title("Repartition des coefficients de la TCD (chrominances)
    ")
# plt.plot(uniqu, np.log10(counts), 'b.', label = "Chrominance
    bleue")
# uniqu, counts = np.unique(Cr, return_counts=True)
# plt.plot(uniqu, np.log10(counts), 'r.', label = "Chrominance
    rouge")
# plt.legend(loc = 'best')
# plt.savefig(f"[path]/Graphe C {time()}.jpg")
```

Longueur d'un code et entropie de Shannon

```
def longueur(L, alphabet):
    d, total = compte(L)
    p = {x:d[x]/total for x in d}
    l = {x:len(alphabet[x]) for x in d}
    S = 0
    for x in d:
        S += p[x]*l[x]
    return S


def H(L):
    d, total = compte(L)
    p = {x:d[x]/total for x in d}
    S = 0
    for x in d:
        S += p[x]*np.log2(p[x])
    return -S
```

Aplatir un bloc

```
def zigzag(bloc):
    n = len(bloc)
    L = []
    for i in range(n):
        if i%2:
            for j in range(i+1):
                L.append(bloc[j][i-j])
        else:
            for j in range(i+1):
                L.append(bloc[i-j][j])
    for i in range(n-1):
        if i%2:
            for j in range(i+1, n):
                L.append(bloc[j][n+i-j])
        else:
            for j in range(i+1, n):
                L.append(bloc[n+i-j][j])
    return L
```

Aplatir un bloc

```
def inverse_zigzag(L):
    n = int(len(L)**.5)
    bloc = np.zeros((n,n), dtype = int)

    c = 0
    for i in range(n):
        if i%2:
            for j in range(i+1):
                bloc[j][i-j] = L[c] ; c+=1
        else:
            for j in range(i+1):
                bloc[i-j][j] = L[c] ; c+=1
    for i in range(n-1):
        if i%2:
            for j in range(i+1, n):
                bloc[j][n+i-j] = L[c] ; c+=1
        else:
            for j in range(i+1, n):
                bloc[n+i-j][j] = L[c] ; c+=1
    return bloc
```

Codage et décodage

```
def codage(L, alphabet):
    n = len(L)
    T = np.full(n, '1111111111111111')
    C = {x:len(alphabet[x]) for x in alphabet}
    total = 0
    for i in range(n):
        x = L[i]
        T[i] = alphabet[x]
        total += C[x]
    return T, total


def decodage(T, alphabet):
    n = len(T)
    decode = {alphabet[x]:x for x in alphabet}
    L = np.full(n, list(decode.values())[0])
    for i in range(n):
        L[i] = decode[(T[i])]
    return L
```

Codage de Huffman

```
def compte(L):
    d = {}
    total = 0
    for x in L:
        if x in d:
            d[x] += 1
        else:
            d[x] = 1
        total += 1
    return d, total


def huffman(L):
    d = compte(L)[0]
    K = list(d.keys())
    n = len(K)
    A0 = {i:(K[i],d[K[i]],None,None) for i in range(n)}
    c0 = n-1
    def aux(A, c): # creation de l'arbre
```

Codage de Huffman

```
        if len(A) == 1:
            return A
        K = list(A.keys())
        x1, dm1 = K[0], A[K[0]][1]
        x2, dm2 = K[1], A[K[1]][1]
        if dm1 > dm2:
            x1, x2 = x2, x1
            dm1, dm2 = dm2, dm1
        for x in A:
            dx = A[x][1]
            if dx<dm1:
                x1,dm1 = x,dx
            elif dx<dm2 and x!=x1:
            x2,dm2 = x,dx
            a1 = A.pop(x1)
            a2 = A.pop(x2)
            c += 1
            A[c] = (None, dm1+dm2, a1, a2)
            return aux(A,c)
```

Codage de Huffman

```
    A = aux(A0, c0)
    K = list(A.keys())
    arbre = A[K[0]] # arbre obtenu
    alphabet = {x:[] for x in d}
    def aux2(chemin, arb): # recuperation des encodages
        x, dx, gauche, droite = arb
        if x != None:
            alphabet[x] = chemin
        if gauche != None:
            aux2(chemin+'0', gauche)
        if droite != None:
            aux2(chemin+'1', droite)
    aux2('', arbre)
    return alphabet
```

Algorithme Move-to-front

```
def mtf(L):
    n = len(L)
    T = np.full(n, L[0])
    d = {}
    for x in L:
        d[x] = None
    c = 0
    for x in d:
        d[x] = c
        c += 1
    for i in range(n):
        y = d[L[i]]
        for x in d:
            if d[x]<y:
                d[x]+=1
        d[L[i]] = 0
        T[i] = y
    return T, d
```

Algorithme Move-to-front

```
def imtf(T, d):

    n = len(T)
    L = []

    for i in range(n):
        d2 = {d[x] : x for x in d}
        t = T[-i-1]
        y = int(d2[0])

        for x in d:
            if int(d[x])<=int(t):
                d[x] = int(int(d[x])-1)

        d[y] = int(t)
        L.append(y)

    return L[::-1]
```

Transformée de Burrows-Wheeler

```
def lexico(a, b):
    for x, y in zip(a[0], b[0]):
        if x < y:
            return False
        if x > y:
            return True
    return False

def fusion(L1, L2, comparaison):
    L, M = L1, L2
    l, m = 0, 0
    liste = []
    while l<len(L) and m<len(M):
        if comparaison(L[l], M[m]):
            liste.append(M[m])
            m += 1
        else:
            liste.append(L[l])
            l += 1
```

Transformée de Burrows-Wheeler

```
def merge_sort(L, comparaison):

    n = len(L)
    if n == 1:
        return L

    return fusion(merge_sort(L[int(n//2):], comparaison),
        merge_sort(L[:int(n//2)], comparaison), comparaison)

def BWT(L):
    n = len(L)
    M = np.concatenate((L, L))
    permu = [(L,True)] + [(M[i:n+i],False) for i in range(1, n)]
    trie = merge_sort(permu, lexico)
    R = np.full(n, L[0])
    for i in range(n):
        R[i] = trie[i][0][-1]
        if trie[i][1]:
            position = i
    return R, position
```

Transformée de Burrows-Wheeler

```
def fusion2(L1, L2):
    L, M = L1, L2
    l, m = 0, 0
    liste = []

    while l<len(L) and m<len(M):
        if L[l][0] > M[m][0]:
            liste.append(M[m])
            m += 1
        else:
            liste.append(L[l])
            l += 1

    if l == len(L):
        liste += M[m:]
    else:
        liste += L[l:]
    return liste
```

Transformée de Burrows-Wheeler

```
def lexico2(a, b):
    for x, y in zip(a, b):
        if x < y:
            return False
        if x > y:
            return True
    return False

def IBWT(R, position):
    n = len(R)
    T = np.full((n,n), 0, dtype = int)
    D = np.full((n,n), 0, dtype = int)
    for i in range(n):
        T[i][0] = R[i] ; D[i][0] = R[i]
    P = np.eye(n, k=1)
    for i in range(n-1):
        T = np.array(merge_sort(list(T), lexico2))
        T = np.dot(T, P) + D
    T = np.array(merge_sort(list(T), lexico2), dtype = int)
    return T[position]
```

Compression finale

```
def compression(img = A):
    h, l = len(img), len(img[0])
    hc, lc = h//8, l//8

    Y, Cb, Cr = tcdY(matY(img)), tcdC(matCb(img)), tcdC(matCr(img))
    comp = [Y, Cb, Cr]

    blocs = np.full((hc, lc, 3, 64), "1111111111111")
    position = np.zeros((hc, lc, 3), dtype=int)
    d = np.full((hc, lc, 3), {})
    alphabet = np.full((hc, lc, 3), {})

    total = 0
    for c in range(3):
        for i in range(hc):
            for j in range(lc):
```

Compression finale

```
                L = zigzag(comp[c][i][j])
                R, position[i][j][c] = BWT(L)
                M, d[i][j][c] = mtf(L)
                ALP = huffman(M)
                blocs[i][j][c], t = codage(M, ALP)
                alphabet[i][j][c] = ALP
                total += t

        # C = rassemble(c)
        # L = zigzag(C)
        # R, position = BWT(L)
        # M, d = mtf(R)
        # alphabet = huffman(M)
        # T, t = codage(M, alphabet)
        # total += t

    return total, lc, hc, blocs, position, d, alphabet
```

Décompression finale

```
def decompression(blocs, position, d, alphabet):
    hc, lc = len(blocs), len(blocs[0])
    Y = np.zeros((hc, lc, 8, 8), dtype = int)
    Cb = np.zeros((hc, lc, 8, 8), dtype = int)
    Cr = np.zeros((hc, lc, 8, 8), dtype = int)
    comp = [Y, Cb, Cr]

    for c in range(3):
        for i in range(hc):
            for j in range(lc):
                T = decodage(blocs[i][j][c], alphabet[i][j][c])
                M = imtf(T, d[i][j][c])
                R = IBWT(M, position[i][j][c])
                B = inverse_zigzag(R)
                comp[c][i][j] = B

    Y = rassemble(tcdiY(Y))
    Cb = rassemble(tcdiC(Cb))
    Cr = rassemble(tcdiC(Cr))
    return rgb(Y, Cb, Cr)
```

Calcul de la distance entre les images

```
def ecart(M, Mref):
    h, l = len(M), len(M[0]) ; S = 0
    for i in range(h):
        for j in range(l):
            for c in range(3):
                S += abs(M[i][j][c]-Mref[i][j][c])
    return S/h/l

def tcos(M):
    Y, Cb, Cr = matY(M), matCb(M), matCr(M)
    Y1, Cb1, Cr1 = tcdiY(tcdY(Y)), tcdiC(tcdC(Cb)), tcdiC(tcdC(Cr))
    return rgb(rassemble(Y1), rassemble(Cb1), rassemble(Cr1)), rgb(
        rassemble(Y1), Cb, Cr)

# LC, L = tcos(A)
# N2, N8, N25 = naif(A,2), naif(A,8), naif(A, 25)
# print(ecart(LC,A)) ; print(ecart(L,A))
# print(ecart(N2,A)) ; print(ecart(N8,A))
# print(ecart(N25,A))
```