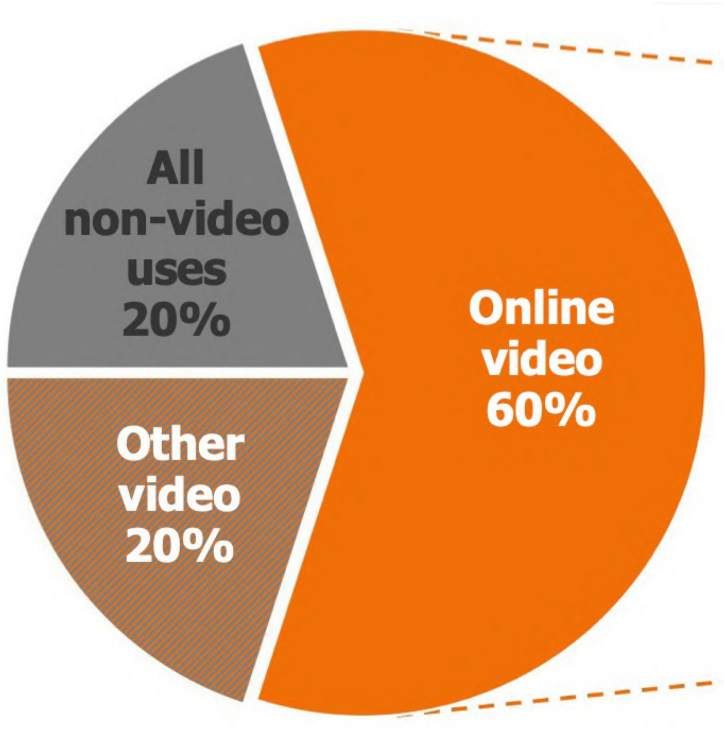
Étude de différentes méthodes de compression et décompression d’images

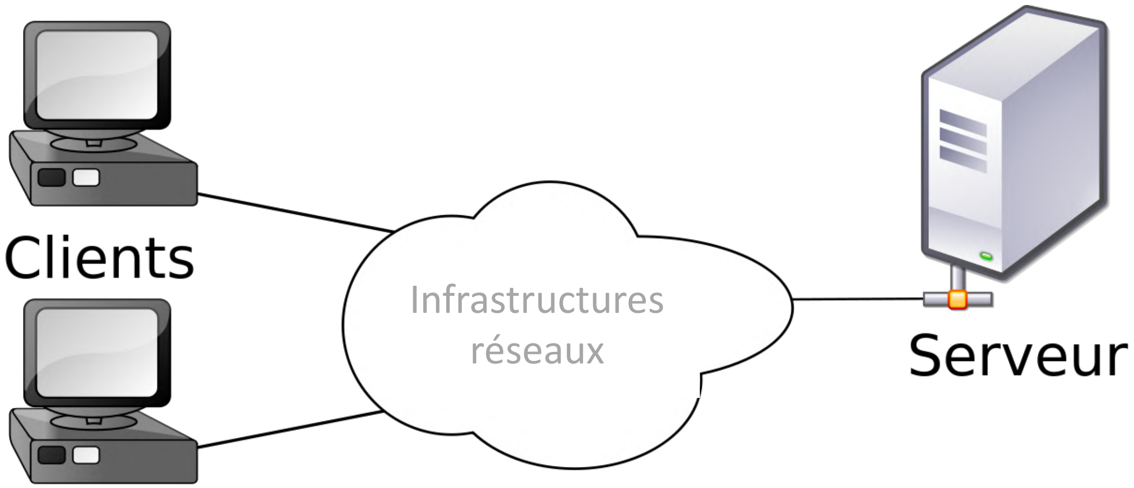
Nathan Foucher PSI

Numéro d’inscription : 17225

Enjeux



The shift Project : Distribution of online data flows between different uses in 2018 in the world [1]



Eléments majeurs composants l'infrastructure d'internet

Les infrastructures délivrants les vidéos en lignes ont générées 306 Méga tonnes de C02 en 2018

Comment fonctionne la compression du jpeg et pourquoi utilise-t-il la DCT plutôt de que la DFT ? Est-il possible d'adapter le procédé afin d'utiliser un système optique pour réduire la demande en calculs tout en respectant une qualité d'image suffisante ?

I. Le fonctionnement de la compression par le format jpeg

I.1 Présentation du format JPEG
Pourquoi la DCT ou DFT ?

- Permet de réaliser le spectre des fréquences spatiales composants une image

- Hautes fréquences correspondent à des variations très peu visibles dans l'image, donc peuvent êtres supprimées

$$f_{fenêtre} = \frac{1}{90} pixels^{-1}$$

$$f_{grille} = \frac{1}{6} pixels^{-1}$$

Plus petit détail visible : $d = 2\sin(3.10^{-4})\,L \;\approx 6mm$ à 10m

I.1 Les bases de la DTF

- Transformée de Fourier (analyse spectrale d'un signal continu) :
  - $X(f) = \int_{-\infty}^{+\infty} x(t) e^{-i2\pi f t} dt$

- x étant échantillonné aux instants nTe :
  - $X\left(\frac{k}{nTe}\right) \approx Te \sum_{k=-\infty}^{+\infty} x(kTe) e^{-i2\pi f kTe}$

- x connu sur N échantillons on calcul X pour $\mathrm{f} = k\frac{1}{NTe}$ avec $Te = \frac{N}{N}1$:
  - $X\left(\frac{k}{NTe}\right) \approx Te \sum_{n=0}^{N-1} x(nTe) e^{-i2\pi\frac{kn}{N}}$

  - Est à valeurs dans C.

$$\boxed{X(k) \approx \sum_{n=0}^{N-1} x(n) e^{-i2\pi\frac{kn}{N}}}$$

Transformée de Fourier discrète

$$\boxed{x(k) \approx \frac{1}{N} \sum_{n=0}^{N-1} X(n) e^{i2\pi\frac{kn}{N}}}$$

Transformée de Fourier discrète inverse

I.2 Les bases de la DCT

- Variant de la DFT, fonctionne avec un cosinus pour noyau :

  - $X(k) \approx \sum_{n=0}^{N-1} x(n)\cos[\frac{\pi}{N}\left(k+\frac{1}{2}\right)n]$ DCT
  - $\mathrm{x}(k) \approx \frac{1}{2}X(o) + \sum_{n=1}^{N-1} X(n)\cos[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k]$ DCT inverse
  - Est à valeurs dans R.

- DFT et DCT en 2D :

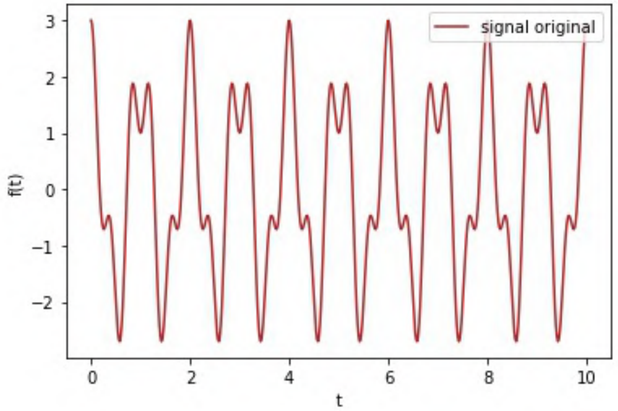
$$\boxed{X_{DFT}(x,y) \approx \sum_{v=0}^{V-1}\sum_{u=0}^{U-1} x(v,u)e^{-i2\pi(x\frac{v}{V}+y\frac{u}{U})}}$$

$$\boxed{X_{DCT}(x,y) \approx \sum_{v=0}^{V-1}\sum_{u=0}^{U-1} x(v,u)\cos[\frac{\pi}{V}\left(v+\frac{1}{2}\right)x]\cos[\frac{\pi}{U}\left(u+\frac{1}{2}\right)y]}$$

I.3 Exemple utilisations pour l’analyse fréquentielle 1D

- Signal original :

$$u(t) = 2\cos(2\pi 10t) + \cos(2\pi 25t)$$



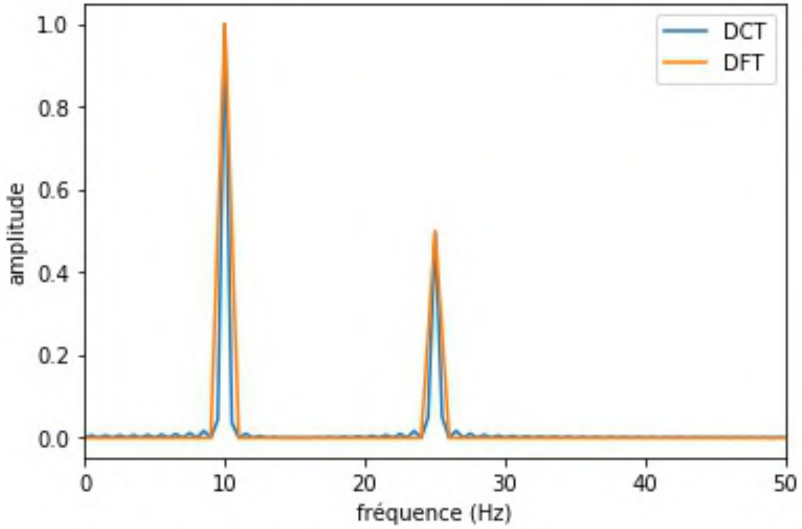
Graphique signal original (500 points)

```python
j=complex(0,1)

def DFT(u,N,k):
    a=0
    for i in range (N):
        a+=u[i]*np.exp(-j*2*np.pi*i*k/N)
    return(a)

def DCT(u,N,k):
    a=0
    for i in range (N):
        a+=u[i]*np.cos((np.pi/N)*(i+(1/2))*k)
    return(a)
```

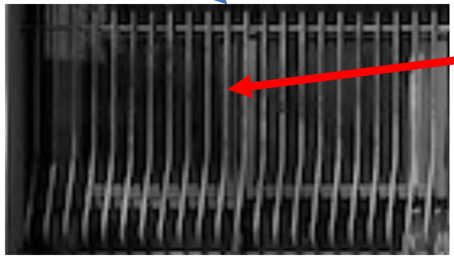
Fonctions python



Spectres en sortie

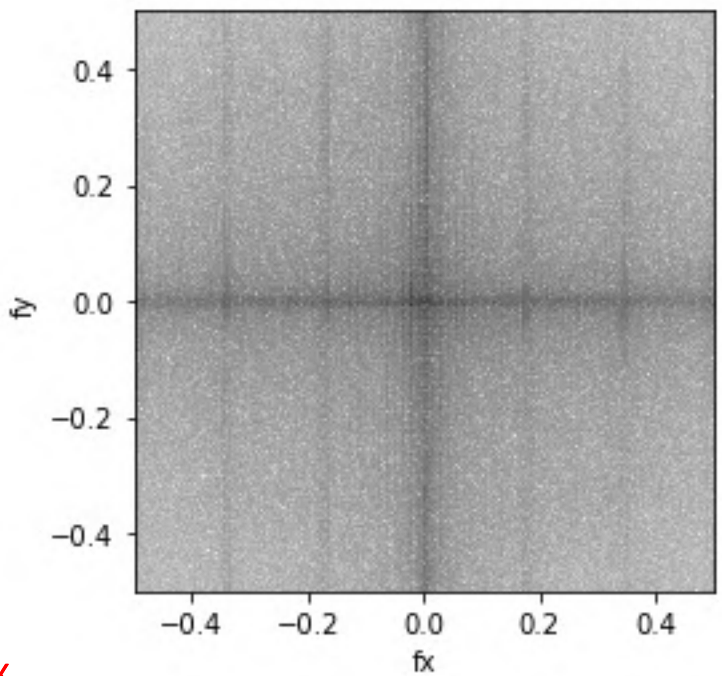
I.3 Exemple utilisations pour l’analyse fréquentielle 2D

Image Originale



Grille originale

6 pixels entre deux barreaux

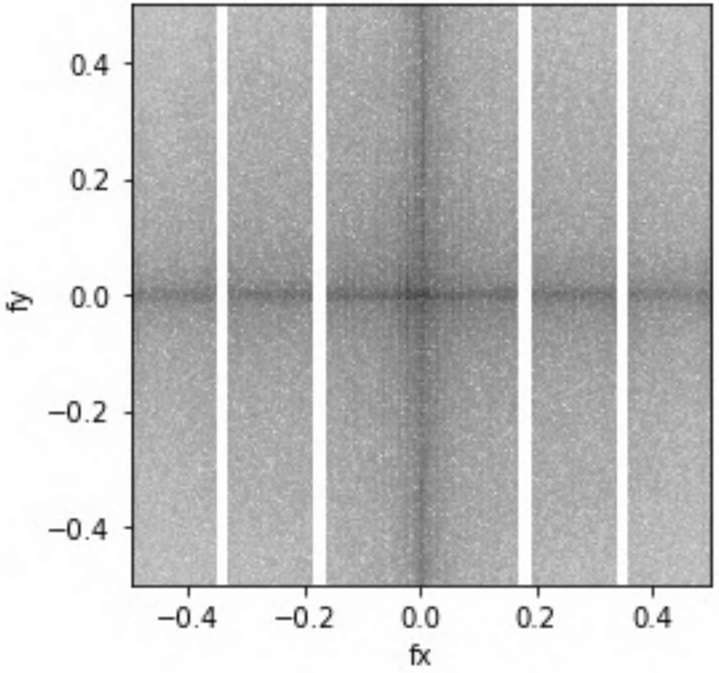


Spectre de sortie

On applique un filtre coupe bande

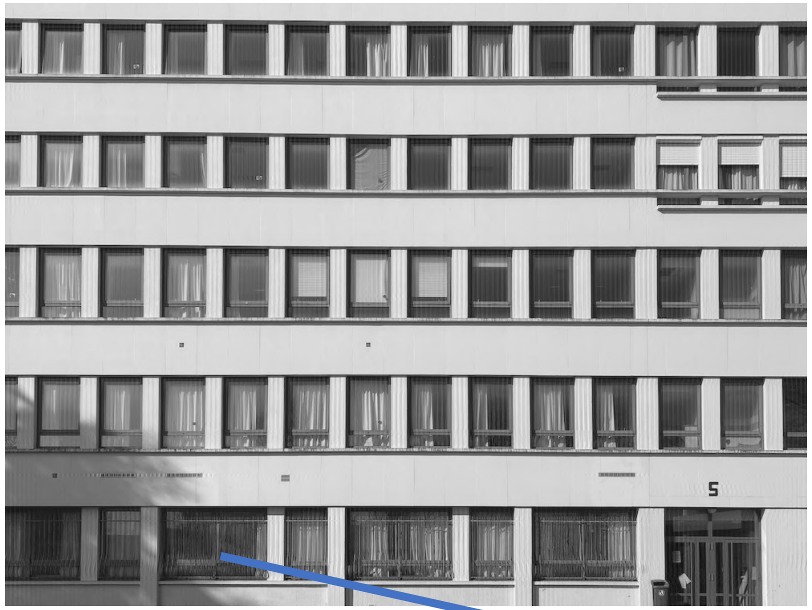
$f = \frac{1}{6} pixels^{-1}$ $f_p = 0.1 pixels^{-1}$

I.3 Exemple utilisations pour l’analyse fréquentielle 2D



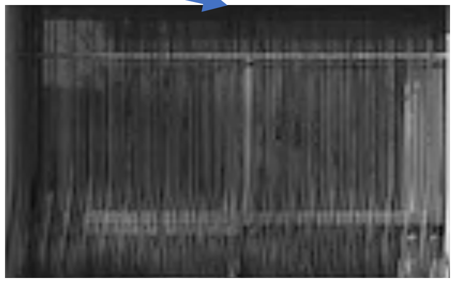
Spectre de sortie

Image de sortie



Grille originale

VS

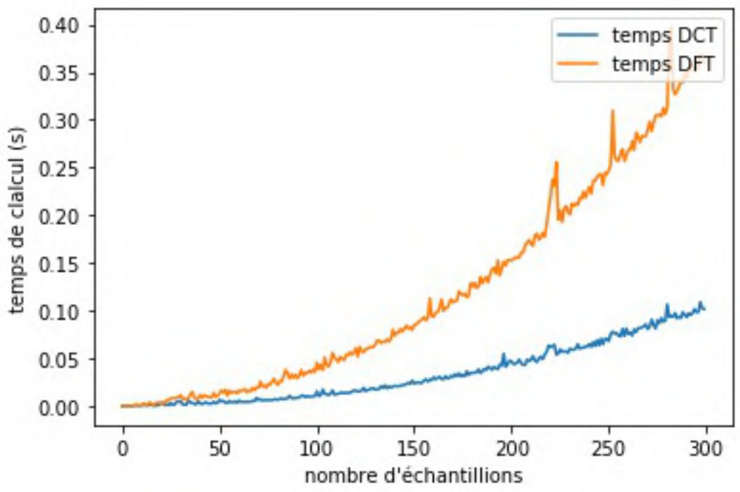


Grille après filtrage

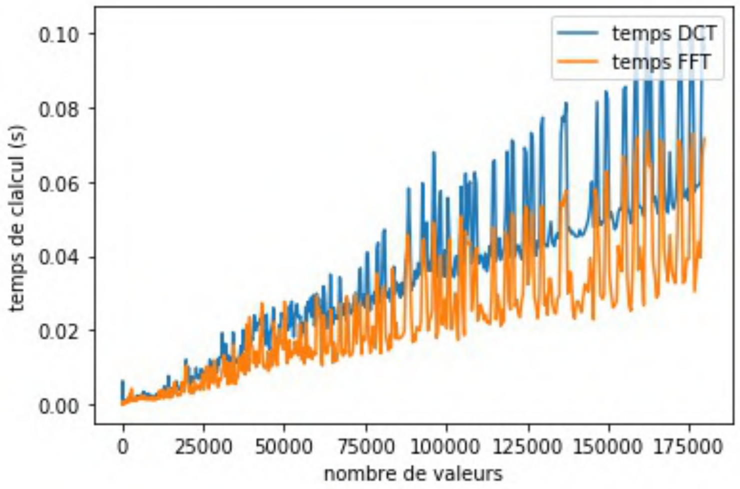
I.4 Le choix de la DCT

- Temps de calcul :
  - Cause : calcul de nombres complexes nécessaire pour la DFT donc 2N multiplications contre N pour la DCT
  - Problème résolu avec la FFT

Comparaison temps DCT et DFT :

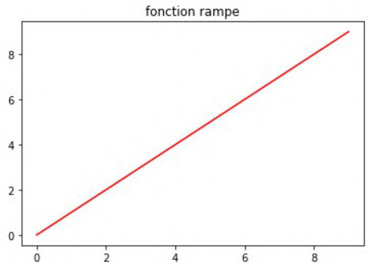


Comparaison temps DCT et FFT :



I.4 Le choix de la DCT

- Nombre de coefficients nécessaires : exemple avec avec la fonction rampe $u(t) = t$



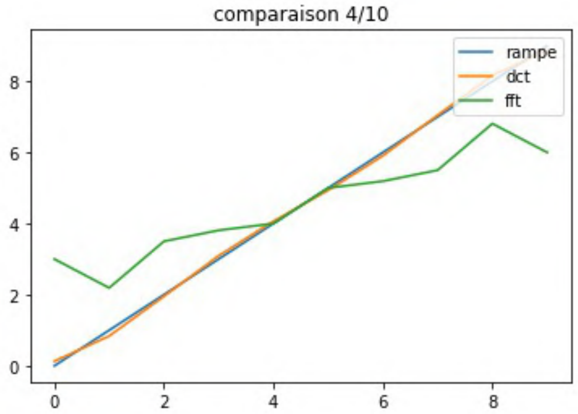
0	1	2	3	4	5	6	7	8	9

FFT

(45.0+0.0j)	(-5.0+15j)	(-5.0+6.8j)	(-5.0+3.6j)	(-5.0+~~1.6j)~~	(-5.0-~~1.3e-~~15j)	(-5.0-~~1.6j)~~	(-5.0-~~3.6j)~~	(-5.0-~~6.8j)~~	(-5.0-~~15.3j~~)

DCT

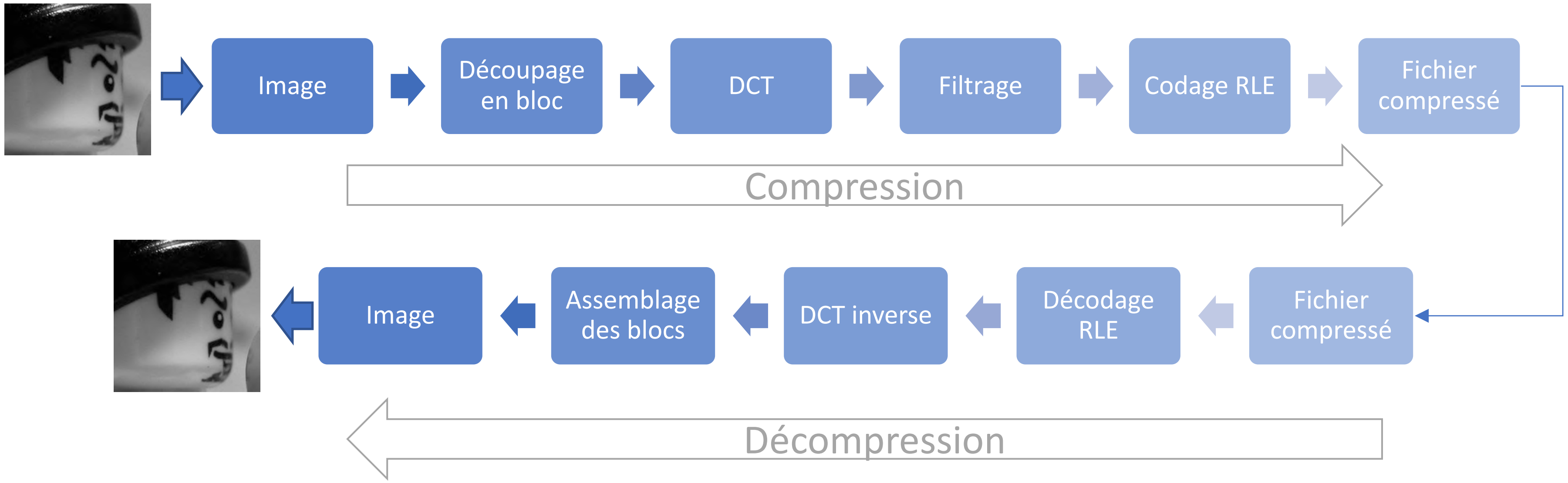
14	-9	0	-0,96	~~0~~	~~0,31~~	~~0~~	~~0,12~~	~~0~~	~~0,035~~



II. Étude du fonctionnement de la compression par le format jpeg

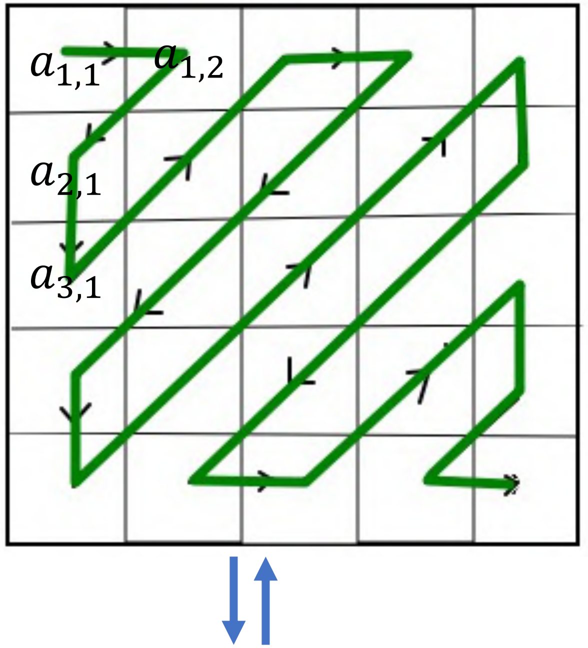
II.1 Codage d’un algorithme de compression

- Étapes clés de la compression :



II.1 Codage d’un algorithme de compression

- Codage zigzag



$a_{1,1}$	$a_{1,2}$	$a_{2,1}$	$a_{3,1}$		[…]

```python
def zizag(M):
    #assert M.shape[0]==M.shape[1] #vérifions que la matrice est carrée
    n=len(M)

    sections=[0]*(n*2-1)
    for i in range (n*2-1):
        sections[i]=[]

    for i in range(n):
        for j in range(n):
            a=i+j
            if(a%2 ==0):

                #ajout au début
                sections[a].insert(0,M[i][j])
            else:

                #ajout à la fin de la liste
                sections[a].append(M[i][j])
    F=[]
    for i in sections:
        for j in i:
            F.append(j)
    return(F)
```

II.1 Codage d'un algorithme de compression

- Codage RLE

```
def RLE(M):
    F=[]
    b=0
    for i in range (len(M)):
        if M[i]==0:
            b+=1
        else :
            if b==0 :
                F.append(M[i]) #si pas de 0 on ajoute la valeur
            else :
                F.append("#{int}".format(int=b)) # rajoute#nb de 0
                F.append(M[i])
            b=0
    #faire pour les elements restants
    if b!=0 :
        F.append("#{int}".format(int=b))
    return(F)
```

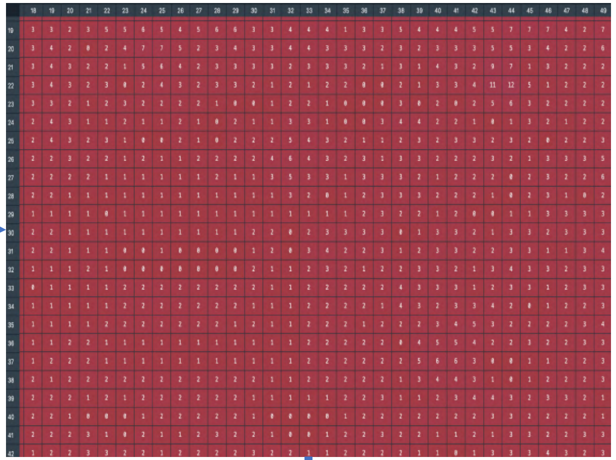
[1, 5, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 12, 6, 0, 123] → [1, 5, 8, '#10', 1, '#7', 12, 6, '#1', 123]

25 coefficients

10 coefficients

II.1 Codage d'un algorithme de compression

Image originale



	0	1	2	3	4	5	6	7
0	2	2	5	3	4	3	4	17
1	1	2	3	2	5	7	7	18
2	3	4	4	2	3	6	5	10
3	0	0	2	2	1	5	6	8
4	1	1	2	2	3	6	7	8
5	2	2	2	3	5	4	4	4
6	2	1	2	3	5	4	4	2
7	2	2	0	0	1	2	2	2

Bloc 8*8

Application de la DCT

DCT Bloc 8*8

	0	1	2	3	4	5	6	7
0	29.5	-15.39…	5.93558	-3.43…	2	-3.22…	3.22396	-0.240…
1	9.13508	-8.093…	5.99261	-7.08…	2.841…	-3.52…	3.74694	-0.643…
2	-0.151…	1.43653	1.50888	-2.02…	2.923…	-2.48…	0.478553	0.6591…
3	2.18592	-1.028…	-1.5574	-1.16…	2.224…	-0.20…	-0.1608…	1.2548
4	-3.25	-0.632…	1.7685	-1.59…	-0.25	1.112…	0.349854	-0.442…
5	-1.468…	2.65421	-2.1378	-0.31…	0.133…	1.485…	-0.0063…	0.5198…
6	-1.019…	4.24945	0.2285…	-0.13…	-0.12…	-0.59…	-0.2588…	-0.137…
7	1.49658	1.24056	-0.014…	0.475…	-0.58…	0.008…	0.164944	0.2775…

	0	1	2	3	4	5	6	7
0	29.5	-15.…	5.93…	0	0	0	0	0
1	9.135…	-8.0…	5.99…	0	0	0	0	0
2	-0.15…	1.43…	1.50…	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

Bloc 8*8 filtré

II.1 Codage d’un algorithme de compression

Bloc 8*8 filtré

	0	1	2	3	4	5	6	7
0	29.5	-15.…	5.93…	0	0	0	0	0
1	9.135…	-8.0…	5.99…	0	0	0	0	0
2	-0.15…	1.43…	1.50…	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	29.5	-15.3984	5.93558	0	0	0	0	0	24.625	5.70368	2.16279	0	0	0	0	0	29.375	3.83…	-4.03…	0
1	9.13…	-8.09341	5.99261	0	0	0	0	0	7.488…	4.71826	-0.085…	0	0	0	0	0	-3.247…	-9.1…	1.570…	0
2	-0.1…	1.43653	1.50888	0	0	0	0	0	-2.14…	1.52104	-2.575…	0	0	0	0	0	1.50271	4.44…	1.118…	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	11.1…	-0.2620…	-0.856…	0	0	0	0	0	13	-1.42911	1.44186	0	0	0	0	0	24.875	-1.6…	1.532…	0
9	2.97…	0.37206	-0.943…	0	0	0	0	0	1.573…	1.11852	-0.732…	0	0	0	0	0	3.83679	6.45…	-7.49…	0
10	-0.9…	0.181111	-1.316…	0	0	0	0	0	0.748…	-1.07359	0.375	0	0	0	0	0	2.83249	0.50…	-3.95…	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	8.75	0.207867	0.6532…	0	0	0	0	0	15	-2.17358	1.75209	0	0	0	0	0	19.875	-5.2…	0.242…	0
17	0.24…	0.576641	0.7044…	0	0	0	0	0	-4.79…	-0.0276…	-1.232…	0	0	0	0	0	2.12358	-5.3…	4.622…	0
18	0.46…	-0.0751…	0.4267…	0	0	0	0	0	2.365…	-1.96543	2.00888	0	0	0	0	0	0.5692…	-0.6…	2.4205	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	12	0.109932	-8.497…	0	0	0	0	0	14.375	-0.6652…	-1.374…	0	0	0	0	0	13.125	2.01…	-2.16…	0
25	-0.6…	-3.01182	0.5493…	0	0	0	0	0	4.622…	-1.5498	1.50581	0	0	0	0	0	3.5096	-0.3…	-0.91…	0
26	-0.1…	0.86478	-1.103…	0	0	0	0	0	0.028…	-1.82902	3.7526	0	0	0	0	0	1.3578	0.95…	-0.08…	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Ensemble des blocs
8*8

Parcourt en Zigzag

5	float64	1	5.935576033089691
6	float64	1	0.0
7	float64	1	5.992611070960034
8	float64	1	1.4365340662206922
9	float64	1	0.0
10	float64	1	0.0
11	float64	1	0.0
12	float64	1	1.5088834764831844
13	float64	1	0.0
14	float64	1	0.0

Nb de coefficients :
36 864

Codage RLE

5	float64	1	5.935576033089691
6	str	2	#1
7	float64	1	5.992611070960034
8	float64	1	1.4365340662206922
9	str	2	#3
10	float64	1	1.5088834764831844
11	str	3	#23
12	float64	1	11.125000000000002
13	str	2	#7

Nb de coefficients :
8013

Fichier compressé

II.1 1er tests

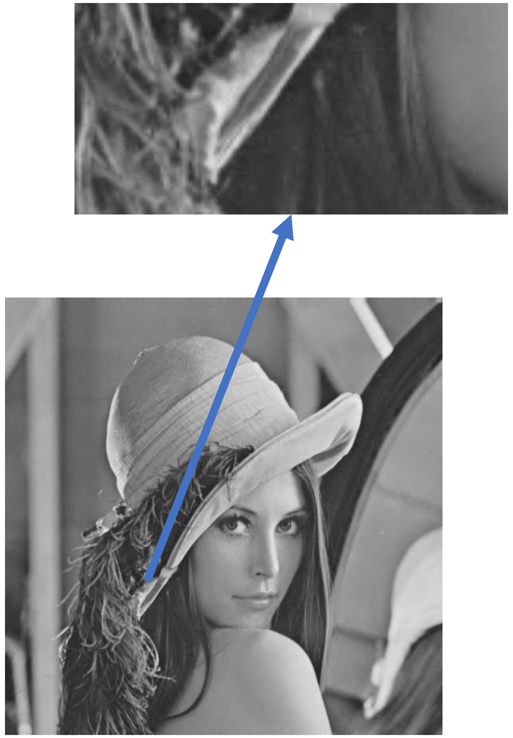


Image Originale (512*512 pixels)

$\approx 8\ Mo$

Compression 43%

$\approx 4{,}7\ Mo$

Compression 84%

$\approx 1{,}3\ Mo$

Compression 98%

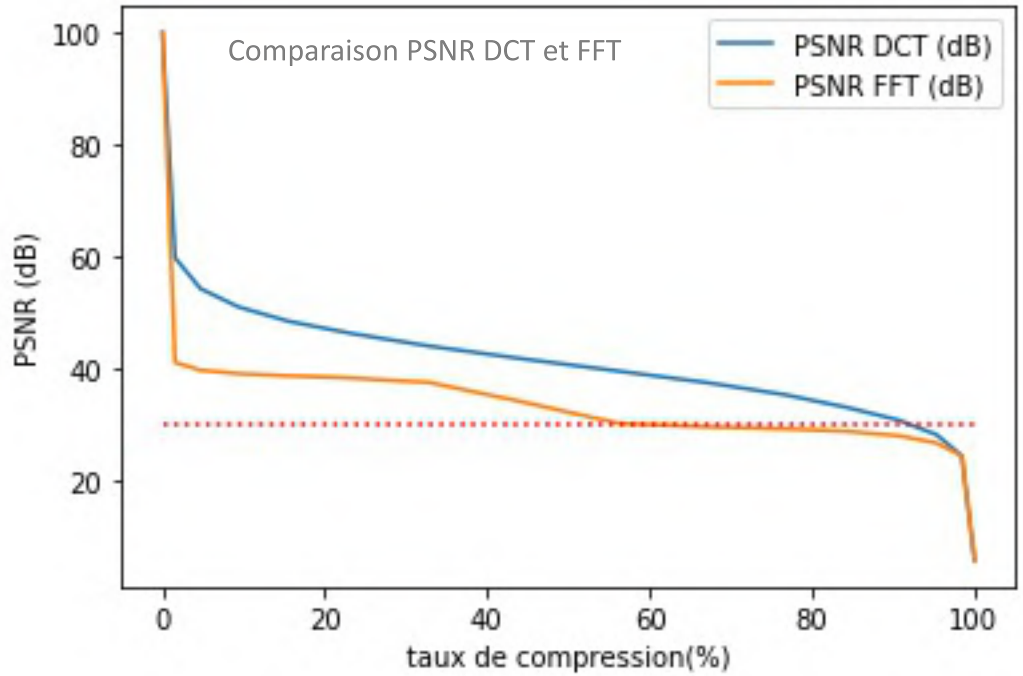
$\approx 0{,}16\ Mo$

$$taux\ compression = \frac{Nb\ coeff\ supprimés}{Nb\ coeff\ total} * 100$$

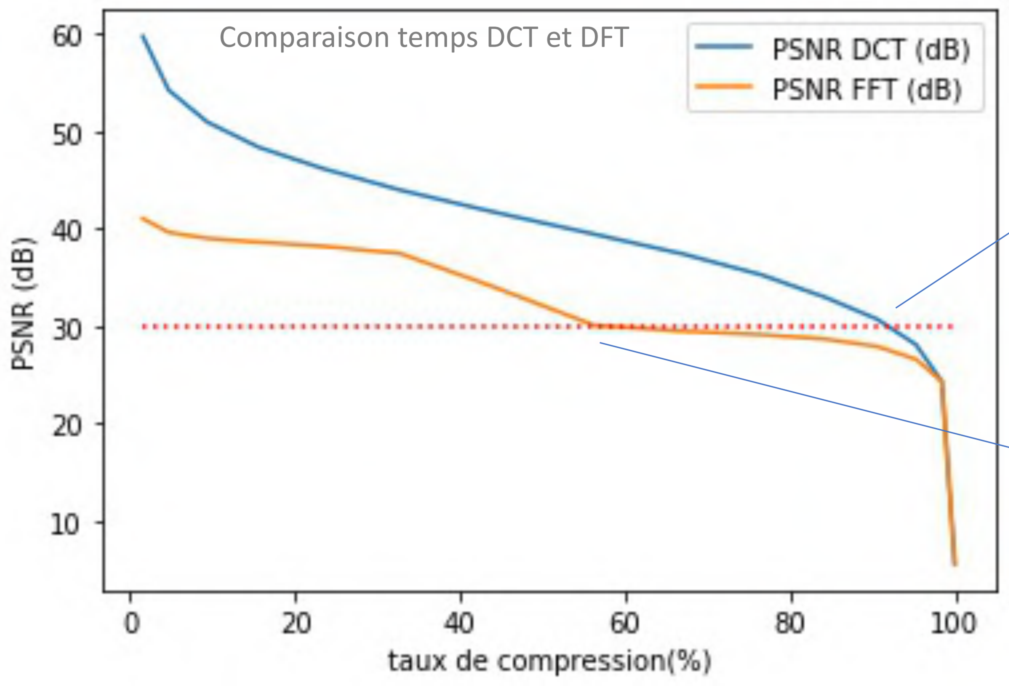
II.2 Comparaison

$$PSNR_{dB} = 10\log(\frac{255^2}{EQM}) \qquad EQM = \frac{1}{N^2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}|I_O(i,j) - I_c(i,j)|$$

Image originale 512*512 pixels



II.2 Comparaison



DCT: compression max sans perte de qualité :90%

DFT: compression max sans perte de qualité :55%

II.* Conclusion de cette partie

Compression 84% par DCT

Compression 84% par DFT

- Le choix de la DCT semble évident tant la perte de qualité avec DFT est importante à taux de compression équivalent

- Néanmoins l'argument d'un temps de compression/décompression moins élevé semble obsolète grâce à FFT

- Il reste cependant élevé dans les deux cas ce qui induit une consommation importante des appareils réalisant la compression/décompression

III. Étude d’une approche optique

III.1 Présentation du système

Hypothèses : Conditions de Gauss, approximation de Fraunhofer (onde incidente plane, $f \gg \frac{a^2}{\lambda}$)

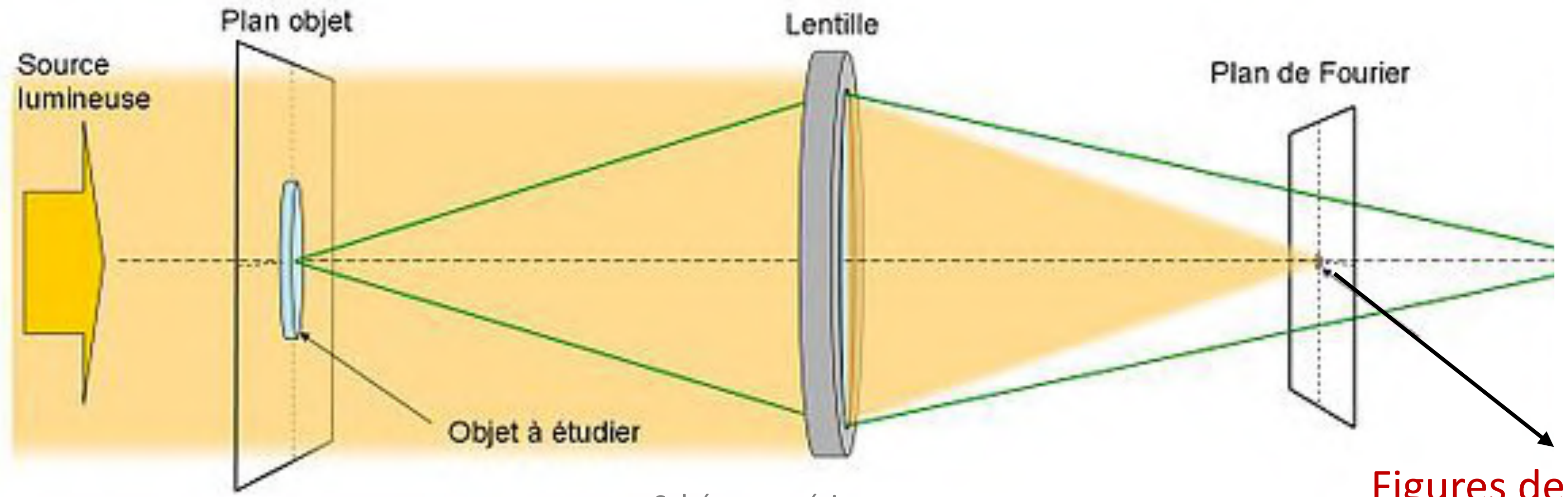
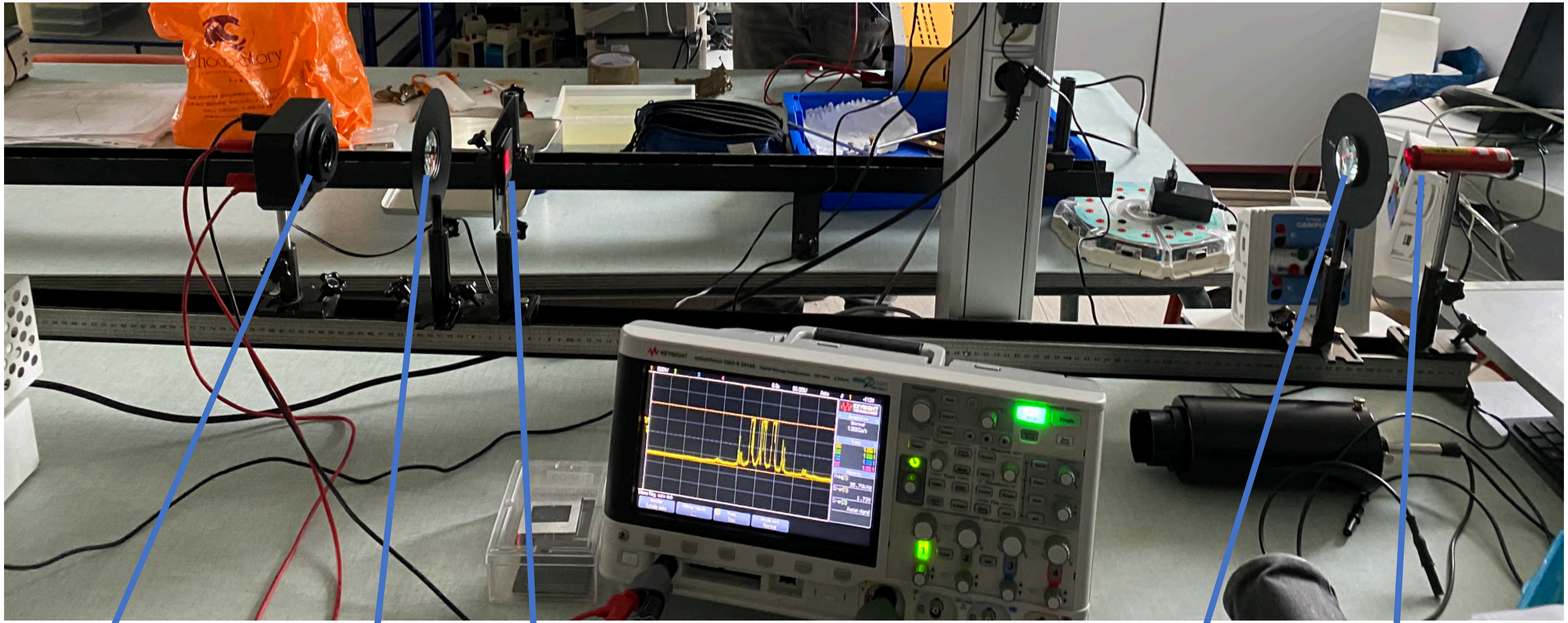


Schéma expérience

Plan de fourrier sur le plan focale image de la lentille

Figures de diffractions amplitudes proportionnelles à $|F|^2$

III.1 Présentation du système



Capteur CCD
(sur le plan de
Fourier)

Lentille
convergente

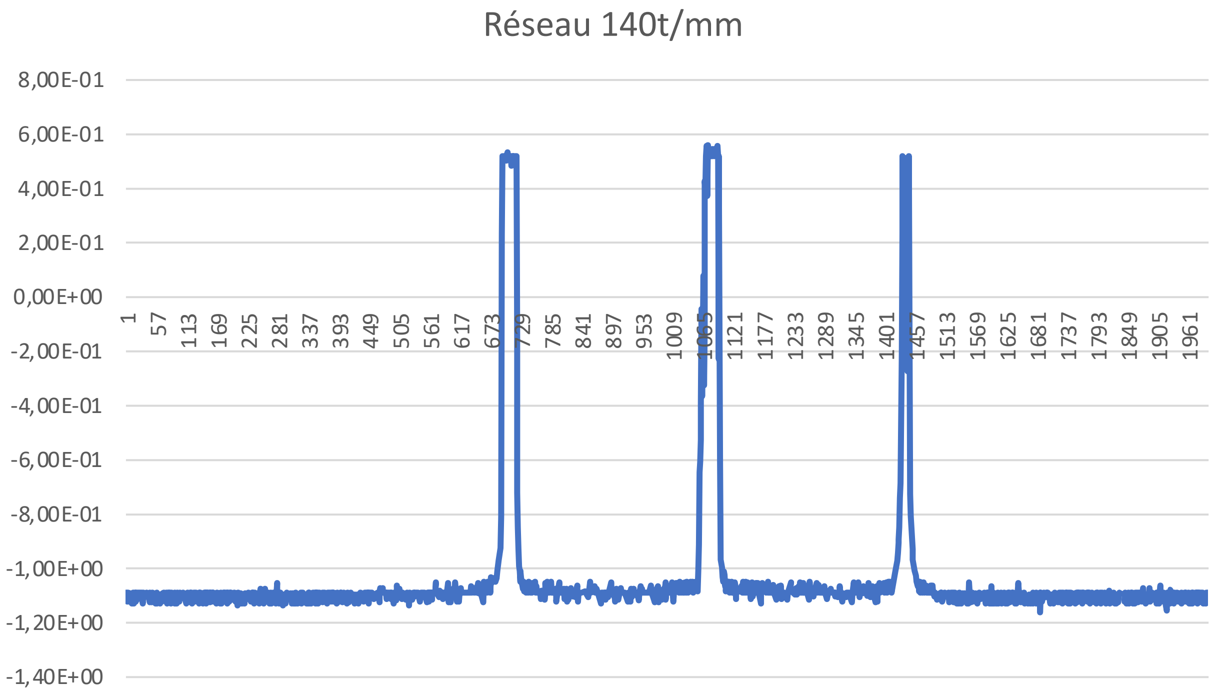
Objet

Lentille
convergente

Laser

III.2 Expérience

- Essais :
  - Avec réseau 140t/mm
  - Laser $\lambda = 650nm$



Données :

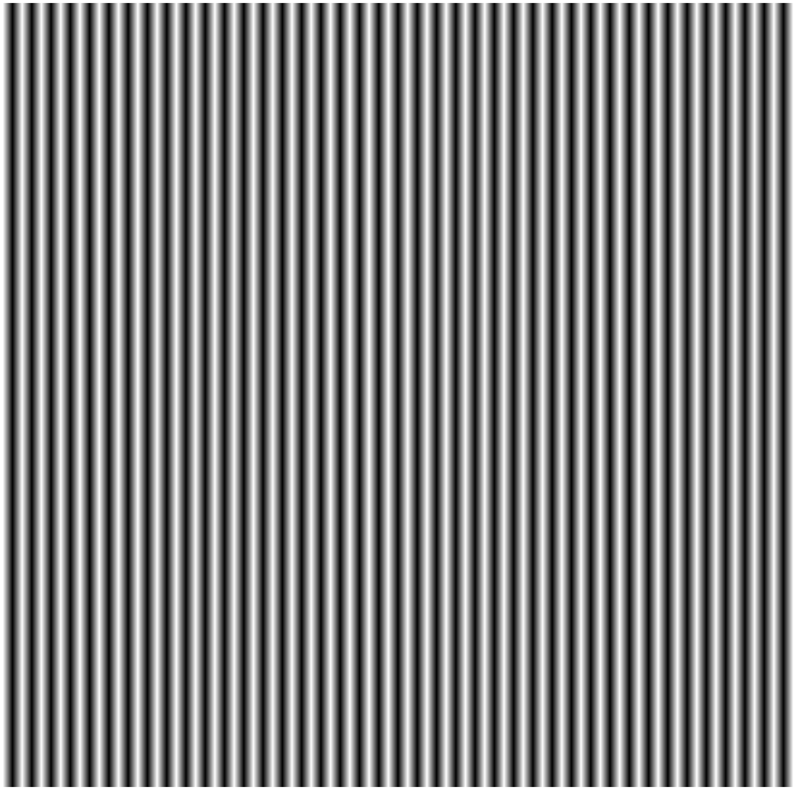
Nb pixels capteur : 2048 pixels

Longueur détecteur : 28.67 mm

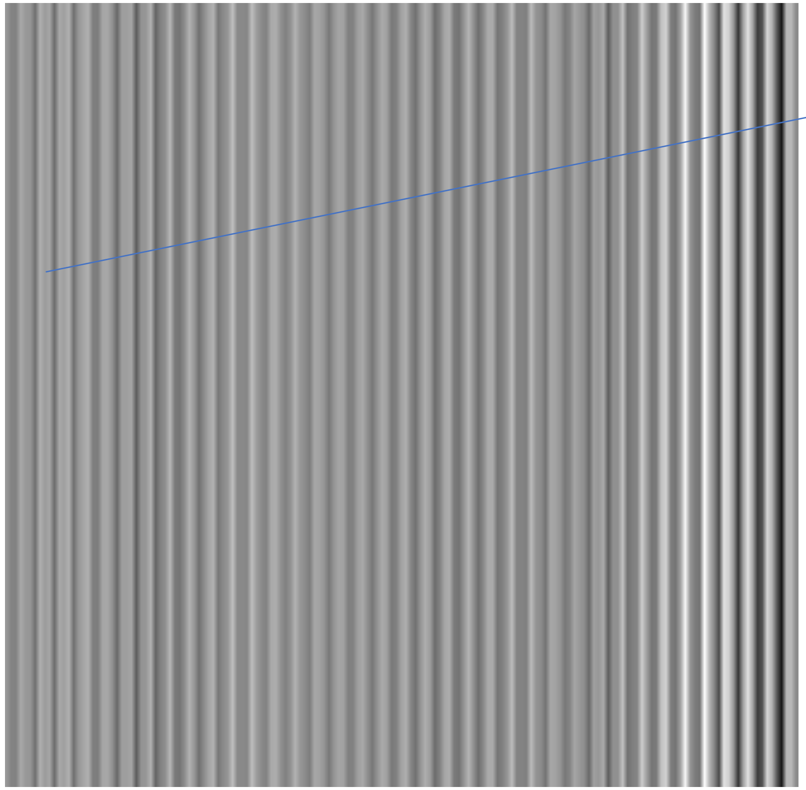
1 pixel = 14 µm

III.2 Expérience

- Résultat attendu



- Résultat expérimental après application de la transformée de Fourier inverse



Périodicité :
4 pixels

PSNR=28 dB
Temps de compression : 0sec

IV. Conclusion

- L’utilisation de la DCT au lieu de DFT à été démontré notamment sur le point de la fidélité de l’image.
- Il semble effectivement possible d’utiliser un système optique pour limiter l’utilisation du processeur lors du processus de compression.
  - En revanche les résultats obtenus, bien que montrant un fonctionnement relatif de cette méthode, sont peu suffisants pour pourvoir conclure sur une implémentation à grande échelle [1]

[1] A new approach for optical colored image compression using the JPEG standards (Abdulsalam G. Alkholidi)

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
@author: Nathan Foucher N°:17225
"""

# ==============================================================================
# module transformée en bloc
# ==============================================================================

import numpy as np
import scipy.fftpack as scifft



def add_matrix(M,n,c,m,p1,p2):  #injecte les sous matrices de taille p1*p2
                                #dans la matrice M à partir de l'indice (n,c)
    for i in range (p1):
        for j in range (p2):
            M[n+i][c+j]=np.copy(m[i,j])
    return()

def add_matrix_rounded(M,n,c,m,p1,p2):  #pareil que add_matrix mais arrondi
    for i in range (p1):                #et supprime les coeff <0
        for j in range (p2):
            if np.abs(m[i,j]) >= 0 :
                M[n+i][c+j]=np.abs(np.round(m[i,j]))
            else :
                 M[n+i,c+j]=0
    return()

def DCT(M,compressionDCT): #applique la DCT à des blocs de 8*8 d'une matrice M
    x,y =M.shape
    assert x==y #vérifie que l'image est bien carrée
    F=np.zeros((x,y),dtype=np.complex128)
    q=x//8
    #pour les blocs de 8
    p=8
    for i in range (q): #def la sous matrice
        n=i*8 #def le debut de la ligne la sous matrice à prendre
        for j in range (q):
            c=j*8 #def le debut de la colonne la sous matrice à prendre
            m=M[n:n+8,c:c+8]#sous matrice de taille 8*8
            mDCT=scifft.dctn(m,norm = 'ortho') #calcul de sa DCT
            mDCT=compressionDCT(mDCT) #compression selon la fct
            add_matrix(F, n, c, mDCT, p,p) #ajout dans la matrice finale

    return(F)

def DCTi(M): #applique la DCT inverse à des blocs de 8*8 d'une matrice M
    x,y =M.shape
```

```
    assert x==y #vérifie que l'image est bien carrée
    F=np.zeros((x,y))
    q=x//8
    #r=x%8
    p=8
    for i in range (q): #def la sous matrice
        n=i*8 #def le debut de la ligne la sous matrice à prendre
        for j in range (q):
            c=j*8 #def le debut de la colonne la sous matrice à prendre
            m=M[n:n+8,c:c+8]#sous matrice de taille 8*8
            mDCTi=scifft.idctn(m,norm = 'ortho') #calcul de sa DCT inversere
            add_matrix_rounded(F, n, c, mDCTi, p,p) #ajout dans matrice finale
    return(F)

def FT(M,compressionFT):  #applique la DFT à des blocs de 8*8 d'une matrice M
    x,y =M.shape
    assert x==y #vérifie que l'image est bien carrée
    F=np.zeros((x,y),dtype=np.complex128)
    q=x//8
    #r=x%8
    #pour les blocs de 8
    p=8
    for i in range (q): #def la sous matrice
        n=i*8 #def le debut de la ligne la sous matrice à prendre
        for j in range (q):
            c=j*8 #def le debut de la colonne la sous matrice à prendre
            m=np.copy(M[n:n+8,c:c+8])#sous matrice de taille 8*8
            mFT=np.fft.fft2(m) #calcul de sa DFT (domaine fréquentiel)
            #np.fft.fftshift(mFT)
            mFT=compressionFT(mFT) #compression selon la fct
            add_matrix(F, n, c, mFT, p,p) #ajout dans la matrice finale
    return(F)

def FTi(M):     #applique la DFT inverse à des blocs de 8*8 d'une matrice M
    x,y =M.shape
    assert x==y #vérifie que l'image est bien carrée
    F=np.zeros((x,y),dtype=np.complex128)
    q=x//8
    #r=x%8
    p=8
    for i in range (q): #def la sous matrice
        n=i*8 #def le debut de la ligne la sous matrice à prendre
        for j in range (q):
            c=j*8 #def le debut de la colonne la sous matrice à prendre
            m=M[n:n+8,c:c+8]#sous matrice de taille 8*8
            mFTi=np.real(np.fft.ifft2(m)) #calacul de sa DFT inverse
            #np.fft.ifftshift(mFTi)
            add_matrix_rounded(F, n, c, mFTi, p,p) #ajout dans la matrice fina
    F=np.real(F)
    return(F)
```

```
def normalize_img(IMG): #normalise les valeurs des coeff et les code sur 8bits
    norm=np.zeros((IMG.shape),dtype=float)
    maxi=np.amax(IMG)
    mini=np.amin(IMG)
    print(maxi,mini)
    for i in range (len(IMG)):
        for j in range (len(IMG)):
            norm[i][j]=int(255-255*(maxi-IMG[i][j])/(maxi-mini))
    return(np.array(norm))

def filtre_coupe_bande_horiz(fft,fi,ff): #fi: freq finitiale #ff : freq finale
    n=fft.shape[1]
    for j in range (n//2,n):
        if (j-n//2)/n > fi and (j-n//2)/n<ff:
            for i in range (fft.shape[0]) :
                fft[i][j]=0
                fft[i][-j]=0
    return(fft)


# ==============================================================================
# module PSNR
# ==============================================================================

import numpy as np

def PSNR(original, compressed):
    mse = np.mean((original - compressed) ** 2) #calcul la moyenne des diffs
    if(mse == 0):  # MSE=0 si il n'y a pas de bruit dans le signal
        return 100  #dans ce cas rien ne sert de calculer le PSNR
    max_pixel = 255.0
    psnr = 20 * np.log10(max_pixel / np.sqrt(mse))
    return psnr

# ==============================================================================
# module codage RLE
# ==============================================================================
import numpy as np


def zigzag(M):
    #assert M.shape[0]==M.shape[1] #vérifions que la matrice est carrée
    n=len(M)

    sections=[0]*(n*2-1)
    for i in range (n*2-1):
        sections[i]=[]

    for i in range(n):
        for j in range(n):
            a=i+j
            if(a%2 ==0):
```

```
                #ajout au début
                sections[a].insert(0,M[i][j])
            else:

                #ajout à la fin de la liste
                sections[a].append(M[i][j])
    F=[]
    for i in sections:
        for j in i:
            F.append(j)
    return(F)

def zigzagi(M):
    n=int(np.sqrt(len(M)))
    F=[]
    a=0
    #création d'un tableau contenant les listes diagonales
    for i in range (1,n+1): #jusqu'a diagonale
        b=a+i
        F.append(M[a:b])
        a=b
    for i in range (n-1,0,-1): #après diagonale
        b=a+i
        F.append(M[a:b])
        a=b
    for i in range (len(F)):    #inverser les listes à un indice impaire
        if (i%2!=0):
            F[i].reverse()
   #ajout des listes dans la mtrice finale
    A=np.zeros((n,n),dtype=complex)
    n=len(F)
    for i in range(n//2):     #matrice finale avant diagonale
        for j in range (len(F[i])):
            A[i-j][j]=F[i][j]
    for i in range(n//2+1):     #matrice finale après diagonale
        for j in range (len(F[i])):
            A[n//2-(i-j)][n//2-j]=F[n-1-i][len(F[n-1-i])-1-j]
    return(A)


def RLE(M):
    F=[]
    b=0
    for i in range (len(M)):
        if M[i]==0:
            b+=1
        else :
            if b==0 :
                F.append(M[i]) #si pas de 0 on ajoute la valeur
            else :
                F.append("#{int}".format(int=b)) # rajoute#nb de 0
```

```
                F.append(M[i])
            b=0
    #faire pour les elements restants
    if b!=0 :
        F.append("#{int}".format(int=b))
    return(F)

def RLEi(M):
    F=[]
    for i in range (len(M)):
        if "#" in str(M[i]):
            l=len(M[i])
            for i in range (int(M[i][1:l])):
                F.append(0)
        else :
            F.append(M[i])
    return(F)

    F=[]
    for i in range (len(M)):
        if "#" in str(M[i]):
            for i in range (int(M[i][1])):
                F.append(0)
        else :
            F.append(M[i])
    return(F)
```