2021

Rapport de TP
Compression des images fixes par JPEG

Réalisé par :

- **AREZKI Yasmine**

1. Introduction

On distingue les techniques de compression conservatrice qui permettent de reconstituer, en fin de processus, une image identique à l'image initiale, et les techniques de compression non conservatrice ou dites avec perte. Ce sont ces dernières qui nous intéressent dans le cadre de ce TP. Ces méthodes peuvent réaliser une compression très poussée, moyennant une certaine perte d'information. L'image reconstituée en fin de processus diffère de l'image initiale, mais la différence de qualité n'est pratiquement pas perçue par l'œil humain.

Le principe de la compression JPEG consiste à supprimer les détails les plus fins d'une image, autrement dit, les détails que le système visuel humain ne peut détecter.

Nous allons dans ce TP réaliser la suite des opérations à effectuer pour compresser une image en passant par les trois phases majeures de la compression d'images avec perte :

a. Transformée en cosinus discrète bi-dimensionnelle (DCT)
b. Quantification
c. Codage entropique

2. Environnement de développement

Pour réaliser ce TP, nous avons choisi le langage Python, étant donné que c'est le langage le plus utilisé en vision par ordinateur. Ceci est dû à la présence de plusieurs librairies d'algèbre linéaire et de traitement d'images optimisées et adaptées. Dans ce travail pratique, nous utilisons la librairie **NumPy** pour les structures de données matricielles, **Scipy** pour l'appel des fonctions de Transformée en Cosinus Discrète bidimensionnelle (TCD) et **Matplotlib** pour la visualisation et la comparaison des résultats.

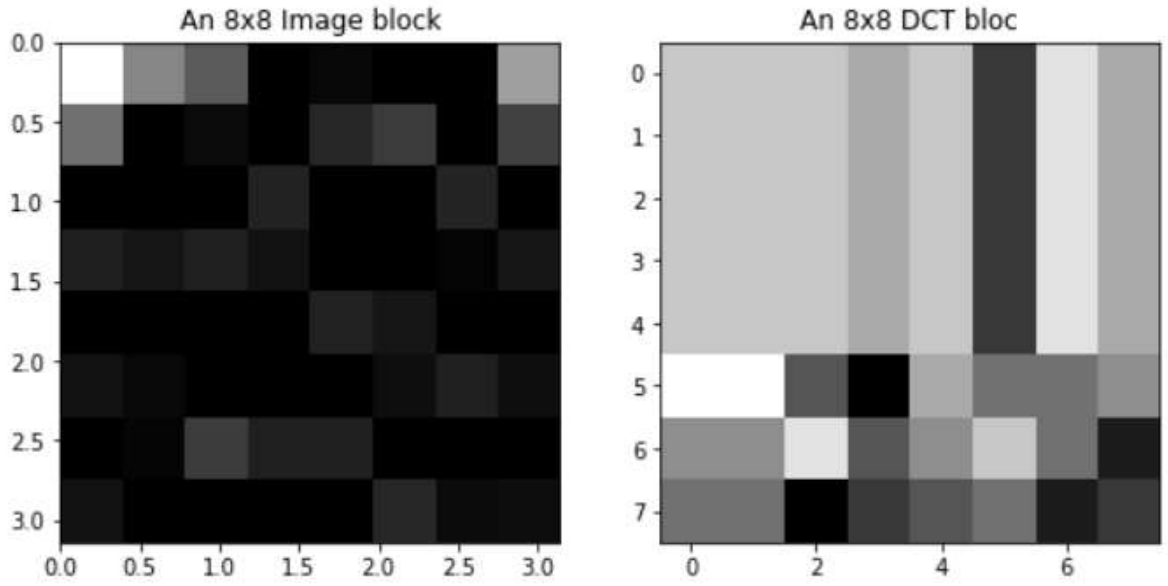
Pour faciliter la gestion des librairies, nous utilisons le gestionnaire de packages Anaconda, qui permet de créer des environnements virtuels séparés. Un environnement virtuel est créé pour ce TP. Pour l'écriture du code python, nous utilisons le notebook interactif Jupyter.

Nous allons décrire les étapes de réalisation de ce TP par parties en suivant l'énoncé.

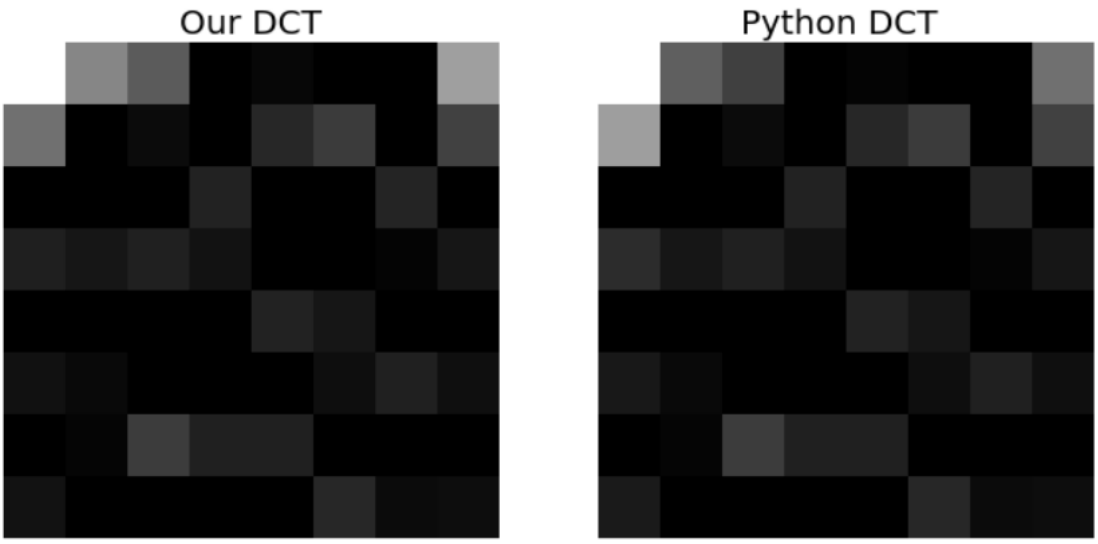
3. Partie 1 : transformation DCT et DCT inverse

Nous avons commencé par la programmation de la fonction qui calcule la DCT directe basée sur les produits matriciels. Nous avons découpé la formule en sous formules afin de les traduire en un ensemble de vecteurs et matrices. Le résultat final de la DCT est calculé à partir de ces vecteurs et matrices crées, en appliquant un produit matriciel.

Après application de la DCT directe sur un bloc de l'image de Lenna, voici l'affichage du résultat :

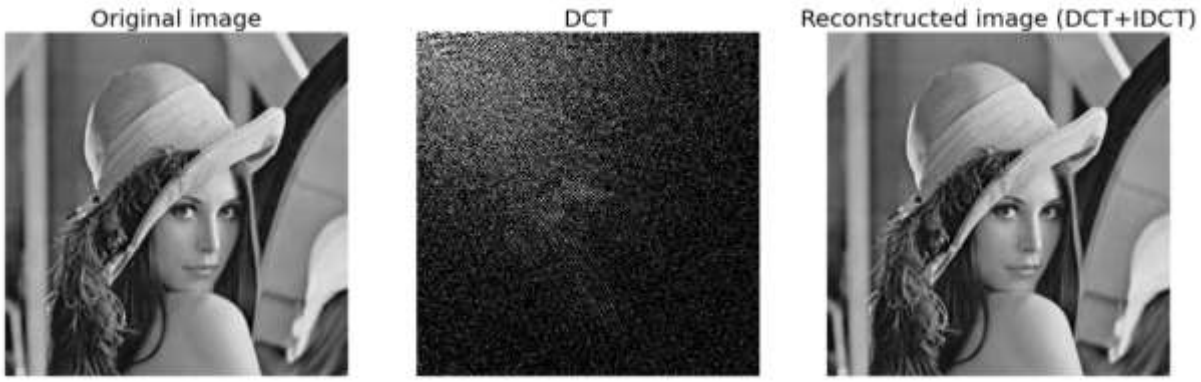


Nous avons ensuite appliqué la DCT fournie par la librairie **Scipy** de python. Il existe, en théorie, 8 types de DCT, seuls les 4 premiers types sont implémentés dans Scipy. La DCT directe se réfère généralement au DCT de type 2, et la DCT Inverse se réfère généralement au DCT de type 3 qui est-elle même la IDCT(Inverse DCT). Nous avons choisi le mode de normalisation « ortho » étant donné qu'il est l'équivalent de la fonction DCT de Matlab.



En comparant notre DCT avec celle de python, nous remarquons que leurs résultat est similaire.

Nous avons appliqué le DCT sur l'image entière de Lenna.

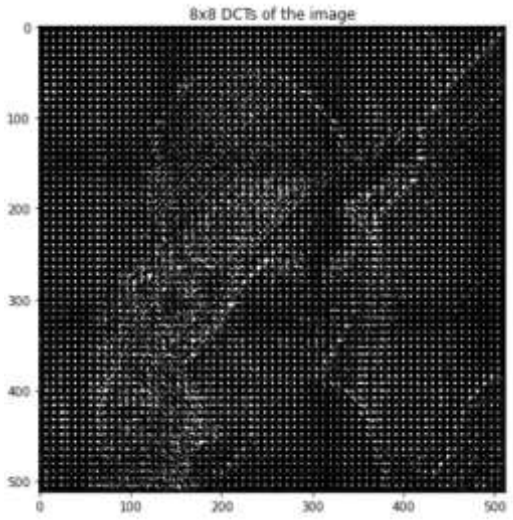


Commençons par analyser le résultat de la DCT. Nous pouvons constater que les pixels noirs qui représentent les fréquences élevées sont majoritairement dans le coin droit du bas de la matrice, or l'œil humain discerne mal ces fréquences, tandis que dans le coin gauche du haut, on retrouve les basses fréquences (les pixels blancs).

En comparant l'image originale avec l'image reconstituée (DCT+DCT inverse), nous pouvons affirmer qu'elles sont presque similaires.

Conclusion : Le principe de la DCT (transformation en cosinus discrète bidimensionnelle) est de transformer la matrice de départ (obtenue au moyen de l'étape précédente) en matrice de fréquences permettant de révéler les caractéristiques propres à une compression efficace. Elle prend donc un ensemble de points d'un domaine spatial et les transforme en une représentation équivalente dans le domaine fréquentiel. Après compression, la transformation inverse permet de revenir à l'image de départ (enfin presque).

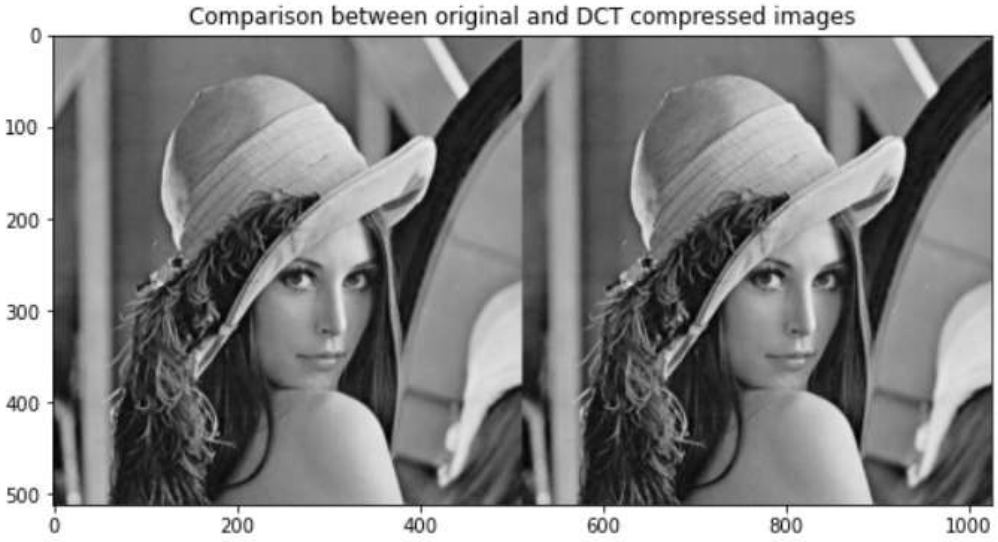
Nous poursuivons ce travail par un découpage de l'image initiale en bloc de taille 8× 8 pixels. Nous appliquons ensuite la DCT sur chaque bloc. Voici le résultat obtenu :



Pour reconstruire l'image initiale, nous avons juste besoin de suivre le processus inverse, c'est-à-dire, décomposer l'image DCT en bloc de taille 8x8 pixels, puis appliquer la DCT inverse sur chaque bloc comme suit :

```
im_dct = np.zeros(imsize)
#Decomposer la DCT en bloc 8*8 et appliquer la DCT inverse sur chaque bloc
for i in r_[:imsize[0]:8]:
    for j in r_[:imsize[1]:8]:
        im_dct[i:(i+8),j:(j+8)] = idct2( dct[i:(i+8),j:(j+8)] )

#Affichage du resultat
fig = plt.figure(figsize=(9,15))
plt.imshow( np.hstack( (im, im_dct) ) ,cmap='gray')
plt.title("Comparison between original and DCT compressed images" )
```

```
Text(0.5, 1.0, 'Comparison between original and DCT compressed images')
```



Nous avons besoin de prendre les deux images originale et reconstruite et les comparer pour déterminer si elles sont identiques ou presque identiques d'une manière ou d'une autre. La méthode consiste à utiliser des algorithmes tels que l'erreur quadratique moyenne (MSE). Nous définissons notre fonction **mse**, qui prend deux arguments : imageA et imageB (c'est-à-dire les images que nous voulons comparer pour la similitude) et retourne la valeur du MSE calculé.

```
def mse(imageA, imageB):
    # the 'Mean Squared Error' between the two images is the
    # sum of the squared difference between the two images;
    # NOTE: the two images must have the same dimension
    err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
    err /= float(imageA.shape[0] * imageA.shape[1])

    # return the MSE, the lower the error, the more "similar"
    # the two images are
    return err

# im : original image / im1 : Reconstructed image
print(mse(im,im1))
```

```
2.2089802571951822e-27
```

Le MSE = 2.20, ce qui est une valeur très petite qui signifie que les deux images originale et reconstruite sont presque similaires.

4. Partie 2 : Quantification et codage en zigzag

La quantification représente la phase non conservatrice du processus de compression JPEG. Elle permet, moyennant une diminution de la précision de l'image, de réduire le nombre de bits nécessaires au stockage. Pour cela, elle réduit chaque valeur de la matrice DCT en la divisant par un nombre (quantum), fixé par une table (matrice 8 x 8) de quantification.

En choisissant des tailles de pas de quantification très élevés, le résultat est un taux de compression important pour une qualité d'image médiocre. En choisissant des tailles de pas petites la qualité de l'image reste excellente et les taux de compression n'auront rien d'extraordinaire.

Nous avons déclaré une fonction qui permet de créer la table de pondération correspondante au facteur de qualité récupéré en entrée.

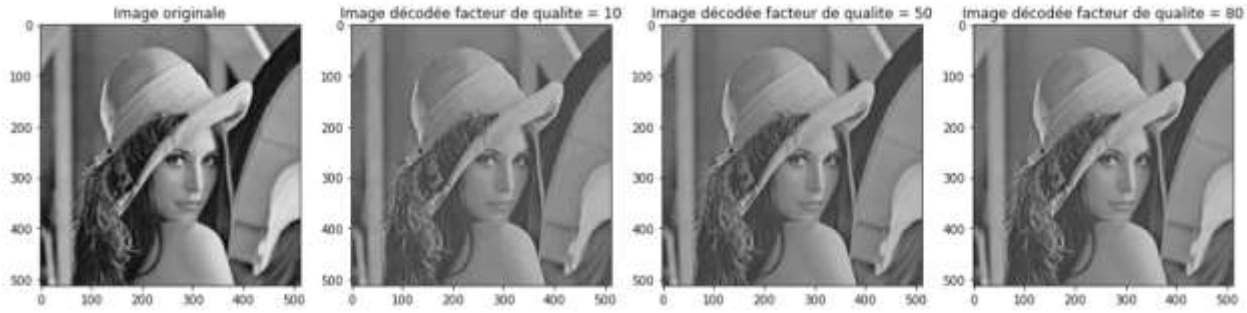
```
### PARTIE 2 #####

def table_de_ponderation (Q) :
    TabQ = np.zeros((8,8))
    Tab50 = 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]
            ])

    if(Q<50):
        E = 50/Q
    else :
        E = 2 - (Q/50)

    TabQ = Tab50 * E + 0.5

    return TabQ
```

Nous avons codé ensuite les fonctions suivantes :

```
def ponderation(Bloc , TabQ):
    BlocPondere = np.zeros((8,8))
    for i in range(0,8):
        for j in range(0,8):
            BlocPondere[i,j] = Bloc[i,j] / TabQ[i,j]
            BlocPondere[i,j] = round(BlocPondere[i,j])
    return BlocPondere


def quantifier(imsize,dct,TabQ):
    dct_quantifie= np.zeros(imsize)
    for i in r_[:imsize[0]:8]:
        for j in r_[:imsize[1]:8]:
            dct_quantifie[i:(i+8),j:(j+8)] = ponderation( dct[i:(i+8),j:(j+8)],TabQ)
    return dct_quantifie

def deponderation(BlocPondere , TabQ):
    Bloc = BlocPondere * TabQ
    return Bloc

def quantification_inverse(dct_quantifie,TabQ):
    dct= np.zeros(imsize)
    for i in r_[:imsize[0]:8]:
        for j in r_[:imsize[1]:8]:
            dct[i:(i+8),j:(j+8)] = deponderation( dct_quantifie[i:(i+8),j:(j+8)],TabQ)
    return dct
```

Ces fonctions permettent de pondérer les coefficients de la DCT et les quantifier, ainsi que de revenir dans le sens inverse (dépondération, déquantification).

Nous avons calculé la DCT inverse pour les différents facteurs de qualité demandés.




```
Facteur de qualité = 10 , mse =  17239.781143971424
Facteur de qualité = 50 , mse =  15592.043783331965
Facteur de qualité = 80 , mse =  12909.136993514512

Facteur de qualité = 10 , psnr =  5.765486126372702
Facteur de qualité = 50 , psnr =  6.201773152334113
Facteur de qualité = 80 , psnr =  7.021831512492294
```

Nous remarquons que plus on augmente le facteur de qualité, la qualité de l'image compressée obtenue augmente, l'erreur quadratique moyenne diminue ce qui est très logique. Plus le taux de compression diminue, plus le **psnr** augmente.

Le codage de la matrice DCT quantifiée se fait en parcourant les éléments dans l'ordre imposé par une séquence particulière appelée séquence zigzag.

On lit les valeurs en zigzags inclinés à 45° en commençant par le coin supérieur gauche et en finissant en bas à droite. Cette séquence a la propriété de parcourir les éléments en commençant par les basses fréquences et de traiter les fréquences de plus en plus hautes. Puisque la matrice DCT contient beaucoup de composantes de hautes fréquences nulles, l'ordre de la séquence zigzag va engendrer de longues suites de 0 consécutifs. Ceci facilite de nouveau les étapes de compression suivantes : les codages RLE et HUFFMAN. Voici l'exemple du code :

```
Bloc de 8*8 choisi aléatoirement :
 [[159 161 159 164 157 153 154 147]
 [160 164 164 157 156 152 149 142]
 [159 163 161 158 153 152 148 141]
 [166 162 163 158 153 155 145 143]
 [161 160 159 160 156 152 148 139]
 [161 160 161 160 155 149 144 139]
 [162 161 163 157 152 152 143 138]
 [163 162 158 155 156 153 145 145]]

 Son tableau monodimentionnel equivalent aprés une lecture en ZIGZAG  :
 [159. 161. 160. 159. 164. 159. 164. 164. 163. 166. 161. 162. 161. 157.
 157. 153. 156. 158. 163. 160. 161. 162. 160. 159. 158. 153. 152. 154.
 147. 149. 152. 153. 160. 161. 161. 163. 162. 163. 160. 156. 155. 148.
 142. 141. 145. 152. 155. 157. 158. 155. 152. 149. 148. 143. 139. 144.
 152. 156. 153. 143. 139. 138. 145. 145.]
```

5. Partie 3 : Principe de l'entropie

L'objectif à ce stade est de comprimer et stocker toute l'information dans le minimum de symboles, donc de place mémoire des ordinateurs. C'est une phase de compression conservatrice. Le codage proprement dit fait correspondre à un symbole ou à un ensemble de symboles un certain code. Il s'accompagne d'un modèle statistique calculant, de manière plus ou moins complexe, la probabilité d'apparition du symbole à coder. Ainsi, un symbole ayant une grande probabilité d'occurrence sera codé sur très peu de bits et inversement.

Il a été prouvé que l'algorithme d'Huffman est une des meilleures méthodes de codage à codes de longueur fixée (aujourd'hui, une méthode arithmétique dérivant d'Huffmann a été trouvée, mais cette méthode-ci est la plus simple à expliquer).

L'algorithme de Huffman consiste à construire progressivement un arbre binaire en partant des nœuds terminaux.

- On sélectionne les deux symboles les moins probables, on crée deux branches dans l'arbre et on les étiquette par les deux symboles binaires 0 et 1.

- On actualise les deux listes en rassemblant les deux symboles utilisés en un nouveau symbole et en lui associant comme probabilité la somme des deux probabilités sélectionnées.

- On recommence les deux étapes précédentes tant qu'il reste plus d'un symbole dans la liste.

6. Partie 4 : décompression

Le processus JPEG de compression conduit au stockage ou au transport des informations sur l'image sous un volume mémoire réduit. La restitution de l'image nécessite ensuite que l'on fasse le chemin inverse dans un processus de décompression.

Lors de la restitution de l'image (décompression), il suffira de réaliser l'opération inverse en suivant ces étapes :

1. Rétablissement de la matrice DCT quantifiée, en suivant le chemin inverse de la méthode de Huffman.

2. Rétablissement de la matrice DCT déquantifiée en multipliant les éléments de la matrice par le coefficient correspondant de la matrice de quantification.

3. Rétablissement de la matrice initiale à l'aide d'une transformation DCT inverse.

La matrice de pixels de sortie n'est plus exactement la même que la matrice d'entrée, mais la perte de données doit rester peu perceptible.

7. Conclusion

A l'heure actuelle la norme JPEG fait partie des algorithmes les plus utilisés parce ce qu'elle atteint des taux de compression très élevés pour une perte de qualité indécelable par le système visuel humain. De plus elle s'adapte aux besoins de l'utilisateur en permettant de choisir la qualité souhaitée grâce au paramétrage de la quantification