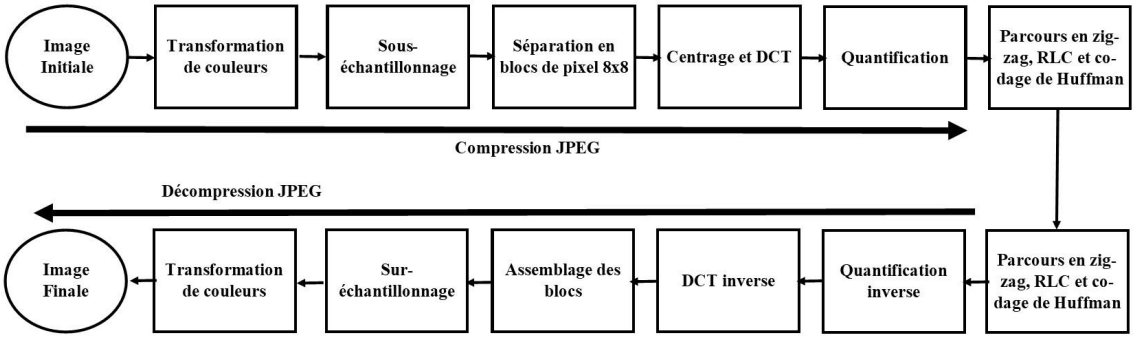
CR JPEG

Jimmy-Antoine PAN / Alix HAVRET
2G1TD1TP2

Projet : Réalisation d'un codeur/décodeur JPEG simplifié

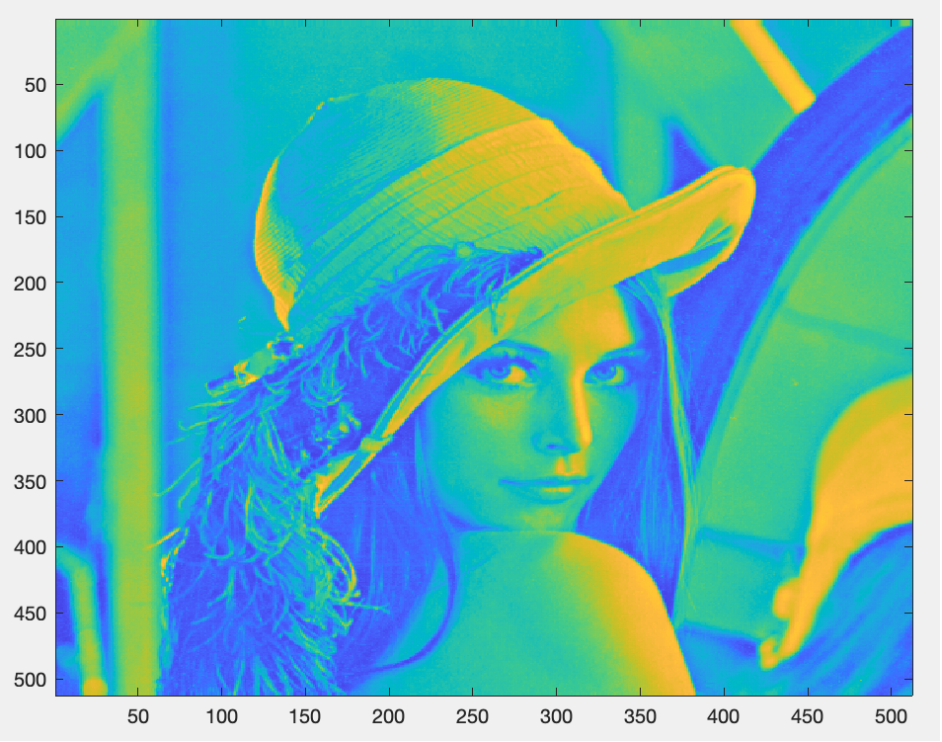
L'objectif du projet de 8 heures est de réaliser un codeur et un décodeur JPEG.

On commencera par la compression de l'image ainsi que sa décompression, mais quand elle est en gris (avec des nuances de gris différents).
Voici les étapes décrites dans le projet sous forme de graphique :



1. Structure d'une image (non compressée)

Dans un premier temps nous allons lire l'image et la faire passer dans des niveaux de gris différents avec la fonction "rgb2gray(I)".



Chaquepixel de l'image sera alors codé entre 0(le noir, ici bleu) et 255(le blanc, ici jaune).
Nous nommons la matrice de l'image image2

2. Encodage d'une data nuit

Nous avons une matrice image2 de taille 512x512, or nous devons travailler avec des blocs de 8x8 pour faire l'encodage. Le nombre de pixel total est dim_img=512*512=262144
Donc il nous faut avoir un multiple de 8 dans le nombre de colonnes et de lignes.
On traitera ensuite chaque bloc séparément.

Notre cas : 512x512
Donc ici aucune modification. Nous séparons la matrice en blocs de 8x8. La partie padding est cependant là pour ajouter des colonnes et des lignes de 0 jusqu'à obtenir un nombre de lignes/colonnes multiple de 8.

Nous travaillons maintenant avec des blocs de tailles 8x8

2.1. Centrage

On commence par enlever 128 à chaque valeur des blocs 8x8 car on veut avoir des valeurs comprises entre -128 et 127. Le but de cette partie est de centrer en 0 notre matrice pour réduire les valeurs des éléments de la matrice (en valeur absolue). Cela nous permet de préparer la suite dont le but est d'annuler le plus d'éléments de notre matrice.

2.2. Transformée en cosinus

Dans cette partie nous calculons la DCT (Transformée en cosinus) pour chacun des blocs de taille 8x8.

Le calcul de la DCT ne peut pas se faire sur une image entière d'une part parce que cela générerait trop de calculs et d'autre part parce que le signal de l'image doit absolument être représenté par une matrice carrée. C'est pour cela que nous avons créé divers blocs de 8x8.

Objectif de la DCT :

$$F(u,v) = C_u C_v \sum_{i=0}^{7}\sum_{j=0}^{7} f(i,j) cos\left(\frac{(2i+1)u\pi}{16}\right) cos\left(\frac{(2j+1)v\pi}{16}\right)$$

$$f(i,j) = \sum_{u=0}^{7}\sum_{v=0}^{7} C_u C_v F(u,v) cos\left(\frac{(2i+1)u\pi}{16}\right) cos\left(\frac{(2j+1)v\pi}{16}\right)$$

avec $C_0 = \frac{1}{\sqrt{(8)}}$ et $C_u = \frac{1}{2}$ si $u \neq 0$.

Nous allons appliquer à chacun des termes les formules de la DCT à l'aide de la fonction "dctmx" et nous obtenons des blocs 8x8 que l'on nomme dct

2.3. Quantification

La quantification nous permet de gagner plus de place car la DCT ne fait aucune compression.
Nous allons diviser termes à termes les matrices 8x8 après DCT par les termes de la matrice de quantification Q. Une grande partie des termes de Q est plus grande que celle de la matrice dct que l'on obtient. On aura donc beaucoup de termes qui seront proches de 0.

Nous allons effectuer un arrondi avec round() : on obtient des blocs F. De cette façon, nous réduisons le nombre d'éléments de notre matrice et donc le nombre de symboles.

Nous perdons cependant de l'information avec cet arrondi. Nous déterminerons si cet arrondi a un gros impact à la fin du TP.

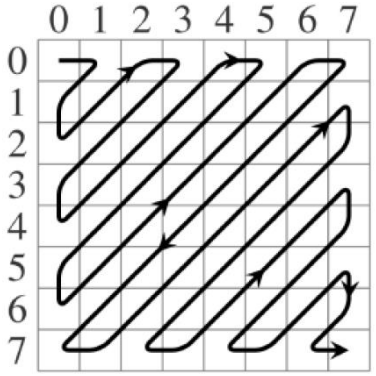
$$\hat{F}(u,v) = round\left(\frac{F(u,v)}{Q(u,v)}\right)$$

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

Matrice de quantification dans le cas d'une image en nuances de gris.
A l'issue de la quantification nous allons obtenir de nombreux zéros et avec le codage RLC on va faire des paquets avec les zéros pour réduire la taille des matrices et donc compresser notre image.

2.4. Parcours en zigzag, RLC, et codage de Huffman

On effectue ensuite un parcours en zigzag de chaque bloc afin d'obtenir les vecteurs qui contiennent des longues suites de 0 que l'on nomme zig.
Le but de ce parcours en zigzag est que nous puissions rassembler les 0 entre eux et donc compresser notre image sans perte de données supplémentaire à l'aide du RLC (Run Length Coding).



En effet, une fois le vecteur contenant les 64 valeurs du bloc 8×8 construit, nous effectuerons un codage RunLength pour obtenir un vecteur avec toutes les valeurs pour obtenir des vecteurs vrlc qu'il faudra rassembler.

Méthode RunLength : Nous allons remplacer les 0 par le 257 suivi de n avec n qui correspond au nombre de 0. On rassemble ensuite les vecteurs obtenus par concaténation en un seul vecteur que l'on nomme Vrlc.
Remarque: avec notre méthode, nous partons du principe qu'il y a plus de suites composées de deux 0 et que de suites composées d' un ou deux 0. Dans le cas contraire, nous n'effectuons bien évidemment aucune compression. Nous avons choisi de ne pas séparer les cas pour alléger le code. Le résultat quant à lui sera inchangé.

Une fois compressée, nous pouvons avoir les nouvelles dimensions de la matrice de l'image initiale.
Nous sommes passés d'une image de taille 512x512, soit 262144 pixels en tout, à une image de 1x51737 soit 51737 pixels ! **On obtient un taux de compression T=5,07** : la compression est un succès !

Nous déterminer son code de Huffman à partir des éléments de la matrice compressée. En utilisant la fonction unique(), nous déterminons le vecteur contenant toutes les valeurs différentes, dites symboles, de notre matrice compressée : ce vecteur se nomme symboles.
En comptant la récurrence de chaque symbole dans Vrlc, nous pouvons déterminer statistiquement la probabilité de présence de chaque symbole. On détermine ainsi le vecteur correspondant au dictionnaire d'un arbre Huffman avec la fonction huffmandict() que l'on nomme dico.

On termine la compression par un encodage.

3. Décodage d'une data unit

Maintenant que nous avons compressé une image, nous allons effectuer la décompression en appliquant les opérations inverses de toutes les étapes effectuées jusqu'à maintenant. Il faudra ensuite évaluer les pertes d'information, c'est à dire voir si l'image décompresser correspond bien à l'image de départ

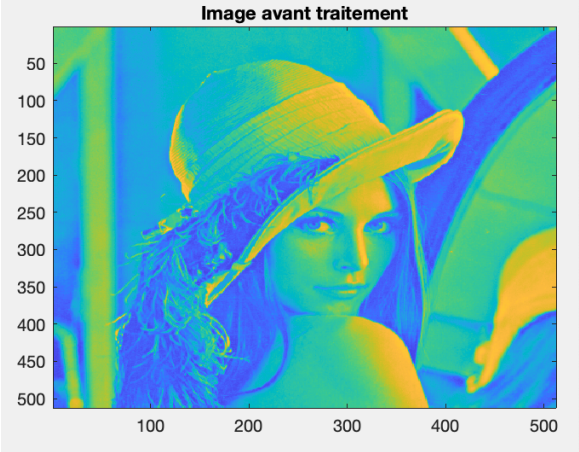
Tout d'abord, nous allons décoder notre encodage avec notre dictionnaire. Nous créons une fonction *InverseRunLength()* qui prend en paramètre le message decodé, le nombre de ligne et colonne de la matrice de l'image initiale. Cette fonction restaure les 0 que l'on a supprimé et remplacé par 257 suivi de n où n correspond au nombre de 0 supprimé. Cette étape correspond à la décompression. Nous obtenons zigInverse.

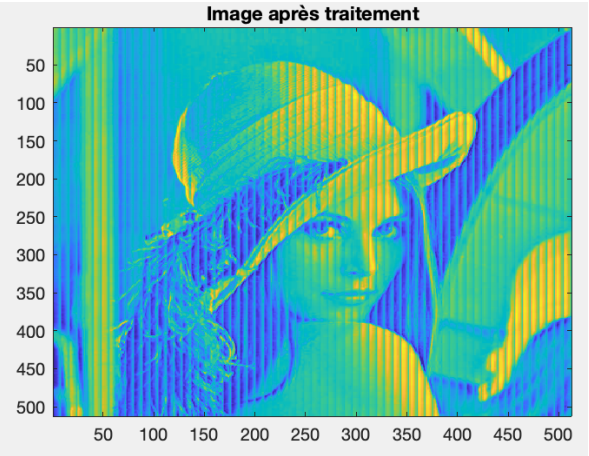
Nous travaillons maintenant avec des blocs de tailles 8x8
Nous reconstruisons ensuite les blocs 8x8 et à l'aide de la fonction inverse zigzag avec la fonction Inversezigzagcode on effectue le parcours en zigzag dans l'autre sens de manière à associer le bon pixel de chaque bloc correspondante à l'image de départ. Nous obtenons dctinverse.

Ensuite, nous effectuons la quantification inverse qui consiste à multiplier chaque pixel des blocs par les coefficients de la matrice de quantification terme à terme. Comme mentionné précédemment, nous avons perdu de l'information suite à l'arrondi que nous avons effectué. En effet, il nous a permis de réduire le nombre de symboles et de faciliter les calculs. Nous rassemblons le tout et nous obtenons Finter qui résulte de la concaténation de nos matrices 8x8.
On obtient alors une matrice 8x32768 =262144=dim_img. La décompression est un succès !

Nous re-composons notre image de départ passant de Finter à imagefinal qui est une matrice carrée.

Une fois notre image recomposée nous devrions retrouver à l'affichage une image similaire à celle de départ.

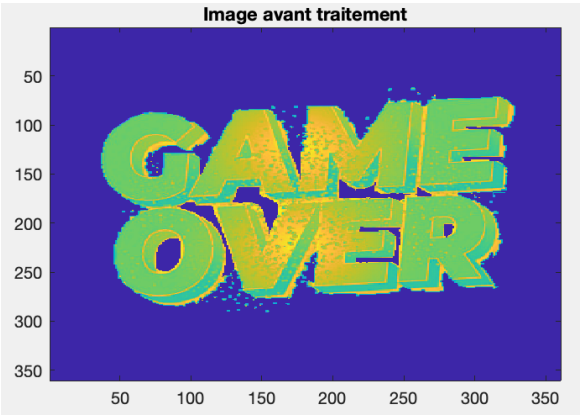




La reconstitution de l'image est un succès ! Cependant, nous observons des défauts sur l'image après traitement, en effet, nous avons arrondi nos valeurs à une étape pour simplifier les calculs, constituant une perte d'information.

Nous avons réessayé notre code sur deux autres images, plus petite et plus grande.

Image de taille 360x360: Taux de compression T=4,97



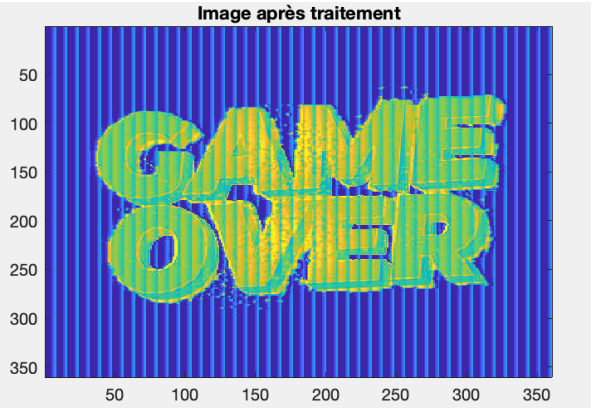
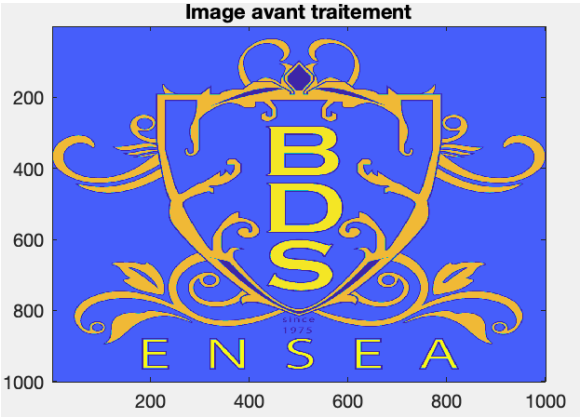
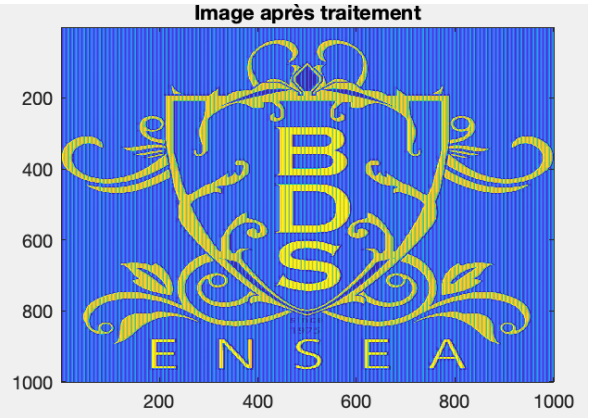


Image de taille 1000x1000: Taux de compression T=6,83





On remarque que plus la taille de l'image est grande, plus le taux de compression augmente. Cependant, le temps d'exécution de l'image augmente également (la complexité est exponentielle).