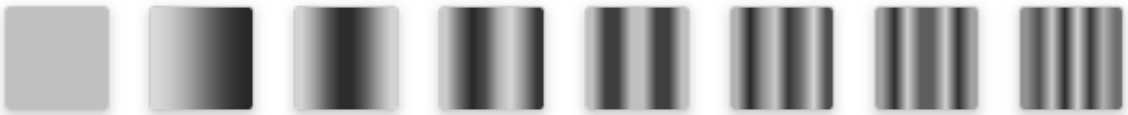
جامعة محمد الخامس بالرباط
Université Mohammed V de Rabat

المدرسة العليا لأساتذة التعليم التقني الرباط
Ecole Normale Supérieure de l'Enseignement Technique - Rabat

Rapport du TP « Compression JPEG »

COMPRESSION JPEG



Réalisé par :

YAICH Hamza

Encadré par :

Pr. JILBAB Abdelilah

Table de matières

Objectif :

L’objectif du TP est la réalisation des étapes du codeur JPEG sous Matlab. On demande de programmer les différents blocs pour réaliser une compression de l’image.

Introduction :

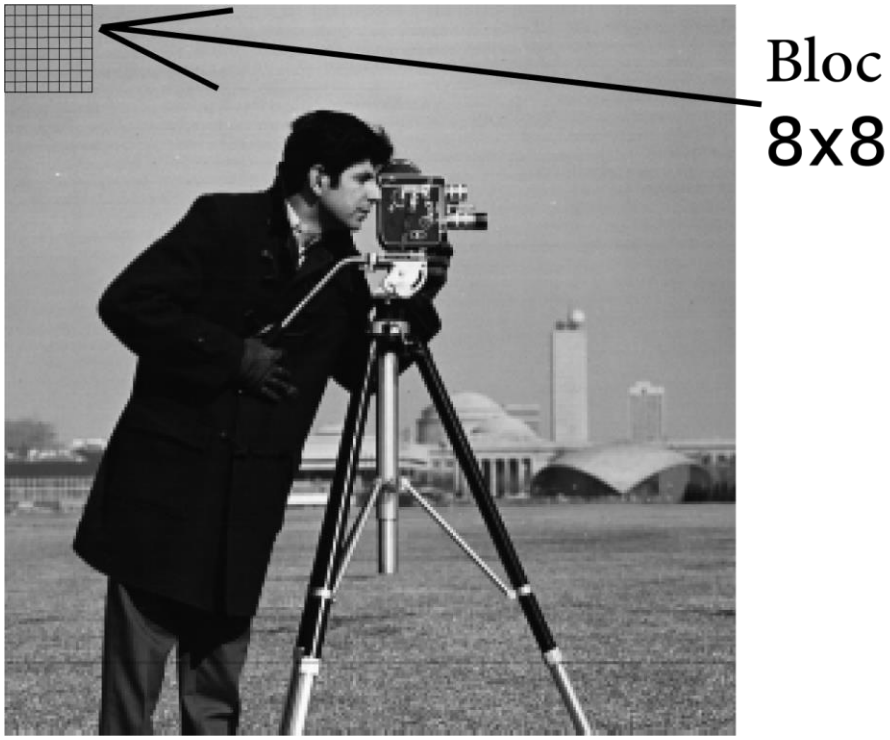
À mesure que notre utilisation et notre dépendance à l'égard des ordinateurs continuent de croître, notre besoin de moyens efficaces de stocker de grandes quantités de données augmente également. Par exemple, une personne disposant d'une page Web ou d'un catalogue en ligne - qui utilise des dizaines, voire des centaines d'images - devra probablement utiliser une forme de compression d'image pour stocker ces images. Ceci est dû au fait que la quantité d'espace nécessaire pour contenir des images non altérées peut être prohibitive en termes de coût. Heureusement, il existe aujourd'hui plusieurs méthodes de compression d'image. Ceux-ci se répartissent en deux catégories générales: la compression d'image sans perte et avec perte. Le processus JPEG est une forme largement utilisée de compression d'image avec perte qui se concentre autour de la transformation cosinus discrète. Le DCT fonctionne en séparant les images en parties de fréquences différentes. Au cours d'une étape appelée quantification, où une partie de la compression se produit réellement, ***les fréquences les moins importantes sont rejetées (les hautes fréquences)***, d'où l'utilisation du terme «avec perte». Ensuite, seules les fréquences les plus importantes qui restent sont utilisées pour récupérer l'image dans le processus de décompression. En conséquence, les images reconstruites contiennent une certaine distorsion; mais comme nous le verrons dans ce TP, ces niveaux de distorsion peuvent être ajustés pendant la phase de compression. La méthode JPEG est utilisée à la fois pour les images en couleur et en noir et blanc, mais dans ce TP, on se concentrera sur la compression exclusivement sur les images en noir et blanc.

Le processus:

Ce qui suit est un aperçu général du processus JPEG. Plus tard, nous présenterons au lecteur un fichier. visite détaillée de la méthode JPEG afin que a. une compréhension plus complète du processus peut être acquise.

1. L'image est divisée en **blocs** de **8x8** pixels (voir l’image ci-dessous).
2. En travaillant de gauche à droite, de haut en bas, le **DCT** est appliqué à chaque bloc.
3. Chaque bloc est ***compressé*** par quantification.
4. Le tableau de blocs compressés qui constituent l'image est stockée dans une quantité d'espace *considérablement réduite*.
5. Lorsque vous le souhaitez, l'image est reconstruite par décompression, un processus qui utilise *la transformation en cosinus discrète inverse* (**IDCT**).

→ La ***quantification*** est un processus dans lequel nous prenons quelques valeurs dans une plage spécifique et les transformons en une valeur **discrète**. Dans notre cas, ***il s'agit simplement d'un nom sophistiqué pour convertir les coefficients de fréquence les plus élevés de la matrice de sortie DCT en 0***.
Lorsque vous enregistrez une image en JPEG, la plupart des programmes d'édition d'image vous demandent de combien de compression vous avez besoin. Le pourcentage que vous y fournissez affecte la quantité de quantification appliquée et la quantité d'informations de fréquence plus élevée perdues. C'est là que la compression avec perte est appliquée. ***Une fois que vous perdez des informations hautes fréquence, vous ne pouvez pas recréer l'image d'origine exacte à partir de l'image JPEG résultante. LOSSY COMPRESSION !***



Principale étape de compression :

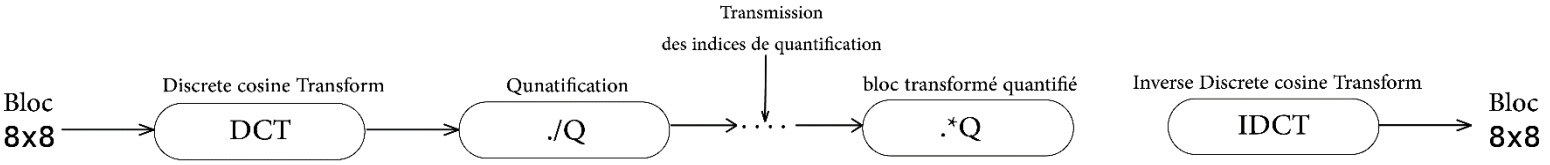


Figure 1

1. DCT et quantification :

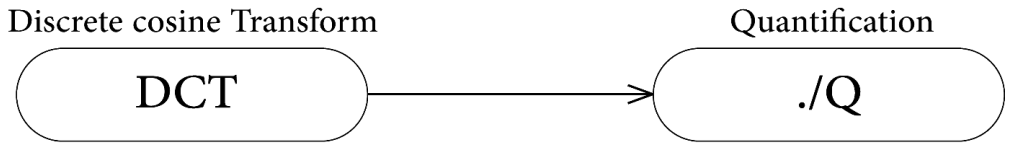


Figure 2

coder.m | decoder.m | compression_f.m

```
1
2   function [ImgTr] = coder(Img,L,Q)
3 -   [ligne,colone]=size(Img);
4
5 -   for i=1:L:ligne
6 -       for j=1:L:colone
7 -           ImgTr(i:i+L-1,j:j+L-1) = dct2(Img(i:i+L-1,j:j+L-1));  % DCT Matrix before quantization  lossless !
8 -           ImgTr(i:i+L-1,j:j+L-1)=ImgTr(i:i+L-1,j:j+L-1)./Q;   % quantize   lossy !
9 -       end
10 -  end
11 -  ImgTr = int16(ImgTr); %Arrondissement
12
```

Figure 3 : fonction « coder.m »

ImgTr

256x256 int16

	1	2	3	4	5	6	7	8	9	10	11
1	78	0	0	0	0	0	0	0	79	-1	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0
9	78	-1	0	0	0	0	0	0	80	0	0
10	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0

Figure 4 : ImgTr (Premier bloc 8x8)

2. Bloc transformé quantifié et DCT inverse :

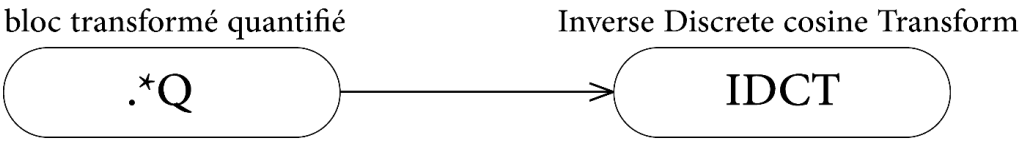


Figure 5

coder.m | decoder.m | compression_f.m

```
1
2   function [ImgRec] = decoder(ImgTr,L,Q)
3   [ligne,colone]=size(ImgTr);
4   ImgTr = double(ImgTr);
5
6   for i=1:L:ligne
7       for j=1:L:colone
8           ImgTr(i:i+L-1,j:j+L-1)=ImgTr(i:i+L-1,j:j+L-1).*Q;    % unquantize
9           ImgRec(i:i+L-1,j:j+L-1) = idct2(ImgTr(i:i+L-1,j:j+L-1)); % inverse DCT
10      end
11  end
12  ImgRec = uint8(ImgRec);
```

Figure 6 : fonction «decoder.m »

ImgRec

256x256 uint8

	1	2	3	4	5	6	7	8	9	10
1	156	156	156	156	156	156	156	156	156	156
2	156	156	156	156	156	156	156	156	156	156
3	156	156	156	156	156	156	156	156	156	156
4	156	156	156	156	156	156	156	156	156	156
5	156	156	156	156	156	156	156	156	156	156
6	156	156	156	156	156	156	156	156	156	156
7	156	156	156	156	156	156	156	156	156	156
8	156	156	156	156	156	156	156	156	156	156
9	154	154	155	156	156	157	158	158	160	160
10	154	154	155	156	156	157	158	158	160	160
11	154	154	155	156	156	157	158	158	160	160
12	154	154	155	156	156	157	158	158	160	160
13	154	154	155	156	156	157	158	158	160	160
14	154	154	155	156	156	157	158	158	160	160
15	154	154	155	156	156	157	158	158	160	160

Figure 7 : ImgRec (Premier bloc 8x8)

3. Ecart quadratique moyen entre l'image originale et l'image reconstituée :

➢ *Code pour modifier la qualité de compression :*

```
quality = 10;
Q = Q * (100 - quality) / 50;    % Q qualité 10%
```

➢ *Matrice de quantification standard : Qualité 50%*

L'erreur quadratique moyenne

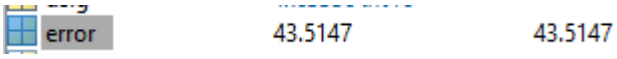


Figure 8 : Différence (Image original, image JPEG_50)

➢ *Matrice de quantification : Qualité 90%*

L'erreur quadratique moyenne

error	6.6544	6.6544

Figure 9 : Différence (Image original, image JPEG_90)

➢ *Matrice de quantification : Qualité 10%*

L'erreur quadratique moyenne

error	70.1755	70.1755

Figure 10 : Différence (Image original, image JPEG_10)

Commentaire :

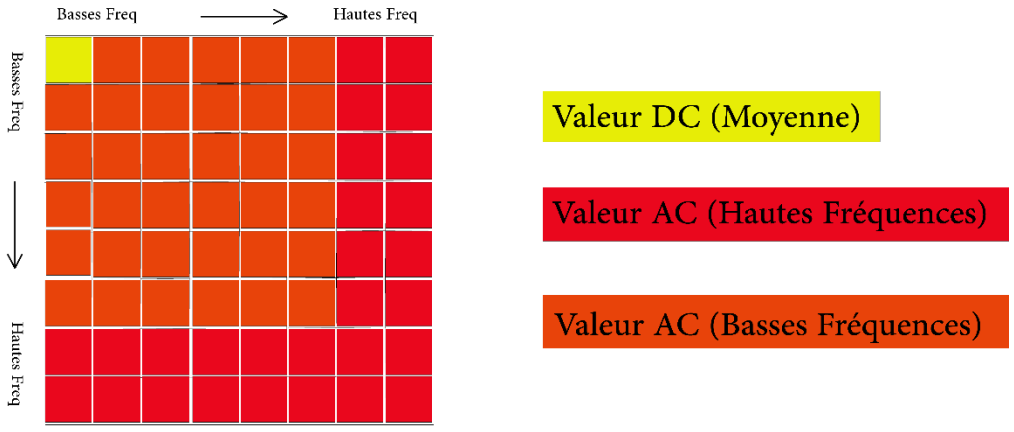
Les figures ci-dessus montrent Matrice des erreurs réalisées par les pertes.

On constate qu'en modifiant le niveau de compression (de 90% jusqu'à 10%), l'image reconstruite est plus dégradé, ainsi que l'erreur quadratique moyenne entre l'image originale et l'image « JPEG» reconstruite est plus en plus importante autant qu'on diminue la qualité.

Ce qui se manifeste dans les trois figures, il y'a de nombreux pixels blancs dans la *Figure 10* car la compression a beaucoup atténué la qualité de l'image original

Remarque :

Même si les humains ne peuvent pas voir les informations à haute fréquence, si vous ***supprimez trop d'informations des blocs d'image 8x8***, l'image globale semblera pixélisée.



4. Teste avec 3 types de matrice de quantification *(Standard, Uniforme, Dégradé)*

- *Matrice de quantification standard :*

```
Q =

    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 uniforme :*

```
function q = q_avg(Q)

Q = [16 11 10 16 24 40 51 61;  % Q standard : qualité 50%
     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];

[ligne,colone]=size(Q);
```

```
avg1 = (Q(1,2) + Q(2,1))/2;
avg2 = (Q(1,3) + Q(2,2) + Q(3,1))/3;
avg3 = (Q(1,4) + Q(2,3) + Q(3,2) + Q(4,1))/4;
avg4 = (Q(1,5) + Q(2,4) + Q(3,3) + Q(4,2)+ Q(5,1))/5;
avg5 = (Q(1,6)+ Q(2,5) + Q(3,4) + Q(4,3) + Q(5,2)+ Q(6,1))/6;
avg6 = (Q(1,7)+ Q(2,6) + Q(3,5) + Q(4,4) + Q(5,3)+ Q(6,2) + Q(7,1)) / 7;
avg7 = (Q(1,8)+ Q(2,7) + Q(3,6) + Q(4,5) + Q(5,4)+ Q(6,3)+ Q(7,2)+
Q(8,1))/8;
avg8 = (Q(2,8)+ Q(3,7) + Q(4,6) + Q(5,5) + Q(6,4)+ Q(7,3) + Q(8,2)) / 7;
avg9 = (Q(3,8)+ Q(4,7) + Q(5,6) + Q(6,5) + Q(7,4)+ Q(8,3))/6;
avg10 = (Q(4,8) + Q(4,7) + Q(5,5) + Q(7,4)+ Q(8,4))/5;
avg11 = (Q(5,8) + Q(6,4) + Q(4,6) + Q(8,5))/4;
avg12 = (Q(6,8) + Q(7,7) + Q(8,6))/3;
avg13 = (Q(7,8) + Q(8,7))/2;

q =  (avg2-avg1) + (avg3-avg2) + (avg4-avg3) + (avg5-avg4) + (avg6-avg5) +
(avg7-avg6) + (avg8-avg7) + (avg9-avg8)+ (avg10-avg9)+ (avg11-avg10)+
(avg12-avg11)+ (avg13-avg12);


q = round(q/12);
```

En appelant cette fonction dans le script compression.m

```
q = q_avg(Q);                          % q le pas moyenne
Q= q*ones(L,L);                  % Q moyenne
```

le résultat est :

q 8 8

Q =

8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8

Remarque :

Les Valeurs AC et la valeur DC sont atténuées de la même facteur.

- *Matrice de quantification dégradée : versions dégradée avec un pas de quantification multipliés par un facteur 2p, à essayer pour p = 1*

Q =

1	2	4	8	16	64	128	256
2	4	8	16	64	128	256	512
4	8	16	64	128	256	512	1024
8	16	64	128	256	512	1024	2048
16	64	128	256	512	1024	2048	4096
64	128	256	512	1024	2048	4096	8192
128	256	512	1024	2048	4096	8192	16384
256	512	1024	2048	4096	8192	16384	32768

On peut voir la détérioration de la qualité d'image dans ce cas !

Car la matrice dégradé attenue agressivement les hautes fréquences

Les Valeurs AC sont atténuées et la valeur DC est préservée.

- *Matrice de quantification dégradée : versions dégradée avec un pas de quantification multipliés par un facteur 2p, à essayer pour p = 2*

Q =

1	4	16	64	256	1024	4096	16384
4	16	64	256	1024	4096	16384	65536
16	64	256	1024	4096	16384	65536	262144
64	256	1024	4096	16384	65536	262144	1048576
256	1024	4096	16384	65536	262144	1048576	4194304
1024	4096	16384	65536	262144	1048576	4194304	16777216
4096	16384	65536	262144	1048576	4194304	16777216	67108864
16384	65536	262144	1048576	4194304	16777216	67108864	268435456

On peut voir encore mieux la détérioration de la qualité d’image dans ce cas là où p = 2 !

Les Valeurs AC sont beaucoup atténuées et la valeur DC est préservée.

Dans ce cas, on a supprimé trop d'informations des blocs d'image 8x8, même si les humains ne peuvent pas voir les informations à haute fréquence, l'image globale semble pixélisée

5. Histogramme des indices de quantification

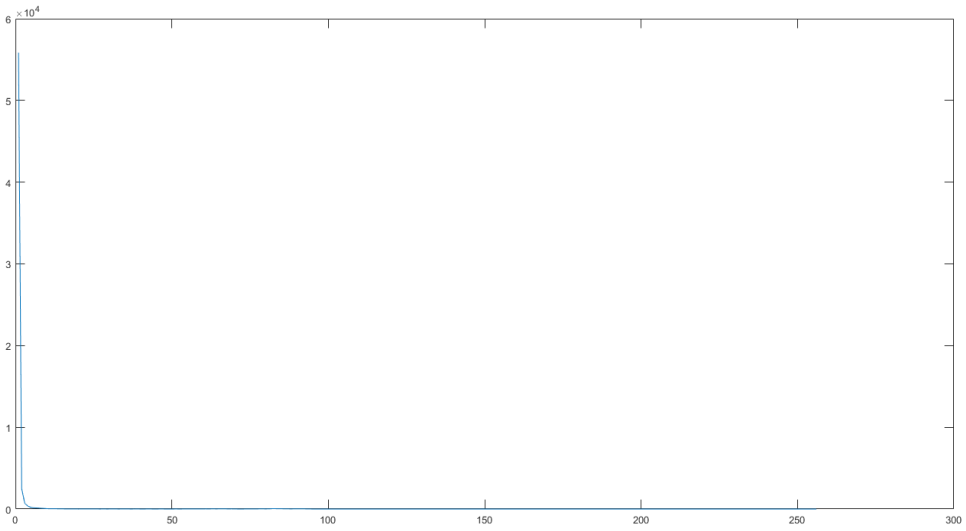


Figure 11 : Histogramme de l'image quantifiée

Interprétation :

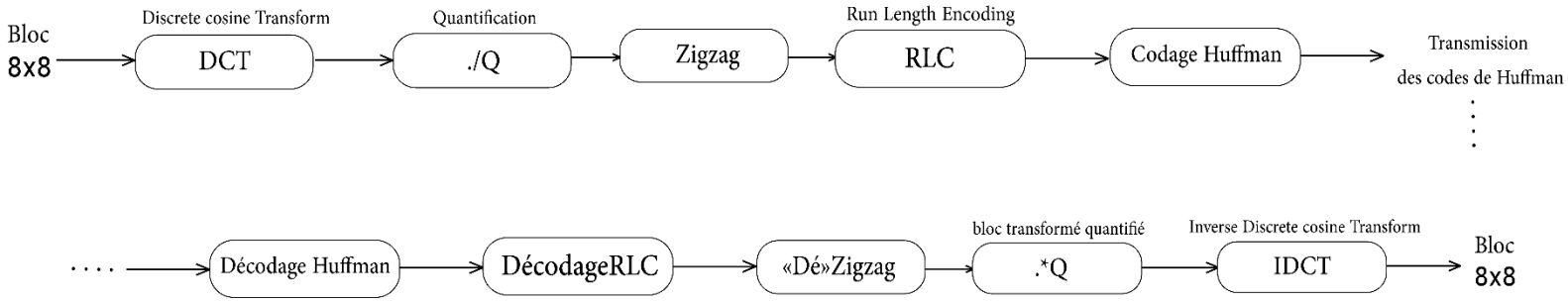
- Après la quantification, on aperçut que l'image quantifié contient beaucoup de **Zéros,** comme le montre l'histogramme dans la *Figure 11 ;*
- On sait que pour chaque pixel, il faut un octet (8bits) pour le représenter, sans faisons appel à une technique de codage sans pertes, donc il faut **256 * 256 * 8 bits**, c'est-à-dire qu'il faut transmettre 524.288 bits.

  Mais ! On a beaucoup de redondance de 0, pourquoi pas l'exploiter pour minimiser la taille du fichier transmis.

- Ce résultat est ensuite compressé selon un algorithme RLC basé sur la valeur 0 (le codage RLC intervient uniquement sur cette dernière), puis un codage entropique de type Huffman.

Et cela ce qu'on voir dans la section suivante.

Les autres étapes de la compression JPEG:



1. Lecture en zigzag et Run Length Coding :

a. Fonctionnement de la fonction Zigzag :

Tout d'abord, le balayage en zig-zag n'est pas une propriété de DCT lui-même mais plutôt une partie du processus de codage d'image basé sur la transformation.

Le balayage en zig-zag est utilisé après la quantification des coefficients DCT par ***NxN*** bloc d'image, et avant le codage de la catégorie / longueur de série des coefficients quantifiés.

Les images naturelles typiques de ***NxN*** (N = 8 blocs les plus typiques) sont *passe-bas qui sont reflétées dans leurs coefficients DCT comme emballant la plupart de l'énergie du signal dans les coefficients d'indice inférieurs et en les distribuant en outre de manière circulaire autour de l'origine.*

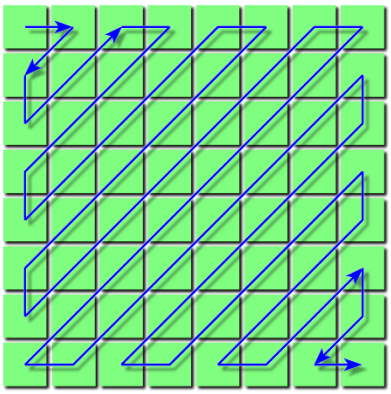
Le balayage en zig-zag de ces coefficients commence donc à partir de coefficients d'ordre inférieur et se déplace à travers un motif circulaire de l'intérieur vers l'extérieur.

La séquence 1D résultante, après un certain nombre de coefficients initiaux non nuls (dont le nombre est basé sur le réglage de qualité), la plupart des coefficients quantifiés et **scannés ZZ** restants seront nuls ne nécessitant aucun bit à coder; d'où la compression.

Le balayage en zig-zag ordonne donc les coefficients DCT de manière efficace pour que cette phase de codage RLC (de type {longueur plage nulle, valeur}) tire parti de leur structure ...

La matrice quantifiée C'est maintenant prête pour l'étape finale de compression. Avant le stockage, tous les coefficients de C sont convertis par un encodeur en un flux de données binaires (01101011). Après quantification, il est assez courant que la plupart des coefficients soient égaux à zéro. JPEG tire parti de cela en codant des coefficients quantifiés dans la séquence en zig-zag illustrée à la figure 1. L'avantage du codage de la séquence quantifiée se comprime très bien. La séquence du Bloc (8x8) se poursuit pour toute l'image en faisant les mêmes étapes.

L'avantage réside dans la consolidation de séries de zéros relativement importantes.



b. Code Matlab de la fonction Zigzag :

```
R = double(ImgTr(zigzag(256)));                %Zigzag
```

```
>> size(R)

ans =

           1       65536
```

La séquence 1D résultante « R »

c. codage RLC :

(Run Length Coding). Le principe employé pour ce codage est très simple: toute suite d'octets de même valeur est remplacée par la valeur, à laquelle on associe le nombre d'occurrences suivantes.

d. Code Matlab de RLC :

```
function [d,c]=rlc_enc(x); %x est le vecteur sortie de la fonction
zigzag

ind=1;
d(ind)=x(1);
c(ind)=1;

for i=2 :length(x)
    if x(i-1)==x(i)
       c(ind)=c(ind)+1;
    else ind=ind+1;
         d(ind)=x(i);
         c(ind)=1;
    end
end
```

2. Codage sans pertes – Codage de Huffman:

a. Fonctionnement de la fonction Huffman :

Après la quantification, le codage Huffman est l'un des contributeurs ***les plus significatifs aux économies de taille de fichier dans la compression JPEG.***

Il s'agit d'une méthode de codage purement statistique consistant à coder les lettres suivant leur probabilité d'apparition. À la lettre la plus fréquente est associé le mot de code le plus court; le mot de code le plus long s'utilise pour la lettre la moins probable.

b. Introduire cette fonction dans le programme de codage complet :

```
%%
%%%%%%%% Huffman dictionbary %%%%%%%%%
temp=1;
temp2=0;
[row, col] = size(ImgTr);
pixel_count = row*col;
symbol = reshape(ImgTr,[1,row*col]);
symboll = unique(symbol);
len = length(symboll);
for i = 1:len
    k = ImgTr == symboll(i);
    count(temp) = sum(k(:));
    prob_pix(temp) = double(count(temp)/pixel_count);
    temp2 = temp2 + prob_pix(temp);
    prob_cum(temp) = temp2;
    temp = temp+1;
end
[dict,avg_len] = huffmandict(symboll,prob_pix);
%%
comp = huffmanenco(R,dict);   %Huffman
dsig = huffmandeco(comp,dict);   %Decodage Huffman
```

c. Expériences sur les images :

Code complet et les fonctions implémentées :

```
clc
clear all
close all
I = imread('Cameraman.bmp');

figure(1), imshow(I, [0 255]);
L = 8 ;
Q = [16 11 10 16 24 40 51 61;  % Q standard : qualité 50%
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];
%%
[ImgTr] = coder(I,L,Q);                    %DCT et Quantization
figure(2), imshow(ImgTr, [0 255]);
%%
%%%%%%%%%%%%%%%% Les autres étapes de la compression JPEG %%%%%%%%%%%%%
R = double(ImgTr(zigzag(256)));                %Zigzag
[d,c] = rlc_enc(R);                            %RLC
%%
%%%%%%%% Huffman dictionbary %%%%%%%%%
temp=1;
temp2=0;
[row, col] = size(ImgTr);
pixel_count = row*col;
symbol = reshape(ImgTr,[1,row*col]);
symbol1 = unique(symbol);
len = length(symbol1);
for i = 1:len
    k = ImgTr == symbol1(i);
    count(temp) = sum(k(:));
    prob_pix(temp) = double(count(temp)/pixel_count);
    temp2 = temp2 + prob_pix(temp);
    prob_cum(temp) = temp2;
    temp = temp+1;
end
[dict,avg_len] = huffmandict(symbol1,prob_pix);
%%
comp = huffmanenco(R,dict);   %Codage Huffman
dsig = huffmandeco(comp,dict);   %Decodage Huffman
%%
x = rlc_dec(d,c);                                    %Decodage RLC
%%
out=Dezigzag(x,length(ImgTr),length(ImgTr));         %de_Zigzag
%%
[ImgRec] = decoder(out,L,Q);                          %IDCT et
UnQuantization
%%
figure(3), imshow(ImgRec, [0 255]);      %affichage de l'image compressée
```

```
[D] = Distimage(I,ImgRec);                  %difference (image original -
image compressé)
figure(4), imshow(D, [0 255]);
[H] = histogramme(ImgTr,8);
figure(5), plot(H);
%%
error = immse(I,ImgRec);                    %MSE ( mean squarred error ==
erreur moyenne quadratique )
```

[dict,avglen] = huffmandict(symbols,prob) generates a binary Huffman code dictionary, dict, for the source symbols, symbols, by using the maximum variance algorithm. The input prob specifies the probability of occurrence for each of the input symbols. The length of prob must equal the length of symbols. The function also returns average codeword length avglen of the dictionary, weighted according to the probabilities in the input prob.

Conclusion :

C'est beaucoup de travail pour voir une simple photo! Mais ce que j'aime à ce sujet, c'est que vous pouvez voir à quel point JPEG est une technologie très centrée sur l'humain. Il s'appuie sur les bizarreries de notre perception pour atteindre des taux de compression bien supérieurs à ce qui est possible avec des techniques à usage général. Et maintenant que vous comprenez comment fonctionne JPEG, vous pouvez imaginer combien de ces techniques peuvent être étendues à d'autres domaines. Par exemple, l'application du codage delta à une vidéo peut entraîner une réduction considérable de la taille du fichier, car il y a souvent des zones qui ne changent pas du tout entre les images (comme l'arrière-plan).