RAULIN Julie

LE STRAT Claire

2G1TP5

Compte rendu de TP de majeure signal S8

TP de compression d'images par ondelettes

I- Introduction

La technique de compression à base d'ondelettes offre une grande finesse au niveau de l'analyse du signal et permet de mieux s'adapter aux propriétés locales de l'image. La compression en ondelettes est en elle-même sans pertes, c'est l'introduction facultative d'une quantification ou d'un seuillage qui entraine la perte irréversible d'informations.

La première transformation par ondelettes est une technique inventée par Alfréd Haar en 1909, avec l'ondelette du même nom. En 1984, Jean Morlet, un ingénieur français, les utilise pour la prospection pétrolière et introduit le terme même d'ondelette qui fut traduit en anglais par wavelet, à partir des termes wave (onde) et le diminutif let (petite). Yves Meyer (prix Abel 2017), rassembla en 1986 toutes les découvertes précédentes puis définit les ondelettes orthogonales. La même année, Stéphane Mallat fit le lien entre les ondelettes et l'analyse multirésolution. Enfin, Ingrid Daubechies mit au point en 1987 des ondelettes orthogonales appelées ondelettes de Daubechies, et utilisées dans le standard JPEG 2000.

L'utilisation de cette transformation en imagerie consiste à décomposer une image en une myriade de sous-bandes, c'est-à-dire des images de résolution inférieure. La transformation en ondelettes provient d'une analyse multirésolution de l'image. On considère des espaces d'approximations et des espaces capturant les détails perdus entre chaque niveau d'approximation. Les bases ondelettes se situent sur les espaces "détails". Le choix de l'ondelette mère est très important et fait toujours l'objet d'expérimentations pour adapter l'analyse du signal image au système visuel humain.

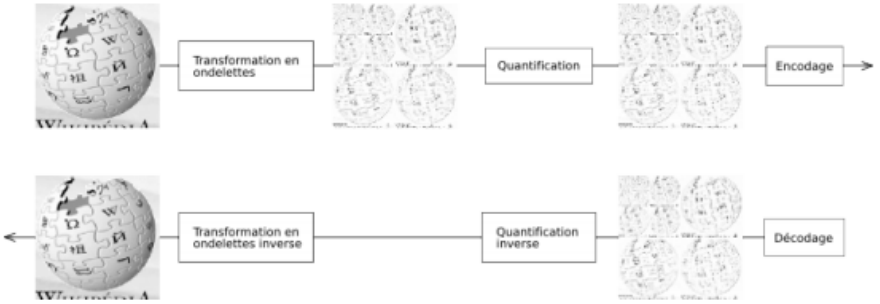


FIGURE 1 – Schéma de compression d'image par ondelettes.

La compression par ondelettes se compose des étapes suivantes (Figure 1) :

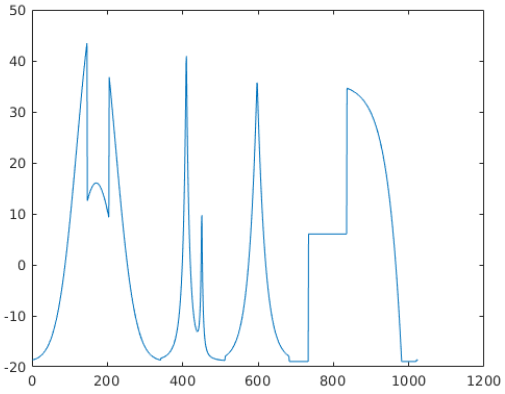
— Transformations par ondelettes.
— Quantification : les valeurs des images de détails inférieures à un certain niveau sont éliminées, en fonction de l'efficacité recherchée. C'est cette étape qui introduit des pertes.
— Codage des valeurs restantes.

II- Décomposition en ondelettes et reconstruction parfaite

Dans cette partie, nous nous intéressons à l'analyse de signaux à l'aide de leurs d'ondelettes.

— On charge le fichier PieceRegSig avec l'instruction load. Celle-ci permet de créer un certain nombre de signaux test (sig). On affiche la courbe correspondante avec plot

```
load("PieceRegSig.mat")
figure(1);
plot(sig)
```



2.1 Première décomposition dans la base de Haar

Objectif : apprendre l'adressage partiel d'un vecteur, et comment sous-échantillonner un signal.

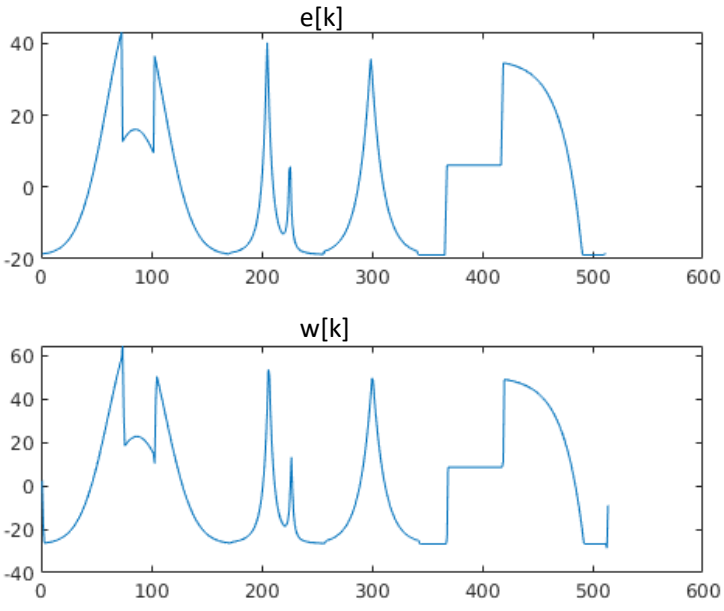
— On crée un vecteur e de longueur moitié contenant les moyennes de deux valeurs consécutives de sig, la première des deux valeurs utilisées ayant toujours un indice impair.

Mathématiquement, cela s'écrit e[k] = (sig[2k − 1] + sig[2k])/2. On utilise pour cela des indexages partiels dans sig. 2

—Puis on crée un vecteur w de longueur moitié contenant les demi variations de deux valeurs consécutives de sig, soit w[k] = (sig[2k] − sig[2k − 1])/2 ;

On trace e et w.

```
n=length(sig);
ind1=[1:2:n]; %impairs
ind2=[2:2:n]; %pairs
e=(sig(ind1)+sig(ind2))/2; %approximation
w=(sig(ind2)-sig(ind1))/2; %détails

figure(2);
subplot(2,1,1);
plot(e)
title('approximation dans la base de Haar')
subplot(2,1,2);
plot(w)
title('detail dans la base de Haar')
```



décomposition dans la base de Haar

On remarque que e diffère peu du signal d'origine, surtout dans ses parties lisses. Par contre, w n'est important qu'au voisinage des sauts. C'est normal, puisqu'on le calcule par différences finies.

2.2 Convolution

La convolution de deux suites $(a_n)_{n\in Z}$ et $(b_n)_{n\in Z}$ est définie par : $(a * b)_n = P_{k\in Z}\, a_k b_{n-k}$

Le script convHaar.m qui nous est fournit compare des implémentations « à la main » de la décomposition sur le base de Haar avec des implémentations utilisant l'opérateur de convolution.

Nous allons implémenter une cascade simple par un banc de filtres de Daubechies.

—Le fichier 'PieceRegSig' a déjà été chargé avec l'instruction 'load' plus haut dans le TP.

— On charge le banc de filtres de Daubechies à deux moments nuls à partir du fichier 'Daub4'.

```
load("Daub4")
```

Quatre variables sont alors chargées dans Matlab : h, g, rh, rg.

Les deux premiers sont les filtres d'analyse (respectivement passe-bas et passe-bande). Les seconds sont les filtres de reconstruction.

— On vérifie bien que g s'obtient à partir de rh en changeant un signe sur deux ; même observation pour rg et h.

Le support du filtre rh de Daubechies-4 est [0,3], h est le miroir du filtre rh, son support est [-3,0] (en effet, le miroir F d'un filtre f est defini par F(n) = f(−n)).

— On implémente le schéma fonctionnel de décomposition dans Figure 2

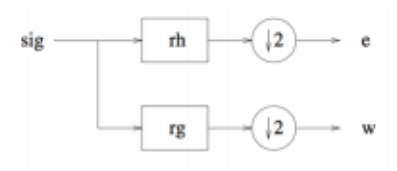


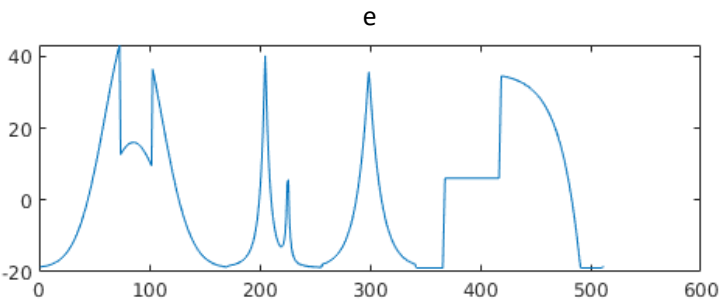
FIGURE 2 – Décomposition en ondelettes.

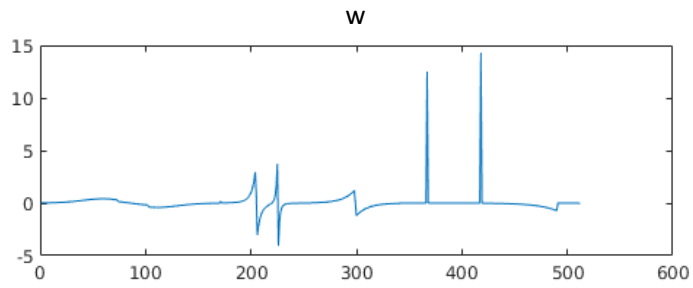
Code d'implémentation :

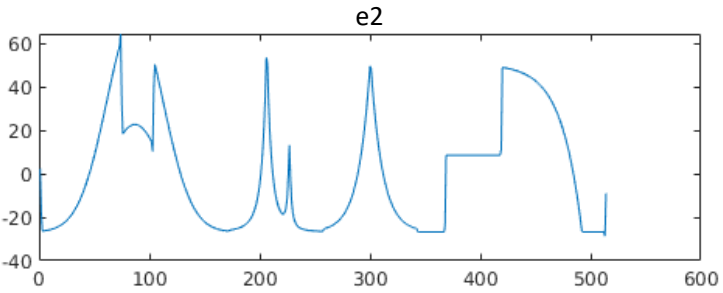
```
ec = conv(sig,[0.5 0.5]); % convolution du signal avec [0.5 0.5]
wc = conv(sig,[0.5 -0.5]);%convolution du signal avec [0.5 -0.5]


econv = conv(rh, sig) ; % convolution entre le signal et rh
e2 = econv(1:2:length(econv));% sous-echantillonnage

wconv=conv(rg, sig); % convolution entre le signal et rg
w2=wconv(1:2:length(wconv));% sous-echantillonnage
```

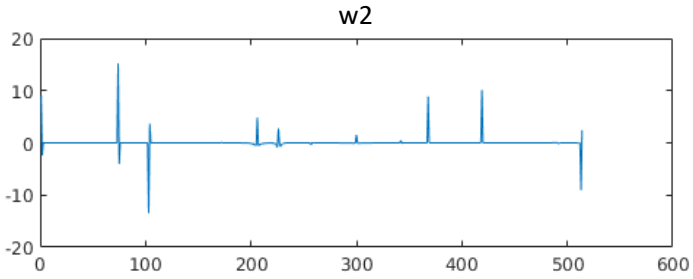
```
figure(3)
subplot(2,1,1);
plot(e)
subplot(2,1,2);
plot(e2)

figure(4)
subplot(2,1,1);
plot(w)
subplot(2,1,2);
plot(w2)
```

Que remarque-t-on aux bords de e et w ? Quelle est la cause de ce phénomène ?









On remarque que les valeurs sont différentes mais l'allure reste la même. Par contre les valeurs sur les bords changent beaucoup plus. Ce phénomène s'explique par le fait qu'on compresse à pertes.

Par exemple **: e2[1]= 2.41** (signal compressé) contre **e[1]= -18.64** (signal initial)

2.3 Reconstruction

Les filtres de Daubechies sont des filtres à reconstruction parfaite. Cela que le diagramme de Figure 3 suivant fonctionne :

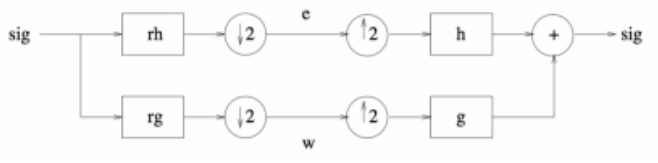


FIGURE 3 – Banc de filtres d'ordre 2.

— la partie gauche du schéma (banc d'analyse) a été implémentée dans ce source lors de l'exercice précédent ; il reste à effectuer la reconstruction (banc de synthèse).

—On implémente le sur-échantillonnage par insertion de zéros, puis la convolution :

La convolution de x (x correspondant à h ou g) avec rh s'écrit :

$y[n] = r_h[0]*x[n] + r_h[1]*x[n-1] + r_h[2]*x[n-2] + r_h[3]*x[n-3]$.

y est nul si n-3 > 1023 ou n < -3

le support de y est donc [-3,1026]. Pour récupérer sig , il faut éliminer trois valeurs à gauche et trois valeurs à droite.

```
e_surech=zeros(1,n-1);
e_surech(1:2:length(econv))=e2;
w_surech=zeros(1,n-1);
w_surech(1:2:length(econv))=w2;
e_surech_conv=conv(e_surech,h);
w_surech_conv=conv(w_surech,g);
sig2=e_surech_conv + w_surech_conv;
```

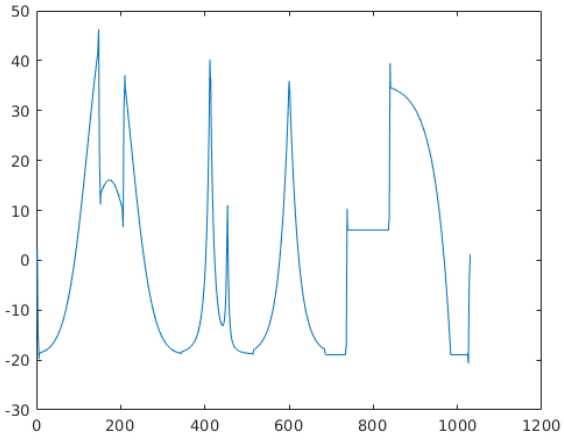
e_surech_conv correspond à la convolution entre le signal e2 suréchantillonné et le filtre g

w_surech_conv correspond à la convolution entre le signal w2 suréchantillonné et le filtre g

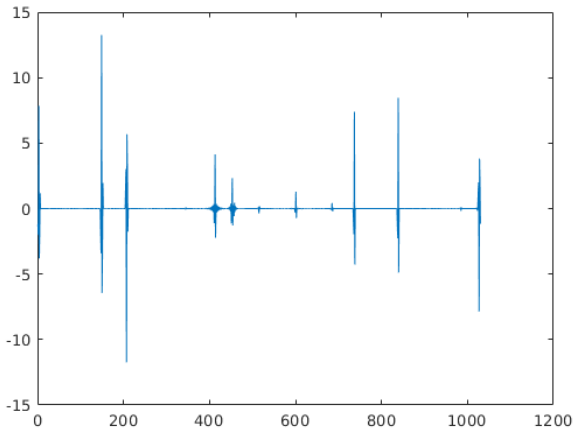
— On trace les signaux e_surech_conv et w_surech_conv la différence entre la reconstruction et le signal d'origine.

```
figure(5)

plot(e_surech_conv)
figure(6)
plot(w_surech_conv)
figure(7)
plot(sig_final-sig)
```

e_surech_conv

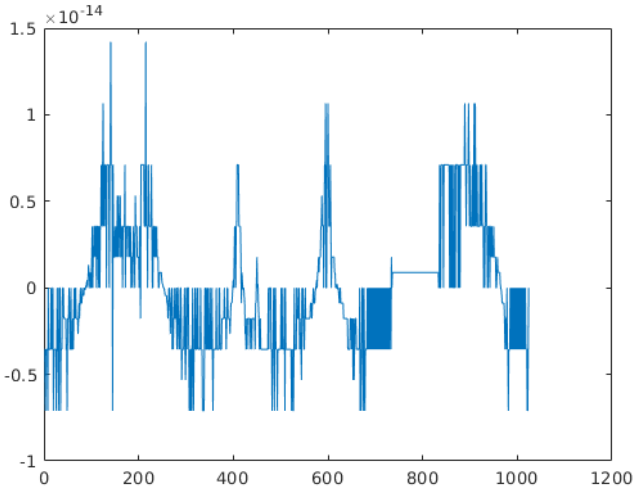


w_surech_conv



différence entre la reconstruction et le signal d'origine

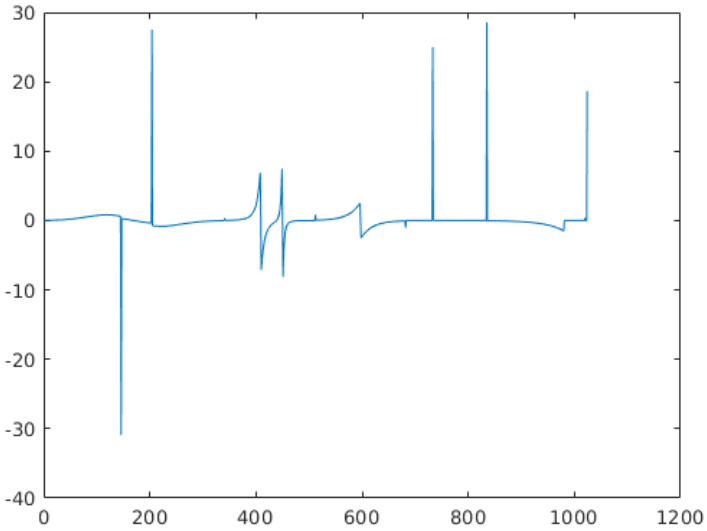
sig_final - sig



La reconstruction numérique est quasiment parfaite. L'erreur est de l'ordre de 10^-14 donc le signal de sortie correspond bien au signal de départ.

Pour vérifier l'importance de la troncature lors de la reconstruction, qui peut s'interpréter comme une synchronisation avec le signal de départ, nous refaisons la troncature en la décalant de 1

```
sig_final2=sig2(5:length(sig2)-2);

figure(8)
plot(sig_final2-sig)
```



La reconstruction numérique est mauvaise. L'erreur est grande donc le signal de sortie ne correspond pas au signal de départ.

En effet, un décalage implique que les 2 signaux ne sont plus synchronisés d'où l'importance de la troncature lors de la reconstruction.

III- Compression des signaux 1D

Afin de faciliter le reste du TP, plusieurs fonctions et fichiers sont fournies :

- `GetFiltres.m` : renvoyer les jeux de filtres associés aux différentes ondelettes utilisées
- `DownFilter.m` : réaliser la décompostion
- `UpFilter.m` : réaliser la reconstruction
- `SignauxTypiques.m` : générer des signaux de types différents.

Les ondelettes utilisées pour la compression sont des ondelettes biorthogonales, qui correspondent à des filtres de longueurs impaires. La norme JPEG2000 précise deux types :

- les ondelettes 5/3 : $rh[n]$ et $g[n]$ de longueur 5, $rg[n]$ et $h[n]$ de longueur 3.
- les ondelettes 9/7 : $rh[n]$ et $g[n]$ de longueur 9, $rg[n]$ et $h[n]$ de longueur 7.

Les ondelettes orthogonales (telles que les ondelettes de Haar ou de Daubechies dans les parties précédentes) sont associées à des filtres de même longueur, obligatoire paire. Ces ondelettes sont moins bien adaptées à la compression.

Remarque : les filtres orthogonaux étant toujours de longueurs paire, le fichier `GetFiltres.m` leur ajoute un zéro en fin de réponse pour les rendre compatibles avec les fichiers `DownFilter.m` et `UpFilter.m`. Pour réaliser le banc de filtres, vous serez confronté(e)s au problème du retard introduit par les filtre, qui doit être compensé pour effectuer la reconstruction. Un paramètre (nommé type) est prévu pour cela dans les fonctions `DownFilter` et `UpFilter`. Le script `banc_ex.m` explique comment renseigner ce paramètre ('a_o' : 'approximation + orthogonal', 'd_o' : detail + orthogonal', 'a_b' : 'approximation + biorthogonal', 'd_b' : 'detail + biorthogonal').

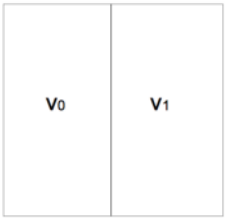
Complétons le script banc_ex.m pour réaliser le banc de filtres (Figure 3) :

— Utiliser de différents signaux et vérifier la reconstruction parfaite, pour les ondelettes Haar et 5/3 (tracer les "approximation" et "detail") — Effectuer ensuite la reconstruction après une compression brutale obtenue par suppression de la composante w. — Comparer les amplitudes des erreurs de reconstruction pour les ondelettes 5/3 et 9/7, pour le signal 'ligneLena'. Conseil : pour de différentes ondelettes, verifier la capacité à concentrer l'énergie des signaux sur peu de coefficients, la complexité de l'implémentation.

IV- Compression d'une image

Une image est un tableau de taille $[M, N]$ avec M le nombre de lignes de l'image et N le nombre de colonnes. Dans ce TP, on suppose que M et N sont pairs. Nous allons travailler avec les images en niveau de gris où chaque pixel est codé sur un octet (valeurs comprises entre 0 et 255). La décompression d'une image se fait en appliquant une décomposition mono-dimensionnelle sur chacune des ses lignes, suivi d'une décomposition sur chacune de ses colonnes. La recomposition se fait de la même façon. L'ordre des opérations (lignes puis colonnes ou colonnes puis lignes) est indifférent.

La décomposition d'une ligne de N points donne deux signaux $v_0[n]$ et $v_1[n]$ de $N/2$ points chacun, que l'on concatène dans cet ordre, pour retrouver un signal de N points qui peut remplacer la ligne d'origine. On obtient alors une image composée de deux parties :



On peut effectuer la même chose sur les colonnes, ce qui donne une image composée de quatre parties :

Si M et N sont multiples de quatre, la décomposition dyadique peut être effectuée sur deux niveaux, en décomposant la composante v_{00}.

—On écrit un script qui effectue la décomposition et la reconstruction parfaite d'une image, sur un niveau.

On travaille sur l'image lena dont on extrait une matrice image grâce à imread.

```
image=imread('lena.png');
```

1) Décomposition de l'image

D'abord, on crée **l'image de détail et d'approximation de l'image lena.**

```
[M,N]=size(image);
[rh, rg, h, g] = GetFiltres('9/7');
```

```
photo=zeros(M,N);


for (i=1:N)
    ligne=image(i,:);
    approxh=DownFilter(ligne,rh,'a_h');
    detail_vertical=DownFilter(ligne,rg,'d_b');
    photo(i,1:N/2)=approxh;
    photo(i,((N/2)+1):N)=detail_vertical;

end
```

```
colormap('gray')
figure(8)
imagesc(photo)
title("détail et approximation de l'image lena")
```

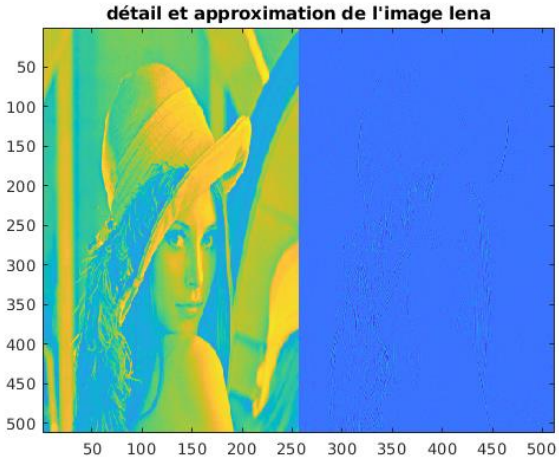
Pour cela on récupère les filtres rh, rg, h et g à l'aide la commande GetFiltres.

Pour chaque ligne de l'image, on applique le filtre DownFilter :

- avec rh pour obtenir l'approximation
- avec rg pour obtenir le détail.

Et on stocke la nouvelle ligne dans la matrice photo

Avec imagesc on affiche photo. Attention, il faut utiliser imagesc et non imshow pour afficher l'image correctement. On observe alors bien l'approximation à gauche et les détails à droite.



Ensuite, on applique détail et approximation de l'image précédente pour obtenir la **décomposition finale.**

```
photo1=transpose(photo);
for (i=1:N)
    ligne=photo1(i,:);
    approxv=DownFilter(ligne,rh,'a_h');
    detail_horizontal=DownFilter(ligne,rg,'d_b');
    photo1(i,1:N/2)=approxv;
    photo1(i,((N/2)+1):N)=detail_horizontal;
end
```

Pour cela on récupère photo et on prend sa transposée photo1.

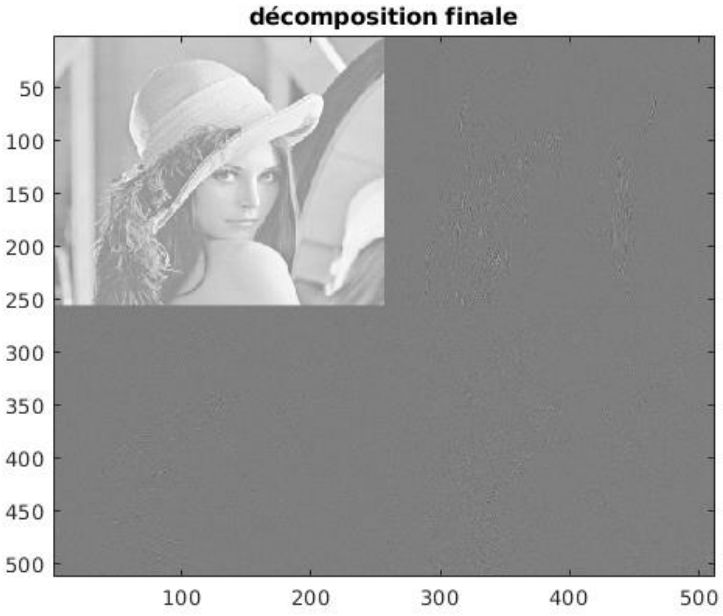
Pour chaque ligne de photo1, on applique le filtre DownFilter :

- avec rh pour obtenir l'approximation
- avec rg pour obtenir le détail

Et on stocke la nouvelle ligne dans la matrice photo1

Avec imagesc on affiche photo1 et on obtient bien le résultat escompté.

```
figure(9)
photo_decompose = transpose(photo1);
imagesc(photo_decompose)
title("décomposition finale")
```



2) Reconstitution de l'image

On crée une fonction Reconstitution qui prend en argument une matrice photo_decompose correspondant à l'image lena décomposée comme ci_dessus et qui renvoie une matrice photo_reconstitue_lena correspondant à l'image lena recomposée.

```
function photo_reconstitue_lena = Reconstitution(photo_decompose)

[N,N]=size(photo_decompose);
[rh, rg, h, g] = GetFiltres('9/7');
partie_1 = transpose(photo_decompose(1:N/2,:));
partie_2 = transpose(photo_decompose((N/2)+1:N,:));
f1=zeros(N,N);
f2=zeros(N,N);
photo_reconstitue=zeros(N,N);

for (i=1:N)
    ligne=partie_1(i,:);
    f1(i,1:N)=UpFilter(ligne,h,'a_h');
end

filter1=transpose(f1);


for (i=1:N)
    ligne=partie_2(i,:);
    f2(i,1:N)=UpFilter(ligne,g,'d_b');
end
```

```
filter2 = transpose(f2);

photo_reconstitue=filter1+filter2;
```

Pour cela on récupère les filtres rh, rg, h et g à l'aide la commande GetFiltres.

On sépare la matrice en argument en 2 parties : partie 1 : approximation et détail de lena (la partie haute de l'image)

partie 2 : approximation et détail du détail de lena. (la partie basse de l'image)

On applique le filtre UpFilter:

- avec h pour la partie 1 => et on stocke les lignes dans une matrice f1
- avec g pour la partie 2 => et on stocke les lignes dans une matrice f2

Ensuite on stocke dans photo_reconstitue la somme de la transposée de f1 et de la transposée de f2.

```
partie_3 = photo_reconstitue(:,1:N/2);
partie_4 = photo_reconstitue(:,(N/2)+1:N);
photo_reconstitue_lena=zeros(N,N);
filter3 = zeros(N,N);
filter4 = zeros(N,N);

for (i=1:N)
    ligne=partie_3(i,:);
    filter3(i,1:N)=UpFilter(ligne,h,'a_h');
end


for (i=1:N)
    ligne=partie_4(i,:);
    filter4(i,1:N)=UpFilter(ligne,g,'d_b');
end

photo_reconstitue_lena=filter3+filter4;
```

Maintenant, on divise photo_recupere en la partie gauche et la partie droite de l'image.

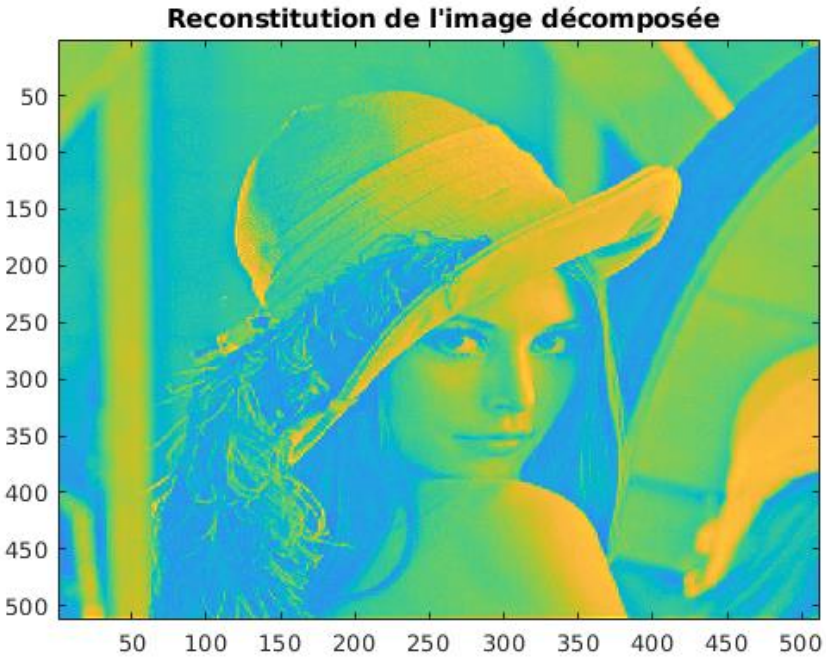
On applique le filtre UpFilter:

- avec h pour la partie 3 => et on stocke les lignes dans une matrice filter3
- avec g pour la partie 4 => et on stocke les lignes dans une matrice filter4

Finalement on obtient l'image reconstituée photo_reconstitue_lena qui est la somme de filter 3 et filter 4.

Maintenant , on retourne dans le main. Et il reste à appliquer la fonction Reconstition à notre photo_decompose. On affiche l'image reconstituée.

```
image_finale= Reconstitution(photo_decompose);
colormap('gray')

figure(10)
imagesc(image_finale)
title(" Reconstitution de l'image décomposée")
```



—On effectue une compression en supprimant la composante v11. On met donc à 0 les photo_decompose[i,j] du quart inférieur droit comme codé ci-dessous. Puis on applique Reconstitution a cette photo ayant v11 en moins = « photo_decompose_abime » , et on l'affiche.

```
photo_decompose_abime=photo_decompose;
for (i=((N/2)+1): N)
    for (j=((N/2)+1): N)
        photo_decompose_abime(i,j)=0;
    end
end
image_abime= Reconstitution(photo_decompose_abime);

figure(11)

colormap('gray')
imagesc(image_abime)
title(" Reconstitution de l'image décomposée sans v11")
```

Calculons le PSNR qui mesure la qualité de reconstruction de l'image compressée par rapport à l'image originale.

PSNR = 10 $\log_{10}$ (2552 * MSE) avec $MSE = \frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}(f_{i,j} - \bar{f}_{i,j})^2$.

```
C=0;
for (i=1:N)
    for (j=1:N)
        C = C + (cast(image(i,j),'double')-image_abime(i,j))^2;
    end
end
MSE=C/(N*N);
PNSR=10*log(255*255/MSE);
```

On obtient alors

```
MSE =

       31.724

>> PNSR

PNSR =

       76.255
```

En comparant l'image abimée avec l'image originale, on obtient un PNSR de 76.255.

Or, un PSNR entre 30 et 50 dB pour la compression d'images est très bien pour la perception humaine. Nous ne sommes pas dans cette intervalle. Cela peut s'expliquer par le fait qu'en pratique, la compression est faite de façon plus élaborée, en quantifiant différemment les différents signaux v00, v10, v01 et v11, afin de rendre les défauts imperceptibles. Ici les défauts sont donc perceptibles, et en effet la reconstitution de l'image décomposée sans v11 est un peu flou (voir plus haut).

Projet : réalisation d'un codeur/décodeur JPEG simplifié

1. Encodage d'une data unit

Le codeur JPEG traite des blocs de taille 8x8. Le première étape est donc compléter l'image initiale pour obtenir une nouvelle image de taille égale à un multiple de 8. Pour cela, on écrit le code suivant :

```
m = mod(M,8);
n = mod(N,8);
while (m~=0)
    ligne = image_depart(M,:);
    image_depart = [image_depart;ligne];
    m = m+1;
end
while (n~=0)
    colonne = image_depart(:,N);
    image_depart = [image_depart,colonne];
    n = n+1;
end
[M2,N2] = size(image_depart); % nouvelle taille de l'image
```

A chaque fois que la taille de l'image n'est pas un multiple de 8, on duplique la dernière ligne jusqu'a obtenir une taille dont la valeur est un multiple de 8.

1.1 Centrage

Pour effectuer un centrage, on soustrait 128 à chaque valeur de pixel :

```
function bloc_centrage = Centrage(bloc)
[N,M]=size(bloc);
bloc_centrage=zeros(N,M);
for (k=1:N)
    for (l=1:M)
        bloc_centrage(k,l)=(cast(bloc(k,l),'double'))-128;
    end
end
end
```

Ainsi, les valeurs sont comprises entre -128 et 127. Le centrage est une première étape de la compression puisqu'au lieu de coder sur 8 bits les valeurs de 0 à 255, nous avons maintenant besoin de 7 bits.

1.2 Transformée en cosinus

Pour réaliser la transformée en cosinus, on utilise la fonction D=dctmtx(8) qui permet de calculer la DCT d'un bloc de 8x8. Ainsi, pour calculer la DCT d'une image de taille 8x8, on utilise : D*size(image_8x8)*D'.

```
D = dctmtx(8);
DCT_bloc_centree = D*bloc_centree*D';
```

1.3 Quantification

La quantification est l'étape qui permet de gagner le plus de place. Elle consiste à diviser point à point la matrice 8x8 par une autre matrice 8x8 appelée matrice de quantification (Q). Le but est d'atténuer les hautes fréquences car l'oeil humain n'y est que très peu sensible. Ainsi, les informations peu essentielles seront remplacées par les zéros et on gagnera de la place lors de l'application du Run Length Code.

Ici, Q_RGB_luminescence=

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

Q_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

On réalise le code suivant sur chacun des blocs de taille 8x8 :

```
bloc_RGB_luminescence = round(DCT_bloc_centree ./Q_RGB_luminescence);
bloc_chrominance = round(DCT_bloc_centree ./Q_chrominance);
```

On créer également une fonction Quantification qui prend en argument une matrice 8x8 ainsi qu'un facteur de qualité qui varie entre 1 et 100 :

```
function bloc_quantifie = Quantification(DCT_bloc_centree,fq)
s=0;
if fq<50
    s=5000/fq;
else
    s=200-2*fq;
end
bloc_quantifie=round((s*DCT_bloc_centree+50)/100);
```

1.4 Parcours en zigzag, RLC, codage de Huffman

```
function vrlc = RunLength(bloc_quantifie)
vrlc=[];

% zigzag
col = [1 2 1 1 2 3 4 3 2 1 1 2 3 4 5 6  5 4 3 2 1 1 2 3 4 5 6 7 8 7 6 5 4 3 2 1 2 3 4 5 6 7 8 8 7 6 5 4 3 4 5 6 7 8 8 7 6 5 6 7 8 8 7 8];
lig = [1 1 2 3 2 1 1 2 3 4 5 4 3 2 1 1 2 3 4 5 6 7 6 5 4 3 2 1 1 2 3 4  5 6 7 8 8 7 6 5 4 3 2 3 4 5 6 7 8 8 7 6 5 4 5 6 7 8 8 7 6 7 8 8];

zigzag = zeros(1,64);
for k = 1:64
    zigzag(k) = bloc_quantifie(lig(k),col(k));
end
i=1;
while i<63
    if (zigzag(i)==0 && zigzag(i+1)==0 && zigzag(i+2)==0)
        C=3;
        j=3;
        while (zigzag(i+j)==0 && (i+j)<64)
            C=C+1;
            j=j+1;
        end
        vrlc=[vrlc 257 C];
    else
        C=1;
        vrlc=[vrlc zigzag(i)];
    end
    i=i+C;
end

end
```

Après la quantification, beaucoup de valeurs sont à 0. Pour compresser l'image, on peut alors optimiser ces zeros à l'aide du RLC. La première étape de cette partie consiste à transformer le bloc 8x8 en un vecteur de 64 valeurs qui sera appelé VRLC. On utilise ainsi le parcours en zigzag :

Le vecteur de ligne et de colonne sert à donner la position de chaque valeur du bloc de 8x8 dans le vecteur de 1x64. Le parcours ci contre montre la place de chaque valeur.

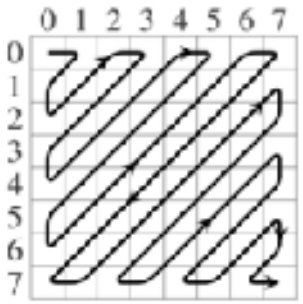


FIGURE 4 – Ré-ordonnancement Zigzag.

Une fois que nous avons notre vecteur remplis, nous cherchons une suite de 3 zéros. En effet, nous avons besoin de 2 bits pour coder le nombre de zéro consécutif, il n'est donc pas nécessaire de détecter une suite de zero inférieur ou égale à 2.

Lorsque l'on trouve au moins 3 zeros consécutifs, le programme évalue les valeurs d'après : si c'est encore un zero, le compteur C s'incrémente de 1, sinon, on sort de la boucle. Finalement, on substitue au vecteur VRLC des deux bits suivants : [257 C]. Cela signifie que nous avons une suite de C zeros.

Finalement, toutes ces fonctions sont mises dans une boucle afin de parcourir toute l'image :

```
VRLC=[];
for (i=1:M2/8)
    for (j=1:N2/8)
        bloc = image_depart( (i-1)*8+1 : i*8, (j-1)*8+1 : j*8);
        bloc_centree = Centrage(bloc);
        % le centrage permet de compresser sur moins de bits et donc
        % d'obtenir une meilleure DCT
        D = dctmtx(8);
        DCT_bloc_centree = D*bloc_centree*D';
        bloc_RGB_luminescence = round(DCT_bloc_centree ./Q_RGB_luminescence);
        bloc_chrominance = round(DCT_bloc_centree ./Q_chrominance);
        VRLC = [VRLC RunLength(bloc_RGB_luminescence)];
    end
end
```

Pour réaliser du code d'Huffman, nous avons besoin de créer un dictionnaire. Dans ce dictionnaire, nous retrouvons toutes les différentes valeurs du vecteur VRLC ainsi que la probabilité liée au nombre de fois que l'on retrouve ce symbole dans le vecteur. Le code est le suivant :

```
symb= unique(VRLC);
prob = zeros(1, length(symb));
nbrsymb= length(symb);
for s=1:nbrsymb
    for i=1:length(VRLC)
        if symb(s) == VRLC(i)
            prob(s)=prob(s)+1;
        end
    end
end
prob=prob./length(VRLC);


dico=huffmandict(symb,prob);
encode=huffmanenco(VRLC,dico);
```

Nous avons ainsi terminer de compresser l'image initiale.

2. Décodage d'une data unit

Le décodage d'une data unit va défaire toutes les étapes précédentes les unes après les autres.

```
% HUffman
signaldeco = huffmandeco(encode,dico);
```

On utilise le même dictionnaire pour le décodage.

On décompresse ensuite VRLC afin de retrouver le nom nombre de zero dans le vecteur.

```
% VRLC décodé (RunLength inverse) :

VRLCdeco= [];
for i=1:length(signaldeco)
    if signaldeco(i)==257
        VRLCdeco=[VRLCdeco zeros(1, signaldeco(i+1))];
        i=i+2;
    else
        VRLCdeco=[VRLCdeco signaldeco(i)];
        i=i+1;
    end
end
```

```
% zigzag inverse

col = [1 2 1 1 2 3 4 3 2 1 1 2 3 4 5 6  5 4 3 2 1 1 2 3 4 5 6 7 8 7 6 5 4 3 2 1 2 3 4 5 6 7 8 8 7 6 5 4 3 4 5 6 7 8 8 7 6 5 6 7 8 8 7 8];
lig = [1 1 2 3 2 1 1 2 3 4 5 4 3 2 1 1 2 3 4 5 6 7 6 5 4 3 2 1 1 2 3 4  5 6 7 8 8 7 6 5 4 3 2 3 4 5 6 7 8 8 7 6 5 4 5 6 7 8 8 7 6 7 8 8];
bloc = zeros(8);
```

```
function bloc_quantifie_inverse = Quantificationinverse(DCT_bloc,fq)
s=0;
if fq<50
    s=5000/fq;
else
    s=200-2*fq;
end
bloc_quantifie_inverse=round((s*DCT_bloc+50)/100);
```

A l'aide de la fonction de quantification inverse ci-dessus, on réalise le décodage :

```
% fonction inverse de zigzag
k=1;
for i=1:8:M
    for j=1:8:M
        for q=1:64
            bloc(lig(q),col(q))=VRLCdeco(k);
            k=k+1;
        end
        % Quentification inverse
        bloc_quantifie_inverse = Quantificationinverse(bloc,20);
        blocquantification=round(bloc_quantifie_inverse.*Q_chrominance);
        % Fonction inverse DCT
        D = dctmtx(8);
        blocDCT=inv(D)*blocquantification*inv(D');
        % Décentrage
        for t=1:length(blocDCT)
            for h=1:length(blocDCT)
                blocdecentrage(t,h)=(cast(blocDCT(t,h),'double'))+128;
            end
        end
        imdeco(i:i+7,j:j+7)=blocdecentrage;

    end
end
```

Pour chaque ligne et chaque colonne et par pas de 8 (taille d'un bloc), on réalise toutes les fonctions inverses.

Malheureusement, l'image en sortie ne ressemble pas à celle de départ.