

[Date]

Compression des images numériques

Compte rendu des travaux dirigés

Guénon Marie et Favreau Jean-Dominique
VIM / MASTER SSTIM

Table des matières

Préambule :	2
Librairies :	3
TD1	4
1. Filtre de Haar, Analyse	4
Méthode directe	4
Filtrage	6
2. Reconstitution, Synthèse	7
Inversion directe	7
Synthèse	8
3. Récursivité	9
Analyse	9
Synthèse	10
TD2	11
1. Quantificateur scalaire	11
2. Caractéristique Entrée / Sortie	12
3. Distorsion	13
TD3	15
1. Quantification	15
2. Entropie	15
3. Reconstruction	16
TD4	19
TP5	22
1. Huffman	22
2. Comparaison	23

But : réaliser une transformée en ondelettes (analyse / synthèse) au moyen du filtre de Haar.

Préambule :

Après une première approche avec Scilab, nous avons préféré utiliser le langage de programmation C++ pour implémenter ce projet. En effet, ce langage nous permet d'avoir une approche simplifiée du traitement d'image ainsi que des calculs optimisés : les calculs matriciels et les itérateurs sont déjà implémentés et optimisés pour le genre de calculs que nous allons devoir effectuer. Nous obtiendrons donc ainsi de meilleurs résultats, autant au niveau qualitatif, qu'au niveau de la rapidité d'obtention.

Librairies :

De manière native, C++ ne sait pas charger d'image. Nous avons donc du installer la librairie :

OpenCv2

```
#include<opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/core/core_c.h>
#include <opencv2/highgui/highgui.hpp>
```

Cette librairie permet de charger, afficher et traiter les images.

```
cv::Mat input = cv::imread("lena.bmp",CV_LOAD_IMAGE_COLOR);
cv::namedWindow("input", CV_WINDOW_NORMAL );
cv::imshow("input",br);
```



TD1

1. Filtre de Haar, Analyse

Méthode directe

Une fois l'image à traiter chargée, nous allons la traiter. Pour ce faire, nous allons la parcourir par bloc de quatre pixels, que l'on nommera $\begin{pmatrix} x & y \\ z & t \end{pmatrix}$. On applique alors les quatre filtres de Haar à chaque point considéré de notre image de départ :

- Moyenne des pixels avoisinants, on applique le filtre : $\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$ et on obtient l'image résultante :

$$I_0 = x + y + z + t$$

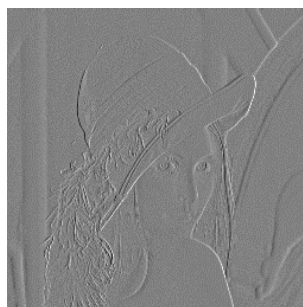
```
*ptr img0++ = val11 + val12 + val21 + val22;
```



- Gradient X, on applique le filtre : $\begin{pmatrix} 1 & -1 \\ 1 & -1 \end{pmatrix}$ et on obtient l'image résultante :

$$I_1 = x - y + z - t$$

```
*ptr img1++ = val11 - val12 + val21 - val22;
```



- Gradient Y, on applique le filtre : $\begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix}$ et on obtient l'image résultante :
 $I_2 = x + y - z - t$

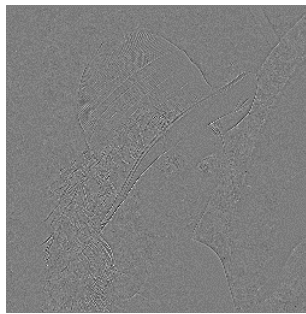
```
*ptr img2++ = val11 + val12 - val21 - val22;
```



- « Gradient diagonal », on applique le filtre : $\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$ et on obtient l'image résultante :

$$I_3 = x - y - z + t$$

```
*ptr img3++ = val11 - val12 - val21 + val22;
```



Ce qui donne, une fois tous les filtres réunis ensemble, on obtient¹ :



¹ Les images qui sont affichées ici ont été retouchées pour pouvoir être visualisable. En effet, une fois les filtres appliqués, les valeurs obtenues ne sont pas des images et doivent être translatées et étirées entre 0 et 255.

Filtrage

Avant de filtrer l'image, nous devons lui appliquer un traitement préalable. En effet, pour éviter les effets de bords et les problèmes aux limites, nous devons insérer une ligne et une colonne supplémentaire à la fin de l'image. Pour cela, nous insérons par miroir (copie du pixel précédent) ces lignes à droite et en bas de l'image.

Après quoi nous appliquons les quatre filtres vus précédemment mais sur l'intégralité de l'image :

- Moyenne des pixels avoisinants $\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$:

$$I_0 = x + y + z + t$$

```
*ptr_img0++ = val11 + val12 + val21 + val22;
```

- Gradient X $\begin{pmatrix} 1 & -1 \\ 1 & -1 \end{pmatrix}$:

$$I_1 = x - y + z - t$$

```
*ptr_img1++ = val11 - val12 + val21 - val22;
```

- Gradient Y $\begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix}$:

$$I_2 = x + y - z - t$$

```
*ptr_img2++ = val11 + val12 - val21 - val22;
```

- « Gradient diagonal » $\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$:

$$I_3 = x - y - z + t$$

```
*ptr_img3++ = val11 - val12 - val21 + val22;
```

Pour finir, nous effectuons le sous échantillonnage sur les quatre filtres obtenus : nous prenons un pixel sur deux en fonction du filtrage effectué : Nous considérons ici que le numéro du premier pixel est (0,0)

- Moyenne des pixels avoisinants $\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$:
On prend les colonnes et les lignes paires.
- Gradient X $\begin{pmatrix} 1 & -1 \\ 1 & -1 \end{pmatrix}$:
On prend les colonnes impaires et les lignes paires.
- Gradient Y $\begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix}$:
On prend les colonnes paires et les lignes impaires.
- « Gradient diagonal » $\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$:
On prend les colonnes et les lignes impaires.

```
for(int row=0; row<rows2;row++)
{
    const int row_x2 = row<<1;
    const int row_x2p1 = row_x2+1;
    ptr_img0 = img0[row_x2];
    ptr_img1 = img1[row_x2];
    ptr_img1++;
    ptr_img2 = img2[row_x2p1];
    ptr_img3 = img3[row_x2p1];
    ptr_img3++;
    for(int j=0; j<cols2; j++)
    {
        *ptr_dst0+=*ptr_img0;
        ptr_img0+=2;
        *ptr_dst1+=*ptr_img1;
        ptr_img1+=2;
        *ptr_dst2+=*ptr_img2;
        ptr_img2+=2;
        *ptr_dst3+=*ptr_img3;
        ptr_img3+=2;
    }
}
```

2. Reconstitution, Synthèse

Inversion directe

Comme vu précédemment, nous avons une image de départ et nous l'avons parcouru par blocs de quatre pixels que nous avons nommé $\begin{pmatrix} x & y \\ z & t \end{pmatrix}$. Cette image a été analysée en quatre images grâce aux filtres décrits ci-dessus :

$$\begin{cases} I_0 = x + y + z + t \\ I_1 = x - y + z - t \\ I_2 = x + y - z - t \\ I_3 = x - y - z + t \end{cases}$$

Pour reconstituer l'image de départ, Il suffit d'inverser le système présenté ici et nous obtenons les formules suivantes :

$$\begin{cases} x = \frac{I_0 + I_1 + (I_2 + I_3)}{4} \\ y = \frac{I_0 + I_2 - (I_1 + I_3)}{4} \\ z = \frac{I_0 + I_1 - (I_2 + I_3)}{4} \\ t = \frac{I_0 + I_3 - (I_1 + I_2)}{4} \end{cases}$$

```
*ptr11 = (val11 + val12 + val21 + val22)>>2;
*ptr12 = (val11 + val21 - (val12 + val22))>>2;
*ptr21 = (val11 + val12 - (val21 + val22))>>2;
*ptr22 = (val11 + val22 - (val21 + val12))>>2;
```

En appliquant ces formules sur les filtres obtenus précédemment, on obtient à gauche l'image d'origine et à droite l'image reconstruite :



Synthèse

De la même manière que dans l'analyse de l'image, avant de filtrer l'image, nous devons lui appliquer un traitement préalable. En effet, pour éviter les effets de bords et les problèmes aux limites, nous devons insérer une ligne et une colonne supplémentaire à la fin de l'image. Pour cela, nous insérons par miroir (copie du pixel précédent) ces lignes à droite et en bas de l'image.

Après quoi, on sur-échantillonne les quatre images. Nous ajoutons des pixels noirs là où on les avait enlevés précédemment :

```
void sur_ech_v(const cv::Mat_<T>& src, cv::Mat_<T>& dst, bool hf)
{
    const int rows = src.rows;
    const int cols = src.cols;
    const int rowsx2 = rows<<1;
    const int size = cols * sizeof(T);
    dst = cv::Mat_<T>(rowsx2,cols);
    T* ptr_dst = dst[0];
    if(!hf)
    {
        for(int row=0; row<rows; row++)
        {
            memcpy(ptr_dst, src[row], size);
            ptr_dst+=cols;
            memset(ptr_dst,0,size);
            ptr_dst+=cols;
        }
    }
    else
    {
        for(int row=0; row<rows; row++)
        {
            memset(ptr_dst,0,size);
            ptr_dst+=cols;
            memcpy(ptr_dst, src[row], size);
            ptr_dst+=cols;
        }
    }
}
```

Sur-échantillonnage vertical

Ensuite, nous appliquons les filtres :

- Filtre basse fréquence (1 1)

```
*ptr_dst+=((*ptr_src1++)+(*ptr_src2++))>>2;
```

- Filtre haute fréquence (1 -1)


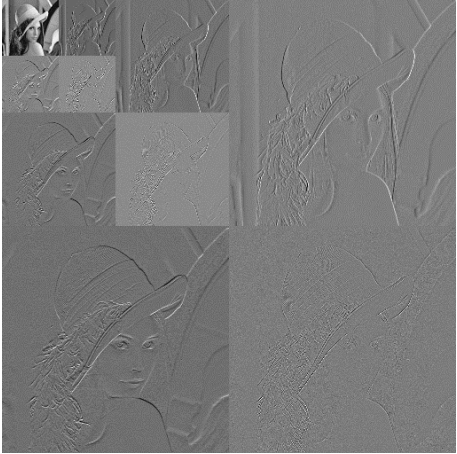

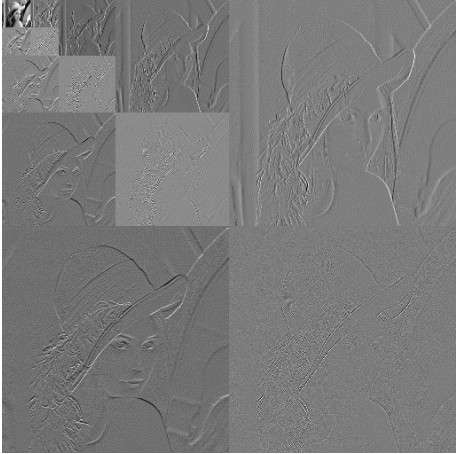
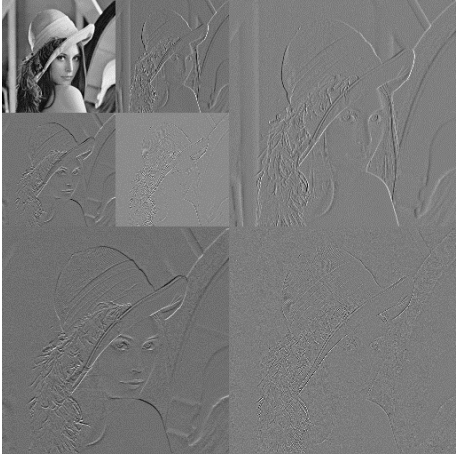
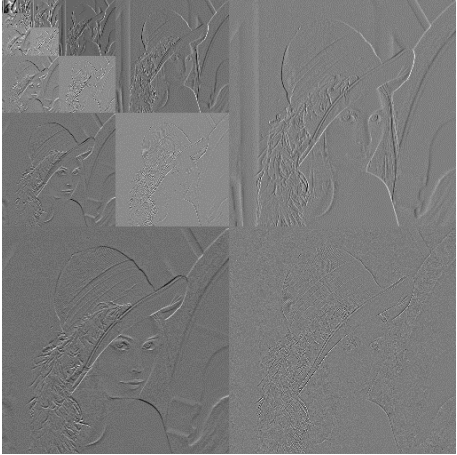
```
*ptr_dst+=((*ptr_src1++)-(*ptr_src2++))>>2;
```

En sommant les résultats obtenus, on obtient l'image d'origine à et à droite l'image reconstruite :



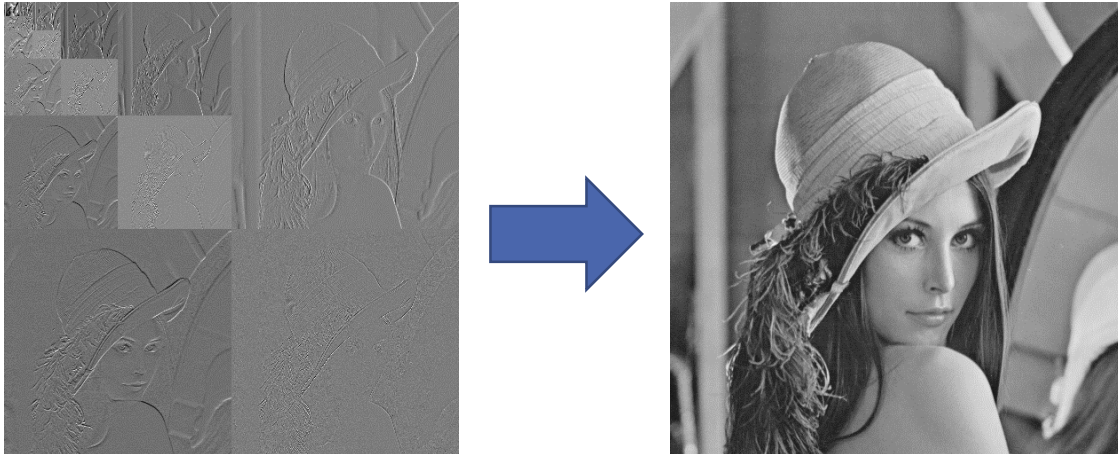
3. Récursivité

Analyse

Nombre d'itération	Résultat obtenu	Nombre d'itération	Résultat obtenu
0		3	
1		4	
2		5	

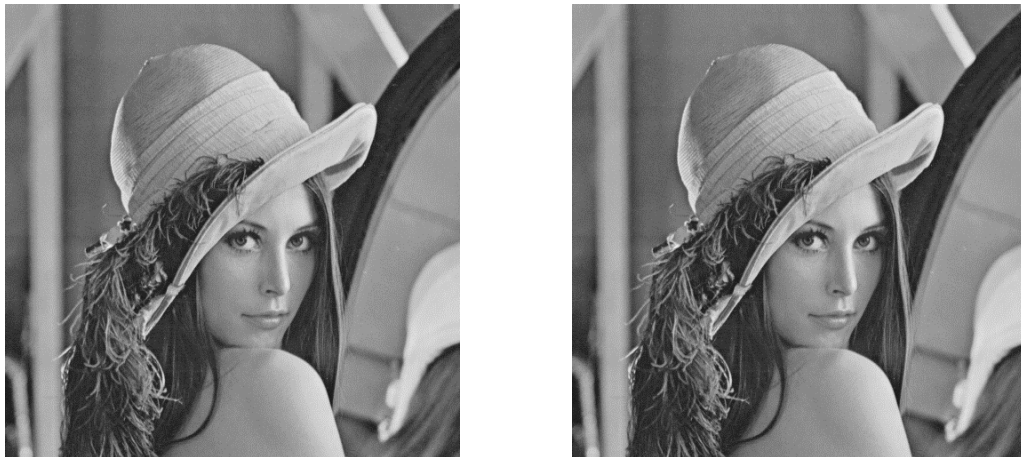
Synthèse

Nous avons appliqué cinq fois notre synthétiseur (méthode directe) sur l'image obtenue après analyse et nous avons eu le résultat suivant :



Pour vérifier la cohérence de nos résultats, nous avons de plus effectué un calcul de distance entre les deux images (norme de la différence) et nous avons constaté que celle-ci était nulle. Et donc nous pouvons dire que nos résultats sont bons.

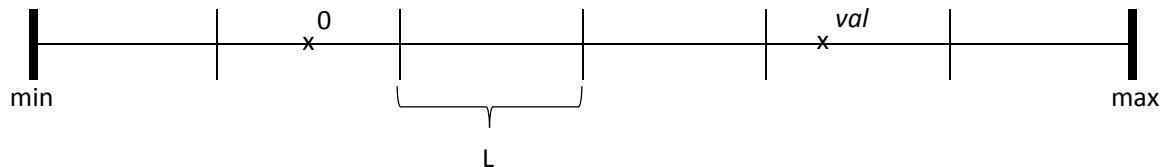
A droit image originale, à gauche image obtenue après analyse et synthèse :



TD2

1. Quantificateur scalaire

Nous cherchons ici à construire un quantificateur scalaire uniforme à L niveaux de quantification comme suit :



Sur cet exemple, \min et \max sont les bornes des valeurs atteignables. Nous avons de plus ici six intervalles, L est la longueur de chacun de ces intervalles. val est une valeur quelconque de l'intervalle $[\min, \max]$. Ici, val est sur le quatrième intervalle (nous comptons à partir de zéro).

De manière générale, nous cherchons à calculer le représentant de chaque classe i (numéro d'intervalle) telle que 0 soit centré au milieu d'un intervalle. Pour cela, nous calculons le pas de quantification centré en zéro:

$$\Delta = \frac{\max - \min}{L}$$

A partir de là, nous pouvons calculer l'indice de la classe contenant la valeur val :

$$i = E\left(\frac{val}{\Delta} + 0,5\right) - E\left(\frac{\min}{\Delta} + 0,5\right)$$

Où $E(.)$ est la partie entière. D'où nous obtenons la valeur du représentant de val :

$$representant = \left(i + E\left(\frac{\min}{\Delta} + 0,5\right)\right) * \Delta$$

Soit définit de manière itérative tel que :

$$\begin{cases} representant_{n+1} = representant_n + \Delta \\ representant_0 = E\left(\frac{\min}{\Delta} + 0,5\right) * \Delta \end{cases} \quad (1)$$

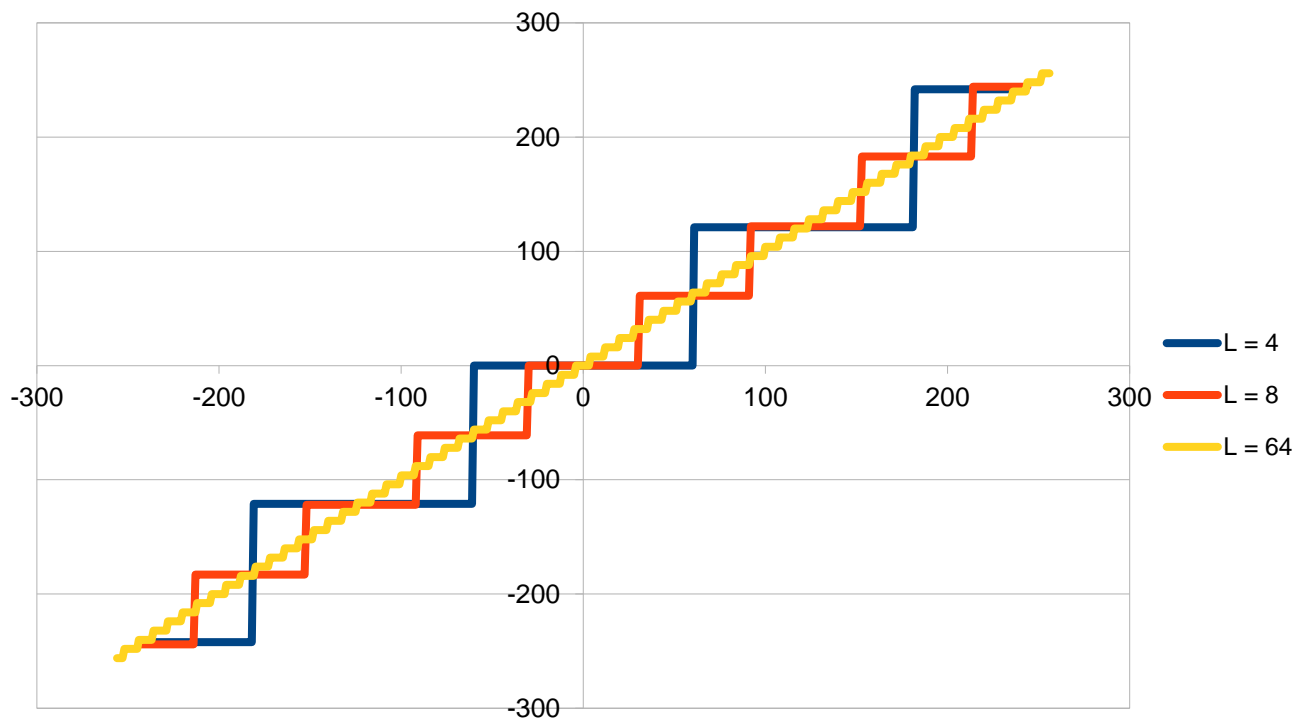
2. Caractéristique Entrée / Sortie

Nous cherchons à avoir un quantificateur centré en zéro, pour cela, on doit fixer les bornes dans lesquelles on va l'appliquer. C'est-à-dire, nous calculons les valeurs suivantes :

$$\text{MAX} = \max(|\min(\text{image})|, |\max(\text{image})|)$$

$$\text{MIN} = -\text{MAX}$$

Ensuite, nous pouvons appliquer notre algorithme défini en (1) ce qui nous donne les résultats suivants pour trois nombres de niveaux différents :



Nous pouvons constater que les caractéristiques obtenues coupent bien les axes en (0,0) ce qui est cohérent avec ce qui était demandé. De plus, nous pouvons voir que la tailles des pas sont homogènes, hormis au tout début et à la toute fin de la courbe. Ce qui est logique puisque notre système est sur-contraint et c'est là que se répercute la variable liée (en trop). Ceci est aussi lié au fait que nous avons forcé la taille de l'intervalle du quantificateur comme entier (pour éviter les problèmes d'arrondis et que chaque pas soit de même longueur).

3. Distorsion

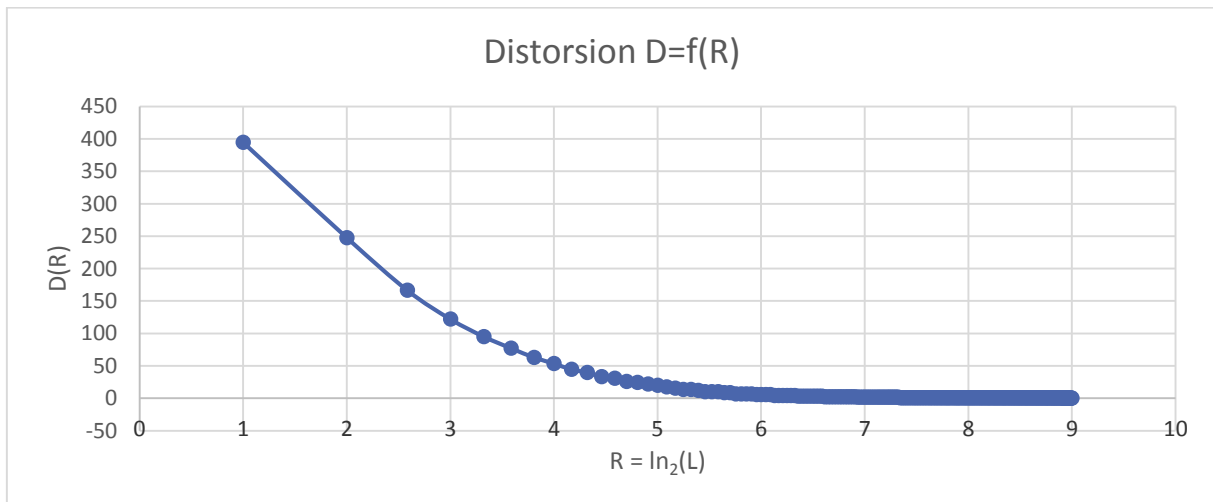
Nous avons la formule de distorsion suivante :

$$D = \frac{1}{\text{lignes} \times \text{colonnes}} \sum_{i \in \tau} (p_{1i} - p_{2i})^2$$

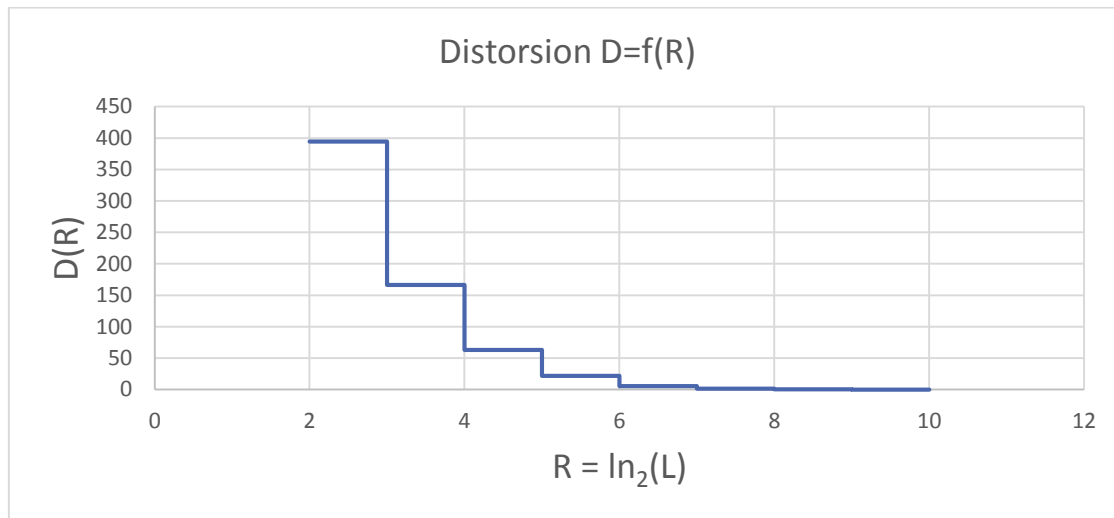
Où p_1 est l'image de référence et p_2 est l'image comparée. Ce qui nous donne l'algorithme suivant :

```
template<typename T, typename U>
double get_distorsion(const cv::Mat_<T>& I1, const cv::Mat_<U>& indice, const std::vector<T>& table_association)
{
    double distorsion(0);
    const int rows = I1.rows;
    const int cols = I1.cols;
    const int rowscols = rows*cols;
    const U* ptr_indice = indice[0];
    const T* ptr_I1 = I1[0];
    for(register int row_col=0; row_col<rowscols; row_col++)
    {
        distorsion += pow(*ptr_I1++ - table_association[*ptr_indice++],2);
    }
    distorsion/=rowscols;
    return distorsion;
}
```

En appliquant notre méthode sur une sous-bande de coefficients d'ondelettes obtenus sur une image de Lena, nous obtenons la courbe suivante pour les L pairs :



Pour le traitement suivant de la distorsion, nous aurons besoin de bits entiers et donc de garder la partie la partie entière supérieur des résultats obtenus :



TD3

1. Quantification

La quantification mise en œuvre ici est la même que celle utilisée au [TD2.1](#). Et nous l'appliquons maintenant à chaque sous bande de l'image :

```
std::vector<std::vector<cv::Mat_<int>>> quantif(img.size());
std::vector<std::vector<std::vector<int>>> table_assoc(img.size());
for(unsigned int i=0; i<img.size(); i++)
{
    quantif[i].resize(img[i].size());
    table_assoc[i].resize(img[i].size());
    for(unsigned int j=0; j<img[i].size(); j++)
    {
        quantificateur_scalaire_uniforme(img[i][j], quantif[i][j], table_assoc[i][j], 8);
    }
}
```

2. Entropie

La formule de l'entropie est la suivante :

$$H(C) = \sum_{i=1}^L p_i \log_2(p_i)$$

Où p_i est le nombre d'éléments codé par le représentant de la classe i , divisé par le nombre de représentants total (probabilité d'apparition du représentant), c'est-à-dire $p_i = Pr\{Q(S) = \hat{s}_i\}$. En appliquant cette formule, nous obtenons l'algorithme suivant :

L'entropie totale suit elle la formule suivante :

$$H_{tot} = \frac{1}{taille\ totale} \sum_{sous-bande} H(C) * taille_sous_bande$$

Le taux de compression est donné par 100 moins la division entre le nombre de bits présents dans l'image obtenue (donné par H_{tot}) et le nombre de bits présents initialement dans l'image (donné par l'entropie de l'image initiale), c'est-à-dire :

$$TC = 100 - \frac{H_{tot}}{H_{image\ initiale}}$$

3. Reconstruction

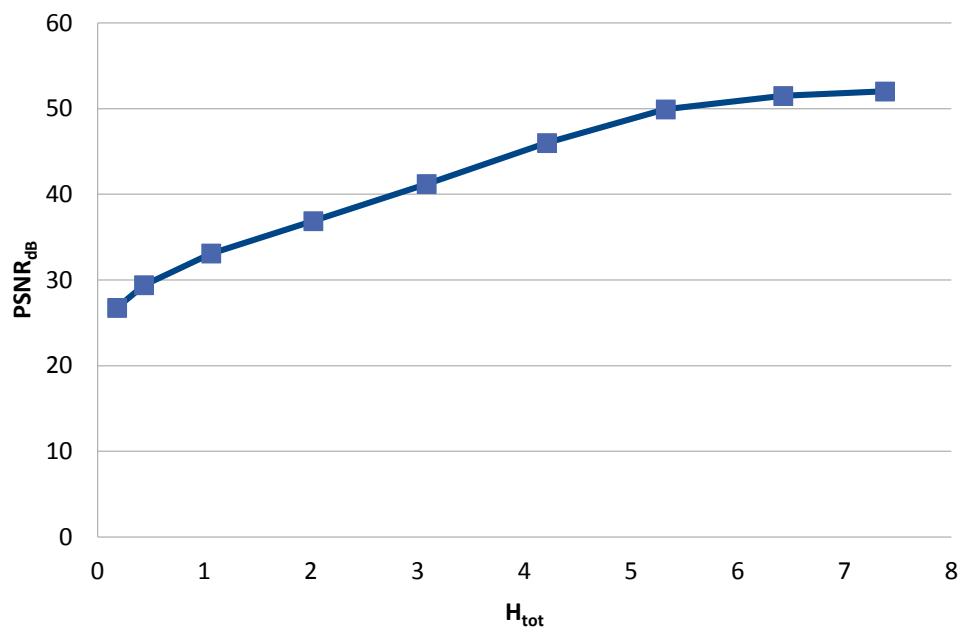
La transformée en ondelette inverse utilisée ici est celle que nous avons développée au [TD1.3](#). Nous mesurons alors son PSNR à partir de la formule suivante :

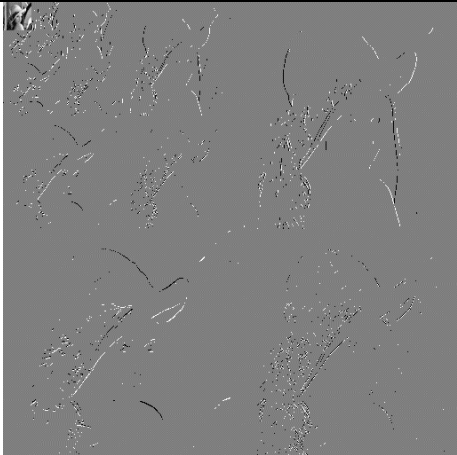
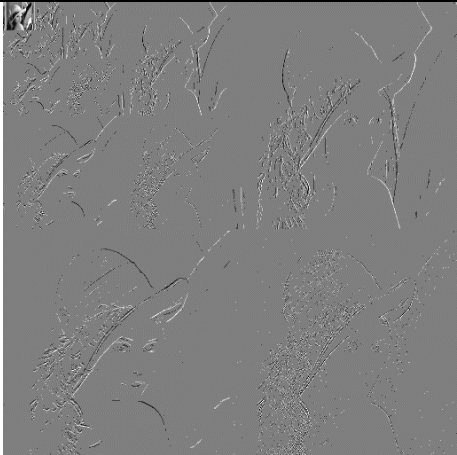
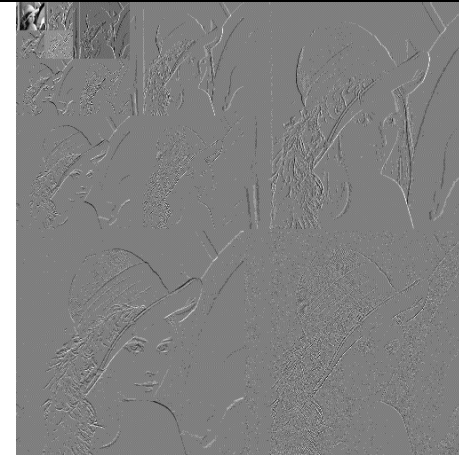
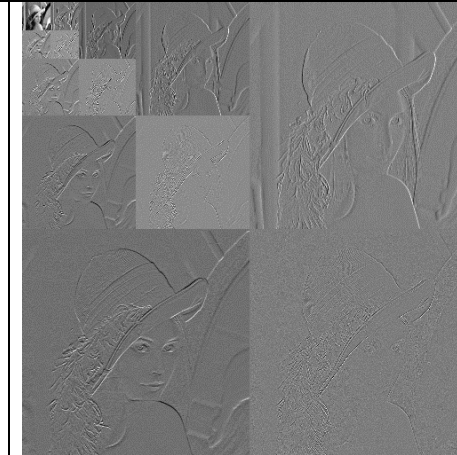
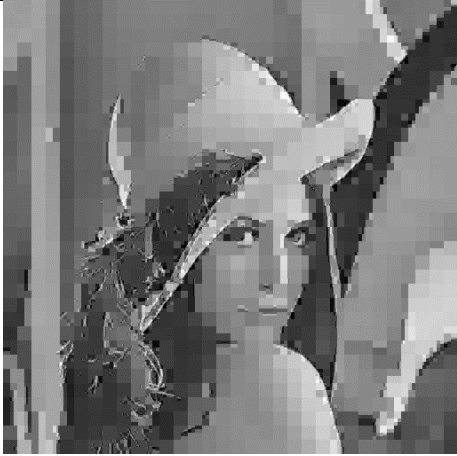



$$PSNR_{dB} = 10 \log_{10} \frac{255^2}{eqm}$$

Où :

$$eqm = \frac{1}{\text{ligne} * \text{colonne}} \sum_i (P_{1i} - P_{2i})^2$$

PSNR_{dB} = fonction(H_{tot})



Nombre d'intervalles	3	5	9	257
Transformée en ondelettes				
Image reconstituée				
Taux de compression (en %)	94,9	90	81,7	5,4
Entropie totale	0,377	0,743	1,36	7,03
PSNR	71,78	74,53	78,12	104,311
EQM totale ($\cdot 10^{-3}$)	4,3	2,28	1,00	0,0024

Avec un petit nombre d'intervalles, on ne garde que peu valeurs, nous avons donc une image très compressée et donc plus légère, mais aussi de moins bonne qualité puisque l'on peut clairement voir des blocs de pixels. A contrario, avec un grand nombre d'intervalles, nous obtenons une image reconstituée de bien meilleure qualité, mais avec un taux de compression bien plus bas et donc une image plus lourde.

De plus, nous pouvons voir ici que plus on utilise d'intervalles pour faire notre quantification, plus l'entropie totale de l'image, son PSNR et son EQM totale diminuent, illustrant bien l'amélioration de la qualité de l'image reconstituée.

```
template<typename T, typename U>
double get_entropy(const cv::Mat_<U>& indice, const std::vector<T>& table_association)
{
    const int rows = indice.rows;
    const int cols = indice.cols;
    const int rowscols = rows*cols;
    const double un_sur_rowscols = 1.0/rowscols;

    const int size_table_assoc = table_association.size();
    std::vector<int> nb_elem(size_table_assoc,0);
    const U* ptr_indice = indice[0];
    for(register int i=0; i<rowscols; i++)
    {
        nb_elem[*ptr_indice++]++;
    }
    double pi;
    double entropy(0);
    int* ptr_nb_elem = nb_elem.data();
    for(register int i=0; i<size_table_assoc; i++)
    {
        pi = (*ptr_nb_elem++) * un_sur_rowscols;
        if(pi!=0)
            entropy -= pi * log2(pi);
    }
    return entropy;
}
```

```
template<typename T, typename U>
double get_eqm(const cv::Mat_<U>& src_0, const cv::Mat_<T>& src_1)
{
    const int rows = src_0.rows;
    const int cols = src_0.cols;
    const int rowscols = rows*cols;
    double eqm = 0;

    const U* ptr_0 = src_0[0];
    const T* ptr_1 = src_1[0];
    for(register int row_col=0; row_col<rowscols; row_col++)
    {
        eqm+=pow(*ptr_0++-*ptr_1++,2);
    }
    eqm/=rowscols;
    return eqm;
}
```

```
template<typename T, typename U>
inline double get_psnr(const cv::Mat_<U>& src_0, const cv::Mat_<T>& src_1)
{
    return 10*log10(65025/get_eqm(src_0, src_1));
}
inline double get_psnr(double eqm)
{
    return 10*log10(65025/eqm);
}
```

TD4

On initialise la pente λ que nous cherchons, la précision ε avec laquelle on veut notre résultat et le débit maximal autorisé R_{SB} .

On cherche les points R_i vérifiant l'équation :

$$\frac{w_i}{a_i} \frac{\partial D_i}{\partial R_i}(R_i) = \lambda (*)$$

On calcule le débit total :

$$R_{SB} = \sum_i R_i$$

On vérifie la distance du débit total calculé par rapport au débit maximal autorisé :

$$|R_{SB} - R_{MAX}| < \varepsilon$$

Si cette inégalité est respectée, on considère qu'il y a convergence. Sinon, on choisit une autre valeur de λ et on relance l'algorithme à l'étape (*). De plus, on utilise les paramètres suivants :

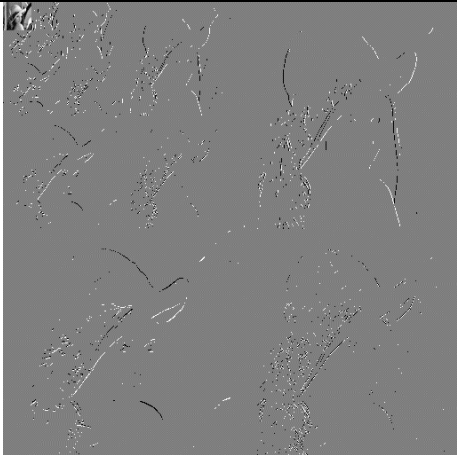
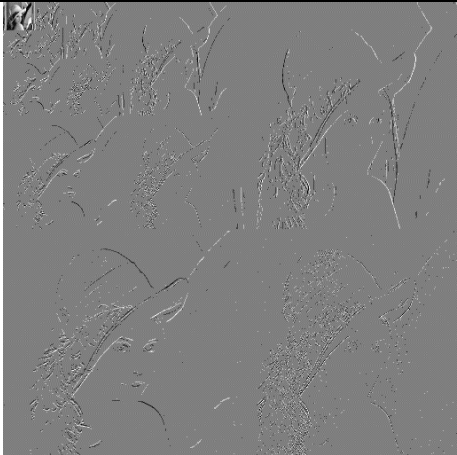
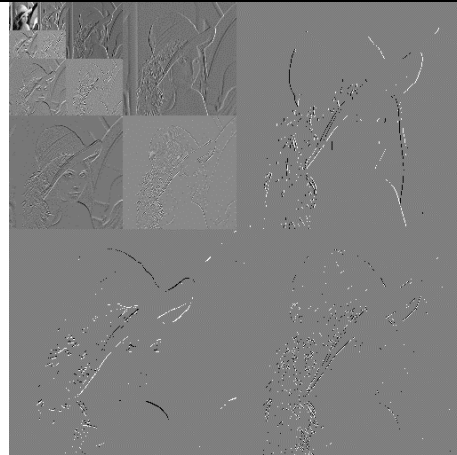
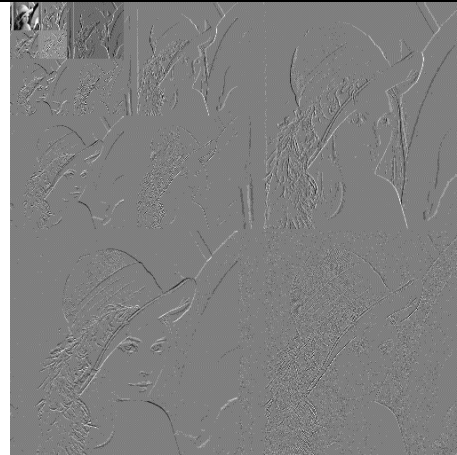




$w_i = 1$ (car nous utilisons la méthode de Haar)

$$a_i = \frac{1}{2^{2i}}$$

$$D(R) = D^{\sigma^2(R)}$$

Notons ici que si $|\lambda|$ augmente, on augmente la compression.

```
template<typename T> double allocation_optimale(size_t profondeur, double lambda, const cv::Mat_<T>& src)
{
    const double ai = 1.0/(1<<(2*profondeur));
    const double lambda_ai = lambda*ai;
    cv::Mat_<T> dst = cv::Mat_<T>(src.rows,src.cols);
    std::vector<T> table_association;
    std::vector<double> R, D;
    for(int L=1; L<=512; L*=2)
    {
        quantificateur_scalaire_uniforme(src,dst,table_association,L);
        R.push_back(log2(double(L)));
        D.push_back(get_distorsion(src,dst,table_association));
    }
    double* ptr_R = R.data();
    double* ptr_D = D.data();
    const int size = R.size();
    std::vector<double> derive(size-1);
    double* ptr_deriv = derive.data();
    --ptr_deriv;
    for(int i=0; i<size-1; ++i)
    {
        const double& r = *ptr_R;
        const double& d = *ptr_D;
        *++ptr_deriv = (d-*++ptr_D)/(r-*++ptr_R);
    }
    ptr_deriv = derive.data();
    double tmp_1 = *ptr_deriv++;
    double tmp_2 = *ptr_deriv++;
    double tmp_val;
    for(int i=1; i<size-2; i++)
    {
        tmp_val = (tmp_1+2*tmp_2+*ptr_deriv)/4;
        tmp_1=tmp_2;
        tmp_2=*ptr_deriv;
        *(ptr_deriv-1)=tmp_val;
        ptr_deriv++;
    }
    ptr_deriv = derive.data();
    ptr_deriv++;
    int i=1;
    for(; *ptr_deriv<lambda_ai && i<size; ++ptr_deriv,++i);
    return R[i];
}
```


Nombre d'intervalles	3	5	Optimal	9
Transformée en ondelettes				
Image reconstituée				
Taux de compression (en %)	94,9	90	82,8	81,7
Entropie totale	0,377	0,743	1,27	1,36
PSNR	71,78	74,53	78,41	78,12
EQM totale ($\cdot 10^{-3}$)	4,3	2,28	0,93	1,00

Sur la transformée en ondelettes, nous pouvons voir que le nombre d'intervalles n'est pas le même sur toutes les sous bandes. En effet, les hautes fréquences sont plus attaquées que les basses fréquences, en particulier sur la première couche où l'on ne garde que 3 fréquences, alors que l'on en garde de plus en plus à mesure que l'on descend dans les couches.

Par ailleurs, nous pouvons constater que nous avons un taux de compression de 82,3%, ce qui donne un fichier assez léger. Malgré ce fort taux de compression, nous pouvons voir que l'image reconstituée est meilleure que ses deux voisines, alors que celles-ci ont des taux de compression proches. La même constatation peut être faite sur le PSNR ainsi que sur l'EQM totale.

TP5

1. Huffman

2. Comparaison