

## TI : Compression d'images

Gautier Kasperek  
Alice Sparrow

Dans ce rapport nous décrivons la manière dont nous avons traité sujet. De plus, nous présentons et interprétons les résultats obtenus. Nous détaillons également nos différentes implémentations lorsque cela est nécessaire, l'ensemble du code du plugin est trouvable en annexe du rapport.

<b>TI : Compression d'images</b>	<b>1</b>
<b>1. Transformée en cosinus discrète</b>	<b>2</b>
Transformée 2D directe	2
Validation sur un exemple	4
<b>2. Utilisation de la DCT pour la compression JPEG</b>	<b>5</b>
Traitement par blocs d'une image	5
Quantification	8
Décompression	10
<b>3. Qualité et performance de la compression JPEG</b>	<b>11</b>
Évaluation de la distorsion	11
Influence de la quantification et ratio de compression	13
<b>Annexe</b>	<b>15</b>
Plugin partie 1 :	15
Plugin partie 2 :	18
Quantification	19
Décompression	20
Qualité et performance de la compression JPEG	21
Influence de la quantification	21
Ratio de compression	21

## 1. Transformée en cosinus discrète

### Transformée 2D directe

La fonction forwardDCT1D prend en entrée un tableau de double représentant le signal d'entrée 1D. Cela correspond à une ligne ou une colonne de l'image en niveau de gris. La sortie de la fonction est donc la ligne ou colonne à laquelle on a appliqué la transformée en cosinus discrète.

Pour chaque valeur  $m$  de l'entrée  $f$ , on applique la formule de la DCT, soit le facteur normalisation multiplié par la somme des  $f(m) * \cos($

```
/**
 * Transformation DCT 1D directe (methode de classe)
 * @param f(m) Signal 1D d'entree (double[])
 * @return F(u) Signal transforme (double[])
 */
public static double[] forwardDCT1D(double[] f) {
    int M = f.length; // Taille du signal
    double k = Math.sqrt(2.0 / M); // Facteur de normalisation
    double[] F = new double[M]; // resultat
    for (int u = 0; u < M; u++) {
        double cu = 1.0;
        if (u == 0)
            cu = 1.0 / Math.sqrt(2); // Facteur de normalisation
        double somme = 0.0;
        for (int m = 0; m < M; m++) {
            somme = somme + f[m]*Math.cos(Math.PI*(m +0.5)*u / M);
        }
        F[u] = k * cu * somme;
    }
    return F;
}
```

La fonction InverseDCT1D prends en entrée un tableau 1D correspondant à des valeurs d'une transformé cosinus discrète. La sortie est un tableau une dimension correspondant à l'inverse de la DCT.

```
/**
 * Transformation DCT 1D inverse (methode de classe)
 * @param F(u) Signal 1D transforme (double[])
 * @return f(m) signal inverse (double[])
 */
```

```

*/
public static double[] inverseDCT1D(double[] F) {
    int M = F.length; // Taille du signal
    double k = Math.sqrt(2.0 / M); // Facteur de normalisation
    double[] f = new double[M]; // resultat
    for (int m = 0; m < M; m++) {
        double somme = 0;
        for (int u = 0; u < M; u++) {
            double cu = 1.0;
            if (u == 0)
                // Facteur de normalisation dependant de u
                cu = 1.0/Math.sqrt(2);
            somme = somme + cu * F[u] *
                Math.cos(Math.PI * (m + 0.5) * u / M);
        }
        f[m] = k * somme;
    }
    return f;
}

```

Ces deux méthodes nous permettent de d'appliquer la DCT et l'inverse de la DCT sur des valeurs en une dimension. Nous avons implémenté une méthode permettant d'appliquer successivement la DCT aux lignes puis aux colonnes d'une image. Cela nous permet de calculer la DCT dans deux dimensions grâce à la séparabilité de la fonction.

```

/**
 * Transformation DCT 2D directe (methode de classe) utilisant la
 * separabilite
 * @param fp Signal 2D d'entree et de sortie (MxN) (FloatProcessor)
 */
public static void forwardDCT(FloatProcessor fp) {
    int width = fp.getWidth();
    int height = fp.getHeight();

    // Traiter les lignes
    for(int y = 0; y < height; y++) {
        double[] lineY = forwardDCT1D(fp.getLine(0,y,width,y));
        for(int x = 0; x < width; x++) {
            fp.putPixelValue(x,y,lineY[x]);
        }
    }

    // Traiter les colonnes de l'image resultant du traitement des

```

```

//lignes
for(int x = 0; x < width; x++) {
double[] lineX = forwardDCT1D(fp.getLine(x,0,x,height));
    for(int y = 0; y < height; y++) {
        fp.putPixelValue(x,y,lineX[y]);
    }
}
}

```

La fonction forwardDCT parcourt dans un premier temps toute les lignes de l'image en leur appliquant la DCT sur une dimension. Dans cette méthode on ne crée pas de nouvelle image, on remplace les valeurs des pixels déjà existant. Dans un second temps, grâce à la séparabilité de la fonction, on applique la DCT une dimension sur chacune des colonnes. De la même manière, on remplace les valeurs existantes obtenues par le calcul de la DCT des lignes. Avant chaque utilisation du plugin nous effectuons un centrage des valeurs sur 0 en soustrayant 128 à tous les pixels.

```
fp.subtract(128);
```

### Validation sur un exemple

Après avoir implémenté le calcul de la DCT en deux dimensions, nous appliquons sur l'exemple donné dans le sujet l'image *wikipedia\_extract* de 8 pixels sur 8 pixels. On obtient le résultat suivant :

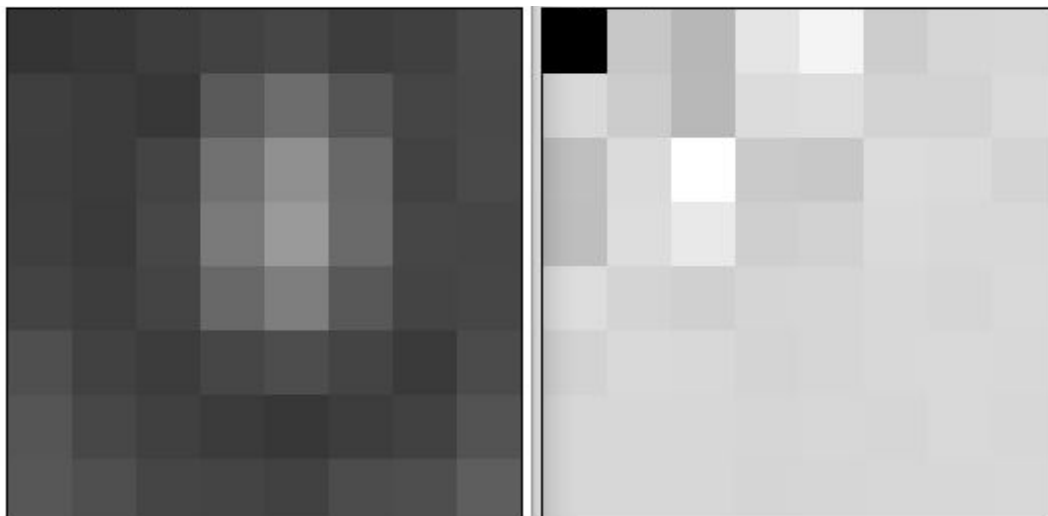


Figure 1 - Image *wikipedia\_extract* à gauche, sa DCT en deux dimensions

Afin de valider notre plugin, on utilise l'outil pixel inspector pour comparer les valeurs obtenues et celles disponibles sur wikipédia.

$$G = \begin{bmatrix} -415.38 & -30.19 & -61.20 & 27.24 & 56.12 & -20.10 & -2.39 & 0.46 \\ 4.47 & -21.86 & -60.76 & 10.25 & 13.15 & -7.09 & -8.54 & 4.88 \\ -46.83 & 7.37 & 77.13 & -24.56 & -28.91 & 9.93 & 5.42 & -5.65 \\ -48.53 & 12.07 & 34.10 & -14.76 & -10.24 & 6.30 & 1.83 & 1.95 \\ 12.12 & -6.55 & -13.20 & -3.95 & -1.87 & 1.75 & -2.79 & 3.14 \\ -7.73 & 2.91 & 2.38 & -5.94 & -2.38 & 0.94 & 4.30 & 1.85 \\ -1.03 & 0.18 & 0.42 & -2.42 & -0.88 & -3.02 & 4.12 & -0.66 \\ -0.17 & 0.14 & -1.07 & -4.19 & -1.17 & -0.10 & 0.50 & 1.68 \end{bmatrix} \quad \downarrow v.$$

Figure 2.a - Résultat de l'exemple sur wikipedia

	0	1	2	3	4	5	6	7
0	-415.38	-30.19	-61.20	27.24	56.12	-20.10	-2.39	0.46
1	4.47	-21.86	-60.76	10.25	13.15	-7.09	-8.54	4.88
2	-46.83	7.37	77.13	-24.56	-28.91	9.93	5.42	-5.65
3	-48.53	12.07	34.10	-14.76	-10.24	6.30	1.83	1.95
4	12.12	-6.55	-13.20	-3.95	-1.88	1.75	-2.79	3.14
5	-7.73	2.91	2.38	-5.94	-2.38	0.94	4.30	1.85
6	-1.03	0.18	0.42	-2.42	-0.88	-3.02	4.12	-0.66
7	-0.17	0.14	-1.07	-4.19	-1.17	-0.10	0.50	1.68

Figure 2b - Résultat obtenu de la DCT de l'image wikipedia\_extract

Nous constatons que les valeurs obtenues sont les mêmes et ainsi valider l'implémentation de notre plugin. Nous allons donc l'utiliser dans les parties suivantes pour étudier les systèmes de compression.

## 2. Utilisation de la DCT pour la compression JPEG

### Traitement par blocs d'une image

Sur une image de grande taille le calcul de la transformation DCT devient compliqué. C'est pourquoi, nous choisissons de découper l'image en blocs de 8\*8 pixels et d'appliquer la transformation DCT bloc par bloc.

```
for(int i = 0; i < fp.getWidth(); i = i + BLOCK_SIZE){
    for(int j = 0; j < fp.getHeight(); j = j + BLOCK_SIZE){
        fp.setRoi(i,j,BLOCK_SIZE, BLOCK_SIZE);
        forwardDCT(fp);
    }
}
```

Nous parcourons donc chaque bloc, la taille de 8 pixels étant défini par la variable globale BLOCK\_SIZE. Pour chaque bloc nous définissons une région d'intérêt de la taille d'un bloc, puis on effectue la transformation DCT sur l'image. Nous avons modifié le calcul de la transformation DCT pour prendre en compte le calcul par bloc.

```

public static void forwardDCT(FloatProcessor fp) {

    int width = BLOCK_SIZE;
    int height = BLOCK_SIZE;

    Rectangle roi = fp.getRoi();
    int xPos = (int)roi.getX();
    int yPos = (int)roi.getY();

    // Traiter les lignes
    for(int y = 0; y < BLOCK_SIZE; y++) {
        double[] lineY = forwardDCT1D(fp.getLine(xPos,y + yPos,xPos
        + BLOCK_SIZE,y + yPos));
        for(int x = 0; x < width; x++) {
            fp.putPixelValue(x + xPos,y + yPos,lineY[x]);
        }
    }

    // Traiter les colonnes de l'image resultant du traitement des lignes
    for(int x = 0; x < BLOCK_SIZE; x++) {
        double[] lineX = forwardDCT1D(fp.getLine(x + xPos,yPos,x +
        xPos, yPos + BLOCK_SIZE));
        for(int y = 0; y < BLOCK_SIZE; y++) {
            fp.putPixelValue(x + xPos,y + yPos,lineX[y]);
        }
    }
}

```

La fonction récupère la région d'intérêt (ROI) définie par le plugin et réalise le calcul de la DCT en ligne puis en colonnes seulement pour le bloc sélectionné. Pour continuer notre étude nous utilisons l'image *lena\_8bits.png* fourni dans le sujet. Nous effectuons donc le calcul de sa transformation DCT ci-dessous.



Figure 3 - A gauche *lena\_8bits.png*, à droite sa transformé DCT

Afin d'observer les caractéristiques de la transformation DCT nous avons sélectionné plusieurs blocs caractéristiques de l'image *lena.png*. Le premier est l'épaule, une zone lisse et clair de l'image, le second est situé dans la bande sombre en haut à droite de l'image et le dernier dans la décoration du chapeau, zone détaillée et contrastée.

	328	329	330	331	332	333	334	335
432	557.44	-10.29	20.89	18.78	18.07	19.76	21.47	23.19
433	71.52	-35.32	-27.33	-25.55	-32.22	-32.77	-34.09	-33.24
434	-60.07	32.95	26.94	26.81	30.42	28.73	30.68	28.65
435	57.70	-28.16	-27.90	-26.33	-27.17	-28.34	-24.37	-29.03
436	-53.47	24.07	24.49	21.73	24.55	25.67	23.65	25.85
437	42.00	-25.01	-18.21	-20.57	-18.78	-21.90	-18.82	-15.98
438	-30.33	18.56	15.14	13.28	15.42	15.38	16.51	18.31
439	19.72	-11.51	-9.87	-10.98	-11.37	-11.52	-11.81	-8.79

Figure 4a - Valeurs de la DCT de *Lena.png* dans la première zone

Nous pouvons constater que sur une surface claire et lisse, une zone de basse fréquence, le coin en haut à gauche possède une forte valeur positive alors que le reste des valeurs restent relativement faible.

	456	457	458	459	460	461	462	463
80	-536.33	-11.14	-23.55	-19.24	-24.71	-22.42	-21.80	-20.58
81	-61.80	24.47	29.86	18.15	25.87	35.98	29.35	30.06
82	56.34	-32.42	-28.18	-36.02	-34.62	-26.29	-26.48	-31.65
83	-49.81	27.22	26.47	27.90	19.91	27.84	27.86	33.91
84	41.74	-23.56	-22.06	-23.19	-24.71	-25.03	-23.58	-20.60
85	-39.86	20.07	13.29	19.37	21.99	23.90	18.86	24.62
86	33.94	-13.87	-15.30	-14.76	-9.97	-5.31	-15.75	-16.37
87	-25.20	1.89	7.53	5.78	5.89	7.36	11.64	7.64

Figure 4b - Valeurs de la DCT de *lena.png* dans la deuxième zone

De la même manière dans une zone sombre et lisse, de basse fréquence. Le bloc contient une grande valeur (en valeur absolue) en haut à gauche puis des valeurs relativement faible par rapport à celle-ci.



	128	129	130	131	132	133	134	135
312	-318.11	130.66	-60.12	82.92	107.53	15.22	8.53	-9.54
313	-123.54	42.24	-10.97	-51.25	75.55	59.25	-42.45	1.60
314	55.99	-25.03	-7.60	-90.54	-73.93	53.11	33.74	-52.39
315	-59.33	18.86	37.02	64.78	14.81	-19.36	25.16	23.79
316	52.73	-10.03	-28.17	-54.69	-29.77	11.40	-2.13	-9.22
317	-44.24	11.57	23.37	51.06	9.78	-14.61	5.09	15.91
318	29.38	-14.42	-15.38	-41.25	-18.17	6.00	-3.63	-7.47
319	-15.62	3.94	13.91	27.69	6.45	-6.02	6.16	1.34

Figure 4c - Valeurs de la DCT de lena.png dans la troisième zone

Alors que dans une zone à forte fréquence, la décoration chapeau, la valeur maximale est plus faible, et les autres valeurs sont plus proche du maximum du bloc.

Nous constatons que les valeurs vont d'une forte valeur absolue en haut à gauche puis tendent vers zéro en se rapprochant du coin en bas à droite.

### Quantification

Nous allons à présent utiliser la matrice de quantification JPEG ( noté QY ci-dessous) pour la composante de luminance afin de filtrer une partie des coefficients DCT.

```
public final static int[][] QY = {
    {16, 12, 14, 14, 18, 24, 49, 72},
    {11, 12, 13, 17, 22, 35, 64, 92},
    {10, 14, 16, 22, 37, 55, 78, 95},
    {16, 19, 24, 29, 56, 64, 87, 98},
    {24, 26, 40, 51, 68, 81, 103, 112},
    {40, 58, 57, 87, 109, 104, 121, 100},
    {51, 60, 69, 80, 103, 113, 120, 103},
    {61, 55, 56, 62, 77, 92, 101, 99}
};
```

Pour ce faire, nous créons une image à partir de la matrice et les appliquons à chaque bloc en divisant les valeurs de l'image par les coefficients de la matrice puis en arrondissant les pixels obtenues à l'entier le plus proche.

```
for(int i = 0; i < fp.getWidth(); i = i + BLOCK_SIZE){
    for(int j = 0; j < fp.getHeight(); j = j + BLOCK_SIZE){
        fp.setRoi(i,j,BLOCK_SIZE, BLOCK_SIZE);
        forwardDCT(fp);
        // Divide de la quantification
        fp.copyBits(bpQuantification, i,j, 6); // 6 = DIVIDE
```



```

    }
}
// Arrondi des valeurs finale
for(int i = 0; i < fp.getWidth(); i++){
    for(int j = 0; j < fp.getHeight(); j++){
        float pixel = (float)(Math.round(fp.getPixelValue(i,j)));
        fp.setf(i,j,pixel);
    }
}
}

```

Nous avons appliqué notre plugin à l'image exemple *extract\_wikipédia* afin de vérifier notre implémentation. Nous obtenons l'image ci-dessous.

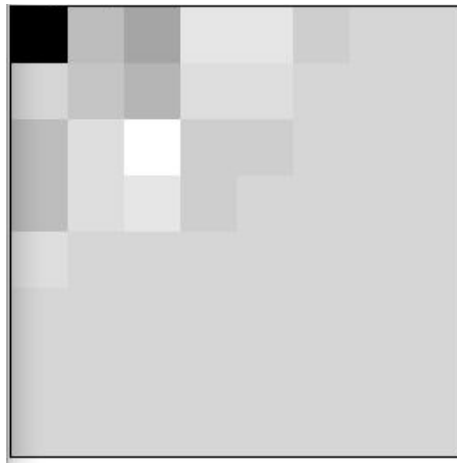


Figure 4 - DCT de *extract\_wikipédia* avec application de la matrice de quantification QY

Comme précédemment nous utilisons le pixel inspector fournit par ImageJ pour comparer nos résultats avec ceux attendus.

	0	1	2	3	4	5	6	7
0	-26.000	-3.000	-6.000	2.000	2.000	-1.000	0.000	0.000
1	0.000	-2.000	-4.000	1.000	1.000	0.000	0.000	0.000
2	-3.000	1.000	5.000	-1.000	-1.000	0.000	0.000	0.000
3	-3.000	1.000	2.000	-1.000	0.000	0.000	0.000	0.000
4	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
5	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
6	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
7	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

$$B = \begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -3 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 5 - En haut les résultats obtenu sur ImageJ, en bas les résultats attendus sur wikipedia

Nous constatons que les résultats obtenus sont les même que les résultats attendus. Cela nous permet de vérifier l'implémentation de notre plugin et continuer l'étude avec un plugin fonctionnel.

### Décompression

Pour visualiser l'image résultat, il est nécessaire d'implémenter le processus de décompression. Nous allons procéder comme suit :

- multiplication des coefficients DCT quantifiés par les éléments de la matrice de quantification
- application de la DCT inverse
- ajout de 128 pour visualiser l'image décompressée

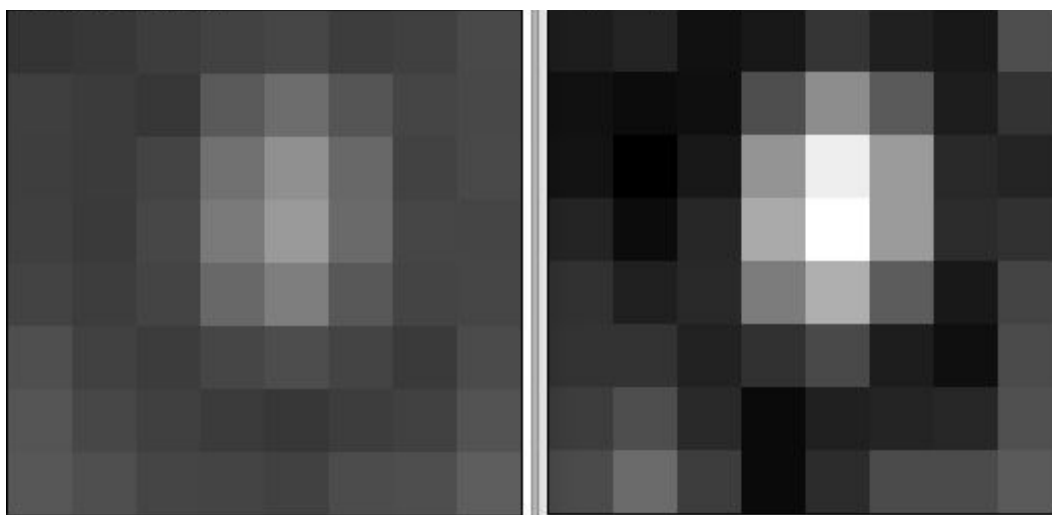


Figure 6 - Image d'origine (à gauche) et Image décompressée (à droite)

Nous avons donc obtenu l'image de droite sur la figure 6. On remarque que l'image décompressée n'est pas identique à l'image d'origine.

Prefs	0	1	2	3	4	5	6	7
0	62.000	65.000	57.000	60.000	72.000	63.000	60.000	82.000
1	57.000	55.000	56.000	82.000	108.000	87.000	62.000	71.000
2	58.000	50.000	60.000	111.000	148.000	114.000	67.000	65.000
3	65.000	55.000	66.000	120.000	155.000	114.000	68.000	70.000
4	70.000	63.000	67.000	101.000	122.000	88.000	60.000	78.000
5	71.000	71.000	64.000	70.000	80.000	62.000	56.000	81.000
6	75.000	82.000	67.000	54.000	63.000	65.000	66.000	83.000
7	81.000	94.000	75.000	54.000	68.000	81.000	81.000	87.000

62	65	57	60	72	63	60	82
57	55	56	82	108	87	62	71
58	50	60	111	148	114	67	65
65	55	66	120	155	114	68	70
70	63	67	101	122	88	60	78
71	71	64	70	80	62	56	81
75	82	67	54	63	65	66	83
81	94	75	54	68	81	81	87

Figure 7 - Ajout de 128 à chaque pixel

Afin de visualiser l'image nous avons dû ajouter 128 à chaque pixels. Nous avons vérifié les résultats que nous avons obtenu en consultant les valeurs disponibles sur Wikipédia, nous pouvons notamment le voir sur l'image ci-dessus.

### 3. Qualité et performance de la compression JPEG

#### Évaluation de la distorsion

Dans cette partie nous allons évaluer la qualité et les performances de la compression par quantification du résultat de la DCT. Pour évaluer la qualité du résultat, on mesure la différence entre l'image obtenue et l'image originale.

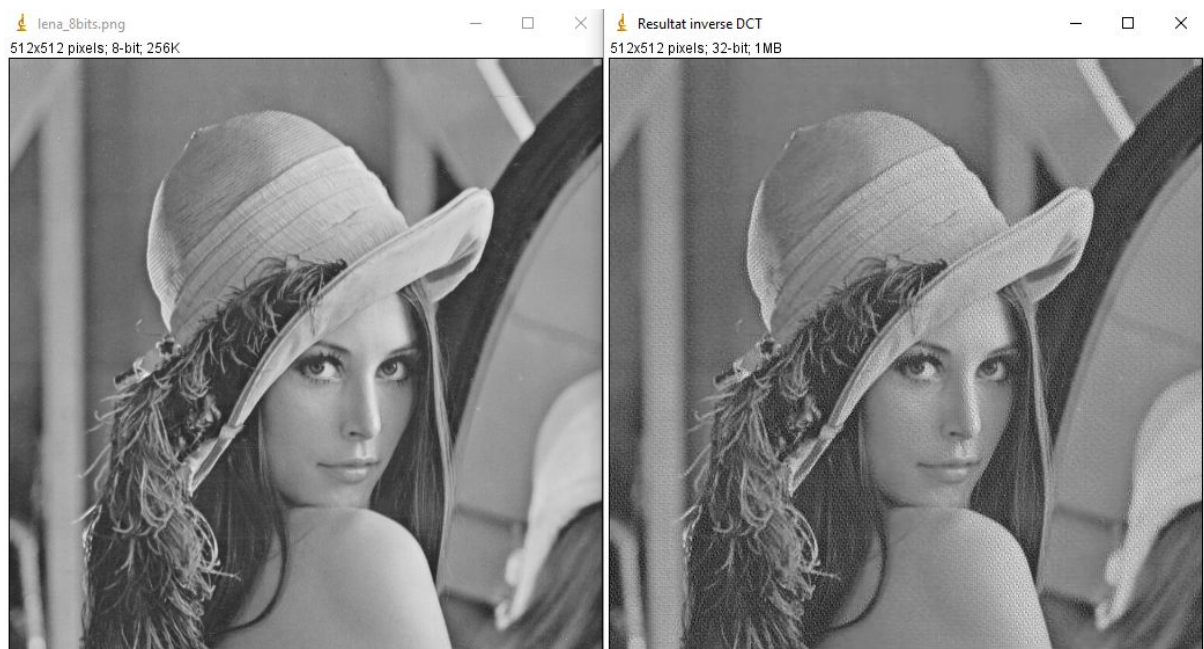


Figure 8 - Image d'origine et image compressée puis décompressée

Nous pouvons noter que les images de la figure 6 sont très semblables cependant sur l'image compressée/décompressée on remarque un petit flou. En effet la compression est avec perte, nous allons donc chercher à évaluer la distorsion de l'image.

	Area	Mean	Min	Max		Area	Mean	Min	Max	
1	262144	128.231	35	240	1	262144	128.225	-16	283	

Figure 9 - Données sur l'image d'origine et l'image compressée et décompressée

On remarque sur la figure 9 que les moyennes de l'image d'origine et l'image compressée/décompressée diffèrent légèrement ce qui témoigne de cette petite distorsion de l'image décompressée.

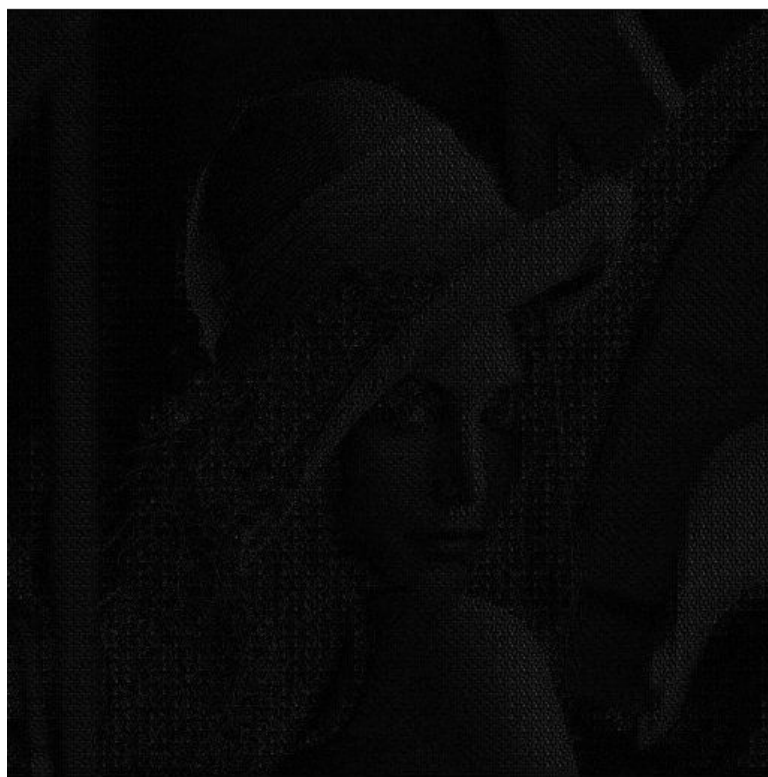



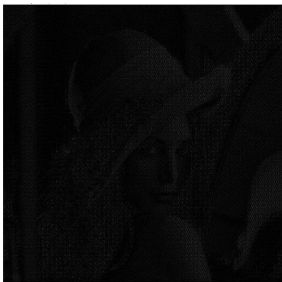

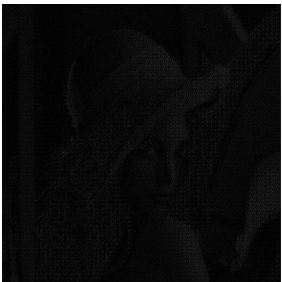


Figure 10 - Différence entre l'image d'origine et l'image compressée puis décompressée







La figure ci-dessus est le résultat de la soustraction entre l'image d'origine et l'image compressée/décompressée. On remarque que l'image est globalement noire, seuls quelques points lumineux sont visibles. On peut conclure de cette soustraction que la distorsion entre les deux images est minime.

### Influence de la quantification et ratio de compression

La valeur min du tableau correspond à la valeur minimale de la différence en valeur absolue entre les pixels de l'image de base et les pixels de l'image compressée. Les valeurs max et moy correspondent également à l'image différence.

Taille de l'image initiale = 1 MB = 1024 kB. Pour trouver ratio de compression, on divise la taille de l'image par la taille de l'image compressée.

	Image compressée	Différence en valeur absolue entre les pixels de l'image de base et les pixels de l'image compressée	min	max	moy	ratio de compression
q = 75			0	87	9.156	$1024/108 = 9.48$
q = 50			0	87	9.156	$1024/108 = 9.48$
q = 25			0	89	11.303	$1204/60 = 17.06$

q = 15			0	88	20.534	$1024 / 44 = 23.27$
q = 7			0	103	9.590	$1024 / 16 = 64$
q = 5			0	93	12.344	$1024 / 8 = 128$

Nous remarquons que le ratio de compression varie en fonction de q. Plus q est faible, plus la compression est grande et inversement.

Pour générer la compression nous avons modifié notre plugin. Nous avons rencontré de nombreux obstacles concernant les différents types de retour des fonctions utilisés, c'est pourquoi nous pensons qu'il existe une version plus optimisée pour effectuer la compression en une ligne de pixel.

```

List<Float> allCoefsBeforeEOB = new ArrayList<Float>();
for(int i = 0; i < fp.getWidth(); i = i + BLOCK_SIZE){
    for(int j = 0; j < fp.getHeight(); j = j + BLOCK_SIZE){
        float[][] region= new float[BLOCK_SIZE][BLOCK_SIZE];
        for(int x=i; x < i+BLOCK_SIZE; x++){
            for(int y=j; y < j+BLOCK_SIZE; y++){
                region[x-i][y-j] = fp.getPixel(x,y);
            }
        }
        FloatProcessor im_region = new FloatProcessor(region);
    }
}

```



```

        List<Float> res_region = coefsBeforeEOB(im_region);
        allCoefsBeforeEOB.addAll(res_region);
    }
}
float[][] tab = new float[1][allCoefsBeforeEOB.size()];
for (int p=0; p<allCoefsBeforeEOB.size();p++){
    tab[0][p]= allCoefsBeforeEOB.get(p);
}
FloatProcessor im_comp = new FloatProcessor(tab);
ImagePlus imP = new ImagePlus("All coef ",im_comp);
FileSaver fs = new FileSaver(imP);
fs.saveAsPng();

```

Ainsi pour chaque bloc, nous récupérons les valeurs dans l'ordre du parcours diagonale jusqu'à la fin du bloc EOB. Puis nous contenons les listes retournés dans un tableau deux dimensions. Seul la première ligne est rempli. Puis grâce à l'environnement imageJ nous créons une image à partir du tableau fini de taille 1 sur le nombre de pixels compressés.

## Annexe

### Plugin partie 1 :

```

import ij.*;
import ij.plugin.filter.*;
import ij.process.*;
import ij.gui.*;
public class validation_ implements PlugInFilter{

    ImagePlus imp;        // Fenetre contenant l'image de reference
    int width;             // Largeur de la fenetre

```



```

int height;          // Hauteur de la fenetre
final static int BLOCK_SIZE = 8;

public int setup(String arg, ImagePlus imp) {
    this.imp = imp;
    return PlugInFilter.DOES_8G;
}

public void run(ImageProcessor ip){
    float[][] imageFloatArray = ip.getFloatArray();
    FloatProcessor fp = new FloatProcessor(imageFloatArray);

    // Subtract 128 pour centrer les valeurs
    fp.subtract(128);
    forwardDCT(fp);
    ImagePlus im = new ImagePlus("Resultat DCT2D",fp);
    im.show();
}

/**
 * Transformation DCT 1D directe (methode de classe)
 * @param f(m) Signal 1D d'entree (double[])
 * @return F(u) Signal transforme (double[])
 */
public static double[] forwardDCT1D(double[] f) {
    int M = f.length; // Taille du signal
    double k = Math.sqrt(2.0 / M); // Facteur de normalisation
    double[] F = new double[M]; // resultat
    for (int u = 0; u < M; u++) {
        double cu = 1.0;
        if (u == 0)
            cu = 1.0 / Math.sqrt(2); // Facteur de normalisation
        double somme = 0.0;
        for (int m = 0; m < M; m++) {
            somme = somme + f[m] * Math.cos(Math.PI * (m + 0.5) * u /
M);
        }
        F[u] = k * cu * somme;
    }
    return F;
}

```

```

/**
 * Transformation DCT 1D inverse (methode de classe)
 * @param F(u) Signal 1D transforme (double[])
 * @return f(m) signal inverse (double[])
 */
public static double[] inverseDCT1D(double[] F) {
    int M = F.length; // Taille du signal
    double k = Math.sqrt(2.0 / M); // Facteur de normalisation
    double[] f = new double[M]; // resultat
    for (int m = 0; m < M; m++) {
        double somme = 0;
        for (int u = 0; u < M; u++) {
            double cu = 1.0;
            if (u == 0)
                // Facteur de normalisation dépendant de u
                cu = 1.0/Math.sqrt(2);
            somme = somme + cu * F[u] * Math.cos(Math.PI * (m
                + 0.5) * u / M);
        }
        f[m] = k * somme;
    }
    return f;
}

/**
 * Transformation DCT 2D directe (methode de classe) utilisant la
 * separabilite
 * @param fp Signal 2D d'entree et de sortie (MxN) (FloatProcessor)
 */
public static void forwardDCT(FloatProcessor fp) {
    int width = fp.getWidth();
    int height = fp.getHeight();

    // Traiter les lignes
    for(int y = 0; y < height; y++) {
        double[] lineY = forwardDCT1D(fp.getLine(0,y,width,y));
        for(int x = 0; x < width; x++) {
            fp.putPixelValue(x,y,lineY[x]);
        }
    }

    // Traiter les colonnes de l'image resultant du traitement des
    //lignes
    for(int x = 0; x < width; x++) {

```

```

        double[] lineX = forwardDCT1D(fp.getLine(x,0,x,height));
        for(int y = 0; y < height; y++) {
            fp.putPixelValue(x,y,lineX[y]);
        }
    }
}

```

### Plugin partie 2 :

Il faut remplacer les méthodes run et forwardDCT par les méthode ci-dessous.

```

public void run(ImageProcessor ip){
    float[][] imageFloatArray = ip.getFloatArray();
    FloatProcessor fp = new FloatProcessor(imageFloatArray);

    // Subtract 128 pour centrer les valeurs
    fp.subtract(128);
    for(int i = 0; i < fp.getWidth(); i = i + BLOCK_SIZE){
        for(int j = 0; j < fp.getHeight(); j = j + BLOCK_SIZE){
            fp.setRoi(i,j,BLOCK_SIZE, BLOCK_SIZE);
            forwardDCT(fp);
        }
    }

    ImagePlus im = new ImagePlus("Resultat DCT2D",fp);
    im.show();
}

/**
 * Transformation DCT 2D directe (methode de classe) utilisant la
 * séparabilité
 * @param fp Signal 2D d'entree et de sortie (MxN) (FloatProcessor)
 */
public static void forwardDCT(FloatProcessor fp) {
    int width = BLOCK_SIZE;
    int height = BLOCK_SIZE;

    Rectangle roi = fp.getRoi();
    int xPos = (int)roi.getX();
    int yPos = (int)roi.getY();
}

```

```

// Traiter les lignes
for(int y = 0; y < BLOCK_SIZE; y++) {
    double[] lineY = forwardDCT1D(fp.getLine(xPos,y + yPos,xPos +
        BLOCK_SIZE,y + yPos));
    for(int x = 0; x < width; x++) {
        fp.putPixelValue(x + xPos, y + yPos,lineY[x]);
    }
}

// Traiter les colonnes de l'image resultant du traitement des lignes
for(int x = 0; x < BLOCK_SIZE; x++) {
    double[] lineX = forwardDCT1D(fp.getLine(x + xPos,yPos,x +
        xPos, yPos + BLOCK_SIZE));
    for(int y = 0; y < BLOCK_SIZE; y++) {
        fp.putPixelValue(x + xPos,y + yPos,lineX[y]);
    }
}
}

```

### Quantification

Fonction **run()** du plugin à modifier

```

public void run(ImageProcessor ip){
    float[][] imageFloatArray = ip.getFloatArray();
    FloatProcessor fp = new FloatProcessor(imageFloatArray);
    bpQuantification.setIntArray(QY);    // initialisation

    // Subtract 128 pour centrer les valeurs
    fp.subtract(128);
    for(int i = 0; i < fp.getWidth(); i = i + BLOCK_SIZE){
        for(int j = 0; j < fp.getHeight(); j = j + BLOCK_SIZE){
            fp.setRoi(i,j,BLOCK_SIZE, BLOCK_SIZE);
            forwardDCT(fp);
            // Divide de la quantification
            if(QUANTIFICATION){
                fp.copyBits(bpQuantification, i,j, 6); // 6 = DIVIDE
            }
        }
    }

    // Arrondi des valeurs finale
    if(QUANTIFICATION){
        for(int i = 0; i < fp.getWidth(); i++){
            for(int j = 0; j < fp.getHeight(); j++){
                float pixel = (float)(Math.round(fp.getPixelValue(i,j)));
            }
        }
    }
}

```

```

        fp.setf(i,j,pixel);
    }
}

ImagePlus im = new ImagePlus("Resultat DCT2D",fp);
im.show();
}

```

## Décompression

A ajouter dans la méthode run()

```

// Inverse
for(int i = 0; i < fp.getWidth(); i = i + BLOCK_SIZE){
    for(int j = 0; j < fp.getHeight(); j = j + BLOCK_SIZE){
        fp.copyBits(bpQuantification,i ,j,5);
        fp.setRoi(i,j,BLOCK_SIZE, BLOCK_SIZE);
        inverseDCT(fp);
    }
}

// Arrondi des valeurs finales + 128
for(int i = 0; i < fp.getWidth(); i++){
    for(int j = 0; j < fp.getHeight(); j++){
        float pixel = (float)(Math.round(fp.getPixelValue(i,j)) + 128);
        fp.setf(i,j,pixel);
    }
}

ImagePlus im2 = new ImagePlus("Résultat inverse DCT", fp);
im2.show();

```

```

//
-----
-----
/**
 * Transformation DCT 2D inverse (methode de classe)
 * @param fp Signal 2D d'entree et de sortie (FloatProcessor)
 */
public static void inverseDCT(FloatProcessor fp) {
    int width = BLOCK_SIZE;
    int height = BLOCK_SIZE;
    Rectangle roi = fp.getRoi();
    int xPos = (int)roi.getX();

```

```

    int yPos = (int)roi.getY();

    for(int y = 0; y < BLOCK_SIZE; y++) {
        double[] lineY = inverseDCT1D(fp.getLine(xPos,y + yPos,xPos
            + BLOCK_SIZE,y + yPos));
        for(int x = 0; x < width; x++) {
            fp.putPixelValue(x + xPos,y + yPos,lineY[x]);
        }
    }

    // Traiter les colonnes de l'image resultant du traitement des lignes
    for(int x = 0; x < BLOCK_SIZE; x++) {
        double[] lineX = inverseDCT1D(fp.getLine(x + xPos,yPos,x +
            xPos, yPos + BLOCK_SIZE));
        for(int y = 0; y < BLOCK_SIZE; y++) {
            fp.putPixelValue(x + xPos,y + yPos,lineX[y]);
        }
    }
}

```

## Qualité et performance de la compression JPEG

### Influence de la quantification

```

int alpha = 0;
if((facteurQ <= 50) && (facteurQ >= 1)){
    alpha = 50/facteurQ;
}
else if((facteurQ <= 99) && (facteurQ >= 51)){
    alpha = 2 - (2*facteurQ/100);
}
else{
    alpha = 1;
}

bpQuantification.setIntArray(QY);    // initialisation
bpQuantification.multiply(alpha);

```

### Ratio de compression

```

List<Float> allCoefsBeforeEOB = new ArrayList<Float>();
for(int i = 0; i < fp.getWidth(); i = i + BLOCK_SIZE){

```

```

    for(int j = 0; j < fp.getHeight(); j = j + BLOCK_SIZE){
        float[][] region= new float[BLOCK_SIZE][BLOCK_SIZE];
        for(int x=i; x < i+BLOCK_SIZE; x++){
            for(int y=j; y < j+BLOCK_SIZE; y++){
                region[x-i][y-j] = fp.getPixel(x,y);
            }
        }
        FloatProcessor im_region = new FloatProcessor(region);
        List<Float> res_region = coefsBeforeEOB(im_region);
        allCoefsBeforeEOB.addAll(res_region);
    }
}
float[][][]tab = new float[1][allCoefsBeforeEOB.size()];
for (int p=0; p<allCoefsBeforeEOB.size();p++){
    tab[0][p]= allCoefsBeforeEOB.get(p);
}
FloatProcessor im_comp = new FloatProcessor(tab);
ImagePlus imP = new ImagePlus("All coef ",im_comp);
FileSaver fs = new FileSaver(imP);
fs.saveAsPng();

```