# Semaine 12 : Compression d'images

# TP12

Maxime CATTEAU
Léane TEXIER

## 1ère partie : Transformée en cosinus discrète

Dans ce TP, nous allons travailler sur la compression des images. Pour cela, commençons par analyser, la transformée en cosinus discrète (que l'on abregera **DCT**). Cette dernière possède les mêmes propriétés que la transformée de Fourier que nous avons étudié dans les TP précédents. Elle est utilisée dans le cadre de la compression d'images (ex. JPEG, MPEG, …). La **DCT** possède également une propriété particulièrement intéressante : elle est dite "séparable". Cela signifie globalement que sommes en mesure de séparer l'application de la **DCT** selon les deux axes de direction *x* et *y*.

### Transformée 2D directe

| |
|---|
| Examiner comment sont implémentées les transformées en cosinus discrètes directe et inverse en 1D (dans le fichier **DCT1D.java**) : expliciter l'entrée, la sortie et la formule de chacune d'elles. |

Analysons dans un premier temps les méthodes utilisées dans le fichier fourni ***DCT1D.java*** :

- Commençons par la signature de la fonction :

```
public static double[ ] forwardDCT(double[ ] f)
```

*Signature de la méthode **forwardDCT** du fichier **DCT1D.java** fourni*

Nous observons que cette méthode prend en paramètre d'entrée un tableau de réels **f**. Cela correspond en fait au signal d'entrée ***f(m)***, c'est à dire la représentation des pixels de l'image étudiée.

- Le corps de la méthode correspond à la formule de la transformée en cosinus discrète que nous avons étudié en cours. Voici le contenu de cette méthode :

```
int M = f.length;    // Taille du signal
double k = Math.sqrt(2.0 / M); // Facteur de normalisation
double[] F = new double[M];            // résulat
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;
```

*Corps de la méthode **forwardDCT** du fichier **DCT1D.java** fourni*

Il s'agit d'une application point par point de la formule du cours :

$$F(u) := \sqrt{\frac{2}{M}}\, c(u) \sum_{m=0}^{M-1} f(m) \cos\left(\pi \frac{(2m+1)u}{2M}\right)$$

Le facteur de normalisation ***cu*** (***c(u)*** dans la formule) se calcule différemment selon les cas :

$$\text{Coef. de normalisation : } c(\alpha) := \begin{cases} 1/\sqrt{2} & \text{si } \alpha = 0, \\ 1 & \text{si } \alpha \neq 0. \end{cases} \quad \text{pour } \alpha \in [u, v]$$

- La valeur de retour est un tableau de réels, tout comme le paramètre d'entrée. C'est à dire que chaque pixel F[u] aura pour valeur le calcul de la formule qui lui est appliquée dans le corps de la méthode. Ainsi, le tableau :

```
double[ ] F = new double[M];
```

Contiendra les valeurs des pixels transformés grâce à :

```
F[u] = k * cu * somme
```

et retournera le tableau rempli des nouvelles valeurs à la fin de la boucle :

```
return F;
```

En utilisant la propriété de séparabilité de la DCT, compléter la méthode de classe implémentant la transformation directe en 2D :

public static void forwardDCT(FloatProcessor f)

qui se trouve dans le fichier **DCT2D.java**. On notera que, dans un souci d'optimisation et contrairement à l'implémentation en 1D, cette méthode transforme sur place (*ang.* « in-place transform ») un tableau de valeurs transmis sous la forme d'une image **FloatProcessor**. Autrement dit, elle ne retourne pas de résultat mais écrase les données d'entrée par celles de sortie.

Comme évoqué précédemment, une propriété intéressante de la **DCT** est la **séparabilité**. Elle nous permet de séparer l'application de la **DCT** sur les deux axes directionnels *x* et *y*. Nous pouvons donc utiliser la méthode décrite lors de la question précédente, ***forwardDCT(...)***, qui implémente la **DCT** uni-directionnelle. Nous proposons cet algorithme :

```
public static void forwardDCT(FloatProcessor fp) {
	double[ ] dct1d;
	int W = fp.getWidth();
	int H = fp.getHeight();

	// Traiter les lignes
	for(int y = 0; y < H; y++){
		dct1d = DCT1D.forwardDCT(fp.getLine(0, y, W-1, y));
		for(int x = 0; x < W; x++){
			fp.putPixelValue(x, y, dct1d[x]);
		}
	}

	// Traiter les colonnes de l'image résultant du traitement des lignes
	for(int x = 0; x < W; x++){
		dct1d = DCT1D.forwardDCT(fp.getLine(x, 0, x, H-1));
		for(int y = 0; y < H; y++){
			fp.putPixelValue(x, y, dct1d[y]);
		}
	}
}
```

*Méthode **forwardDCT(...)** de la classe **DCT2D.java***

Dans cette méthode, nous traitons dans un premier temps les lignes, puis les colonnes. Remarquons que la méthode ne retourne plus rien, elle se contente seulement de "réaffecter" les pixels présents dans le *FloatProcessor fp* grâce à la méthode *putPixelValue()*.

## Validation sur un exemple

| |
|---|
| Écrire un nouveau plugin ImageJ permettant simplement de valider la DCT sur l'image 8x8 en niveau de gris ci-dessus (comparer le résultat obtenu avec celui de l'article wikipedia). |

Les codes étant maintenant réalisés soigneusement, il nous manque la mise en œuvre de ces méthodes, c'est-à-dire un plugin ImageJ qui va nous permettre de valider notre **DCT** sur une image en niveau de gris. Voici le résultat que nous obtenons :

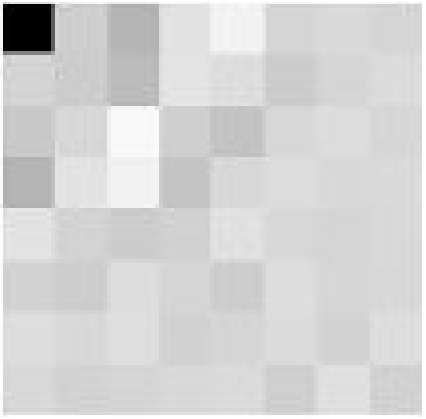


*Image obtenue après l'application du plugin **_ApplyDCT** sur l'image **wikipedia_extract.png***

Nous avons donc obtenu ce résultat en créant le plugin ***_ApplyDCT*** en n'oubliant pas de centrer les valeurs autour de 0 grâce aux lignes :

```
for(int y = 0; y < H; y++){
    for(int x = 0; x < W; x++){
      fp.setf(x, y, fp.getf(x, y) - 128);
    }
}
```

*Morceau de code permettant de centrer les valeurs autour de 0 par soustraction de 128 (qui correspond à $2^{\text{profondeur de bits - 1}}$ soit $2^{8-1}$*

Voici le code de la méthode ***run*** implémentée permettant d'obtenir le résultat ci-dessus :

```
public void run(ImageProcessor ip) {
   FloatProcessor fp = (FloatProcessor) ip.duplicate().convertToFloat();

   int W = ip.getWidth();
   int H = ip.getHeight();

   for(int y = 0; y < H; y++){
     for(int x = 0; x < W; x++){
       fp.setf(x, y, fp.getf(x, y) - 128);
     }
   }

   DCT2D.forwardDCT(fp);

   ImagePlus resultDCT = new ImagePlus("DCT of " + this.imp.getTitle(), fp);

   resultDCT.show();
}
```

*Code de la méthode **run** du plugin **_ApplyDCT.java***

Si on inspecte les pixels de l'image obtenue, par exemple les 3x3 premiers et qu'on les compare aux valeurs de références de la page Wikipédia de notre image de référence, nous observons que nous obtenons les même valeurs, ce qui valide le fonctionnement du plugin que nous avons créé.

## 2ème partie : Utilisation de la DCT pour la compression JPEG

Dans cette seconde partie, analysons comment la DCT peut être utilisée afin d'effectuer une compression JPEG.

### Traitement par blocs d'une image

Modifier la méthode **forwardDCT** (dans **DCT2D.java**) pour n'appliquer la DCT que sur la région d'intérêt (supposée définie au préalable).

L'utilisation de la transformée en cosinus directe peut s'avérer extrêmement coûteuse lorsque l'on travaille sur une image de taille conséquente (par exemple **lena.png** de taille 512x512). Le temps de calcul devient alors beaucoup trop long. Cependant, il existe une technique qui consiste à ne se préoccuper que d'une certaine région d'intérêt. Par exemple, un bloc de pixels de taille 8 comme l'image utilisée pour la question précédente.

En y définissant la constante

```
final static int BLOCK_SIZE = 8;
```

compléter le plugin pour faire tour à tour de chaque bloc 8x8 la région d'intérêt et lui appliquer la DCT.

Nous définissons la taille du bloc de pixels que nous allons utiliser pour les régions d'intérêt (**R**egion **o**f **I**nterest).

```
final static int BLOCK_SIZE = 8;
```

*Définition de la taille d'une **ROI***

Nous devons également modifier notre méthode ***forwardDCT*** *dans* ***DCT2D.java*** afin de prendre en compte les régions d'intérêt. Voici le nouveau code de la méthode que nous proposons :

```
public static void forwardDCT(FloatProcessor fp) {
	double[] dct1d;
	Rectangle rect = fp.getRoi();
	int width = rect.x + rect.width;
	int height = rect.y + rect.height;

	// Traiter les lignes
	for(int y = rect.y ; y < height; y++){
		dct1d = DCT1D.forwardDCT(fp.getLine(rect.x, y, width-1, y));
		for(int x = 0 ; x < rect.width ; x++){
			fp.putPixelValue(x + rect.x, y, dct1d[x]);
		}
	}

	// Traiter les colonnes de l'image résultant du traitement des lignes
	for(int x = rect.x ; x < width; x++){
		dct1d = DCT1D.forwardDCT(fp.getLine(x, rect.y, x, height-1));
		for(int y = 0 ; y < rect.height ; y++){
			fp.putPixelValue(x, y + rect.y, dct1d[y]);
		}
	}
}
```

*Code de la méthode **forwardDCT** prenant en compte les régions d'intérêt **ROI** de la classe **DCT2D.java***

Afin que ces régions soient prises en compte, nous devons également effectuer quelques ajustements sur notre méthode ***run*** de notre PlugIn ***_ApplyDCT.java*** :

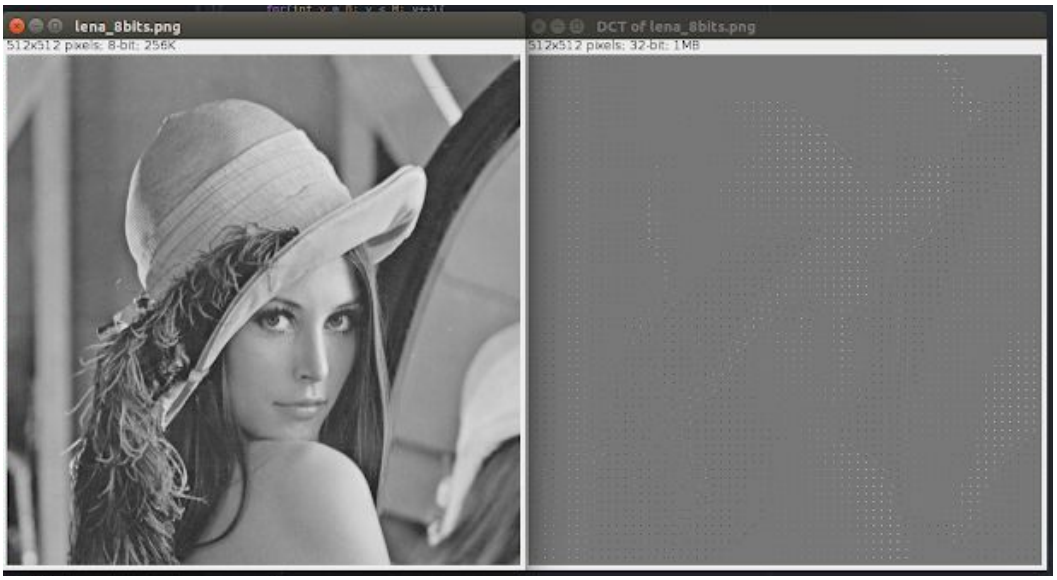
- la déclaration d'une région d'intérêt
  *Rect rect = fp.getROI();*
- l'initialisation des régions d'intérêt grâce à une boucle
- l'application de la DCT2D sur chaque région d'intérêt

```
public void run(ImageProcessor ip) {
   FloatProcessor fp = (FloatProcessor) ip.duplicate().convertToFloat();
   Rectangle rect = ip.getRoi();

   int W = ip.getWidth();
   int H = ip.getHeight();

   for(int rect.y = 0; y < H; y++){
     for(int rect.x = 0; x < W; x++){
       fp.setf(x, y, fp.getf(x, y) - 128);
     }
   }

   int rectHeight = rect.height;
   int rectWidth = rect.width;

   List<Rectangle> subRects = new ArrayList<Rect>();

   for(int y = 0 ; y < Math.ceil(rectHeight / BLOCK_SIZE) ; y++) {
     for(int x = 0 ; x < Math.ceil(rectWidth / BLOCK_SIZE) ; x++) {
           subRects.add(new Rectangle(rect.x + 8 * x, rect.y + 8 * y, Math.min(Math.abs(rectWidth - 8 * x),
BLOCK_SIZE), Math.min(Math.abs(rectHeight - 8 * y), BLOCK_SIZE)));
     }
   }

   for(Rectangle subRect : subRects) {
     ipFloat.setRoi(subRect);
     DCT2D.forwardDCT(ipFloat);
   }

   ImagePlus resultDCT = new ImagePlus("DCT of " + this.imp.getTitle(), fp);

   resultDCT.show();
}
```

*Code de la méthode **run** prenant en compte les régions d'intérêt **ROI** du plugin **_ApplyDCT.java***

Tester le code précédent sur l'image 8 bits ci-dessous (« Lena »). Interpréter le résultat (image de la DCT) et commenter les coefficients DCT obtenus pour 2 ou 3 blocs caractéristiques de l'image.

Nous sommes maintenant en mesure de tester le code du plugin créé sur une image de taille conséquente grâce aux modifications apportées. Nous testons donc l'algorithme sur l'image ***lena.png*** :




*À gauche, l'image de référence **lena.png** en 8-bits, à droite le résultat de l'application de la DCT avec les régions d'intérêt de taille 8x8*

Nous pouvons constater que l'image de référence est plutôt reconnaissable sur l'image obtenue après l'application de la DCT avec régions d'intérêt. De plus, le temps de calcul était relativement correct. Si l'on analyse quelques blocs significatifs de l'image, nous pouvons remarquer quelques singularités : le premier pixel de chaque bloc représente par exemple le coefficient DCT. C'est-à-dire qu'il concentre la majeure partie de l'énergie. Il s'agit en fait de réduire la quantité d'informations liées aux hautes fréquences car nous savons que l'œil humain est plus sensible aux basses fréquences. C'est globalement pour cette raison que sur l'image obtenue après le traitement, nous pouvons beaucoup mieux voir le chapeau (coefficients élevés) plutôt que les cheveux (coefficients faibles).

## Quantification

| |
|---|
| Implémenter l'étape de quantification et valider cette étape grâce à l'exemple de l'article wikipedia (pour arrondir une valeur réelle à l'entier inférieur, utiliser la méthode **Math.round**). |

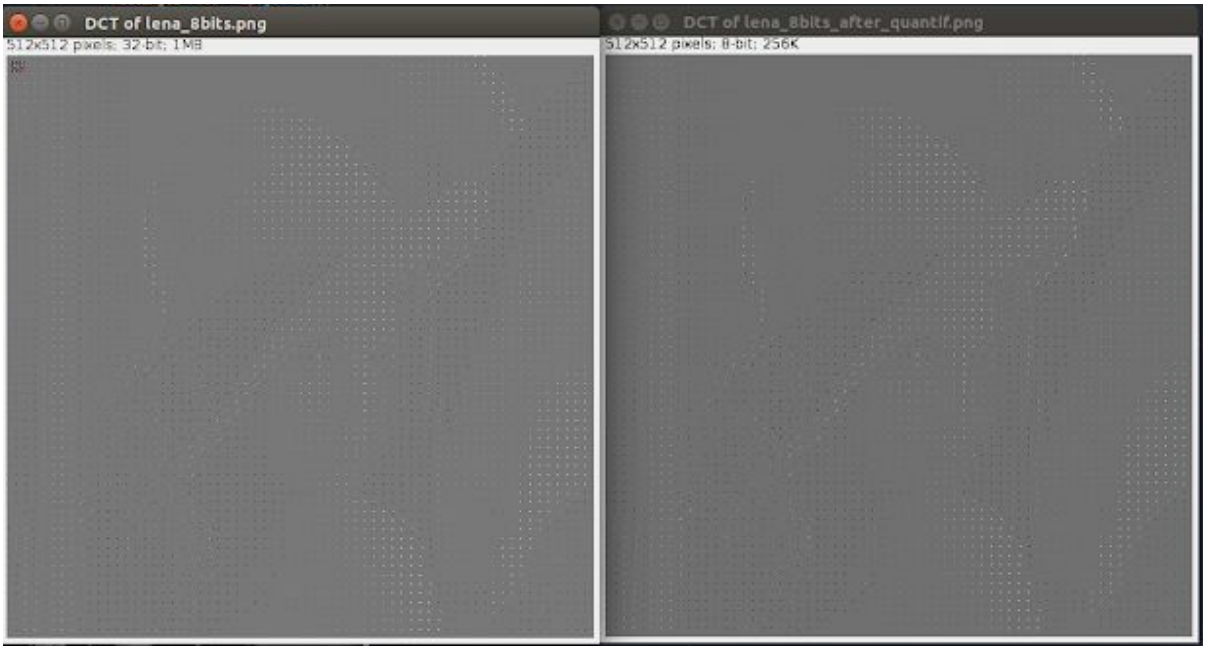
Dans le but de ne garder qu'une partie des coefficients DCT, nous devons implémenter la matrice de quantification JPEG :

```
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}
};
```

*Déclaration de la matrice de quantification JPEG*

Comme évoqué plus haut, l'œil humain est plus sensible aux basses fréquences. C'est pourquoi nous n'allons garder que les coefficients des fréquences basses. C'est la caractéristique "destructive" de JPEG. Chaque valeur de pixel va donc être divisé par sa valeur associée dans la matrice de quantification JPEG.

Après avoir effectué les quelques modifications dans le code de la méthode ***run*** (voir l'intégralité du code de cette méthode en **Annexe 1**), nous obtenons le résultat suivant :




*A gauche l'image **lena.png** avant la quantification, à droite la même image après quantification*

Nous constatons très peu de différences à l'œil nu, toutefois, si l'on observe la valeur des pixels grâce à l'outil Pixel Inspector, nous pouvons observer que les coefficients correspondant aux hautes fréquences ont bien été annulés.

## Décompression

Dans **DCT2D.java**, implémenter la méthode
public static void inverseDCT(FloatProcessor f)
qui réalise la DCT inverse, toujours en utilisant la propriété de séparabilité.

Pour implémenter la méthode ***inverseDCT*** représentant la DCT inverse, il est nécessaire de s'inspirer de la formule sur la DCT inverse du cours :

$$f(m) := \sqrt{\frac{2}{M}} \sum_{u=0}^{M-1} c(u) F(u) \cos\left(\pi \frac{(2m+1)u}{2M}\right)$$

De plus, il est utile d'utiliser également la méthode ***forwardDCT*** écrite dans les questions précédentes. Voici le code associé :

```
public static void inverseDCT(FloatProcessor fp) {
	double[] invDct1d;
	Rectangle rect = fp.getRoi();

	int width = rect.x + rect.width;
	int height = rect.y + rect.height;

	// Traiter les lignes
	for(int y = rect.y; y < height; y++){
		invDct1d = DCT1D.inverseDCT(fp.getLine(rect.x, y, width-1, y));
		for(int x = 0; x < width; x++){
			fp.putPixelValue(x + rect.x, y, invDct1d[x]);
		}
	}
	// Traiter les colonnes de l'image résultant du traitement des lignes
	for(int x = rect.x; x < width; x++){
		invDct1d = DCT1D.inverseDCT(fp.getLine(x, rect.y, x, height-1));
		for(int y = 0; y < height; y++){
			fp.putPixelValue(x, y + rect.y, invDct1d[y]);
		}
	}
}
```

*Code de la méthode **inverseDCT** permettant de réaliser la DCT inverse*

Nous n'avons pour l'instant implémenté qu'une seule étape du processus de décompression que l'on pourrait décomposer en trois étapes principales :

1. Rétablir les coefficients que nous avons quantifié
2. Appliquer la méthode **inverseDCT**
3. Décentrer les valeurs en ajoutant 128 à celles-ci

Il faut maintenant implémenter ce processus dans le plugin ***_ApplyDCT.java***

Dans le plugin, à la suite du processus de compression, implémenter le processus de décompression bloc par bloc puis représenter le résultat. Tester ces étapes de compression-décompression sur l'image de l'article wikipedia et commenter le résultat en le comparant à l'image originale.

Pour implémenter le processus de décompression, nous modifions la méthode **run** afin de différencier la compression et la décompression, qui est le processus inverse de la compression (que nous avons décrit au préalable dans la partie précédente). Voici le code représentant ce processus :

```
public void run(ImageProcessor ip) {
   FloatProcessor fp = (FloatProcessor) ip.duplicate().convertToFloat();
   Rectangle rect = ip.getRoi();

   int W = ip.getWidth();
   int H = ip.getHeight();

   int rectHeight = rect.height;
   int rectWidth = rect.width;


   List<Rectangle> subRects = new ArrayList<Rectangle>();

   for(Rectangle subRect : subRects) {
     fp.setRoi(subRect);
     fp.copyBits(bpQuantification.convertToFloat(), subRect.x, subRect.y, Blitter.MULTIPLY);
     DCT2D.inverseDCT(fp);
   }

   for(int y = rect.y; y < rect.y + rectHeight; y++){
     for(int x = rect.x; x < rect.x + rectWidth; x++){
       fp.setf(x, y, fp.getf(x, y) + 128);
     }
   }

   ImagePlus resultDCT = new ImagePlus("Inverted DCT of " + this.imp.getTitle(), fp);

   resultDCT.show();
}
```

*Code de la méthode **run** du plugin **_ApplyDCT.java** permettant de réaliser la décompression d'une image*

Malheureusement, un problème est présent dans cet algorithme dont la provenance est inconnue ce qui nous empêche d'en déduire des interprétations et par la même occasion de tester sur l'image échantillon de Wikipédia.

Nous pouvons cependant imaginer que l'image décompresser aurait eu quelques erreurs bien visibles sur l'image échantillon. Ces erreurs seraient probablement dues au nombre de coefficients présents dans la matrice de quantification (comme nous avons pu le

voir dans le cours, 1 coefficient représentant une image unicolore). Ainsi, plus le nombre de coefficients est élevé, plus l'image décompressée sera identique à l'image originale.

## 3ème partie : Qualité et performance de la compression JPEG

Dans cette dernière partie, nous allons voir comment évaluer la qualité et la performance de la compression JPEG.

### Évaluation de la distorsion

Sur l'image « Lena », appliquer la compression/décompression, puis évaluer la distorsion en utilisant simplement les fonctionnalités d'ImageJ disponibles dans les menus (par exemple, *Process*/*Image Calculator*/*Difference*, qui retourne la différence absolue pixel à pixel, *Process*/*Math*/* qui permet diverses opérations et *Analyse*/*Measure* qui donne des statistiques sur l'image entière).

Nous allons maintenant tenter d'évaluer la distorsion en utilisant des fonctionnalités d'ImageJ déjà définies et disponibles dans les menus. Nous avons pu utiliser certaines d'entre elles tout au long du module de **T**raitement d'**I**mages. Commençons par l'erreur absolue moyenne (**MAE**). La formule de la **MAE** est :

$$MAE = 1/MN \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} |Y_{m,n} - \hat{Y}_{m,n}|$$

Une autre formule d'évaluation de la qualité de la compression JPEG est l'erreur quadratique moyenne (**MSE**), on associe cette valeur au **P**eak **S**ignal-to-**N**oise **R**atio (**PSNR**), qui permet d'obtenir une valeur en décibels témoignant de la proximité entre une image estimée et une image de base. Voici la formule de la **MSE** :

$$MSE = 1/MN \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} (Y_{m,n} - \hat{Y}_{m,n})^2$$

et celle du **PSNR** :

$$PSNR = 10 \cdot \log_{10}(255^2/MSE)$$

Pour déterminer la **MAE**, on peut essayer d'établir un raisonnement en lien avec la formule :

1. Soustraire l'image estimée à l'image originale ce qui correspondrait au morceau "$Y_{m,n} - \hat{Y}_{m,n}$" de la formule décrite plus haut, nous pouvons effectuer cela grâce à la fonctionnalité ***Process/Image Calculator*** et sélectionner l'opérateur ***Substract***.
2. Mettre les pixels obtenus en valeur absolue. Nous pouvons utiliser la fonctionnalité ***Process/Math/Abs*** pour effectuer cette opération.
3. Récupérer la moyenne

Puisque nous n'avons pas trouvé la solution à notre problème lors de la question précédente, la valeur que nous trouvons n'est pas significative. Passons maintenant au calcul de la **MSE** :

1. Soustraire l'image estimée à l'image originale ce qui correspondrait au morceau "$Y_{m,n}$ - $\hat{Y}_{m,n}$" de la formule, nous pouvons effectuer cela grâce à la fonctionnalité ***Process/Image Calculator*** et sélectionner l'opérateur ***Substract***.
2. Mettre chaque pixel au carré grâce à la fonctionnalité ***Process/Math/Square***
3. Récupérer la moyenne

Même remarque que pour la **MAE**, la valeur obtenue n'est pas significative. Finissons avec le **PSNR** :
Il s'agit cette fois d'une application de la formule grâce au résultat obtenu pour la **MSE** :

$PSNR = 10 * \log_{10}(255^2/\textbf{MSE})$

Le reste du TP se repose sur les résultats obtenus, nous ne pouvons donc pas continuer plus loin.

## Annexe 1

```
public void run(ImageProcessor ip) {
   FloatProcessor fp = (FloatProcessor) ip.duplicate().convertToFloat();
   Rectangle rect = ip.getRoi();

   int W = ip.getWidth();
   int H = ip.getHeight();

   int rectHeight = rect.height;
   int rectWidth = rect.width;

   for(int y = rect.y; y < rect.y + rectHeight; y++){
     for(int x = rect.x; x < rect.x + rectWidth; x++){
       fp.setf(x, y, fp.getf(x, y) - 128);
     }
   }

   List<Rectangle> subRects = new ArrayList<Rectangle>();

   for(int y = 0 ; y < Math.ceil(rectHeight / BLOCK_SIZE) ; y++) {
     for(int x = 0 ; x < Math.ceil(rectWidth / BLOCK_SIZE) ; x++) {
       subRects.add(new Rectangle(rect.x + 8 * x, rect.y+8*y, Math.min(Math.abs(rectWidth - 8 * x),
BLOCK_SIZE), Math.min(Math.abs(rectHeight - 8 * y), BLOCK_SIZE)));
     }
   }

   ImageProcessor quantification = bpQuantification.convertToFloat();

   for(Rectangle subRect : subRects) {
     fp.setRoi(subRect);
     DCT2D.forwardDCT(fp);
     fp.copyBits(quantification, subRect.x, subRect.y, Blitter.DIVIDE);
   }

   ImagePlus resultDCT = new ImagePlus("DCT of " + this.imp.getTitle(), fp);

   resultDCT.show();
}
```

*Code de la méthode **run** pour l'exercice sur la quantification dans le plugin **_ApplyDCT.java***