# Semaine 7 : Transformations locales des images
# TP7

Maxime CATTEAU
Léane TEXIER

## 1ère partie : Introduction

**Note :** La première partie nous a permis de nous familiariser avec l'outil Pixel Inspector proposé par le logiciel ImageJ. Cet outil permet de visualiser la valeur des pixels d'une image.

## 2ème partie : Masque de convolution

Dans le domaine spectral, le filtrage se fait par multiplication. Tandis que, dans le domaine spatial, il se fait par convolution. Un masque de convolution est un filtre linéaire permettant de modifier l'image. L'effet de ce filtre est déterminé selon ses caractéristiques (coefficients, taille). Il s'agit de manière générale d'une matrice carrée (de taille impaire) qui a l'avantage de pouvoir être centrée sur le pixel d'analyse étant donné que sa taille est impaire. De plus, elle est généralement à valeurs symétriques centrées.

Définition et application d'un filtre moyenneur

Avant d'appliquer un filtre moyenneur sur notre image, nous avons tout d'abord analysé notre image de base **I2** en observant sa taille et la valeur de ces pixels.


*Image I2 de base*

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 180 | 180 | 180 | 0 | 0 |
| 0 | 0 | 180 | 180 | 180 | 0 | 0 |
| 0 | 0 | 180 | 180 | 180 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Tableau de pixels de l'image I2*

L'image **I2** étudiée est une image en niveau de gris et qui est carrée de 7 pixels sur 7. Ces 9 valeurs centrales sont à 180 (couleur grise qui tend vers le blanc) et ces autres valeurs sont toutes à 0 (valeur correspondant à la couleur noire).

On définit un filtre **H1** afin d'étudier la convolution de l'image **I2**. Ce filtre n'aura pas la propriété "*normalize kernel*", qui sera désactivée pour cette étape. Cette propriété signifie que nous ne nous préoccupons pas de la normalisation des valeurs du filtre. La somme des coefficients n'est donc pas nécessairement égale à 1, cela implique donc qu'en appliquant le filtre, il est possible de nous retrouver avec une valeur supérieure à 255 qui est la valeur maximale d'un pixel. Dans ce cas là, la valeur est alors mise à 255 (soit à la valeur maximale possible).

Filtre **H1** =

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

Appliquer ce filtre (**H1**) sur l'image **I2** et interpréter le résultat.


*Résultat de l'application du filtre **H1** sur l'image **I2***

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 180 | 255 | 255 | 255 | 180 | 0 |
| 0 | 255 | 255 | 255 | 255 | 255 | 0 |
| 0 | 255 | 255 | 255 | 255 | 255 | 0 |
| 0 | 255 | 255 | 255 | 255 | 255 | 0 |
| 0 | 180 | 255 | 255 | 255 | 180 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Tableau de pixels de l'image I2 après application du filtre **H1***

On observe 3 valeurs distinctes : **0** (noir), **180** (gris qui tend vers le blanc) et **255** (blanc). Ces valeurs sont obtenues après avoir effectué un calcul de convolution discrète. Voici les étapes à effectuer pour chaque pixel de l'image d'origine :

**Étape 1**

Il faut tourner la matrice de poids, ici **H1**, à 180° autour de son centre. Le résultat de cette rotation nous fournit cette nouvelle matrice $\mathbf{H1}_{\theta=180°}$ :

Filtre $\mathbf{H1}_{\theta=180°}$ =

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

Dans ce cas, la matrice est symétrique donc $\mathbf{H1}_{\theta=180°} = \mathbf{H1}$.

**Étape 2**

On superpose le masque obtenu à l'image **I2** afin qu'il soit centré en (*x, y*) (= coordonnées du pixel d'analyse), c'est-à-dire que le pixel d'analyse (i.e. celui sur lequel nous sommes en train de travailler) doit être superposé à la valeur correspondante au centre de la matrice $\mathbf{H1}_{\theta=180°}$.

**Étape 3**

On multiplie le coefficient du masque par le niveau du pixel associé.

Dans notre cas, prenons par exemple le pixel de coordonnées (*0, 0*) de l'image **I2** (image de base), sa valeur est 0. De plus, il s'agit d'un pixel situé au bord de notre image, on doit donc traiter ce problème d'effet de bord en utilisant la stratégie du **miroir (virtuel) de l'image** qui consiste à donner la valeur du pixel d'analyse à son voisin "virtuel" dans la matrice de voisinage. Nous obtenons ainsi la matrice de voisinage **3x3** du pixel situé en (*0, 0*) de l'image **I2** :

Matrice de voisinage 3x3 du pixel (*0, 0*) =

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

**Étape 4**

On additionne chacun des produits trouvés. L'ensemble de la formule est visible ci-dessous :

$$\mathbf{I2'}_{(0,\,0)} = \mathbf{I2}_{(x-1,\,y-1)} \cdot \mathbf{H1}_{\theta=180°\,(-1,\,-1)} + \mathbf{I2}_{(x-1,\,y)} \cdot \mathbf{H1}_{\theta=180°\,(-1,\,0)} + \mathbf{I2}_{(x-1,\,y+1)} \cdot \mathbf{H1}_{\theta=180°\,(-1,\,1)} + \mathbf{I2}_{(x,\,y-1)} \cdot \mathbf{H1}_{\theta=180°\,(0,\,-1)} + \mathbf{I2}_{(x,\,y+1)} \cdot \mathbf{H1}_{\theta=180°\,(0,\,0)} + \mathbf{I2}_{(x,\,y+1)} \cdot \mathbf{H1}_{\theta=180°\,(0,\,1)} + \mathbf{I2}_{(x+1,\,y-1)} \cdot \mathbf{H1}_{\theta=180°\,(1,\,-1)} + \mathbf{I2}_{(x+1,\,y)} \cdot \mathbf{H1}_{\theta=180°\,(1,\,0)} + \mathbf{I2}_{(x+1,\,y+1)} \cdot \mathbf{H1}_{\theta=180°\,(1,\,1)} .$$

Si on remplace le tout par les valeurs correspondantes, on obtient alors : $\mathbf{I2'}_{(0, 0)}$ = 0 **.** 1 + 0 **.** 1 + 0 **.** 1 + 0 **.** 1 + 0 **.** 1 + 0 **.** 1 + 0 **.** 1 + 0 **.** 1 + 0 **.** 1 = **0**. Donc le pixel de coordonnées (*0, 0*) aura pour valeur de niveau **0** sur l'image **I2'**.

On peut effectuer le même processus avec les valeurs des autres pixels. Prenons par exemple le pixel de coordonnées (*2, 1*) sur l'image **I2**. On a alors la matrice de voisinage ci-dessous pour ce pixel :

Matrice de voisinage 3x3 du pixel (*2, 1*) =

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 180 | 180 |

En appliquant la même formule que précédemment, on obtient ainsi : $\mathbf{I2'}_{(2, 1)}$ **=** 0 **.** 1 + 0 **.** 1 + 0 **.** 1 + 0 **.** 1 + 0 **.** 1 + 0 **.** 1 + 0 **.** 1 + 180 **.** 1 + 180 **.** 1 = **360**. La nouvelle valeur du pixel est alors mise à **255** (qui correspond la valeur de niveau maximale pour un pixel).

Si on vérifie sur le tableau de pixels de **I2'** obtenu grâce à l'outil **Pixel Inspector**, on s'aperçoit bien que le pixel en (*0, 0*) vaut **0** et que le pixel en (*2, 1*) vaut **255**.

Appliquer ce filtre sur l'image I2, mais cette fois en cochant la case Normalize kernel. Quel est le filtre appliqué à présent ? Déterminer le coefficient de normalisation du filtre en considérant le niveau d'un pixel donné de l'image filtrée.

Ici, nous allons appliquer le même filtre que précédemment mais en cochant la case Normalize kernel, cela permet alors de normaliser les valeurs. Dans notre cas, les coefficients du filtre **H1** sont tous positifs, il suffit donc de faire la somme **S** de ces valeurs et de prendre son inverse pour retrouver le coefficient de normalisation :

**S =** $H1_{(-1, -1)} + H1_{(-1, 0)} + H1_{(-1, 1)} + H1_{(0, -1)} + H1_{(0, 0)} + H1_{(0, 1)} + H1_{(1, -1)} + H1_{(1, 0)} + H1_{(1, 1)}$ **=** 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 +1 **= 9**

Le coefficient de normalisation est donc **1/9**.


*Résultat de l'application du filtre sur l'image I2 avec l'option "Normalize Kernel"*

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 20 | 40 | 60 | 40 | 20 | 0 |
| 0 | 40 | 80 | 120 | 80 | 40 | 0 |
| 0 | 60 | 120 | 180 | 120 | 60 | 0 |
| 0 | 40 | 80 | 120 | 80 | 40 | 0 |
| 0 | 20 | 40 | 60 | 40 | 20 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Tableau de pixels de l'image I2 après application du filtre avec l'option "Normalize Kernel"*

Afin de trouver ces nouvelles valeurs, il nous suffit de reprendre la même formule utilisée lors de la question précédente et de multiplier le résultat par le coefficient **1/9**.

Par exemple pour $\mathbf{I2'}_{(0,0)}$ on aura **0 * (1 / 9) = 0**. De la même manière, pour $\mathbf{I2'}_{(2,1)}$ on aura **360 * (1 / 9) = 360 / 9 = 40**. Nous pouvons confirmer le résultat de ces calculs grâce au tableau de pixels ci-dessus obtenus grâce à l'outil **Pixel Inspector**.

Effets de bords

Donner le niveau 180 à tous les pixels de la première colonne de l'image **I2** pour obtenir l'image **I3**.

Afin d'obtenir l'image **I3** nous avons modifié l'image **I2** en changeant toutes les valeurs des pixels de la première colonne et en mettant leurs valeurs à 180 (qui est la la valeur des pixels situés au centre de l'image). Nous avons ainsi obtenus l'image et le tableau de pixels correspondant suivant :

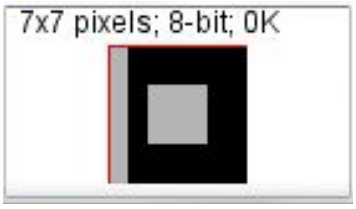



*Image **I3** = **I2** après la mise à niveau de tous les pixels de la première colonne à 180*

| | | | | | | |
|---|---|---|---|---|---|---|
| 180 | 0 | 0 | 0 | 0 | 0 | 0 |
| 180 | 0 | 0 | 0 | 0 | 0 | 0 |
| 180 | 0 | 180 | 180 | 180 | 0 | 0 |
| 180 | 0 | 180 | 180 | 180 | 0 | 0 |
| 180 | 0 | 180 | 180 | 180 | 0 | 0 |
| 180 | 0 | 0 | 0 | 0 | 0 | 0 |
| 180 | 0 | 0 | 0 | 0 | 0 | 0 |

*Tableau de pixels de l'image **I3** obtenue après la modification de l'image **I2***

Appliquer sur **I3** le filtre H1 normalisé.

Puis, nous avons appliqué le filtre **H1** normalisé (i.e. la somme de ces coefficient est égale à 1) sur cette image (**I3**). Nous avons alors obtenus l'image et le tableau de pixels correspondant suivant :


*Image **I3** après application du filtre **H1** normalisé*

| | | | | | | |
|---|---|---|---|---|---|---|
| 120 | 60 | 0 | 0 | 0 | 0 | 0 |
| 120 | 80 | 40 | 60 | 40 | 20 | 0 |
| 120 | 100 | 80 | 120 | 80 | 40 | 0 |
| 120 | 120 | 120 | 180 | 120 | 60 | 0 |
| 120 | 100 | 80 | 120 | 80 | 40 | 0 |
| 120 | 80 | 40 | 60 | 40 | 20 | 0 |
| 120 | 60 | 0 | 0 | 0 | 0 | 0 |

*Tableau de pixels de l'image **I3** après application du filtre **H1** normalisé*

En examinant les niveaux de la première colonne de l'image I3 filtrée, déduire comment ImageJ traite les pixels du bord de l'image lors d'une convolution.

Si on examine les niveaux de la première colonne de l'image **I3** après le filtrage, on s'interroge sur la façon d'on ImageJ va traiter les effets de bords lors de la convolution. Nous avons étudié en cours plusieurs stratégies possibles comme **laisser les pixels inchangés**, **mettre la valeur de ces pixels à 0**, **miroir (virtuel) de l'image** ou encore **l'enroulement**.

Cela ne peut pas être la première solution, qui consiste à laisser inchangés les valeurs de ces pixels, puisque les pixels ont subi une modification. En effet, prenons par exemple, le pixel (0,0) qui est un pixel situé sur un bord sur l'image **I3** sa valeur est de **180** tandis que sa valeur sur l'image **I3** après application du filtre **H1** normalisé est de **120**.

Cela n'est pas non plus la stratégie de l'enroulement d'image, qui consiste à récupérer les valeurs des pixels du bord opposé afin d'effectuer le calcul. En effet, prenons par exemple le pixel (0,0) de notre image, on aurait ainsi eu la matrice de voisinage suivante pour ce pixel :

Matrice de voisinage 3x3 du pixel (*0, 0*) avec la stratégie de l'enroulement =

| | | |
|---|---|---|
| 0 | 180 | 0 |
| 0 | 180 | 0 |
| 0 | 180 | 0 |

D'après le calcul, on aurait ainsi eu :

**I3'**$_{(0, 0)}$ **=** (0 **.** 1 + 180 **.** 1 + 0 **.** 1 + 0 **.** 1 + 180 **.** 1 + 0 **.** 1 + 0 **.** 1 + 180 **.** 1 + 0 **.** 1) / 9 = **720 / 9** = **80**.

Ce qui ne correspond pas à la valeur retrouvée après l'application du filtre **H1** normalisé sur l'image **I3** qui est de **120**.

Ce n'est pas non plus la solution qui consiste à mettre les pixels "manquants" à 0. En effet, prenons par exemple le pixel (0,0) de notre image, on aurait ainsi obtenu la même matrice que celle pour la méthode d'enroulement. Cela implique donc qu'on aurait eu la même valeur pour ce pixel en effectuant le calcul, c'est-à-dire **80** et non **120** qui est la valeur du pixel (0,0) de l'image **I3** après l'application du filtre **H1** normalisé.

Considérons pour le calcul suivant qu'il s'agit de la stratégie du **miroir (virtuel) de l'image** :

Prenons, par exemple, le pixel situé en (0,0) de l'image **I3**. Nous obtenons, avec cette stratégie, la matrice de voisinage ci-dessous :

Matrice de voisinage 3x3 du pixel (*0, 0*) =

| | | |
|---|---|---|
| 180 | 180 | 0 |
| 180 | 180 | 0 |
| 180 | 180 | 0 |

On utilise la même formule de calcul que dans les questions précédentes en n'oubliant pas le coefficient de normalisation ( **1 / 9** ), on obtient ainsi :

**I3'**$_{(0, 0)}$ **=** (180 **.** 1 + 180 **.** 1 + 0 **.** 1 + 180 **.** 1 + 180 **.** 1 + 0 **.** 1 + 180 **.** 1 + 180 **.** 1 + 0 **.** 1) / 9 = **1080 / 9** = **120**.

Nous avons donc retrouvé par le calcul la valeur de **I3'**$_{(0, 0)}$ obtenue après l'application du filtre **H1**. Cette même stratégie peut alors être effectué sur les autres pixels situés sur les bords. Nous ne le détaillerons pas, ici, mais, dans ce cas, cette stratégie fonctionne pour chaque pixel situé au bord de cette image.

Nous pouvons donc en déduire que la stratégie de gestion des bords utilisée par ImageJ semble être celle du **miroir (virtuel) de l'image**.

# 3ème partie : Bruits et débruitage d'image

## Bruit Gaussien d'une image synthétique

Afin de faire cette partie, nous avons créé une image de taille **256x256**, de **8** bits de profondeur, et dont les pixels ont tous un niveau égal à **127**. Suite à cela, nous avons mis du bruit sur l'image.

Grâce au menu **Process/Noise/Add Noise**, ajouter un bruit à cette image. En visualisant l'histogramme de l'image résultante, caractériser le bruit ainsi créé (type et écart-type).

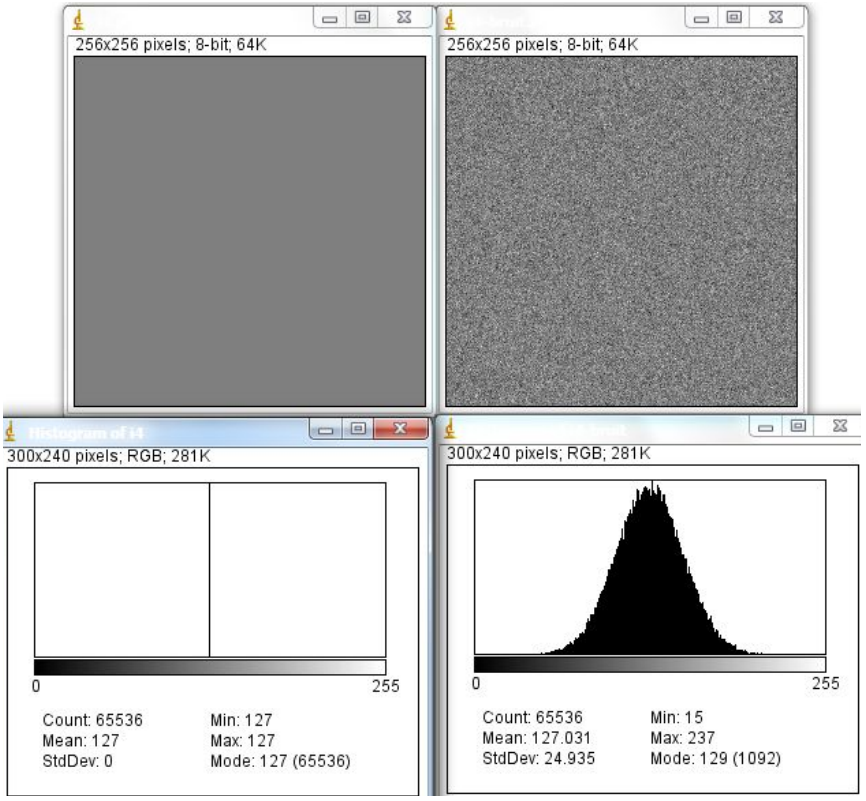



*Image de base **i4** et image bruité à partir d'**i4** avec leur histogramme associé*

D'après les informations indiquées en-dessous de l'histogramme relatif à l'image bruitée, on remarque qu'il y a un écart-type d'environ 25 (valeur de StdDev = Standard Deviation) pour cette image. Le bruit créé est un bruit gaussien. Ces informations peuvent être retrouvées dans la documentation de la commande imageJ **'Process/Noise/Add Noise'** qui nous indique que le bruit créé est un bruit gaussien avec une moyenne nulle et un écart-type de 25. Pour rappel, l'écart-type d'une image détermine le degré de lissage de l'image. Cela explique donc que pour l'image uniforme **I4**, l'écart-type est de 0 étant donné que c'est une image dont tous les pixels ont la même valeur.

Dupliquer ensuite l'image bruitée puis appliquer à cette image le filtre moyenneur H1 (normalisé). Conclure.

Afin de débruiter l'image précédemment bruitée, nous avons appliqué à l'image bruitée le filtre **H1** normalisé.

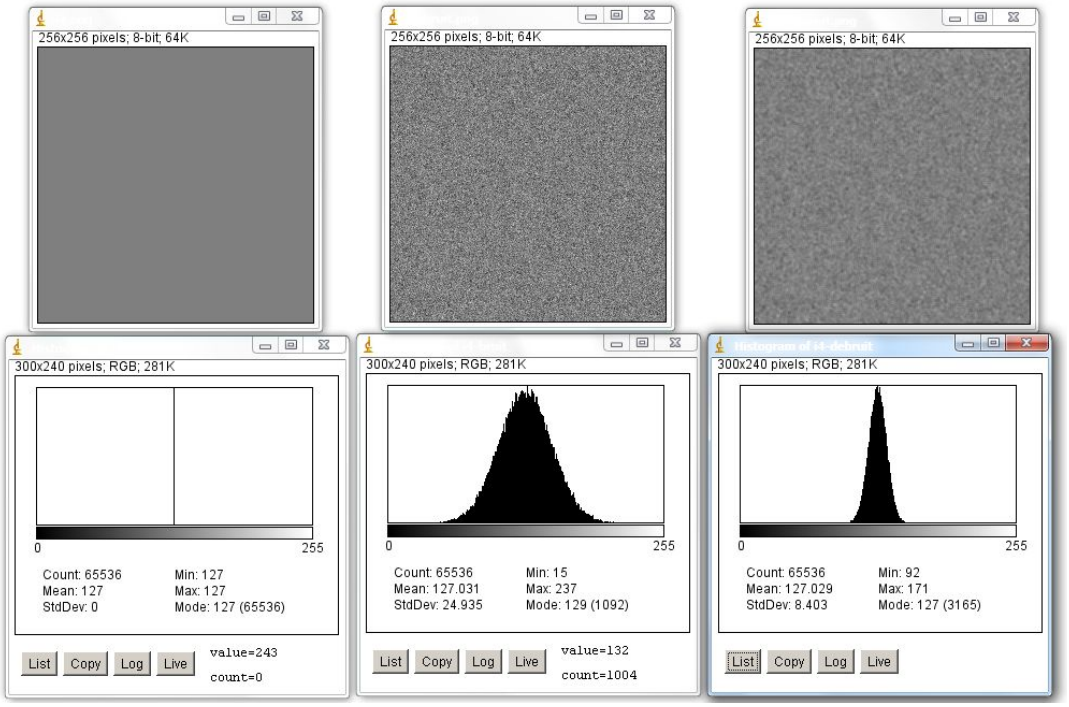



*Image de base **i4**, image bruité à partir d'**i4** et image débruité avec leur histogramme associé*

Grâce à l'histogramme correspondant à l'image débruitée, on remarque que les valeurs minimale et maximale se sont "rapprochées" de celles de l'image de base. De plus, l'écart-type de l'image débruitée est d'environ 8,4 tandis que pour l'image bruitée il était d'environ 25. Étant donné que l'écart-type est réduit, l'image est ainsi plus lissée.

On peut également observer la différence des images en regardant le **PSNR** ( = Peak Signal to Noise Ratio) entre les images deux à deux. Cela permet de mesurer la similarité entre 2 images. Plus la valeur est grande, plus les images sont similaires. Voici les mesures obtenus avec nos images :

PSNR(i4.png,i4-bruit.png) = 20.1946

PSNR(i4.png,i4-debruit.png) = 29.6425

PSNR(i4-bruit.png,i4-debruit.png) = 20.7277

On remarque, ainsi, que l'image **i4** et **i4-debruit** sont les plus similaires.

Pour conclure, l'application du filtre **H1** normalisé sur une image bruitée avec un bruit gaussien d'écart-type 25 permet d'atténuer les bruits même s'il en reste quand même beaucoup. On peut d'ailleurs le remarquer très facilement à l'oeil humain puisque l'image débruitée est très différente de l'image de base **I4**.

Filtres moyenneurs sur une image naturelle

| |
|---|
| Grâce au menu **Process/Noise/Add Specified Noise...**, ajouter un bruit Gaussien d'écart-type σ=10 à l'image gateaux1.png. Comment l'histogramme de l'image est-il modifié? |

Dans cette partie, nous avons travaillé sur l'ajout d'un bruit gaussien d'écart-type 10 sur une image. Nous avons ainsi obtenu l'image ci-dessous (avec son histogramme associé).

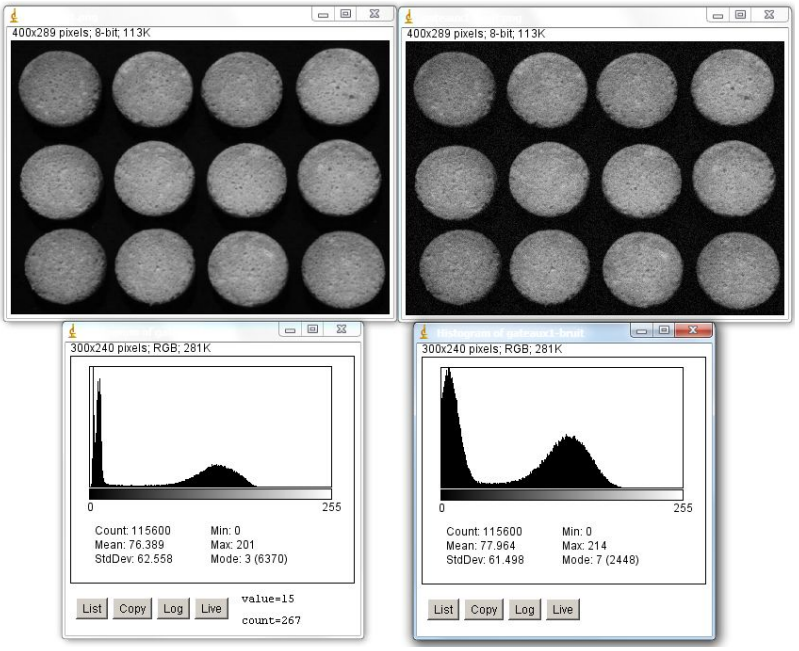


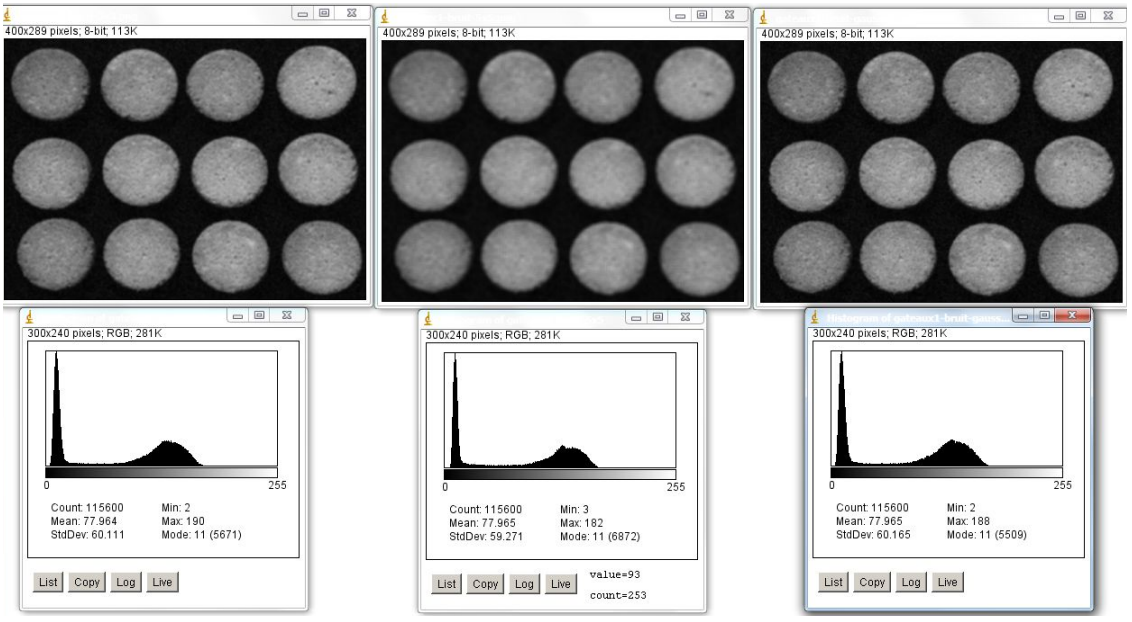

*Image de base **gateaux1.png** et, image bruité à partir de **gateaux1.png** avec leur histogramme associé*

Sur l'histogramme de l'image modifiée, on remarque que les valeurs sont plus éparpillées. Cela se remarque, tout d'abord, par la valeur maximale qui est à **214** sur l'image bruitée contre **201** pour l'image originale. De plus, on remarque grâce au 'Mode' que les valeurs sont répartis plus uniformément : le nombre de pixels qui était faible pour une valeur (entre le min et le max) a alors augmenté et à l'inverse le nombre de pixels qui était élevé pour une valeur a alors diminué.

| |
|---|
| Dupliquer ensuite l'image bruitée 3 fois.<br>* Appliquer à l'image bruitée le filtre moyenneur 3x3 (H1 normalisé).<br>* Appliquer à l'image bruitée le filtre moyenneur 5x5 normalisé dont tous les coefficients sont égaux à 1.<br>* Appliquer à l'image bruitée le filtre Gaussien d'écart-type σ=0.8 défini grâce au menu Process/Filters/Gaussian Blur…<br>Comparer les résultats obtenus. |

Afin de comparer différentes manières de débruitage, nous avons appliqué différents filtres sur l'image précédemment bruitée afin de comparer les résultats obtenus. Nous avons

appliqué un filtre moyenneur normalisé dont tous les coefficient sont 1 et de taille 3x3. Nous avons fait pareil mais avec un filtre de taille 5x5. Nous avons également appliqué sur l'image bruitée un filtre gaussien d'écart-type 0,8. Voici les résultats ainsi obtenus :




*(Dans l'ordre) Image debruitée de **gateaux1.png** bruitée grâce au filtre moyenneur 3x3, au filtre moyenneur 5x5 et au filtre gaussien d'écart-type 0,8 avec leur histogramme associé*

En comparant les différents résultats, à l'oeil humain, nous remarquons que l'image débruitée avec le filtre moyenneur 5x5 semble floue. Afin de comparer les images, il peut être utile de comparer les PSNR.

PSNR entre l'image normale et l'image affectée par le bruit gaussien d'écart-type 10

**PSNR(gateaux1.png, gateaux1-bruit.png) = 28.6117**

PSNR entre l'image normale et le résultat après application du filtre moyenneur 3x3 sur l'image bruitée

**PSNR(gateaux1.png, gateaux1-3x3.png) =31.3908**

PSNR entre l'image normale et le résultat après application du filtre moyenneur 5x5 sur l'image bruitée

**PSNR(gateaux1.png,) = PSNR(gateaux1.png,gateaux1-5x5.png) = 28.8782**

PSNR entre l'image normale et le résultat après application du filtre gaussien d'écart-type 0,8

**PSNR(gateaux1.png,gateaux1-gaussien.png) = 32.1754**

Grâce à cela, on remarque que le filtre gaussien semble être le meilleur étant donné que son PSNR avec l'image d'origine est le plus élevé. On remarque également le filtre moyenneur le plus petit (3x3) est meilleur dans ce cas que le filtre moyenneur plus grand (5x5).

Bruit impulsionnel et filtre médian

Le bruit impulsionnel peut également être appelé “**Poivre et sel**”. Il peut provenir par exemple de pixels défectueux ou d’une erreur de CAN. Un moyen de filtrer ce bruit est le filtre **médian**. Ce filtre présente l’avantage de ne pas créer de nouveau niveau de pixel et de mieux préserver les contours sans altérer le fond. Cependant, il peut être amené à supprimer certains détails fins qui ne sont en réalité pas du bruit. De plus, il peut détruire les coins et est assez coûteux en temps de calcul.

| |
|---|
| Bruiter l’image **I4** grâce au menu **Process/Noise/Salt and Pepper**. Caractériser le bruit ainsi créé (type et pourcentage de pixels affectés). |

En se basant sur l’image **I4**, qui est une image dont tous les pixels sont de niveau 127 (gris). Nous allons appliquer un bruitage impulsionnel afin d’étudier ses caractéristiques et de trouver un moyen de le filtrer.

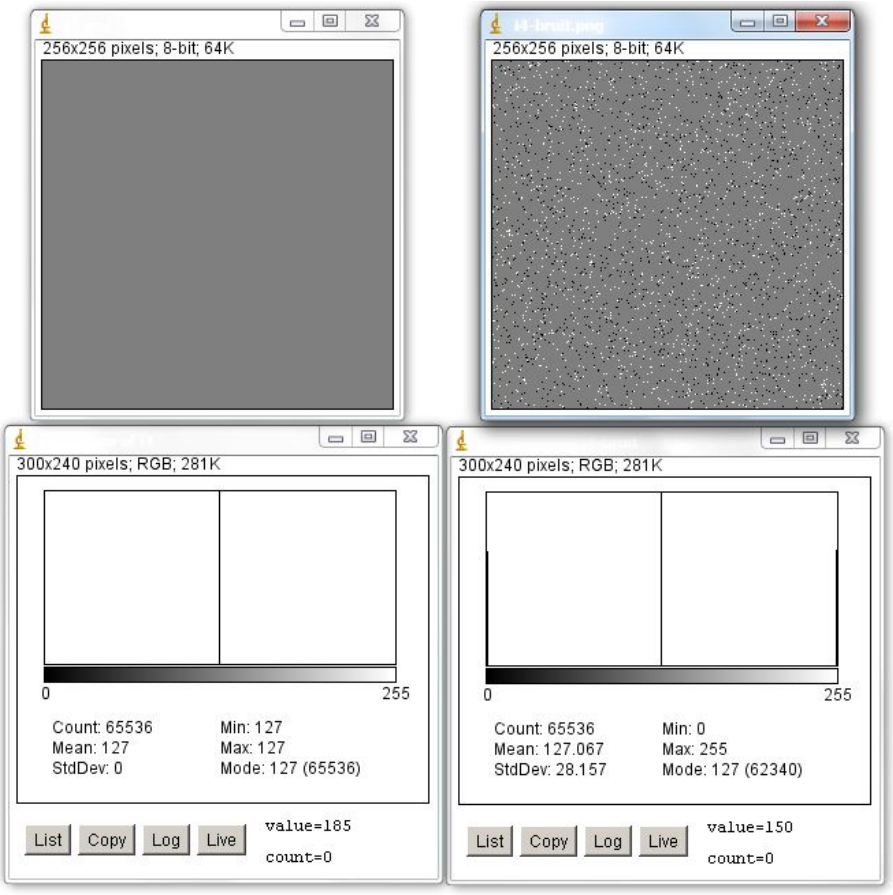



*Image **I4** et son histogramme associé sur la gauche, Image **I4** bruitée et son histogramme associé sur la droite*

Le bruit créé est un bruit dit impulsionnel (ou **Poivre et sel**). C’est en réalité un ajout de pixels de niveau 0 (noir) et de niveau 255 (blanc). Ces pixels sont répartis aléatoirement

sur l'image. On peut, grâce aux valeurs présentes sur l'histogramme, calculer le pourcentage de bruit présent sur cette image :

**Nombre de pixels totaux :** 65 536
**Nombre de pixels de niveau 0 (noirs) :** 1587
**Nombre de pixels de niveau 255 (blancs) :** 1609

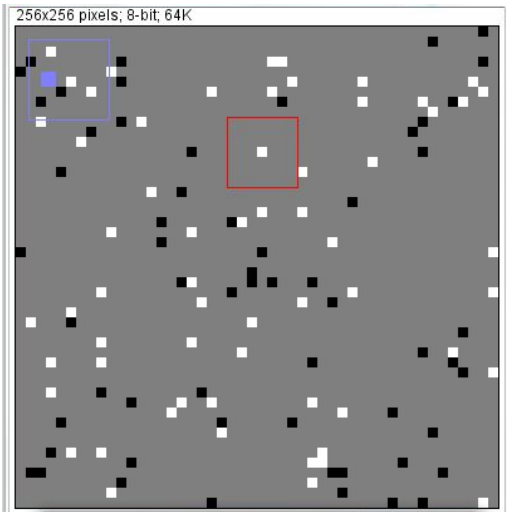
Nous avons donc le pourcentage **p** de bruit suivant :

| **p** = (1587 + 1609) * 100 / 65 536 = **4,88%**. |
|---|

Il y a donc **4,88%** des pixels totaux qui ont été affecté par le bruitage impulsionnel. Cela semble relativement peu mais les pixels étant noirs et blanc, cela ressort beaucoup pour l'oeil humain.

| Comparer l'effet des filtres **moyenneur** 3x3 et **Gaussien** (σ=0.8) au voisinage d'un pixel bruité (noir ou blanc) donné **P**. Conclure sur l'efficacité de ces filtres à réduire ce type de bruit dans une image. |
|---|

A partir de notre image bruitée précédemment avec un bruit impulsionnel de de type **Poivre et Sel**, nous analysons un pixel précis ainsi que la façon dont il a été transformé par le filtre. Le pixel analysé est celui encadré en **rouge** sur l'image ci-dessous :

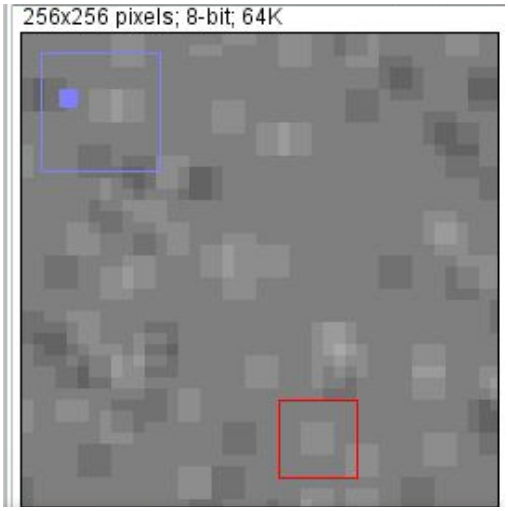



*Partie de l'image **I4** après affectation d'un bruit impulsionnel (**Poivre et Sel**)*

| | | | | | | |
|---|---|---|---|---|---|---|
| 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 127 | 127 | 127 | **255** | 127 | 127 | 127 |
| 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 127 | 127 | 127 | 127 | 127 | 127 | 127 |

*Tableau de pixels du voisinage du pixel analysé (**en gras**) (zone encadrée en rouge sur l'image ci-dessus)*

Nous appliquons sur l'image **I4** bruitée un filtre **moyenneur 3x3** et nous observons ce qui se passe pour le pixel que nous avons choisi précédemment :



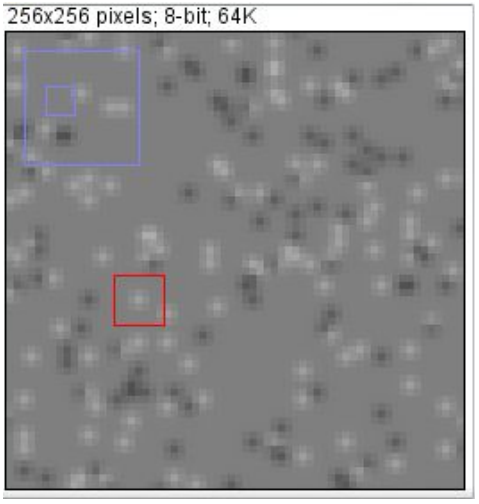

*Application d'un filtre **moyenneur 3x3** sur l'image **I4** affectée par un bruit impulsionnel*

| | | | | | | |
|---|---|---|---|---|---|---|
| 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 127 | 127 | 141 | 141 | 141 | 127 | 127 |
| 127 | 127 | 141 | **141** | 141 | 127 | 127 |
| 127 | 127 | 141 | 141 | 141 | 127 | 141 |
| 127 | 127 | 127 | 127 | 127 | 127 | 141 |
| 127 | 127 | 127 | 127 | 127 | 127 | 141 |

*Tableau de pixels du voisinage du pixel analysé (**en gras**) après l'application du filtre **moyenneur 3x3** sur l'image **I4** bruitée*

Avec le filtre **moyenneur 3x3**, nous constatons que le nombre de pixels modifiés est assez important ce qui rend l'image très différente de l'image **I4** de base à l'oeil humain. On remarque également que notre pixel tend plus vers la valeur d'origine qui est de 127. Cependant, les pixels qui étaient autour (ses 8 proches voisins) et qui avaient la valeur 127 ont été affectés et ont pris la même valeur que celle du pixel transformé qui était à la base blanc (255).

Nous appliquons à l'image **I4** bruitée un filtre **Gaussien** d'écart-type 0,8. Le filtre **Gaussien** présente l'avantage d'éviter le flou et donc de mieux conserver les contours. Il tient compte de la moyenne et de l'écart-type de l'image. Le résultat est visible ci-dessous :




*Application d'un filtre **Gaussien** sur l'image **I4** affectée par un bruit impulsionnel*

| | | | | | | |
|---|---|---|---|---|---|---|
| 127 | 127 | 127 | 127 | 126 | 126 | 126 |
| 127 | 127 | 128 | 128 | 128 | 127 | 127 |
| 127 | 128 | 134 | 142 | 134 | 128 | 127 |
| 127 | 128 | 142 | **159** | 142 | 128 | 128 |
| 127 | 128 | 134 | 142 | 134 | 128 | 134 |
| 127 | 127 | 128 | 128 | 128 | 128 | 142 |
| 127 | 127 | 127 | 127 | 127 | 128 | 134 |

*Tableau de pixels du voisinage du pixel analysé (**en gras**) après l'application du filtre **Gaussien** sur l'image **I4** bruitée*

Avec le filtre **Gaussien**, nous constatons que les valeurs des pixels sont également très proches mais beaucoup d'entre elles ont été modifiées ce qui nous donne encore une fois à l'oeil humain une image très différente de l'image de base. La valeur du pixel analysée se rapproche bien de la valeur initiale qui est de 127. Cependant, si on compare le résultat avec celui obtenu grâce au filtre moyenneur, on remarque alors que cela affecte encore plus de ces voisins.

Grâce au menu **Process/Filters/Median...**, appliquer un filtrage **médian** de rayon 1 sur **I4** bruitée. Commenter le résultat.

En utilisant l'image **I4** affectée par un bruit impulsionnel, nous allons appliquer un filtrage **médian**. D'après le cours, le filtrage **médian** est le plus adapté pour filtrer le bruit impulsionnel (**Poivre et sel**) parmi les autres filtres (**Gaussien, moyenneur 3x3, …**).

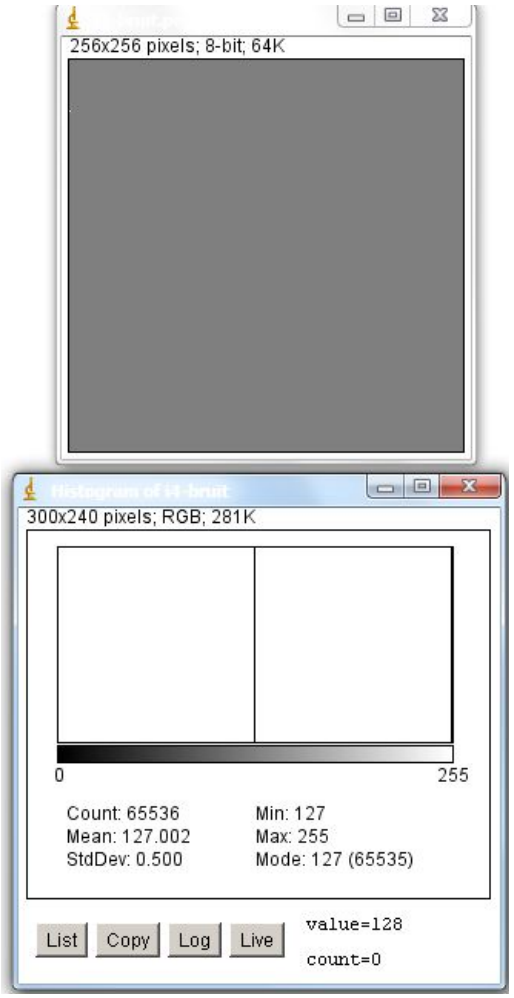



*Image **I4** bruitée après application d'un filtre **médian** (de rayon 1) et son histogramme associé*

Nous constatons à l'oeil humain que l'image est parfaitement filtrée. Il n'y a à priori aucun pixel parasite et il nous serait impossible de faire la différence entre l'image **I4** et l'image **I4** bruitée après filtrage. L'histogramme confirme nos propos puisque la totalité des pixels de l'image se situe sur la valeur 127. Nous pouvons nous intéresser au **PSNR** de l'image débruitée :

**PSNR**(image4,image4-debruit) = Infinity

Nous avons vu dans le cours que si deux images $\mathbf{I_1}$ et $\mathbf{I_2}$ sont égales ($\mathbf{I_1 = I_2}$), alors leur **PSNR** vaut **+∞**. Nous constatons donc ici que l'image **I4** bruitée après un filtrage **médian** est la même que l'image **I4** de base.

Un filtre **médian** de rayon 0.5 permet-il de supprimer totalement le bruit impulsionnel ? Pourquoi (détailler sur des pixels exemples) ?

Nous réalisons la même manipulation que la question précédente sauf que nous modifions la valeur du rayon (0.5 au lieu de 1). Nous obtenons le résultat ci-dessous :

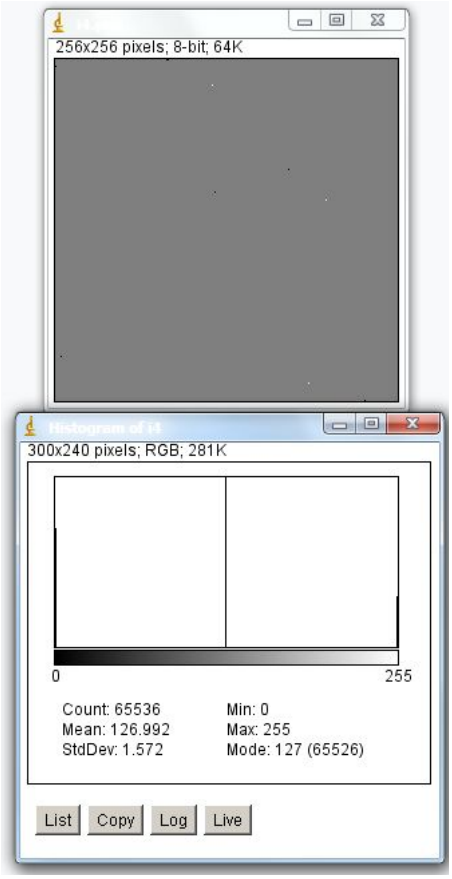



*Image **I4** bruitée après application d'un filtre **médian** (de rayon 0.5) et son histogramme associé*

Nous pouvons apercevoir que l'image filtrée n'est pas parfaite puisqu'il reste un petit nombre de pixels de niveau 0 et 255 (provenant du bruitage impulsionnel **Poivre et Sel**). Nous avons relevé le nombre de pixels blancs et noirs restant :

**Nombre de pixels noirs (niveau 0) restant :** 7
**Nombre de pixels blancs (niveau 255) restant :** 3

Le filtrage **médian** de rayon 0.5 est donc très efficace mais il ne permet pas de supprimer totalement le bruit impulsionnel contrairement au filtrage **médian** de rayon 1 qui lui nous permet d'obtenir une image parfaitement correspondante à l'image de base.

Comparer l'efficacité des filtres **3x3 moyenneur** et **médian** sur l'image gateaux1.png affectée d'un bruit impulsionnel.

Dans un premier temps, nous récupérons l'image '*gateaux1.png*' et nous lui appliquons un bruit impulsionnel (**Poivre et sel**) comme visible sur l'image ci-dessous :

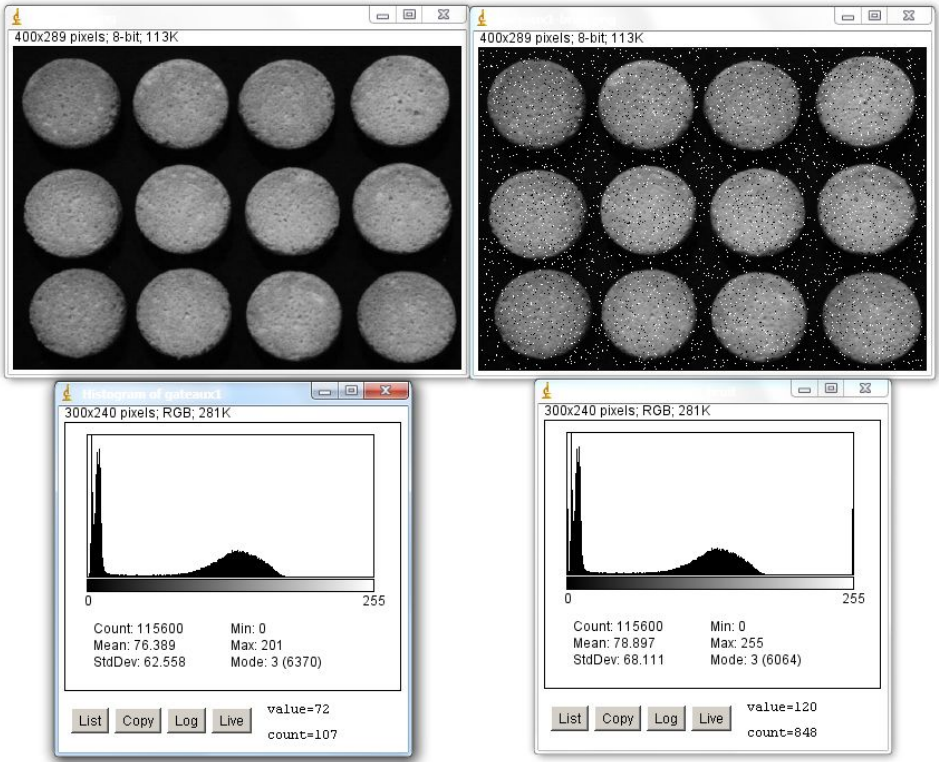



*Image **gateaux1.png** de base et celle affectée par un bruit impulsionnel et leur histogramme associé*

Ensuite, nous appliquons un filtre **3x3 moyenneur** sur l'image bruitée et un filtre **médian** sur cette même image bruitée. Nous constatons le résultat suivant :

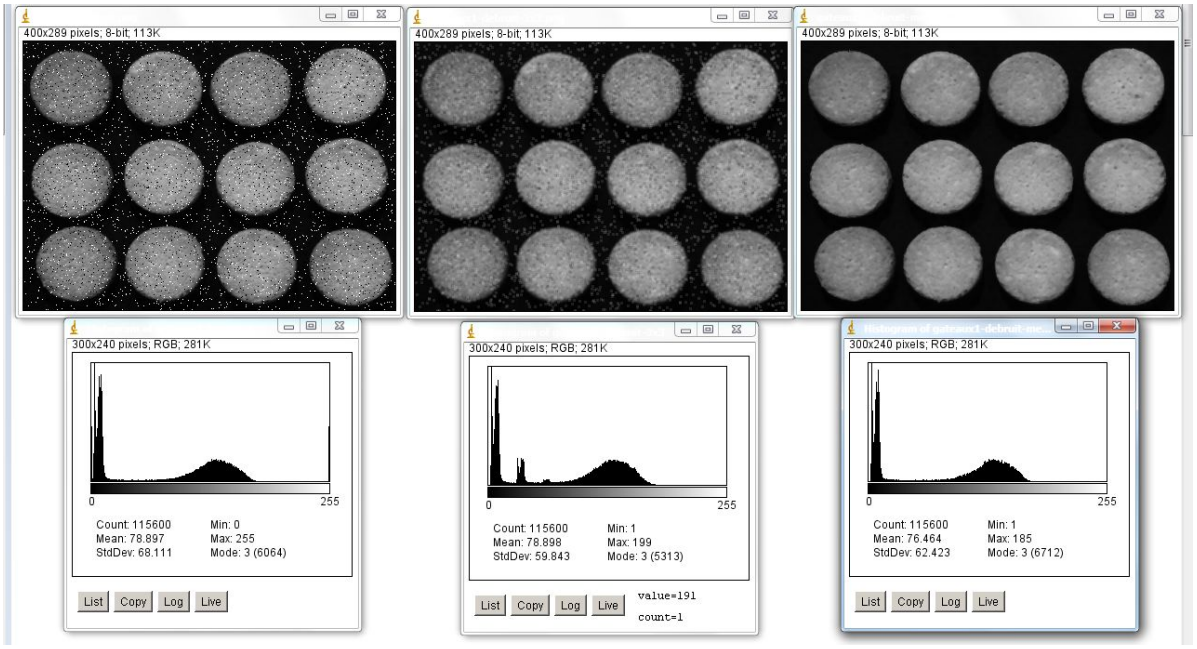



*Image **gateaux1.png** bruitée à gauche, image bruitée avec application d'un filtre **moyenneur 3x3** au centre et image bruitée avec application d'un filtre **médian** à droite ainsi que leur histogramme associé*

Nous pouvons constater qu'à l'oeil humain, le filtre **médian** et le plus efficace puisqu'il ne reste aucun pixel de niveau 255 (blanc) restant en comparaison avec le résultat du filtrage **moyenneur 3x3**. De plus, le **PSNR** nous permet de confirmer notre pensée :

PSNR entre l'image normale et l'image affectée par le bruit impulsionnel

**PSNR**(gateaux1.png,gateaux1-bruit.png) = 17.6792

PSNR entre l'image normale et le résultat après filtrage moyenneur 3x3

**PSNR**(gateaux1.png,gateaux1-debruit-3x3.png) = 25.7305

PSNR entre l'image normale et le résultat après filtrage médian

**PSNR**(gateaux1.png,gateaux1-debruit-median-1.png) = 34.0131

Le **PSNR** le plus élevé est celui correspondant au filtrage **médian** ce qui nous permet de conclure que ce filtrage est le plus efficace dans le cas d'un bruitage impulsionnel.

## 4ème partie : Codage d'un filtre

| |
|---|
| Créer une nouvelle macro qui filtre une image par filtrage min-max. |

Le filtre **min-max** est un filtre de lissage non-linéaire. Il permet de débruiter efficacement en garantissant que la valeur de chaque pixel appartienne à la valeur de ses voisins. D'autre part, il permet de mieux préserver les contours qu'un filtrage médian.

La méthode de calcul est définie par cette formule :

$$I'(x,y)=\begin{cases} I(x,y) & \text{si } i_{min} \leqslant I(x,y) \leqslant i_{max} \\ i_{min} & \text{si } I(x,y) < i_{min} \\ i_{max} & \text{si } I(x,y) > i_{max} \end{cases}$$

La première étape du code consiste à dupliquer l'image résultante après le filtrage, pour cela nous avons dû faire écrire les deux lignes suivantes :

```
newImage("image_result_min_max", "8-bit black", W, H, 1);
image_result_min_max = getImageID();
```

Ensuite, nous avons créé une boucle permettant de traiter l'image pixel par pixel. Durant ce traitement, nous effectuons deux opérations majeures :

**La recherche du min et du max dans le voisinage**

Pour cette opération, nous avons été contraint d'effectuer la gestion des bords. Nous avons choisi la stratégie du **miroir (virtuel) de l'image** qui est la stratégie utilisée par ImageJ pour gérer les effets de bords. Les lignes ci-dessous nous ont permis de réaliser cette stratégie :

```
if(l==-1) l_tmp = 0;
if(l==H) l_tmp = H-1;
if(k==-1) k_tmp = 0;
if(k==W) k_tmp = W-1;
```

Grâce à cela, nous pouvons trouver la valeur de chaque pixel du voisinage du pixel courant (*l, k*) avec de trouver le **min** et **max** du voisinage du pixel (*l, k*) :

```
// calcul du min et du max des voisins
for(k = j-1; k <= j+1; k++){
	for(l = i-1; l <= i+1; l++){
		k_tmp = k;
		l_tmp = l;
		if(!(k==j && l==i)){
			if(l==-1) l_tmp = 0;
			if(l==H) l_tmp = H-1;
			if(k==-1) k_tmp = 0;
			if(k==W) k_tmp = W-1;
			tmp = getPixel(l_tmp, k_tmp);
			if(max < tmp){
				max = tmp;
			}
			if(min > tmp){
				min = tmp;
			}
		}
	}
 }
```

**L'affectation de la nouvelle valeur du pixel**

L'opération d'affectation de la nouvelle valeur du pixel est relativement simple et basique à mettre en place. Toutefois, il ne faut pas oublier de changer d'image courante afin de ne pas modifier l'image de base. La ligne ci-dessous permet cela :

```
selectImage(image_result_min_max);
```

Enfin, il suffit de reprendre la formule indiquée au début de la question et de traiter le pixel selon le cas. Ainsi, soit la valeur est :

- plus petite que tous ces voisins, alors le pixel prend la valeur minimale de ces voisins
- plus grande que tous ces voisins, alors le pixel prend la valeur maximale de ces voisins
- entre le **min** et le **max** de ces voisins, alors le pixel garde sa valeur de base

```
// affectation de la valeur du pixel selon les cas
if(p < min){
	setPixel(i,j,min);
} else if(p > max){
	setPixel(i,j,max);
} else {
	setPixel(i,j,p);
}
```

Tout est prêt, il ne reste plus qu'à tester la macro sur une image modifiée afin de constater si le résultat semble correct. L'intégralité du code de la macro est disponible à la fin du rapport en **Annexes**.

Tester cette macro sur l'image gateaux1.png affectée :
- soit d'un bruit Gaussien d'écart-type 25 (menu **Process/Noise/Add Noise**)
- soit d'un bruit impulsionnel (menu **Process/Noise/Salt and Pepper**)

Bruit Gaussien d'écart type 25

Nous testons dans un premier temps notre macro sur l'image '*gateaux1.png*' affectée par un bruit **Gaussien** d'écart-type 25. Les résultats du bruitage ainsi que du débruitage avec la macro sont visibles sur l'image ci-dessous :

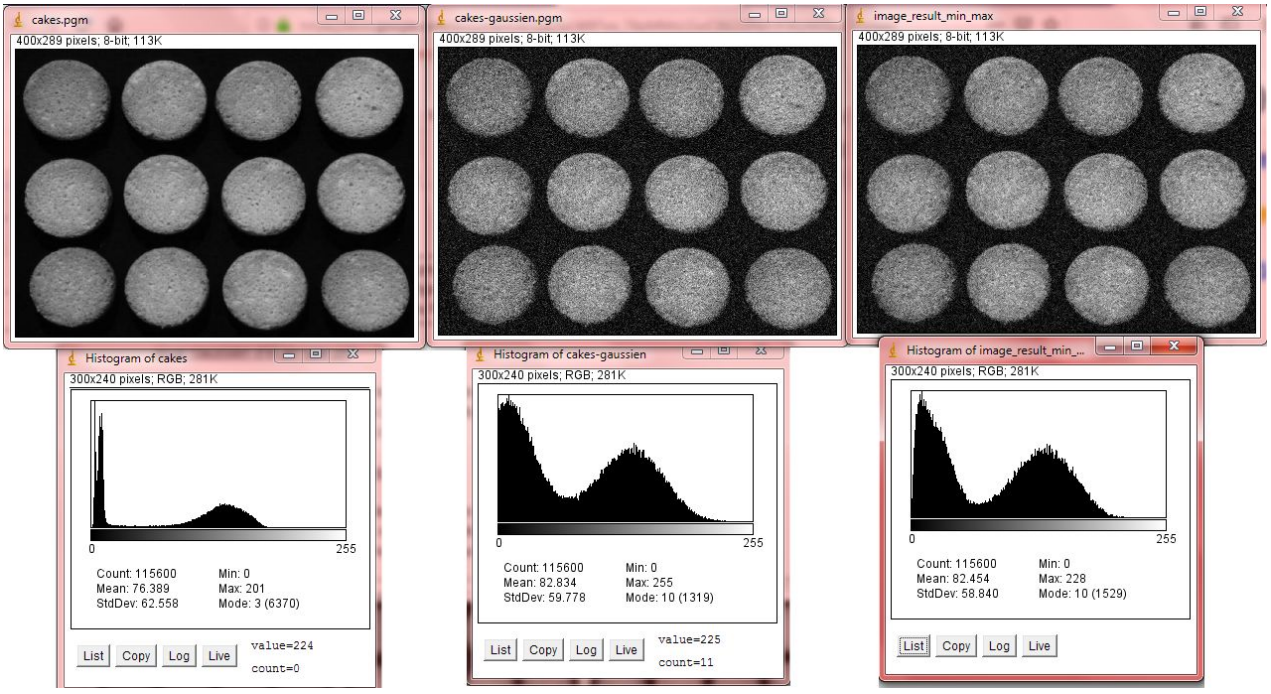



*Image de base **gateaux1.png**, image bruitée et image débruitée suivant la macro avec leur histogramme associé*

Nous observons une différence entre l'image bruitée et l'image débruitée. Toutefois, cette différence reste très légère et le filtre utilisé ne semble pas très efficace face à ce type de bruit. Nous nous intéressons au **PSNR**.

PSNR entre l'image normale et l'image affectée par le bruit Gaussien

**PSNR**(cakes.pgm,cakes-gaussien.pgm) = 20.5372

PSNR entre l'image normale et le résultat après filtrage min-max

**PSNR**(cakes.pgm,image_result_min_max) = 21.8214

Nous pouvons constater que les deux PSNR sont relativement similaire, ce qui confirme la très légère modification entre l'image bruitée et l'image débruitée.

Bruit impulsionnel (type sel et poivre)

Ensuite, nous testons notre macro sur l'image '*gateaux1.png*' affectée par un bruit impulsionnel (de type poivre et sel). Les résultats du bruitage ainsi que du débruitage avec la macro sont visibles sur l'image ci dessous :

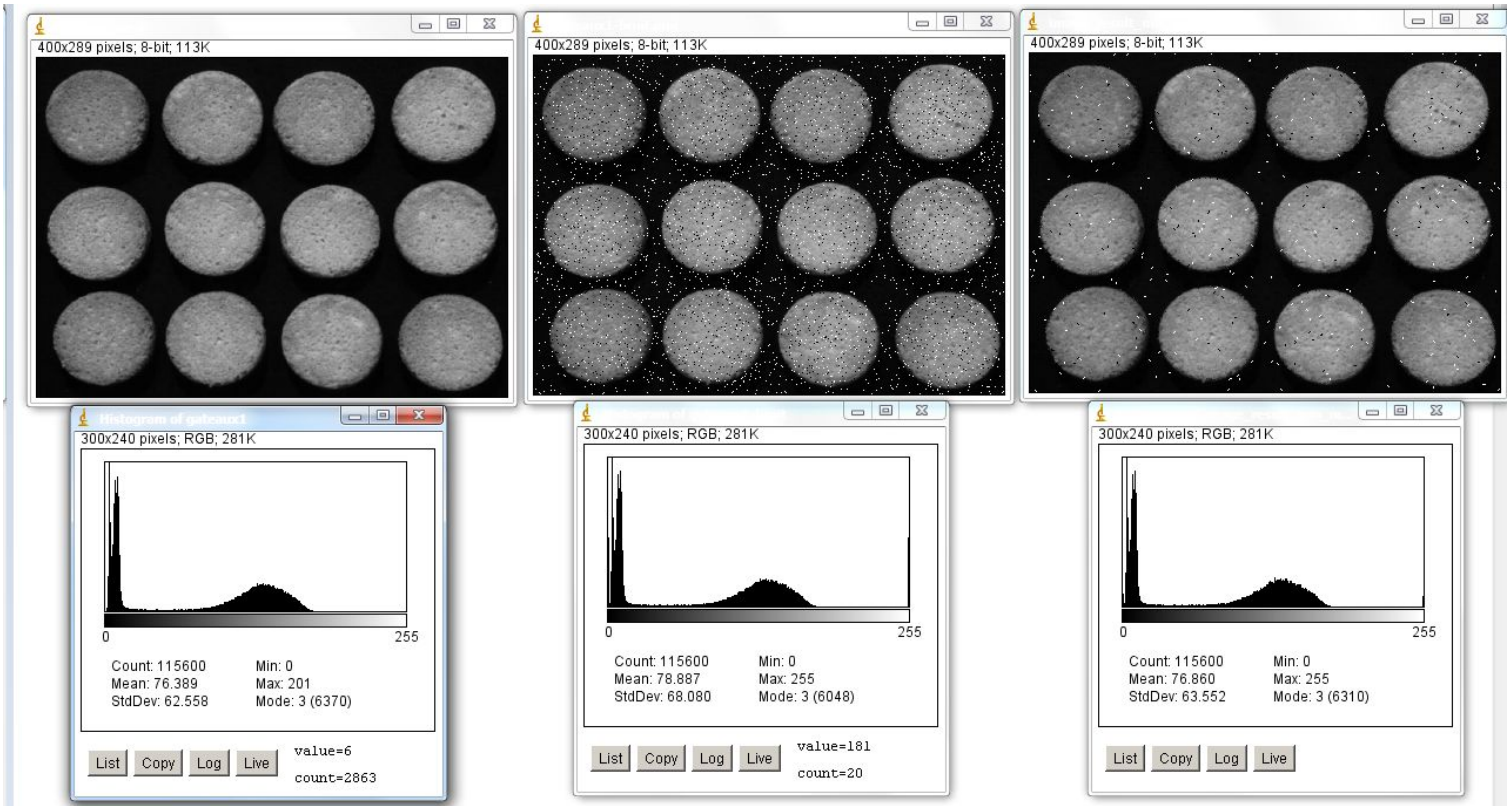



*Image de base **gateaux1.png**, image bruité et image débruitée suivant la macro avec leur histogramme associé*

Dans le but de mieux comprendre ce qui s'est passé, analysons le **PSNR** entre l'image normale et l'image affectée par le bruit impulsionnel dans un premier temps puis ensuite le **PSNR** entre l'image normale et le résultat de l'image bruitée en ayant appliqué le filtrage **min-max**.

PSNR entre l'image normale et l'image affectée par le bruit impulsionnel

**PSNR(gateaux1.png, gateaux1-bruit.png) = 17,617**

PSNR entre l'image normale et le résultat après filtrage min-max

**PSNR(gateaux1.png, image_result_min_max) = 25,0052**

Nous pouvons constater grâce aux PSNR que l'image a bien été en partie débruitée comme on peut le voir à l'oeil nu.

Conclure sur la faculté du filtre min–max à atténuer chacun de ces types de bruits, en la comparant à celle des filtres étudiés à l'exercice précédents.

On s’aperçoit que le filtre **min-max** est une bonne solution pour éliminer le bruit impulsionnel même si le filtre **médian** est beaucoup plus efficace dans ce cas comme nous avons pu le remarquer dans la partie précédente. A l’inverse, le filtre **min-max** n’est pas du tout efficace pour filtrer le bruit **Gaussien**.

# Annexes

```
setBatchMode(true); //images are hidden during macro execution
image_base = getImageID();
selectImage(image_base);
H = getHeight();
W = getWidth();
newImage("image_result_min_max", "8-bit black", W, H, 1);
image_result_min_max = getImageID();
for(j = 0; j < H; j++){
   for(i = 0; i < W; i++){
            min = 255;
            max = 0;
            // récupération de la valeur du pixel
            selectImage(image_base);
            p = getPixel(i,j);
            // calcul du min et du max des voisins
            for(k = j-1; k <= j+1; k++){
                        for(l = i-1; l <= i+1; l++){
                                    k_tmp = k;
                                        l_tmp = l;
                                    if(!(k==j && l==i)){
                                                if(l==-1) l_tmp = 0;
                                                if(l==H) l_tmp = H-1;
                                                if(k==-1) k_tmp = 0;
                                                if(k==W) k_tmp = W-1;
                                                tmp = getPixel(l_tmp, k_tmp);
                                                if(max < tmp){
                                                            max = tmp;
                                                }
                                                if(min > tmp){
                                                            min = tmp;
                                                }
                                    }
                        }
            }
            selectImage(image_result_min_max);
            // affectation de la valeur du pixel selon les cas
            if(p < min){
                        setPixel(i,j,min);
            } else if(p > max){
                        setPixel(i,j,max);
            }else{
                        setPixel(i,j,p);
            }
   }
}
setBatchMode("exit and display"); //show images
```

*Code de la macro utilisé dans la **partie 4** mettant en place le filtrage **min-max***