Semaine 10 : Détection de contours par approches du second ordre

TP10

Maxime CATTEAU
Léane TEXIER

1ère partie : Calcul du Laplacien

Calcul et représentation du Laplacien

Ajouter à la méthode **run** du code de base : * le calcul par convolution du Laplacien de l'image d'entrée dans l'image 32 bits fpLaplacian * l'affichage du résultat dans une nouvelle fenêtre, représentée par un objet ImagePlus

Dans cette première partie, nous allons mettre en place le calcul et la représentation du Laplacien. Afin de mettre cela en place, il est nécessaire d'effectuer le calcul par convolution en fonction du masque. Plusieurs filtres sont possibles et se choisissent lors du lancement du plugin. Le masque appliqué est un filtre 3x3. Dans le but d'effectuer cette opération, nous avons ajouté les lignes suivantes à la méthode **run** :

```
fpLaplacian.convolve(this.MASQUES_LAPLACIENS3x3[this.filtre], 3, 3);
this.imp = new ImagePlus("Résultat du filtre Laplacien", fpLaplacian);
this.imp.show();
```

Code correspondant au calcul et la représentation du Laplacien

La première ligne de ce code permet de convoluer l'image 32bits fpLaplacian suivant le masque choisi. Les deux lignes suivantes permettent de créer une ImagePlus suivant la convolution effectuée et de l'afficher.

Exécuter le plugin sur l'image spores et interpréter l'image obtenue.

Une fois le plugin modifié, nous avons alors exécuté le plugin sur l'image '*spores*' avec différents masques. Pour rappel, voici les masques utilisés dans ce plugin :

- Laplacien1

0	1	0
1	-4	1
0	1	0

- Laplacien2

1	0	1
0	-4	0
1	0	1

- Laplacien3

1	1	1
1	-8	1
1	1	1

- Laplacien4

1	2	1
2	-12	2
1	2	1

- Laplacien5

1	4	1
4	-20	4
1	4	1

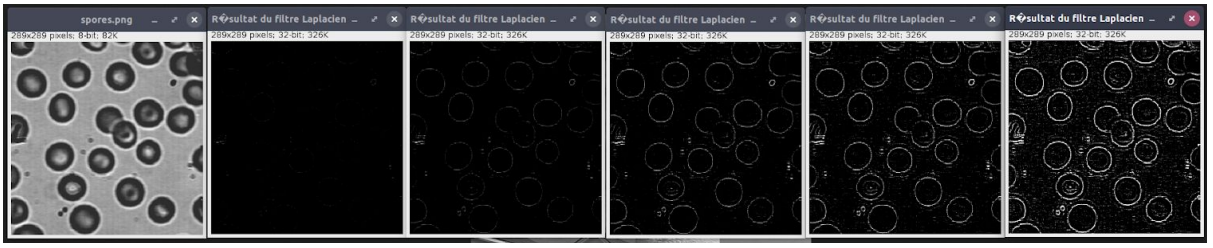


Image 'spores' originale, avec masque Laplacien1, avec masque Laplacien2, avec masque Laplacien3, avec masque Laplacien4 et avec masque Laplacien5

On remarque que suivant le filtre appliqué, les contours sont alors plus ou moins marqués. En effet, plus le filtre donne du poids aux voisins du pixel analysé, plus les contours sont alors visibles sur l'image résultante car plus de pixels ont une valeur se rapprochant du niveau de gris correspondant au blanc.

Mettre en évidence les pixels auxquels le Laplacien a une valeur proche de 0, en utilisant le menu Image/Adjust/Threshold. Quelles remarques pouvez-vous faire sur l'utilisation directe des valeurs nulles du Laplacien pour détecter les points contours ?

Les pixels qui nous intéressent pour mettre les contours en évidence sont ceux ayant une valeur à 0 ou proche de 0. Afin de pouvoir les repérer sur les images obtenues précédemment, nous avons utilisé la commande Threshold proposé par imageJ. Nous avons alors mis le seuil haut à 0.05 et le seuil bas à -0.05. Voici le résultat obtenu :

Image ‘spores’ avec masque Laplacien avec seuil haut à 0.05 et seuil bas à -0.05

Nous pouvons remarquer sur cette image que la plupart des contours sont bien mis en évidence. Nous remarquons qu’il y a également beaucoup de bruit ou des zones, comme en haut à droite de l’image, ou un “paquet” de pixels s’est formé. Cela est dû à des zones ou le niveau de gris est inclus entre les bornes du seuil [-0.05, 0.05] mais qui ne sont pas vraiment des contours. A cause de la double dérivation, le Laplacien est alors plus sensible au bruit.

2ème partie : Seuillage des passages par 0 du Laplacien

Seuillage manuel des passages par 0 du Laplacien

Implémenter cet algorithme sous forme de la méthode :
ByteProcessor laplacienZero(ImageProcessor imLaplacien, float seuil)
qui retourne l'image binaire des passages par 0 du Laplacien stocké dans l'image **imLaplacien**, en utilisant un **seuil** sur ses valeurs.

Dans cette partie, nous avons écrit et exécuté le code permettant d'effectuer un seuillage manuel des passages par 0 du Laplacien. Pour cela, nous avons écrit une méthode ***laplacienZero*** qui prendra en paramètre l'image binaire des passages par 0 ***imLaplacien*** ainsi que le ***seuil***. Cette méthode nous retournera un ***ByteProcessor***.
Cette méthode se déroule en plusieurs étapes, celles-ci sont inspirées du cours concernant les détection de contours.

1ère étape :

La première étape consiste à récupérer le voisinage d'un pixel de coordonnées (*i, j*) afin d'analyser la valeur des voisins. Cette étape est relativement simple et nous nous sommes inspirés du précédent TP :

```
float [ ] voisinage = new float[8];
        for(int k = 0; k < 8; k++){
        int xk = i + dx8[k];
        int yk = j + dy8[k];
        if (xk<0 || xk>=width) continue;
        if (yk<0 || yk>=height) continue;

        voisinage[k] = imLaplacien.getPixelValue(xk, yk)[0];
}
```

Morceau de code permettant de récupérer le voisinage d'un pixel de coordonnées (i, j)

2ème étape :

La seconde étape consiste à trouver dans ce voisinage au moins un pixel supérieur au *seuil* indiqué par l'utilisateur (seuillage manuel) ainsi qu'au minimum un autre pixel inférieur à (*-seuil*). En d'autres termes, nous cherchons deux pixels tel que :

$$m < -S_\Delta \text{ et } M > S_\Delta$$

avec *m* le pixel de valeur minimale, *M* le pixel de valeur maximale et S_Δ le seuil défini par l'utilisateur. Le code correspondant à cette étape est le suivant :

```
if(voisinage[k] > seuil){
        positiveFound = true;
}
if(voisinage[k] < -seuil){
        negativeFound = true;
}
```

Morceau de code permettant de trouver un passage par 0 dans le voisinage

3ème étape :

La dernière étape du traitement consiste à modifier la valeur des pixels dont le voisinage contient un passage par 0. Dans notre cas, nous mettrons en blanc (pixel de valeur 255) les points désignés comme contour et en noir (pixel de valeur 0) les points désignés comme non contour. Voici les quelques lignes réalisant cette opération :

```
if(positiveFound && negativeFound){
        imZeros.set(i,j,0);
}else{
        imZeros.set(i,j,255);
}
```

Morceau de code permettant de modifier la valeur des pixels dont le voisinage contient un passage par 0

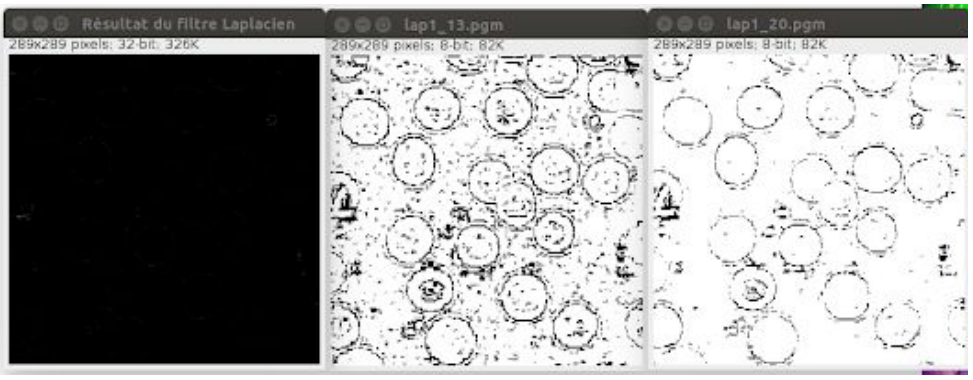
Vous pourrez trouver en **Annexe 1** le code complet de la fonction *laplacienZero*.

Pour l'image *spores*, quel seuil permet d'obtenir les points contours les plus satisfaisants ? Quelles remarques pouvez-vous formuler sur les points contours obtenus ?

Nous avons pu remarquer que le résultat est largement influencé par le seuil que l'on indique, ce qui est logique puisque ce seuil joue dans la détection et la prise en compte des points de passage par 0. Nous avons aussi remarqué que le seuil idéal change selon le filtre que l'on utilise. Nous répertorions donc les seuils idéals pour chacun des 5 filtres laplaciens proposées pour l'image 'spores'.

Avec le filtre *Laplacien 1*

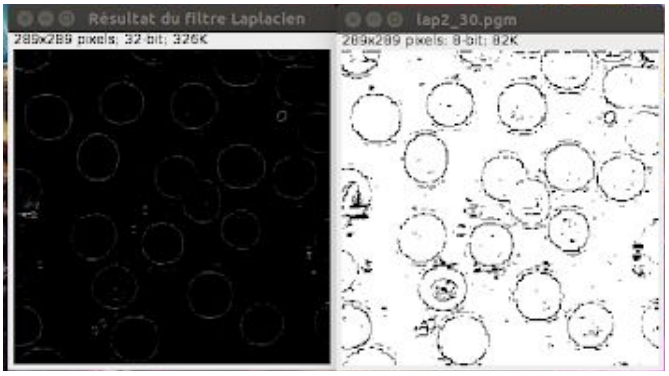
Pour le filtre *Laplacien 1*, la valeur de seuil idéal pour obtenir tous les contours correctement est 13. Cependant il y a beaucoup de bruit présent. Si l'on souhaite obtenir un bon compromis entre peu de bruit et un nombre de contours corrects, il faut utiliser un seuil de 20.



A gauche le filtre Laplacien 1 appliqué sur l'image spores.png, au centre le seuillage manuel avec une valeur de 13, à droite le seuillage manuel avec une valeur de 20

Avec le filtre *Laplacien 2*

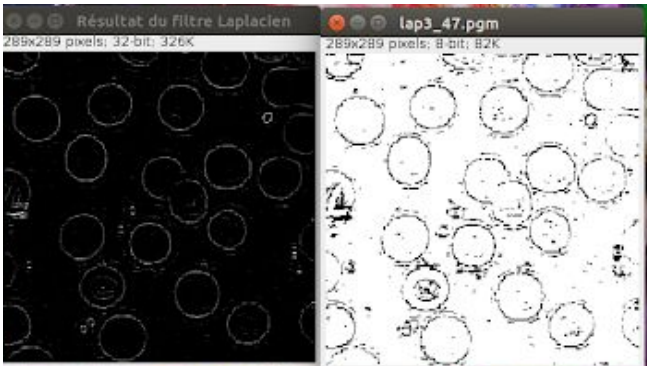
Pour le filtre *Laplacien 2*, nous avons remarqué que le seuil 30 convenait parfaitement. En effet, il y a très peu de bruit et la quasi-totalité des contours sont visibles.



A gauche le filtre Laplacien 2 appliqué sur l'image spores.png, à droite le résultat du seuillage manuel avec une valeur de 30

Avec le filtre *Laplacien 3*

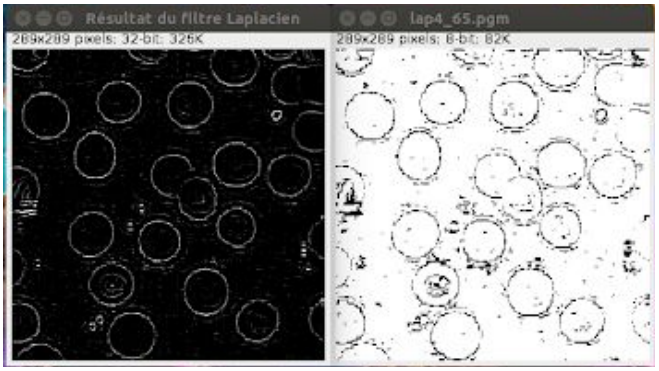
Pour le filtre *Laplacien 3*, nous avons utilisé un seuil de 47 afin d'avoir un bon compromis.



A gauche le filtre Laplacien 3 appliqué sur l'image spores.png, à droite le résultat du seuillage manuel avec une valeur de 47

Avec le filtre *Laplacien 4*

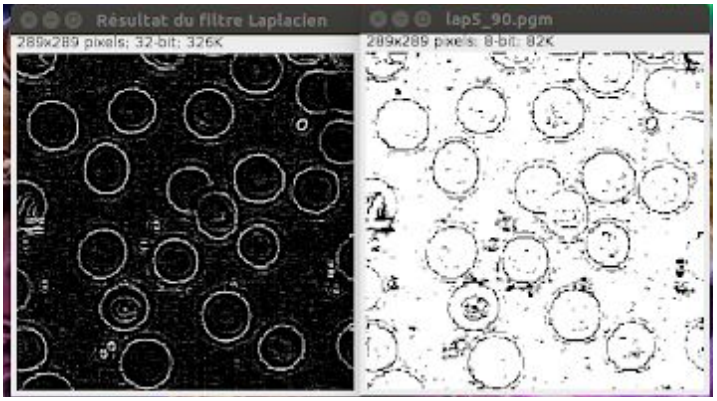
Pour le filtre *Laplacien 4*, nous avons trouvé que le seuil optimal est de 65.



A gauche le filtre Laplacien 4 appliqué sur l'image spores.png, à droite le résultat du seuillage manuel avec une valeur de 65

Avec le filtre *Laplacien 5*

Pour le filtre *Laplacien 5*, nous avons trouvé que le seuil optimal est de 90.



A gauche le filtre Laplacien 5 appliqué sur l'image spores.png, à droite le résultat du seuillage manuel avec une valeur de 90

Nous remarquons donc qu'avec les seuils trouvés, la détection de contours est relativement correcte. Toutefois, elle reste extrêmement sensible aux bruits. En effet, on retrouve beaucoup de pixels isolés, notamment sur les images dont les filtres prennent plus en compte les voisins. Ce bruit est particulièrement visible sur l'image filtrée avec *Laplacien 3*, *Laplacien 4* et *Laplacien 5*. Afin de limiter la présence de bruit, il faudrait appliquer la méthode **LoG** (**L**aplacian **o**f **G**aussian) afin d'effectuer un pré-lissage sur l'image (filtrage passe-bas permettant de "flouter" les zones dont les changement d'intensité sont les plus forts) et donc de réduire les détections de contours qui finalement n'en sont pas.

3ème partie : Utilisation du filtre LoG et détection multi-échelles

Génération de masques pour l'opérateur LoG

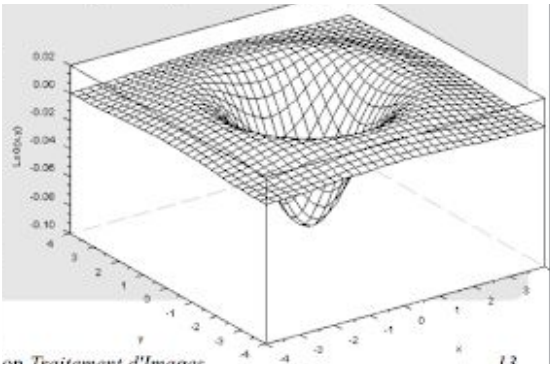
Outre que le masque doit être de taille impaire, comment les paramètres **tailleMasque** et **sigma** sont-ils liés ?

Nous avons remarqué qu'à cause de la double dérivation, le Laplacien est extrêmement sensible au bruit. Il faut donc effectuer un lissage de l'image. Cette opération se réalise en appliquant un pré-filtrage avec un noyau gaussien. Un noyau permet d'effectuer une estimation sur les valeurs représentant potentiellement des contours. Cette opération s'effectue donc avant la détection des points de contour.

La taille du masque ***tailleMasque*** correspond au nombre de lignes et de colonnes du masque ***L**aplacian **o**f **G**aussian* (***LoG***), cette valeur doit être entière et impaire.

sigma représente l'écart-type de la gaussienne. Ainsi, plus le **sigma** sera élevé, plus l'image sera "lissée", c'est-à-dire qu'il y aura beaucoup plus de flou et donc une image très différente de celle de base.

Pour comprendre ce que représentent les paramètres ***tailleMasque*** et ***sigma***, analysons cette fonction du cours :



***tailleMasque** = 9 avec **sigma** = 1,4*

tailleMasque représente donc la "surface" d'impulsion et ***sigma*** représente la taille de l'impulsion, ou dans des termes plus basiques la taille du "chapeau mexicain" ou "cloche Gaussienne". Plus la cloche Gaussienne est large (=plus le sigma est important), plus le flou sera appliqué à l'image et donc plus celle-ci sera affectée (et donc différente de l'image de base).

Pour résumer, la taille du masque impacte sur le nombre de voisins pris en compte et le sigma donne plus ou moins de poids à ces voisins.

Utiliser la méthode **masqueLoG** dans la méthode **run** du plugin pour générer le masque LoG pour un écart-type σ saisi par l'utilisateur, en tenant compte de la remarque de la question précédente.

Afin de générer le masque LoG de l'image pour un écart-type σ saisi par l'utilisateur, nous devons d'abord déclarer une variable qui stockera ce même masque, nous l'appelerons *masqueOfLoG* dans les exemples de code qui suivront. Cette implémentation est relativement simple mais elle nécessite quelques ajustements dans la méthode ***run*** lors de la convolution que l'on souhaite réaliser.

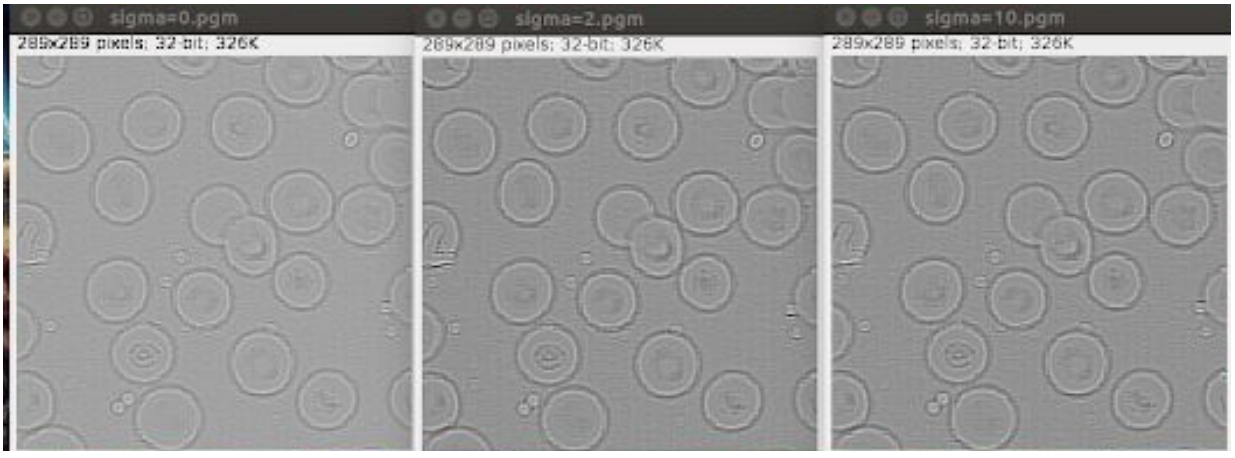
```
float [] masqueOfLoG;
if(sigma != 0.0f){
        masqueOfLoG = masqueLoG(3, sigma);
} else {
        masqueOfLoG = this.MASQUES_LAPLACIENS3x3[this.filtre];
}

FloatProcessor fpLaplacian = (FloatProcessor)(ip.duplicate().convertToFloat());
Convolver conv = new Convolver();
conv.setNormalize(false);
conv.convolve(fpLaplacian, masqueOfLoG, 3, 3);
```

*Morceau de code permettant d'appliquer un masque **LoG** sur l'image*

Il ne faut pas oublier la partie sur le Convolver (i.e. les 3 dernières lignes du code ci-dessus) puisqu'elles permettent d'éviter la normalisation intempestive du masque.

Calculer le LoG de l'image par convolution avec le masque ainsi généré.



*À droite, **LoG** de l'image spores.png avec **sigma** = 0, au centre **LoG** de l'image spores.png avec **sigma** = 2 et à gauche **LoG** de l'image spores.png avec **sigma** = 10*

Nous observons que le passage d'un écart-type de valeur 0 à un écart-type de valeur 2 apporte un certaine lissage de notre image, cela se traduit par des zones un peu plus floues (comme dans le coin haut droit de l'image). Cependant, le passage d'un écart-type de valeur 2 à un écart-type de valeur 10 n'apporte que des changements très minimes voire invisibles.

Si l'on choisit de seuiller les passages par 0 du laplacien ainsi filtré (LoG), comment la valeur du seuil doit-elle être choisie ? Plus précisément, comment choisir ce seuil en fonction de σ et de la taille du voisinage considéré dans la détection des passages par 0 ?

Si l'on choisit de seuiller les passages par 0 du Laplacien, le seuil doit être de plus en plus proche de 0 quand le sigma et la taille de voisinage considérés pour la détection augmente. En effet, dans ce cas, les voisins sont plus pris en compte, on a alors une plus grande précision, il faut donc un seuil plus faible pour détecter les passages par 0.

Utilisation de la norme du gradient

Calculer l'image (binaire) des passages par 0 du LoG de l'image *spores*, en fixant σ=1.4 mais sans seuil (**seuil**=0).

Si nous calculons l'image binaire des passages par 0 du LoG de l'image *spores.png* en indiquant un écart-type de valeur 1.4 et en ne mettant aucun seuil, nous obtenons le résultat suivant :

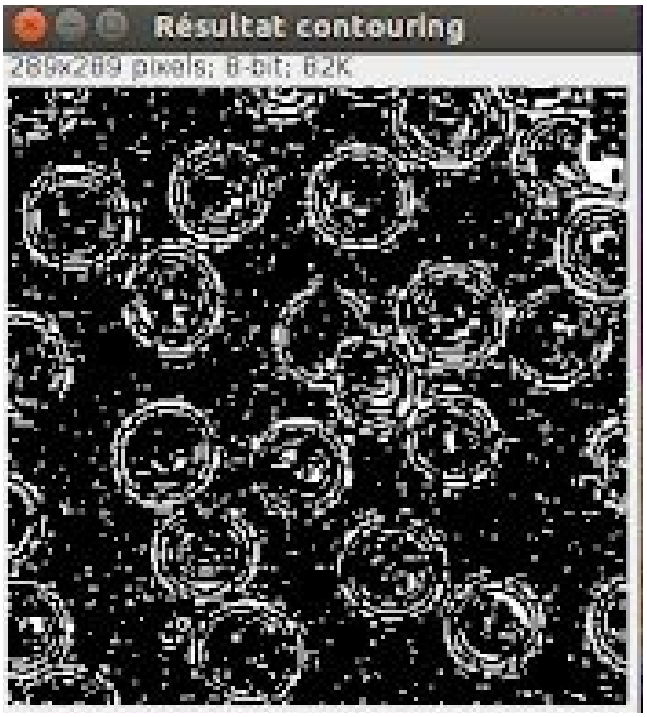
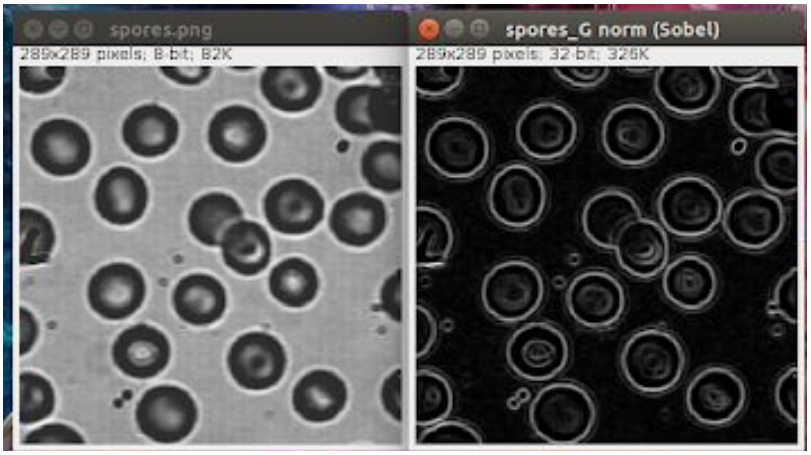


Image binaire des passages par 0 du LoG de l'image spores.png avec σ=1.4 et seuil=0

Nous avons comme résultat une image binaire des passages par 0. La majorité des points de contours sont présents. Toutefois, il reste à régler le problème du bruit (encore assez présent sous forme de pixels isolés) et le problème des contours multiples.

Télécharger et installer le plugin **Gradient**, puis l'utiliser pour calculer l'image de la norme du gradient de l'image *spores*.

Nous téléchargeons le plugin Gradient afin de calculer l'image de la norme du gradient de l'image *spores.png*. Nous obtenons le résultat ci-dessous :



À droite l'image spores.png de base, à gauche la norme du gradient de l'image spores.png

Le filtre utilisé dans ce cas est le filtre de Sobel. Nous obtenons un résultat similaire par rapport aux TP précédents, avec des contours bien marqués malgré un peu de bruit.

Masquer l'image de la norme du gradient ainsi obtenue par celle des passages par 0 du Laplacien obtenue à la question 7, grâce au menu *Process*/*Image Calculator...* et l'opérateur AND.

Nous effectuons la manipulation indiquée par la consigne grâce aux outils qui nous sont proposés par imageJ , nous obtenons le résultat suivant :

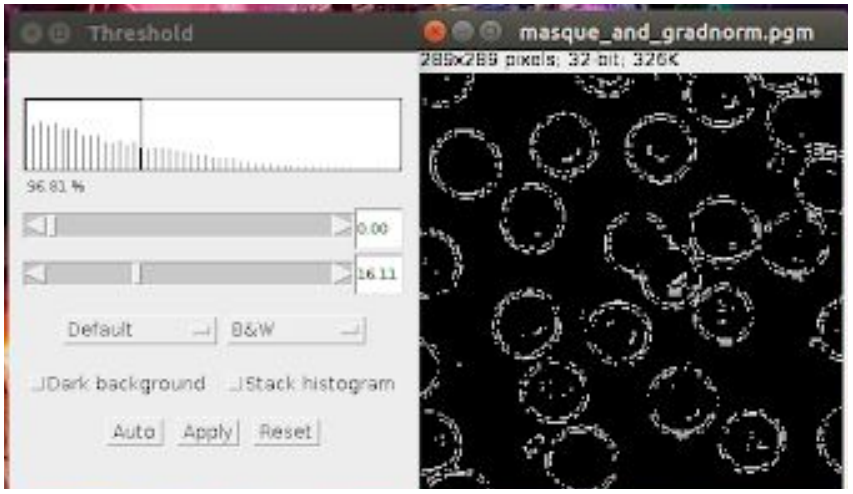
À gauche la norme du gradient de l'image spores.png, au centre le masque obtenu à la question 7, à droite le résultat de l'application du masque sur la norme du gradient grâce à l'opérateur AND

Nous remarquons que le résultat nous fournit un contouring plutôt correct car d'une part le bruit est très atténué (ce qui est l'objectif premier du LoG), d'autre part les contours,

même s'ils ne sont pour la plupart pas fermé intégralement, sont bien visibles et reconnaissables. Il existe des méthodes post-traitement dont le but consiste à effectuer la fermeture des contours.

Appliquer un seuil sur l'image résultante grâce au menu *Image*/*Adjust*/*Threshold*.

Si nous appliquons un seuil de 16.11 sur l'image résultante, nous obtenons un résultat globalement satisfaisant en ce qui concerne les points de contours de l'image :



Ajustement de l'image résultante de la question précédente avec un seuil de 16.11

Comparer les points contours ainsi obtenus à ceux fournis par le plugin **Gradient** qui implémente la méthode de Canny (on fixera par exemple les seuils bas et haut de l'hystérésis respectivement à 8 et 20).

Voici les deux images que nous allons comparer :

À gauche les points de contours obtenus grâce à la méthode de Canny, à droite le résultat des points de contours obtenus grâce à la question précédente

Pour les points de contours obtenus grâce à la méthode de Canny, nous observons des contours très nets et consécutifs, ce qui les rend bien visibles. Toutefois, nous remarquons que d'une part, certains contours ne sont pas fermés (notamment pour les cercles extérieurs des spores, d'autre part, des contours qui n'en sont pas apparaissent, par exemple au centre des cercles.

Pour les points de contours obtenus grâce à la méthode de la norme du gradient pour seuiller les points de passage par 0 du Laplacien, nous avons des contours qui apparaissent saccadés et non fermés. De plus, nous constatons qu'il reste toujours une quantité négligeable de bruit (au centre des cercles par exemple). Par contre, contrairement à la méthode de Canny, il y a très peu de "faux contours" ce qui nous donne une idée assez bonne de la forme générale de l'image.

En conclusion, pour respecter fidèlement les contours d'une image, il faudrait utiliser la méthode de la norme du gradient pour seuiller les points de passage par 0 du Laplacien et compléter cette même méthode avec un post-traitement permettant la fermeture des contours (par exemple la fermeture des contours par extrapolation). Dans le cas de la méthode de Canny, le post-traitement à adopter serait la suppression des contours non-significatifs.

Annexe 1

```
public ByteProcessor laplacienZero(ImageProcessor imLaplacien, Float seuil) {
	int width = imLaplacien.getWidth();
	int height = imLaplacien.getHeight();

	// Image binaire résultat des points contours après seuillage
	ByteProcessor imZeros = new ByteProcessor(width,height);

	int[ ] dx8 = new int[ ] {-1, 0, 1,-1, 1,-1, 0, 1};
	int[ ] dy8 = new int[ ] {-1,-1,-1, 0, 0, 1, 1, 1};

	//boucle sur les pixels
	for(int j = 0; j < height; j++){
		for(int i = 0; i < width; i++){
		//récupérer le voisinage
			float [] voisinage = new float[8];
			boolean positiveFound = false;
			boolean negativeFound = false;

			for(int k = 0; k < 8; k++){
				int xk = i + dx8[k];
				int yk = j + dy8[k];
				if (xk<0 || xk>=width) continue;
				if (yk<0 || yk>=height) continue;

				voisinage[k] = imLaplacien.getPixelValue(xk, yk)[0];

				if(voisinage[k] > seuil){
					positiveFound = true;
				}
				if(voisinage[k] < -seuil){
					negativeFound = true;
				}
			}
			if(positiveFound && negativeFound){
				imZeros.set(i,j,0);
			}else{
				imZeros.set(i,j,255);
			}
		}
	}
	return imZeros;
}
```

*Code complet de la fonction **laplacienZero** de la partie 2*