

8INF804 - Traitement numérique des images



---

Travail Pratique n°2

## Segmentation

---

***Groupe :***

LACLAVERIE Pierre (LACP03119904)

SIMON Thibaud (SIMT15039901)

OUHAOUADA Nihal (OUHN11629909)

REYNAUD Yann (REYY07110005)

# Sommaire

<b>Utilisation du programme</b>	<b>3</b>
Prétraitemet de l'image:	3
<b>Segmentation:</b>	<b>4</b>
Segmentation par seuillage:	4
RandomWalker :	6
Watershed	7
Canny+Filling (détection de contours puis remplissage des formes)	10
SLIC (Simple Linear Iterative Clustering)	14
Quickshift image segmentation	17
Felzenszwalb :	20
Méthode retenue :	21
Extraction des données	22
<b>Conclusion</b>	<b>24</b>

# Utilisation du programme

Le fichier python à utiliser est “segmentation.py”.

Pour pouvoir utiliser le programme il y a deux arguments à donner dans le script : le numéro de l'image et le type de segmentation voulue.

Le numéro de l'image est le dernier numéro du nom de l'image présent dans le dossier Images. Par exemple l'image "Échantillon 1Mod2\_301.png" aura pour numéro 301.

Le deuxième argument à utiliser est le nom de la méthode de segmentation utilisée.

La liste suivante présente toutes les méthodes implémentées:

- thresholding
- rwalker
- watershed
- canny
- slic
- quickshift
- felzen / felzenszwalb

Si un problème survient lors de l'exécution du programme, entrez l'argument h et regardez la sortie standard.

Un exemple d'argument est : 302 watershed.

Pour utiliser ce programme, placez dans le fichier “segmentation.py” dans le même répertoire que le **dossier** Images qui contient les images proposées.

Il ne faut pas supprimer le dossier `resultats` qui est joint au rapport, cela est normal s'il est vide au début.

## Prétraitement de l'image:

Nous avons appliqué plusieurs traitements sur nos images, mais il était convenu que ce prétraitement ne change pas beaucoup les résultats et donc pas très utile peut être que c'est parce que les images à notre disposition n'étaient pas très bruitées, ni de faible ou de très fort contraste.

Parmis les méthodes utilisés pour ce faire, nous citons :

- Transformations morphologiques : comme par exemple la dilatation et l'érosion.
- calibrage du contraste et luminosité: Nous avons réalisé cela avec la méthode **adjust\_brightness\_contrast**, en variant les 2 paramètres alpha pour contrôler la luminosité et beta pour contrôler le contraste. Un tel traitement pourrait par exemple aider à rendre l'image plus claire et permettre une bonne visualisation des données mais n'apporte pas beaucoup en termes d'amélioration de nos algorithmes.
- CLAHE : consiste à étalonner l'histogramme de niveaux de gris. De ce fait, les pixels qui sont proches du noir (respec. blanc) se rapprochent d'un noir (respec. blanc) plus pur.

Par la suite, nous allons adapter pour chaque méthode de segmentation, le prétraitement qui lui convient.

## Segmentation:

Nous distinguons 2 grands types de segmentation : supervisée et non supervisée. Pour chacun de ces deux types, nous essaierons plusieurs méthodes pour choisir celle qui offre les meilleurs résultats.

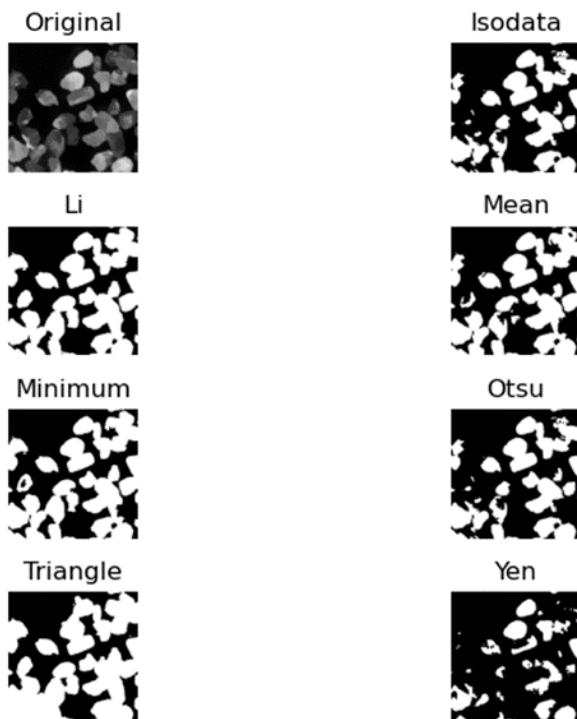
### Méthodes supervisées:

Ces méthodes nécessitent une entrée externe comme par exemple la définition du seuil ou la conversion des images en niveau de gris,...

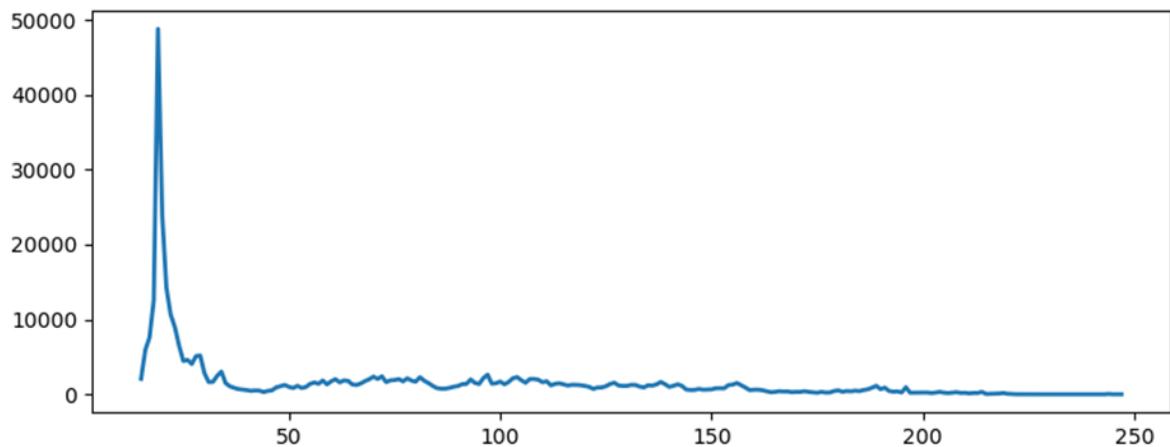
### • Segmentation par seuillage:

C'est la façon la plus simple pour faire de la segmentation. Elle permet d'associer à chaque pixel d'une image un label en fixant un seuil qui va permettre de répartir l'image en deux régions (background et foreground). La première région correspond aux pixels supérieurs à ce seuil tandis que la 2ème région correspond aux pixels qui lui sont inférieurs.

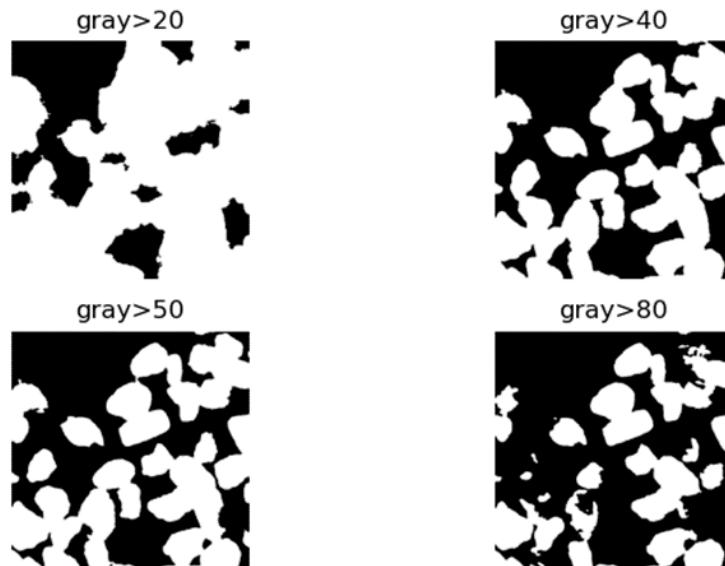
A l'aide de la fonction `filters.try_all_threshold`, nous allons tester différents types de seuillage, l'image suivante résume les résultats obtenus:



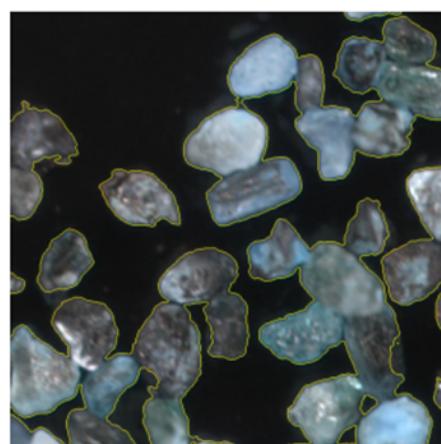
Nous avons aussi réalisé un seuillage binaire, pour cela nous avons tracé l'histogramme de l'image en question (en grayscale) pour nous aider à fixer la valeur du seuil. Nous obtenons les résultats suivants:



Nous avons choisi de varier la valeur du seuil sur la plage [20,80] et de visualiser les résultats.



Comme on peut le voir dans ces images résultantes, deux régions ont été établies. Une comparaison de ces résultats nous mène à conclure avec celui qui nous semble de meilleure segmentation : “segmentation binaire avec seuil = 50”. Nous avons utilisé la fonction **segmentation.mark\_boundaries** pour dessiner les contours.



Nous constatons une bonne segmentation pour les grains non chevauchés. Cependant, pour les grains adjacents l'algorithme n'est plus efficace. En effet, cette méthode ne prend en compte que la distribution des intensités (histogramme) et ne considère pas le problème de voisinage.

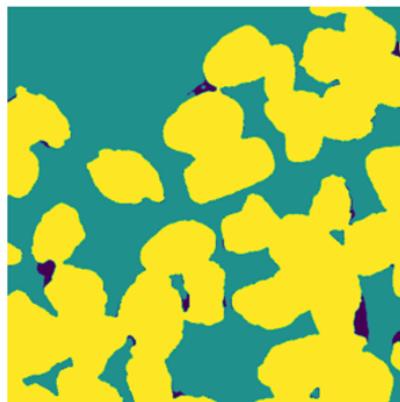
- **RandomWalker :**

La méthode consiste à résoudre une équation de diffusion des traceurs initialisés à la position des marqueurs. Si les pixels voisins ont des valeurs identiques, le coefficient de diffusion est alors plus grand afin d'assurer une diffusion à travers des gradients élevés.

Pour mettre en œuvre cet algorithme, nous allons utiliser la fonction **random\_walker** en conservant les paramètres par défaut c'est-à-dire en mode force brute et avec beta qui vaut 10. Le seul paramètre sur lequel nous allons jouer c'est les markers.

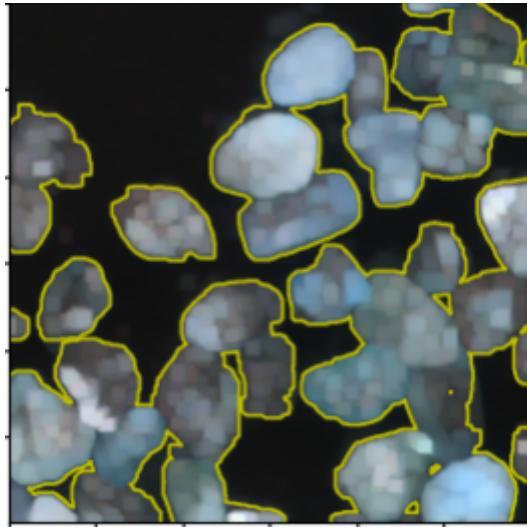
Nous avons défini des marqueurs de même taille que l'image originale et initialisés par des zéros . Le but c'est de mettre ces marqueurs dans des régions sûres, autrement dit , en regardant l'histogramme de la partie threshold, ce sont les pixels compris entre 50 et 55 qui nous retournent ces régions dites d'intérêts. Les markers seront alors définies de la manière suivante :

```
markers = np.zeros_like(gray, dtype=np.uint)
markers[(gray < 50)] = 1
markers[(gray > 55)] = 2
```

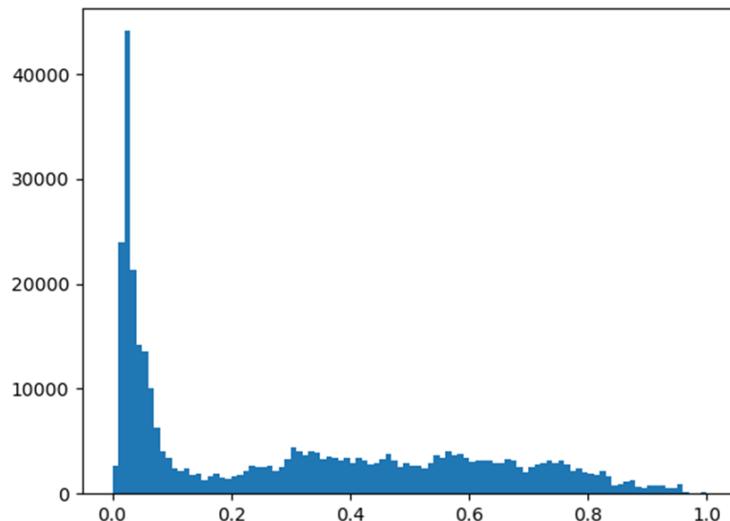


Le paramètre beta n'influe que très très peu le résultat.

Nous obtenons le résultat (Nous avons utilisé la fonction **segmentation.mark\_boundaries** pour tracer les contours) :



Une deuxième façon pour définir nos marqueurs était également possible dans laquelle nous nous servirons d'un histogramme adaptative normalisée entre 0 et 1.

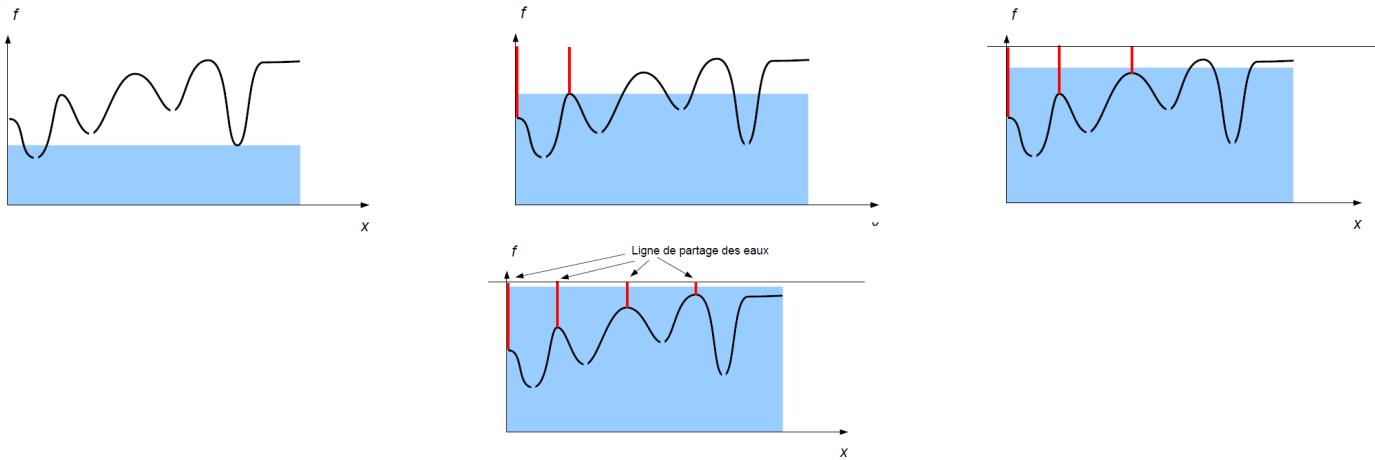


En conclusion, La méthode donne un résultat similaire à celui d'un simple thresholding, ce qui n'est pas très intéressant pour les grains chevauchés mais qui reste de bonne segmentation pour les autres grains non chevauchés.

## • **Watershed**

La ligne de partage des eaux, *watershed* est une technique permettant de segmenter une image *en remplissant* des bassins.

Pour cela, il faut *percer* les minima locaux de l'image, faire *monter* le niveau de l'eau, on construit des *barrages* pour éviter que les eaux des différents bassins ne se mélangent. Les illustrations suivantes sont issues du cours de traitement de l'image, Mr.Ducottet, année 2020.

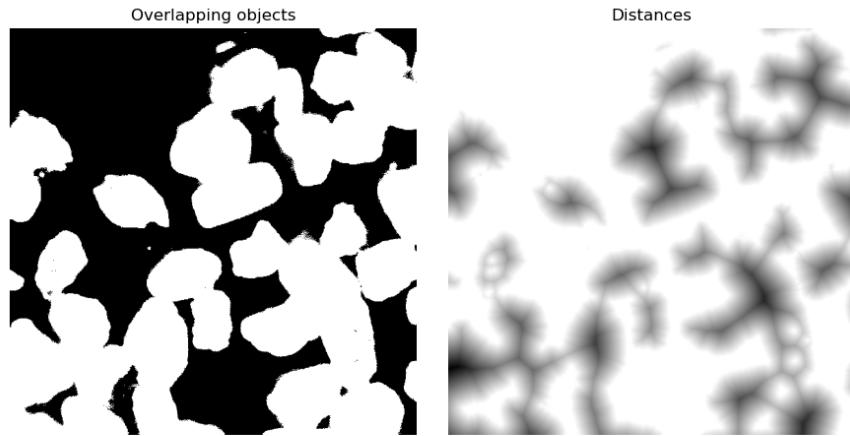


De gauche à droite: les bassins sont remplis, à chaque fois qu'un bassin déborde, une digue - la ligne de partage des eaux- est créée afin de ne pas mélanger les deux eaux.  
Cette ligne va permettre de séparer les régions.

Notre objectif est d'utiliser la méthode de la ligne de partage des eaux non pas sur l'image en elle-même mais sur le résultat de la fonction distance. Celle-ci prend en entrée une image binaire et donne en sortie une image en niveau de gris qui met en avant la distance de chaque pixels de l'objet par rapport au fond. Plus un pixel sera éloigné du fond, plus la distance sera grande.

Le prétraitement est le suivant: convertir l'image en niveau de gris, on applique un filtre laplacien et on soustrait l'image en niveau de gris au laplacien pour accentuer les contours de l'image d'origine. Cela permet de limiter le flou de l'image. Ensuite un threshold automatique en utilisant la méthode du triangle est appliquée, car l'histogramme est assez concentré. On obtient alors une image prétraitée.

L'image de gauche représente l'image prétraitée, cela donne en sortie de la fonction



distance l'image de droite. La distance utilisée est la distance euclidienne.

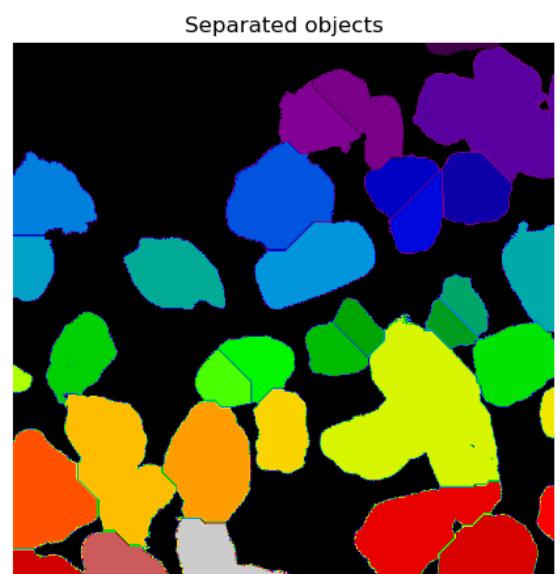
(**scipy.ndimage.distance\_transform\_edt**). Ensuite, il nous faut traiter ce résultat

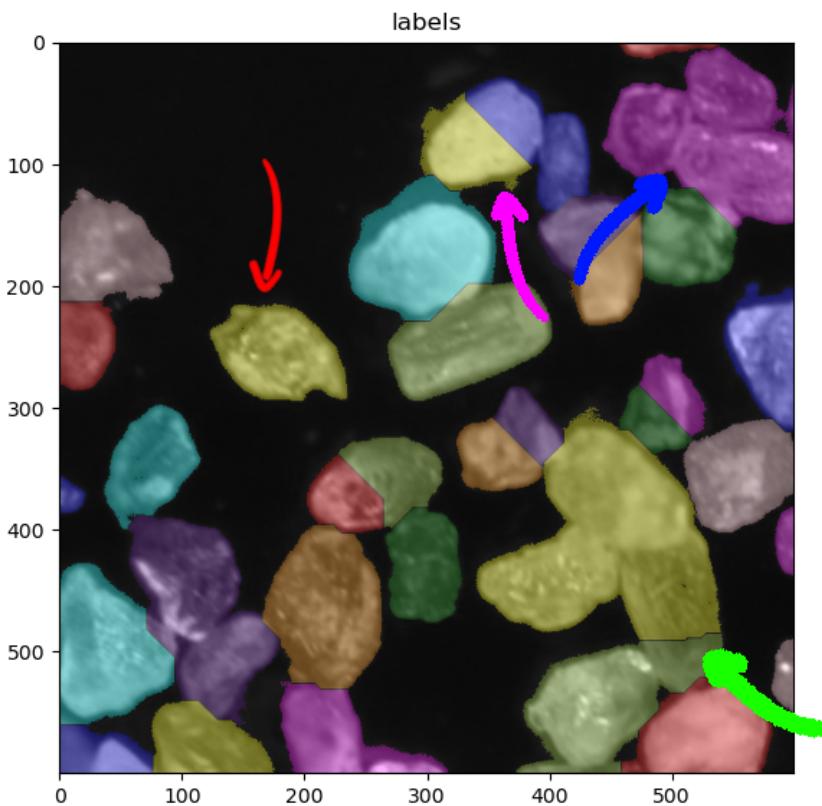
Pour pouvoir segmenter correctement et en particulier séparer les objets qui sont collés, on applique l'opérateur `hmax` (`skimage.morphology.h_maxima`). `hmax` prend en paramètre l'image des distances et un seuil. Si le maximum local est plus important que le seuil `h`, celui-ci sera sauvegardé sinon il sera supprimé. Le seuil a été pris à 10% de la valeur maximale de la distance. Cette valeur permet de se concentrer sur les centres des cailloux. Cette méthode est particulièrement efficace sur des objets convexes se rapprochant de cercles.

Puisque la fonction `distance` renvoie des nombres négatifs, il est à noter que l'on prend l'opposé de l'image des distances pour utiliser la fonction **skimage.segmentation.watershed**.

L'image ci-contre est l'affichage du résultat renvoyé par le `watershed`.

Nous voyons ci-dessous l'aperçu superposé à l'image source, permettant un meilleur visuel.





Chaque couleur représente un objet considéré comme un grain. Au total sur cette image, 34 grains sont détectés. Le résultat de ces traitements n'est pas parfait. En effet, si on superpose les labels trouvés avec l'image initiale, on remarque des différences notables.

Il y a quatre grandes catégories :

- La ***bonne*** segmentation, le grain est assez bien segmenté : flèche rouge
- La ***sur segmentation***, le grain est divisé en plusieurs grains par le programme : flèche violette.
- La ***sous segmentation***, plusieurs grains sont considérés comme étant le même grain par le programme : flèche bleu.
- La ***mauvaise*** segmentation : le label des images s'éloigne fortement de la réalité: flèche verte. Dans le cas soulevé ici 4 grains sont en réalité considérés comme deux. Un grain [réel] est labellisé comme faisant partie à la fois de deux grains [labels].

#### Conclusion sur la méthode watershed :

Cette méthode est particulièrement efficace lorsque l'image présente de fortes régions distinctes, comme par exemple le contour d'un grain blanc sur fond noir. A contrario, si l'objet a des couleurs similaires au fond ou aux objets superposés, cette méthode est peu efficace. Plus les maximaux locaux sont marqués, meilleure sera la segmentation.

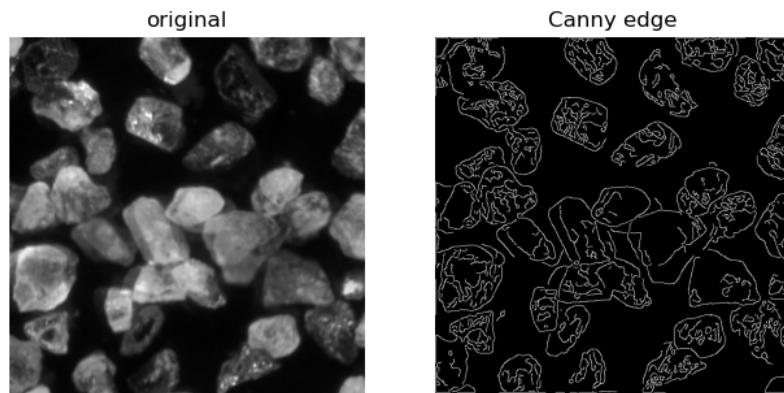
- **Canny+Filling (détection de contours puis remplissage des formes)**

*Méthode de segmentation par détection de contours des objets, puis remplissage des formes retenues, avec filtrage exclusif des petites formes.*

La méthode Canny offre une excellente détection de contours, plus ou moins détaillés en fonction du paramètre de convolution du kernel appliqué à notre image en niveaux de gris.

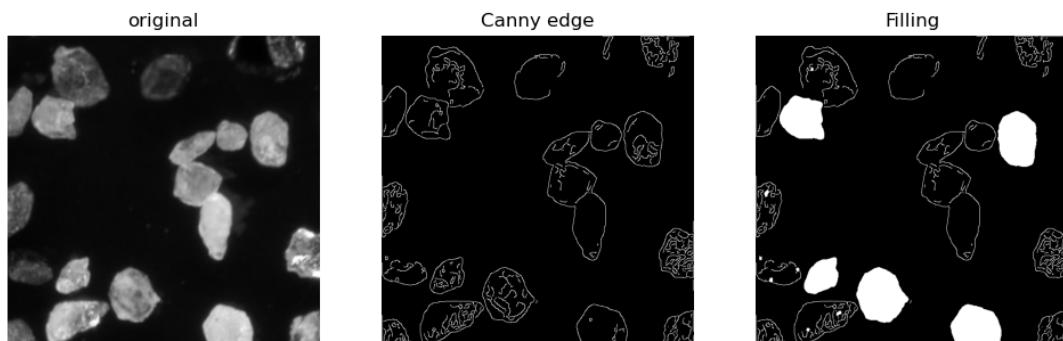
```
from skimage.feature import canny
edges = canny(objects, sigma=1.0)
```

Nous avons laissé le paramètre sigma à 1.0 (défaut) puisque n'impactant que très peu les résultats lors de la sélection de région. Il est cependant utile pour gérer le niveau de détail des contours, notamment pour les images bruitées (puisque  $\sigma$  est la convolution du filtre gaussien appliquée en première passe de l'algorithme Canny).



Une fois les contours détectés, il faut effectuer une morphologie mathématique pour remplir ces contours.

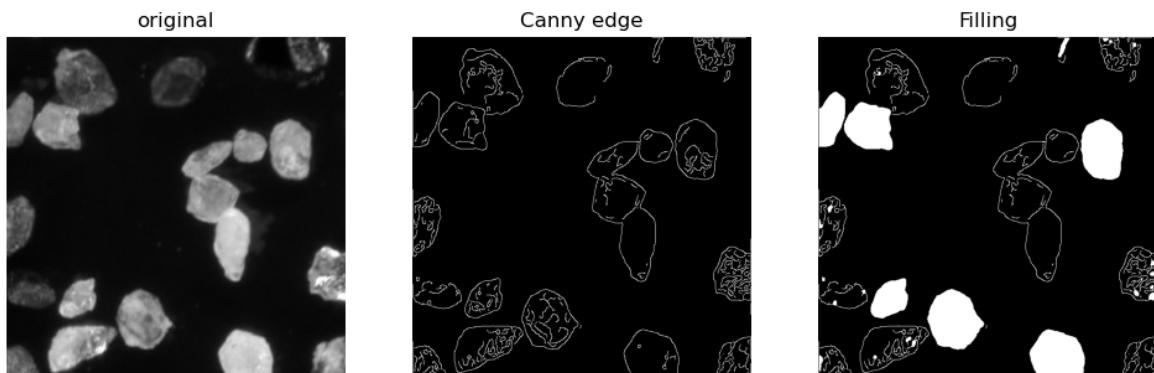
Un [remplissage binaire simple](#) est utilisé, à l'instar de la documentation de scikit image. Cette méthode permet de remplir les "trous" au sein de zones complètement entourées.



On remarque cependant que le remplissage binaire est adapté seulement dans les cas où la forme est parfaitement entourée, ce qui est peu probable pour une sortie de Canny.

Pour répondre à cette problématique, scikit nous permet d'utiliser la fonction `morphology.closing(image, footprint)`. Nous choisirons un disque pour l'empreinte,

avec un rayon initial de 1.5 (disk(1.5))



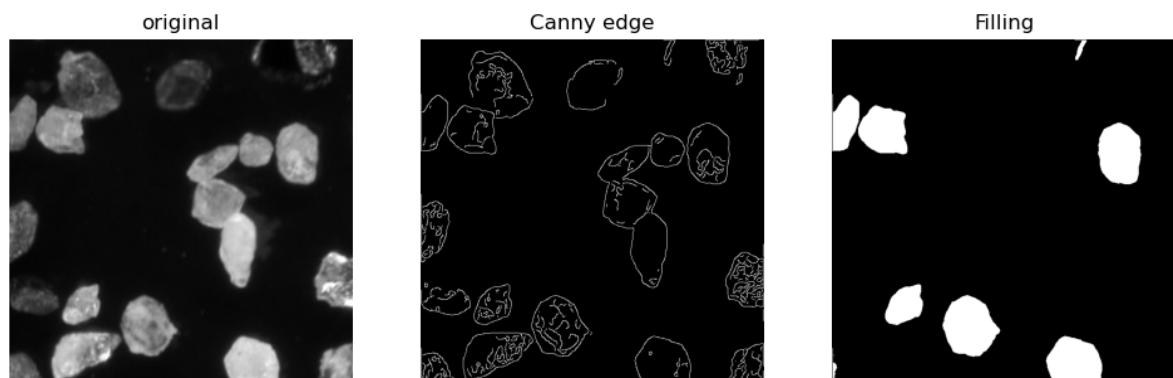
La fonction de closing effectue une opération de dilatation puis d'érosion afin de "fermer" les bords. Documentation Scikit-image:

[https://scikit-image.org/docs/stable/auto\\_examples/applications/plot\\_morphology.html#closing](https://scikit-image.org/docs/stable/auto_examples/applications/plot_morphology.html#closing)

On remarque ainsi quelques cas d'améliorations, surtout en bordures d'image, sans pour autant qualifier la solution de bonne. Nous jouerons avec le rayon du disque de fermeture, pour jauger de cette approche en aval du document.

Afin de retirer les artefacts et contours encore présents sur l'image, une étape de suppression des petits objets est aussi utilisée (taille minimale = 256)

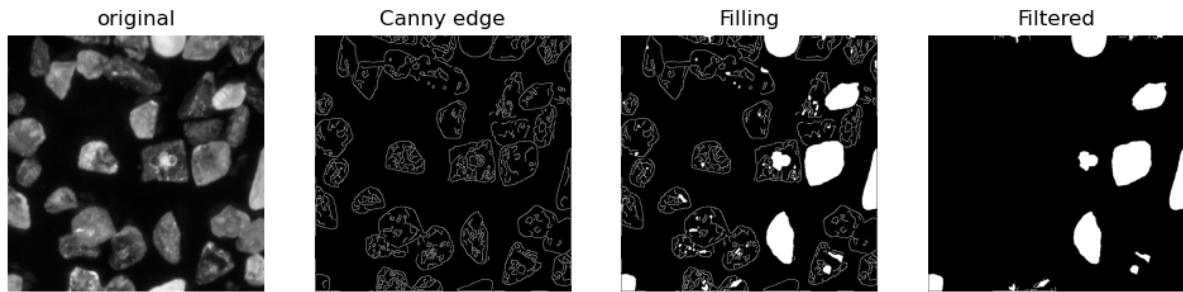
```
morphology.remove_small_objects(fill_obj, min_size=256)
```



En comparant avec l'image précédente, on remarque que seules les régions intéressantes sont conservées.

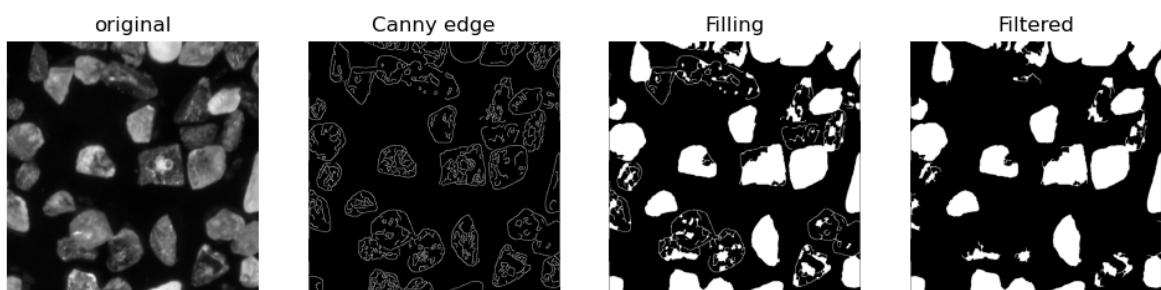
### Essais et analyse sur l'amplitude du disque utilisé pour la fonction de fermeture

*morphology closing avec une empreinte disque de 1.5*



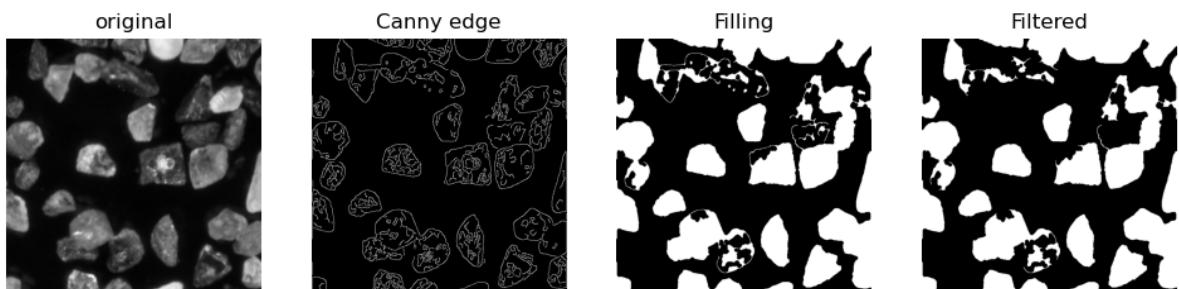
On observe que plus de la moitié des grains ne sont pas retenus, même si le contour est bien défini: Les formes non fermées empêchent le remplissage de région.

*morphology closing avec une empreinte disque de 2.5*



On observe désormais qu'une bonne partie des formes est retenue. Le résultat n'affiche cependant pas la totalité des grains attendus.

*morphology closing avec une empreinte disque de 4*



Avec un grand closing, on apprécie la correction des zones à fermer nous permettant de couvrir à peu près l'ensemble des formes recherchées. Nous nous heurtons cependant à un mur: les objets proches sont fusionnés, empêchant la distinction des grains adjacents (plus probants sur les bords de l'image).

En conclusion, utiliser de façon banale un filtre de Canny pour identifier des régions dans notre cas d'usage n'est pas une bonne solution du fait du manque d'objets reconnus. Ce manque est en grande partie dû aux contours non fermants, générés par l'algorithme. L'application de filtres pouvant améliorer la netteté des contours ne nous ayant pas permis de rendre ceux-ci plus "fermés" (par exemple avec l'application d'un filtre laplacien), la détection de canny étant déjà très efficace quant à la recherche de contours.

Aussi, il est possible d'effectuer une manipulation de fermeture (closing) proposée par scikit image (qui, de façon sous-jacente est une dilatation suivie d'une érosion) afin d'augmenter les chances de fermer les contours. Cette méthode permet des résultats plus proches du but recherché, mais une utilisation à forte intensité (pour tenter de récupérer

l'ensemble des grains) aura pour effet de fusionner les grains adjacents, comme lors de l'utilisation d'un seuil simple.

C'est donc pour ces raisons que la méthode de segmentation par contours (Canny) n'est pas la plus adaptée à notre cas d'utilisation; un compromis important devant être fait entre l'absence d'objets et la distinction entre ceux-ci.

### **Méthodes non supervisées:**

Il s'agit d'une méthode automatique.

- **SLIC (Simple Linear Iterative Clustering)**

Cette méthode utilise les k-means dans l'espace 5D ( $x,y,L,A,B$ ) qui contient l'information de position des pixels dans l'image, mais également l'information colorimétrique. L'utilisation de la distance euclidienne dans l'espace 5D n'est pas appropriée, c'est pourquoi cette méthode utilise une nouvelle distance...

→ Slic permet de segmenter une image en k régions appelées superpixels.

Lien vers la documentation scikit :

<https://scikit-image.org/docs/dev/api/skimage.segmentation.html>

Avant d'appliquer cette méthode, nous avons réalisé une fonction de prétraitement spécifique à SLIC et quickshift qui se constitue des étapes suivantes :

- Thresholding
- Erosion
- Dilatation
- CLAHE

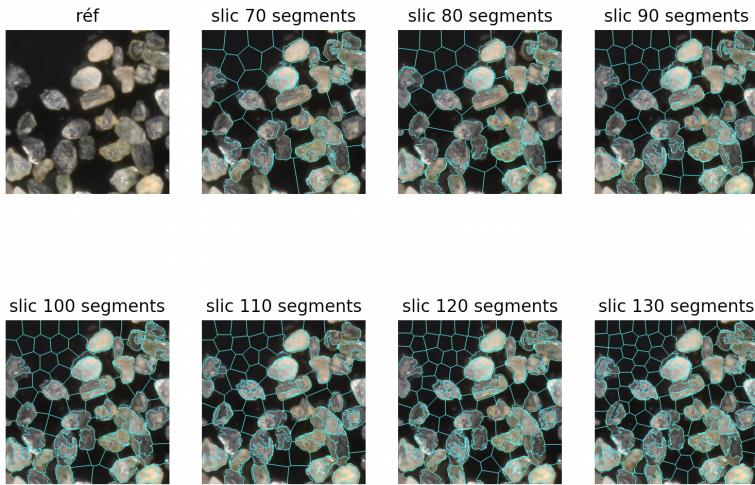
Superpixel : groupe de pixels ayant des caractéristiques similaires

L'utilisation de superpixels permet de diminuer la quantité d'information à traiter sans pour autant diminuer la quantité d'informations brutes de l'image. La fonction slic de scikit prend en arguments différents paramètres :

- `n_segments` : représente le nombre de centre pour l'algorithme k-means
- `compactness` : pour la similarité et la proximité entre les couleurs
- `sigma` : échelle d'approximation de la densité locale

Nous avons alors joué avec les différents paramètres :

### *n\_segments :*



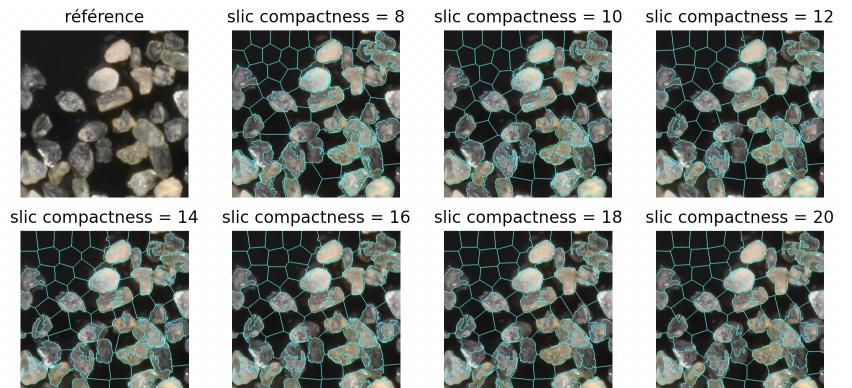
Pour un nombre de segments inférieur ou égal à 80, il nous manque plusieurs contours. Et pour un nombre supérieur à 100, nous rentrons trop dans le détail des grains. On choisira donc une valeur entre 90 et 100 (100 par exemple)

### *compactness :*

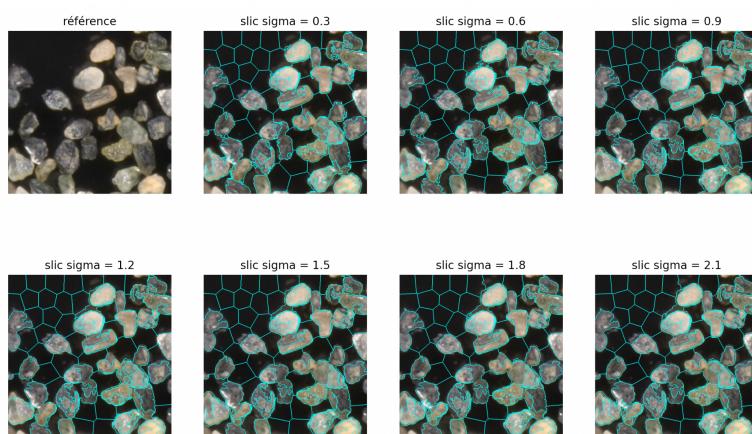
Pour un indice de compactité inférieur à 10, il manque des contours.

De plus la forme des contours change selon l'indice de compactité, on remarque que plus on augmente cet indice, moins l'approximation des contours est bonne.

On retiendra donc un indice de 10.



### *sigma :*

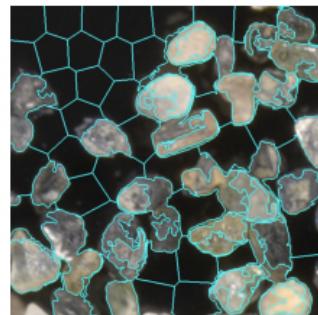


A partir de sigma égal à 2, il ne manque presque plus de contours. On prendra donc cette valeur.

Nous avons également fixé un nombre maximal d'itérations **max\_iter**, en essayant de réduire celui-ci au maximum pour diminuer le temps de chargement du programme, mais pas trop non plus car sinon il manque des segments. La méthode `slic` retourne les contours, il faut alors les juxtaposer sur l'image de base via la fonction **`segmentation.mark_boundaries`**.

On obtient alors la segmentation suivante :

`slic`



Mais avec cette méthode, nous faisons face à un problème : les contours ne sont pas toujours fermés.

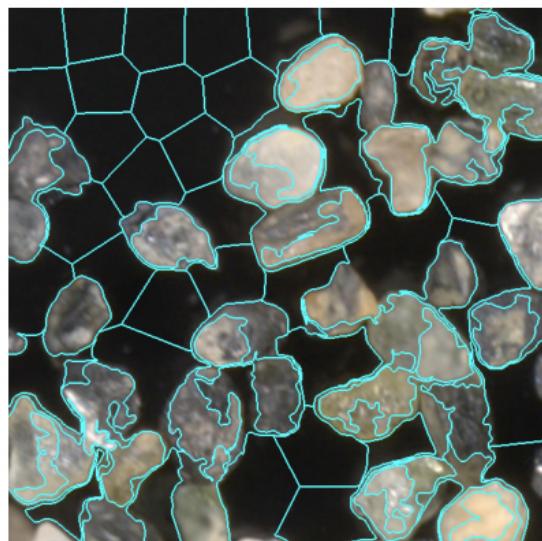
Ainsi nous avons ajouté le paramètre **enforce\_connectivity**, qui, lorsque égal à True, renforce la connexion entre les segments, refermant ainsi les contours.

On a fini par conclure que ces paramètres étaient les plus appropriés pour une meilleure segmentation :

```
segments_slic = slic(im, n_segments=100, compactness=10,
sigma=2.0,start_label=1,convert2lab=True,max_iter=30,
enforce_connectivity=True
)
```

Finalement, nous pouvons observer le résultat :

Optimized Slic



Les segments épousent bien la forme des grains, mais cette méthodologie pose problème dans notre cas car la segmentation se fait également sur fond noir, même en y ajoutant un threshold.

Le résultat n'est pas très agréable à voir, en revanche cette segmentation semblerait adaptée par exemple pour des mosaïques.

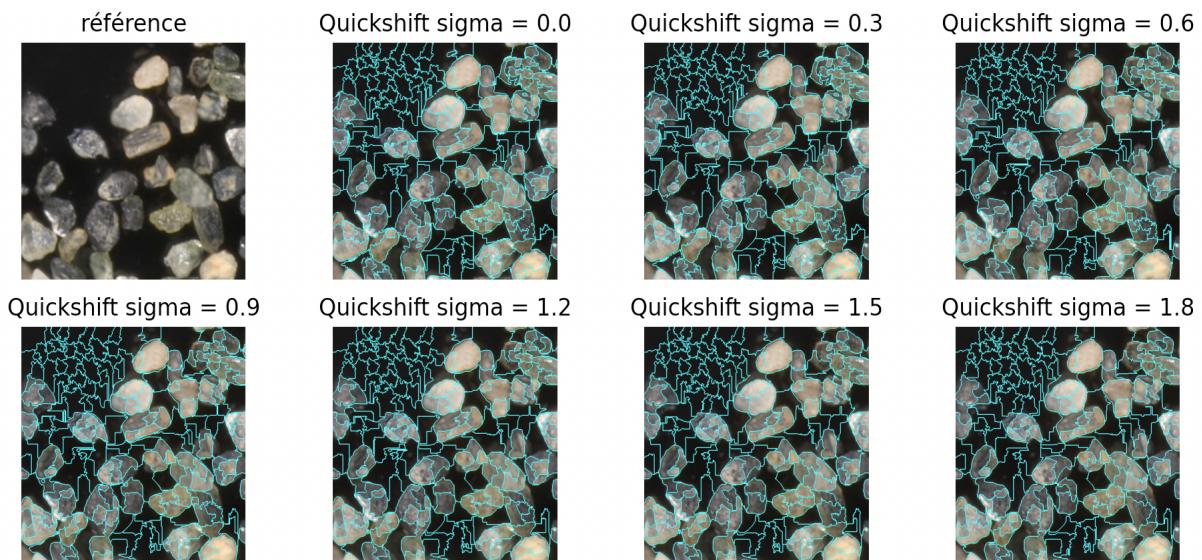
Nous avons une sur segmentation dans les régions les plus sombres, et une sous segmentation dans d'autres régions (par exemple sur le bord droit de l'image nous avons des grains coupés par le bord de l'image qui n'ont pas été segmentés).

### • Quickshift image segmentation

Quickshift est un algorithme de la famille des mode-seeking (tout comme le mean-shift). Celui-ci crée un arbre de liens vers le voisin le plus proche augmentant la densité locale, au lieu de décaler chaque points vers une moyenne locale. Tout comme le slic, cet algorithme s'applique à l'espace 5D mais ne nécessite pas quant à lui que l'image soit dans l'espace LAB.

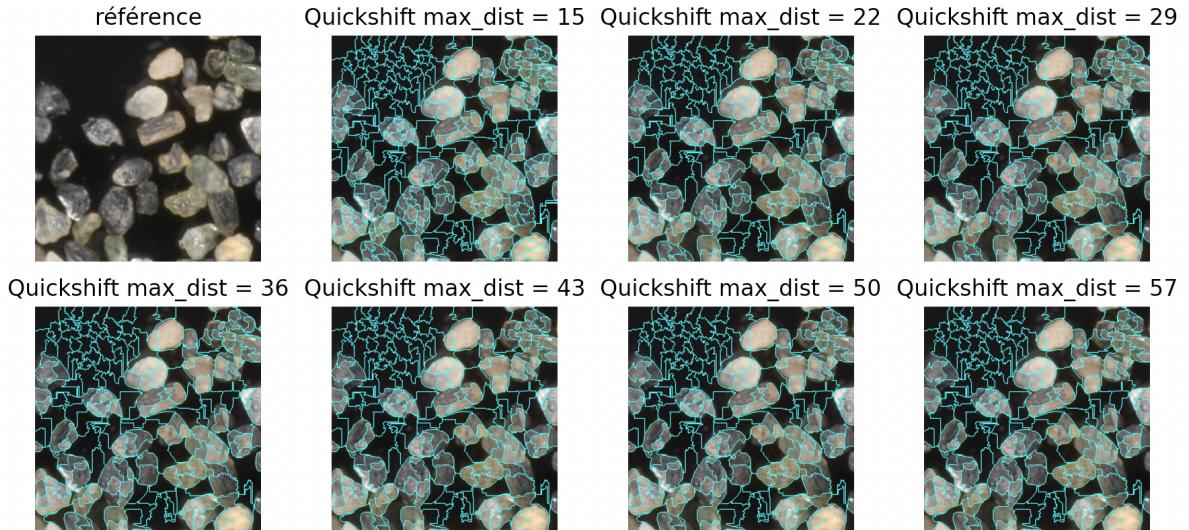
On peut faire varier différents paramètres :

- sigma : échelle d'approximation de la densité locale



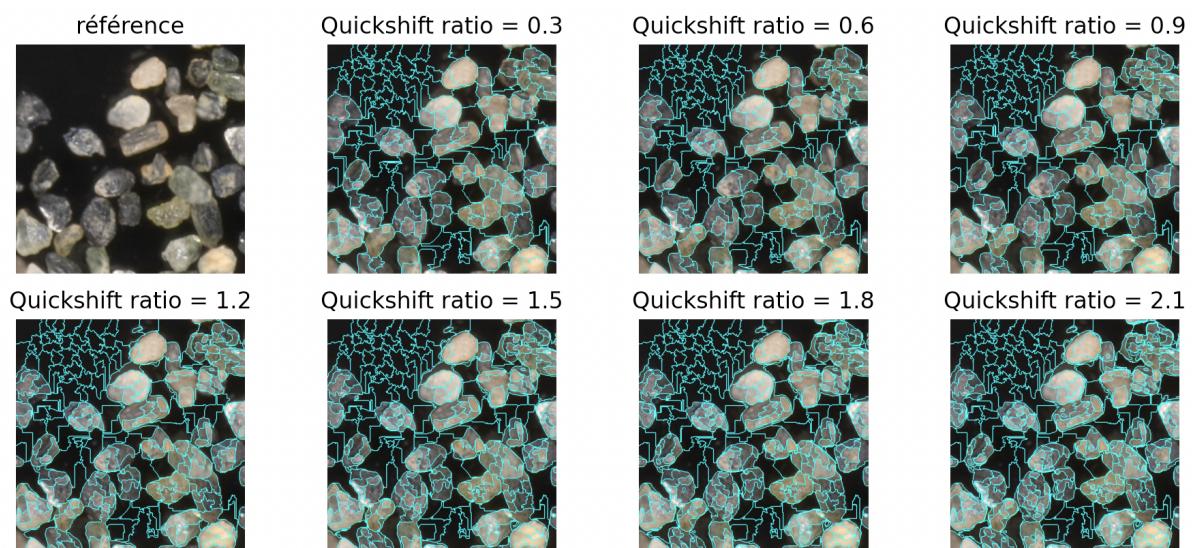
Augmenter Sigma permet une meilleure approximation des contours mais augmente également la complexité des détails. Nous limiterons alors Sigma à 1.

- `max_dist` : limite la profondeur de la segmentation hiérarchique



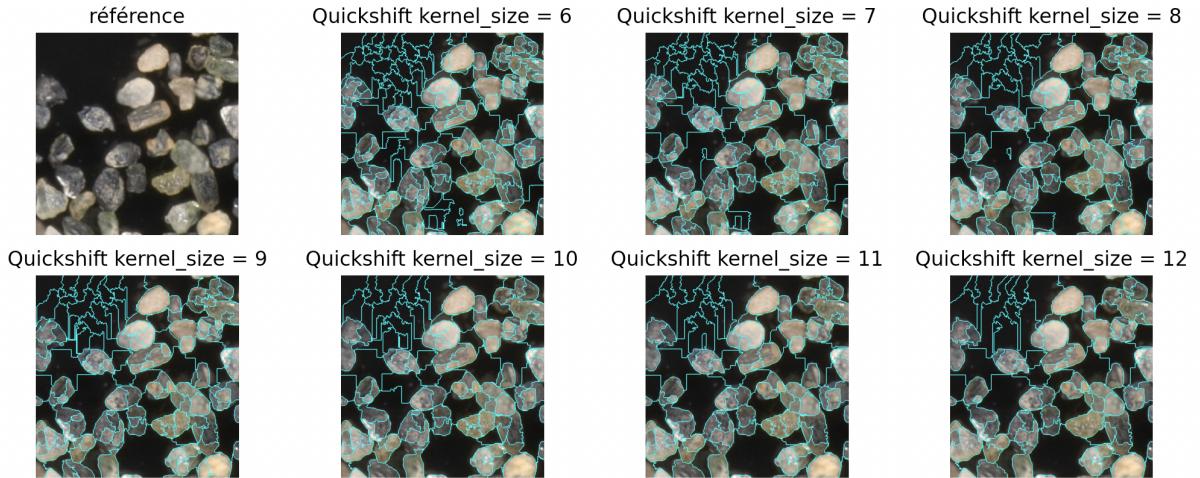
Lorsque la distance est supérieure à 22, nous commençons à perdre des contours (voir le côté droit de l'image), on va donc garder cette valeur `max_dist = 22`.

- `ratio` : pour les distances dans l'espace des couleurs et dans l'espace image



Nous avons une bonne approximation pour un ratio de 1.2, si on prend une plus grande valeur, on rentre plus dans les détails qu'il n'en est réellement nécessaire dans notre cas.

- kernel\_size : taille du noyau gaussien utilisé

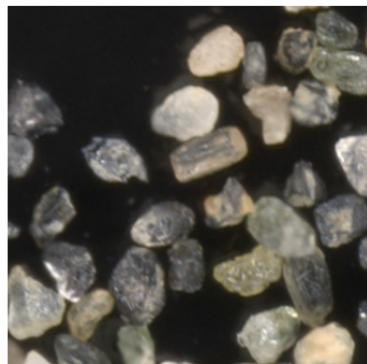


Avec un noyau gaussien faible, nous avons trop de détails/clusters on va donc augmenter la taille de ce noyau pour ne garder que les contours utiles. Lorsque le noyau est supérieur à 9, l'approximation des contours commence à être moins bonne car nous avons perdu des clusters utiles. Donc nous garderons un kernel de taille 9.

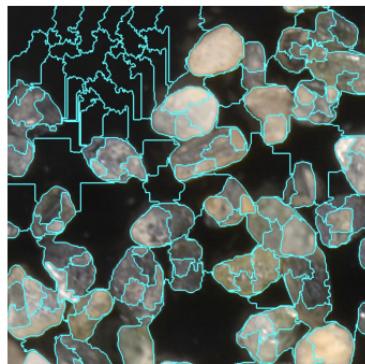
En utilisant les valeurs trouvées pour les paramètres ci-dessus, ainsi que la fonction quickshift ci-dessous on obtient au final la segmentation suivante :

```
segments_quick = quickshift(image4, kernel_size=9, max_dist=22,
ratio=1.2,sigma=1.0)
show_with_matplotlib(img,"référence",1)
show_with_matplotlib(mark_boundaries(img, segments_quick), f"Optimized
Quickshift", 2)
```

référence



Optimized Quickshift



Tout comme le slic, il y a beaucoup de clusters indésirables (que ce soit à l'intérieur des grains ou sur le fond noir). Mais les segments épousent bien la forme des pierres et on obtient une meilleure précision qu'avec slic ( par exemple sur le côté droit nous avons deux grains dont les contours ont été pris en compte contrairement au slic).

En revanche, au niveau de la vitesse, cet algorithme met significativement plus de temps à tourner que pour le slic.

Tout comme le slic, certaines régions sombres sont sur segmentées.

- **Felzenszwalb :**

C'est une méthode basée sur des graphes où les régions connexes représentent les sommets du graphe ainsi que les arcs définissent une relation d'adjacence entre ces régions. Cet algorithme utilise la distance euclidienne minimale entre les pixels dans un arbre couvrant (algorithme de Kruskal) afin de regrouper les régions similaires.

Nous avons utilisé la fonction **segmentation.Felzenszwalb** , elle prend en argument :

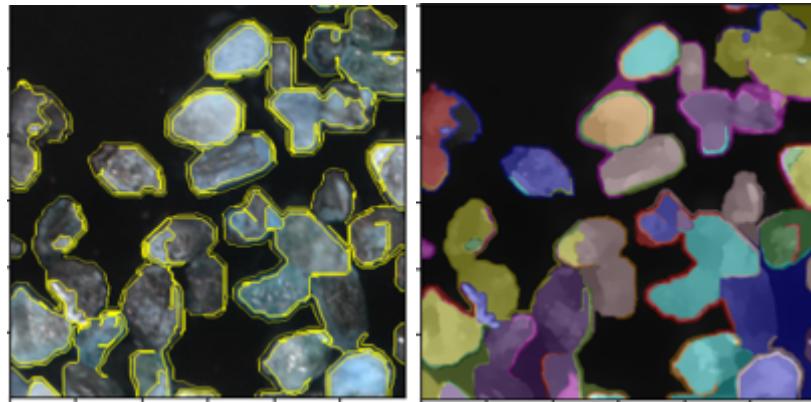
- image à segmenter en RGB
- échelle : plus est élevée, plus les clusters sont grands
- sigma : largeur du noyau gaussien
- min\_size

Après plusieurs tests, nous avons fixé ces valeurs (de telle sorte d'avoir une meilleure segmentation des grains) comme suit :

**segmentation.Felzenszwalb(imageRGB,scale=200,sigma=3,min\_size=200)**

**Remarque :** plus Sigma est petit, plus l'image est bruité . plus sigma est trop grand (supérieur à 250 à peu près), plus l'image résultante est mal segmentée (les labels sont moins clairs).

Nous obtenons le résultat présenté dans la figure suivante:



Pour ce type de problème, ce n'est pas l'algorithme le plus adapté dans le sens où il ne délimite pas correctement l'arrière-plan et les grains. En revanche, en le comparant avec d'autres algorithmes, il est meilleur que SLIC et le quickshift.

## Méthode retenue :

Nous récapitulons les différents résultats des études élaborées précédemment dans le tableau suivant :

Méthode	Avantages	Inconvénients
Thresholding	Bonne délimitation de l'arrière-plan et les grains	mauvaise segmentation pour les grains overlapped
Random walker	Bonne délimitation de l'arrière-plan et les grains	mauvaise segmentation pour les grains overlapped
Watershed	Séparation des grains efficaces quand ils sont distincts du fond.	Méthode demandant beaucoup de calculs. Les grains ayant une couleur proche de leurs voisins ou que le fond sont mal segmentés.
Canny	Bonne segmentation des grains retenus	Beaucoup de grains non retenus à cause de contours non fermés
	Bonne segmentation du contour des grains	-Sur segmentation des grains sombres

SLIC	+/- bonne séparation des grains confondus	-Sous segmentation de grains coupés par le bord de l'image -Segmentation de l'arrière fond -Contours pas toujours fermés
Quickshift	Bonne segmentation du contour des grains +/- bonne séparation des grains confondus	-Sur segmentation des grains sombres -Segmentation du fond -Contours pas toujours fermés -Rapidité
Felzenszwalb	+/- bonne séparation des grains	mauvaise délimitation de l'arrière-plan et les grains

La méthode de segmentation retenue dans notre cas est le [\*\*Watershed\*\*](#).

Au regard des images fournies, la méthode qui utilise le watershed donne les meilleurs résultats. En effet, elle a l'avantage de supprimer peu ou pas de grains lors de son utilisation. Tous les grains sont considérés comme étant des objets. Cependant, la labellisation peut-être améliorée, comme expliqué au paragraphe [Watershed](#).

## Extraction des données

Après avoir segmenté l'image puis l'avoir labellisée, on fait la moyenne des valeurs de chaque canal pour chaque grain, on obtient un tableau semblable à celui-ci:

	Moyenne de B	Moyenne de G	Moyenne de R
Grain isolé 1	20	23	24
Grain isolé 2	80	103	115
Grain isolé 3	114	135	153
Grain isolé 4	74	90	93
Grain isolé 5	98	112	117
Grain isolé 6	100	121	139
Grain isolé 7	106	129	146
Grain isolé 8	130	144	153
Grain isolé 9	77	77	80
Grain isolé 10	92	108	122
Grain isolé 11	94	95	95
Grain isolé 12	133	140	147
Grain isolé 13	108	109	111
Grain isolé 14	78	84	88
Grain isolé 15	69	81	85
Grain isolé 16	86	103	116
Grain isolé 17	85	106	120
Grain isolé 18	70	75	77
Grain isolé 19	85	89	92
Grain isolé 20	89	94	99
Grain isolé 21	82	84	86
Grain isolé 22	84	103	107

Le détail de la dataframe est aussi sauvé sous forme de CSV, retrouvable dans le dossier “resultats/”

	A	B	C	D
1		Moyenne de B	Moyenne de G	Moyenne de R
2	Grain isolé 1	24	23	20
3	Grain isolé 2	115	103	80
4	Grain isolé 3	153	135	114
5	Grain isolé 4	93	90	74
6	Grain isolé 5	117	112	98
7	Grain isolé 6	139	121	100
8	Grain isolé 7	146	129	106
9	Grain isolé 8	153	144	130
10	Grain isolé 9	80	77	77
11	Grain isolé 10	122	108	92
12	Grain isolé 11	95	95	94
13	Grain isolé 12	147	140	133
14	Grain isolé 13	111	109	108
15	Grain isolé 14	88	84	78
16	Grain isolé 15	85	81	69
17	Grain isolé 16	116	103	86
18	Grain isolé 17	120	106	85
19	Grain isolé 18	77	75	70
20	Grain isolé 19	92	89	85
21	Grain isolé 20	99	94	89
22	Grain isolé 21	86	84	82
23	Grain isolé 22	107	103	84
24	Grain isolé 23	80	79	82
25	Grain isolé 24	66	64	59
26	Grain isolé 25	105	98	87
27	Grain isolé 26	87	83	79
28	Grain isolé 27	137	132	117
29	Grain isolé 28	82	81	80
30	Grain isolé 29	120	117	99
31	Grain isolé 30	184	162	126
32	Grain isolé 31	139	126	112
33	Grain isolé 32	77	75	68

## Conclusion

Ce travail pratique a permis de se documenter sur les différentes approches utilisables afin de segmenter et de labelliser une image. D'une approche par histogramme en passant une approche locale ou globale, chaque méthode a ses avantages mais aussi ses inconvénients. Une méthode peut-être adaptée à des cas de figure mais est rarement utilisable dans le cas général. Dans notre cas, la diversité des grains est source de problèmes : de formes et de couleurs différentes, il n'est pas facile d'appliquer un algorithme en particulier. C'est pourquoi les méthodes automatiques sont utilisées, elles permettent d'avoir une marge de manœuvre afin d'avoir des résultats exploitables.

Même si nos approches n'ont pas permises d'avoir la *meilleure* segmentation, elles permettent d'avoir des démarches qui pourront être reprises et améliorées afin d'avoir un meilleur résultat.