Mona Mokart

TP3 IMA201
segmentation des images

1 Détection de contours

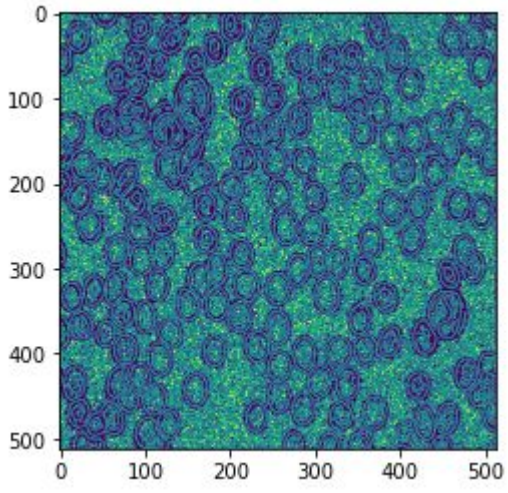
1.1 Filtre de gradient local par masque

Le filtre Sobel calcule le gradient à partir d'un masque de taille 3:

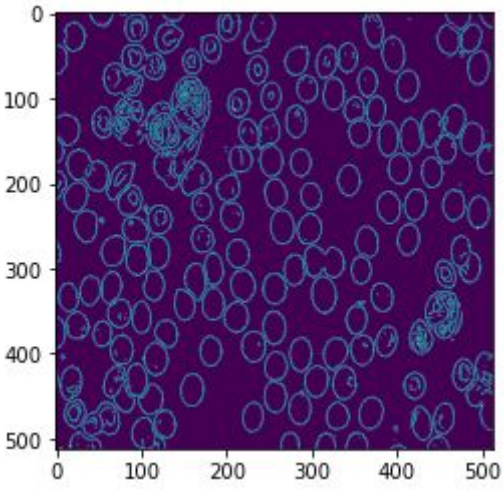
1		-1
2		-2
1		-1

Théorie: Appliquer un passe bas permet d'enlever les grandes fréquences donc les détails, or on ne veut pas les détails on veut les contours. Appliquer un passe bas permet de flouter les détails qui peuvent être important en intensité et donc brouiller l'image en passant par le gradient.

Pratique:



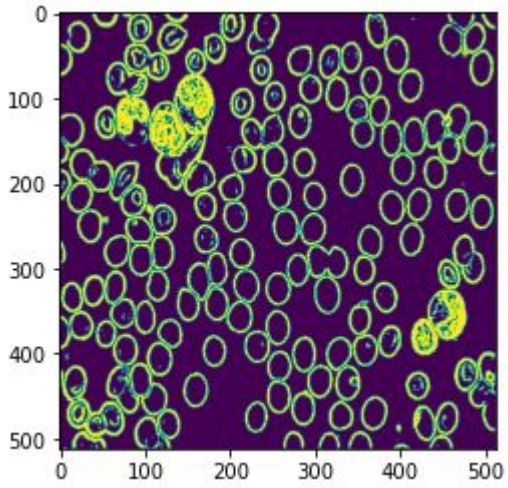
a) contours sur l'image originale



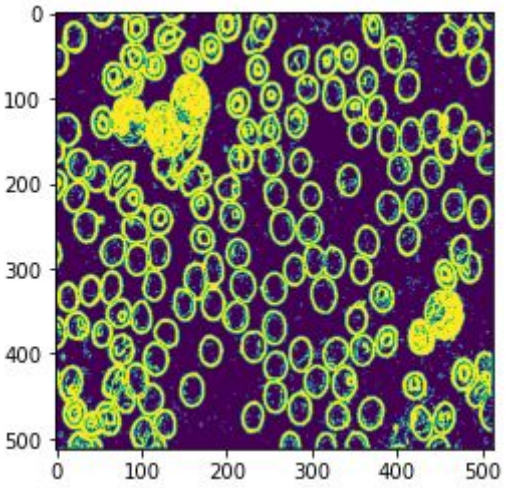
b) contours sur l'image après passe-bas

Pour le même seuil, on remarque une très grande différence si on enlève les hautes fréquences ou non. On observe un bruit très important pour l'image non filtrée

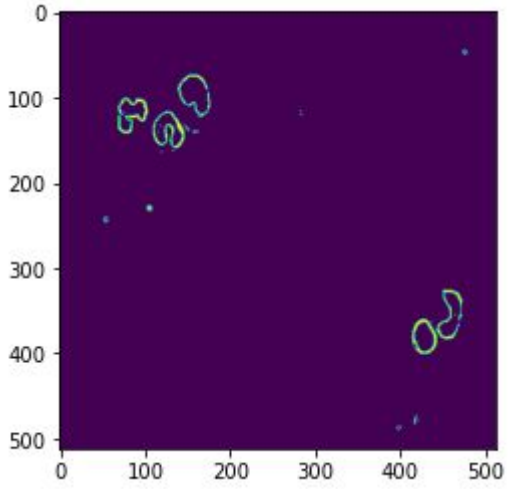
Seuillage norme gradient:



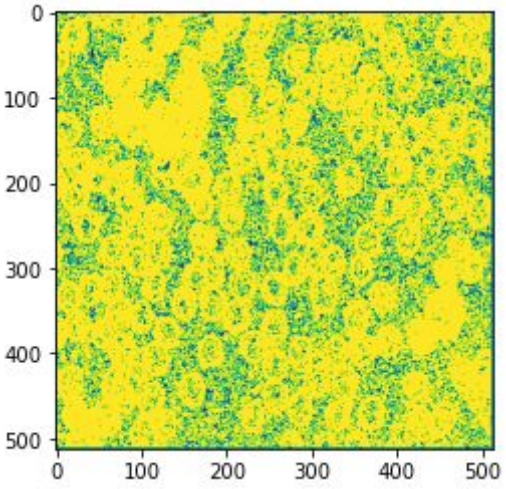
a) seuil= 0.1



b) seuil = 0.05



c) seuil = 0.5



d) seuil = 0.01

On remarque que plus le seuil est grand, plus les contours sont propres, mais aussi moins il y a d'information. Pour le seuil = 0.5 les contours sont biens définis, il y a une bonne discontinuité mais comme il est trop grand, il ne prend que les informations des zones ayant un très grand gradient.

Or quand le seuil est trop petit, cela laisse passer plus d'information que les contours, et donc on observe un bruit.

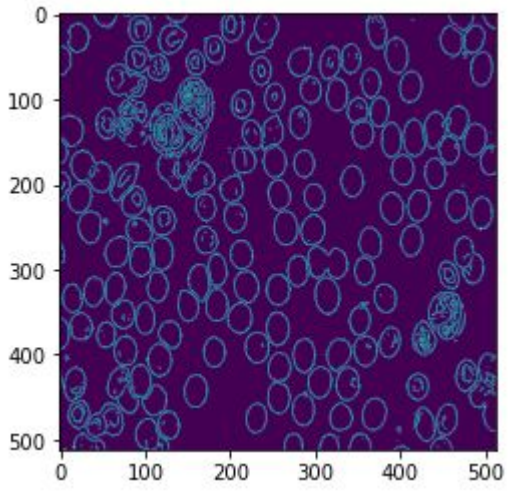
Pour seuil = 0.01 on n'observe pas bien les contours, le seuil est beaucoup trop petit et on observe beaucoup de bruit, le seuil laisse passer toutes les valeurs de la norme du gradient.

Pour seuil = 0.05 on commence à distinguer tous les contours mais il reste un peu de bruit

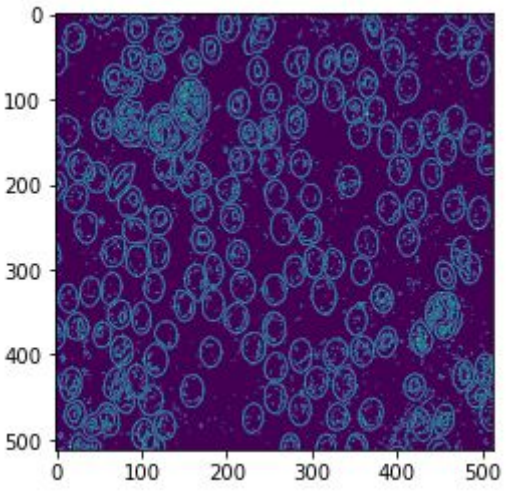
Pou seuil = 0.1 cela semble être le seuil idéal, on distingue tous les contours sans avoir trop de bruit.

1.2 Maximum du gradient ltre dans la direction du gradient

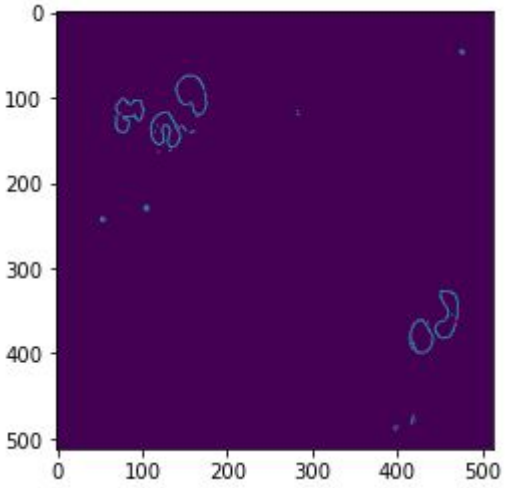
L'utilité du maximum du gradient dans sa direction permet de récupérer au mieux les contours puisqu'ils correspondent à un maximum de "discontinuité" au niveau des valeurs de l'image, donc un maximum du gradient.



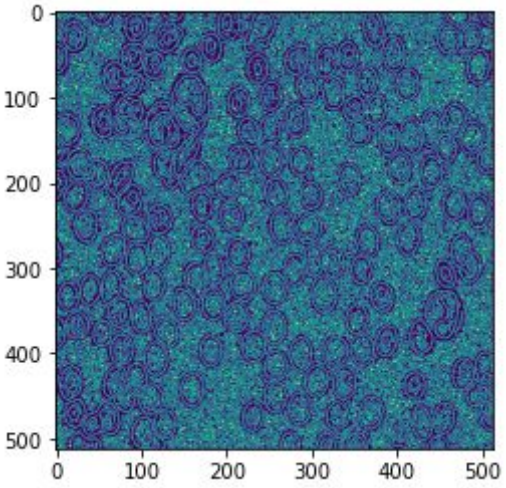
a) seuil = 0.1



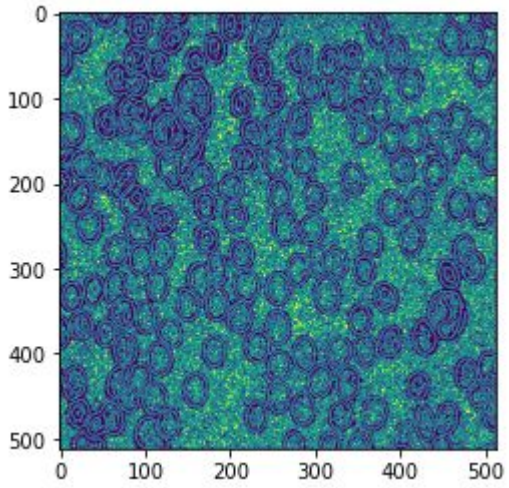
b) seuil = 0.05



c) seuil = 0.5



d) seuil =0.01



e) maxima du gradient dans la direction du gradient

On remarque que la variation du seuil impact les cintours de la même façon qu'avec la norme du gradient. Lorsque le seuil est trop petit on observe trop de bruit, et quand le seuil est trop grand on n'oberve pas tous les contours

Pour seuil = 0.01 on remarque que cela ne change presque rien à l'image du maximum de gradient dans sa direction, le seuil est trop petit et donc laisse passer toutes les valeurs.

Le seuil = 0.1 semble là aussi être le meilleur compromis

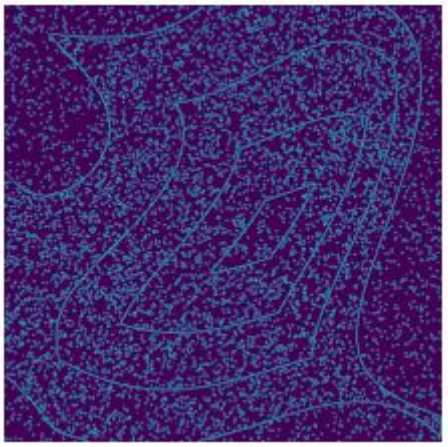
1.3 Filtre récursif de Deriche

```python
def dericheGradX(ima,alpha):


    nl,nc=ima.shape
    ae=math.exp(-alpha)
    c=-(1-ae)*(1-ae)/ae

    b1=np.zeros(nc)
    b2=np.zeros(nc)

    gradx=np.zeros((nl,nc))


#gradx=np.zeros(nl,nc)
    for i in range(nl):
        l=ima[i,:].copy()
        for j in range(2,nc):
            b1[j]=l[j-1] +2*math.exp(-alpha)*b1[j-1] - math.exp(-2*alpha)*b1[j-2]
        b1[0]=b1[2]
        b1[1]=b1[2]
        for j in range(nc-3,-1,-1):
            b2[j]=l[j+1]+2*math.exp(-alpha)*b2[j+1] - math.exp(-2*alpha)*b2[j+2]
        b2[nc-1]=b2[nc-3]
        gradx[i,:]=c*ae*(b1-b2);
```

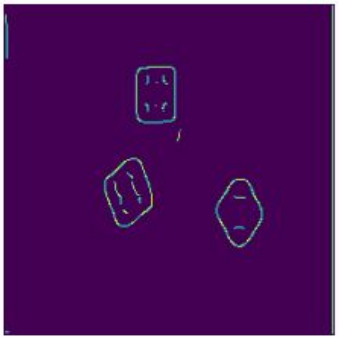
```python
def dericheGradY(ima,alpha):


    nl,nc=ima.shape
    ae=math.exp(-alpha)
    c=-(1-ae)*(1-ae)/ae

    b1=np.zeros(nl)
    b2=np.zeros(nl)

    grady=np.zeros((nl,nc))

    for i in range(nc):
        l=ima[:,i].copy()
        for j in range(2,nl):
            b1[j]=l[j-1]+2*math.exp(-alpha)*b1[j-1] - math.exp(-2*alpha)*b1[j-2]
        b1[0]=b1[2]
        b1[1]=b1[2]
        for j in range(nl-3,-1,-1):
            b2[j]=l[j+1]+2*math.exp(-alpha)*b2[j+1] - math.exp(-2*alpha)*b2[j+2]
        b2[nl-1]=b2[nl-3]
        b2[nl-2]=b2[nl-3]

        grady[:,i]=c*ae*(b1-b2);


    return grady
```

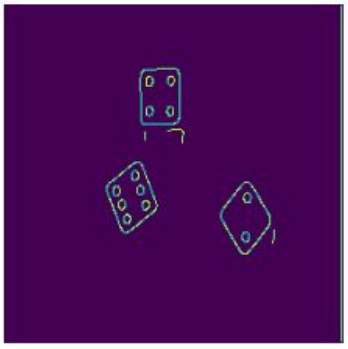
Avec un seuil = 20 :

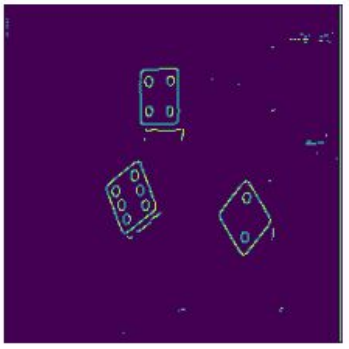
a) α = *1* *b)* α = *0.3*



c) α *= 1.5* *d)* α *= 3*





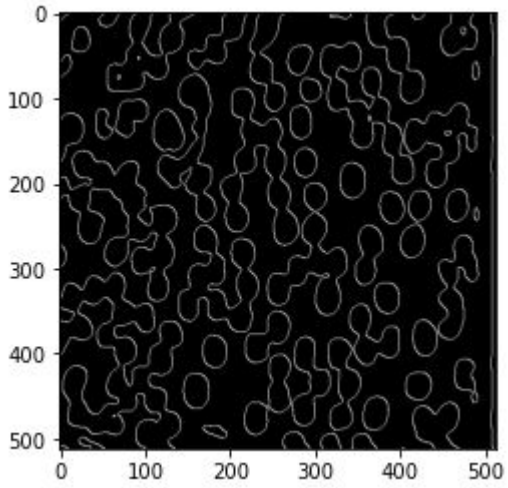


a) α *= 0.3* *b)* α *= 1* *c)* α *= 3*

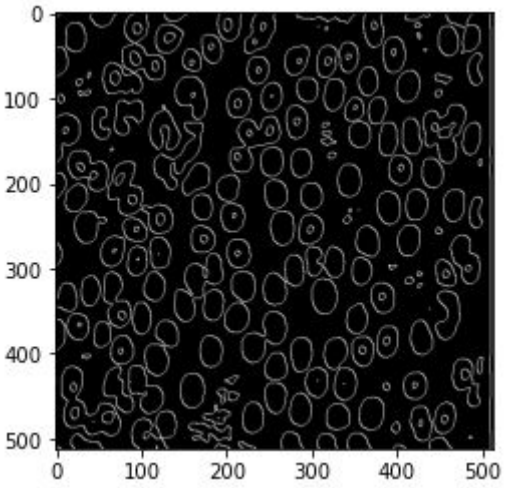
Le temps de calcul ne change pas selon alpha car il n'y a pas d'itération en plus
On remarque que si alpha est trop grand on observe du bruit et si alpha est trop petit on perd des détail.

dericheSmoothX et dericheSmoothY fonctionnent sur le même principe que sobel

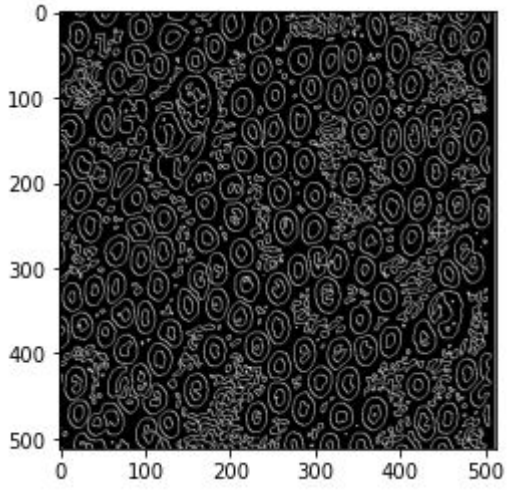
1.4 Passage par zéro du laplacien



a) α *= 0.3*



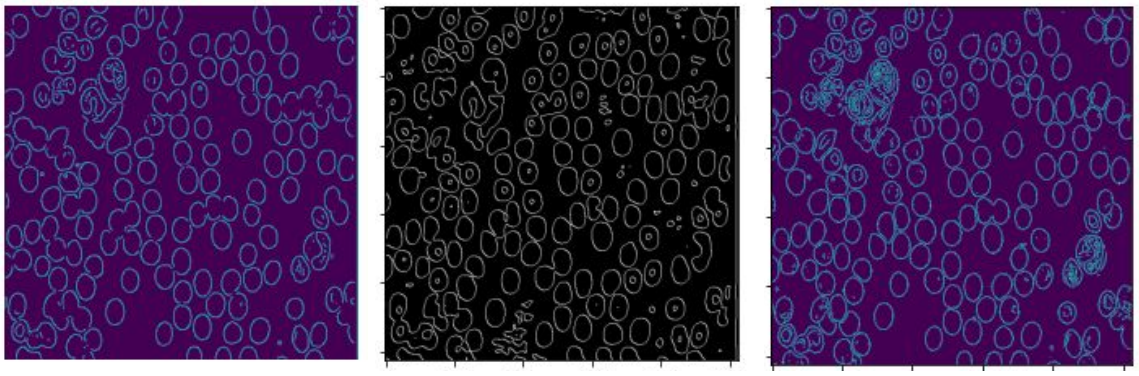
b) α *= 0.5*



c) α *= 1*

Plus alpha est grand, plus on a de détails mais cela reste “propre”, bien défini, ça ne ressemble pas à du bruit. Ici alpha = 0.5 semble correct.

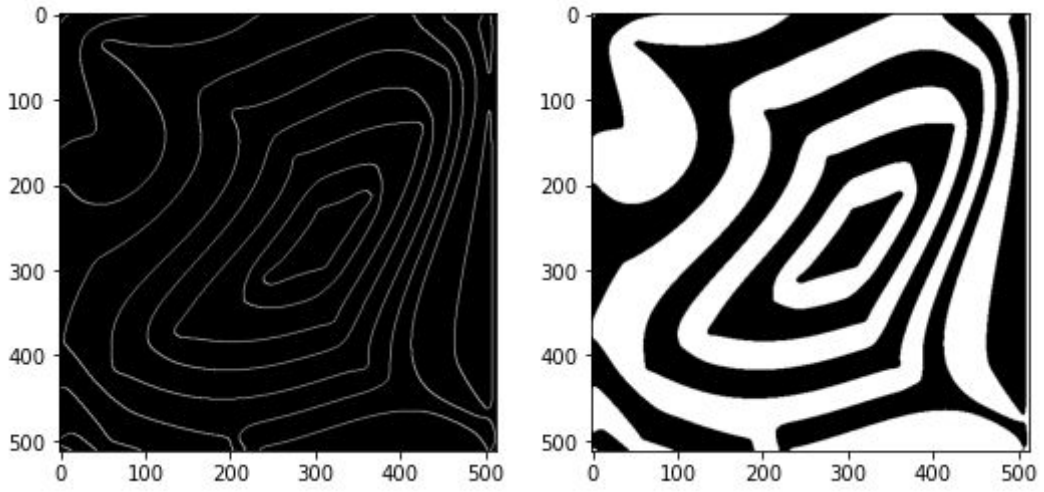
Comparaison des méthodes:



Deriche α = 1 seuil = 13 *Laplacien α = 0.5* *Sobel seuil = 0.1*

Pour le fichier cell.tif, Sobel est plus sensible au bruit que pour le zéro du laplacien ou deriche. (Alors que théoriquement le laplacien est sensible au bruit quand alpha augmente) Entre deriche et la technique du laplacien cela semble se valloir, cependant le laplacien semble être plus performant, car il peut y avoir des détails qui ne sont pas des contours, mais cela ne parait bruité, tandis que les détails chez deriche ressemblent à du bruit.

Quand on utilise ce code pour pyramide.tif, on observe ceci:

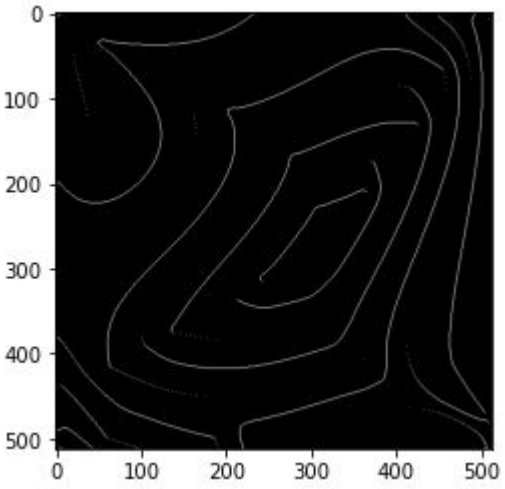


a) *Faux contours* *image 'posneg'*

On remarque donc qu'il y a des faux contours. Mais cela est dû au fait que dès qu'il y a une différence de pixel sur l'image 'poseneg' on marque un contour. Or il faudrait pour éviter cela, marquer le contour que lorsqu'il y a rapport de supériorité entre les pixels:

```
nl,nc=ima.shape
contours=np.uint8(np.zeros((nl,nc)))


for i in range(1,nl):
    for j in range(1,nc):
        if (((i>0) and (posneg[i-1,j] > posneg[i,j])) or
            ((j>0) and (posneg[i,j-1] > posneg[i,j]))):
            contours[i,j]=255


plt.figure('Contours')
plt.imshow(contours, cmap='gray')
```

Avec ce changement dans le code on obtient cela :
On retrouve alors à peu près les bons contours.
Ils sont cependant pas très bien placés en comparaison avec deriche ou sobel



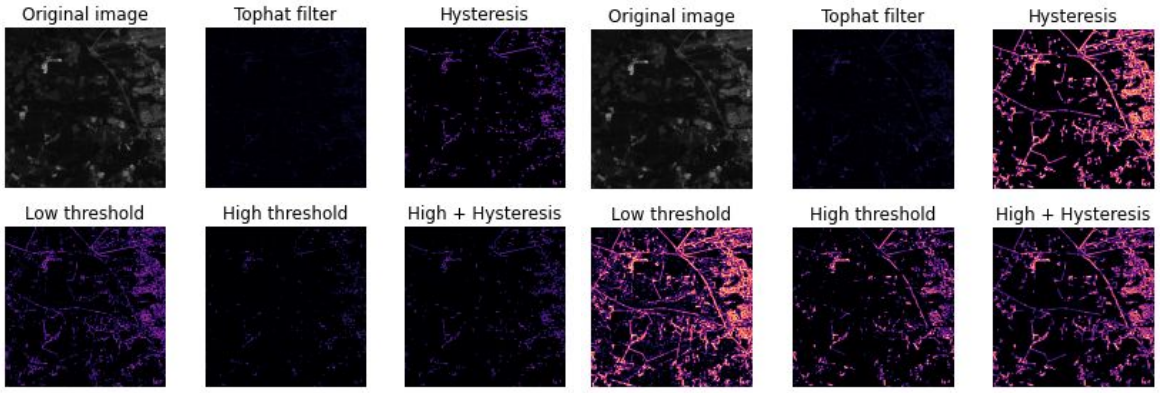
1.5 Changez d'image

Avec pyra-gauss.tif je choisirais deriche car il fait apparaître moins de bruit que sobel et que l'algorithme du zéro du laplacien crée des faux contours et mal placés.
Avec alpha = 0.5 j'obtiens:

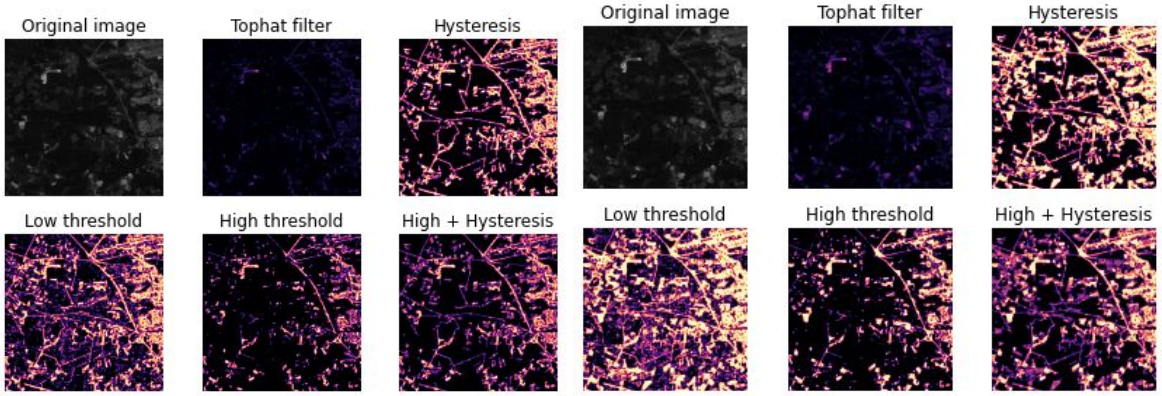
C'est un bon résultat
Si j'avais voulu utilisé sobel j'aurais mis un filtre médian pour flouter le bruit avtn d'appliquer l'algorithme

2 Seuillage avec hystérésis

<u>Pour différents rayons dans la fonction tophat:</u>



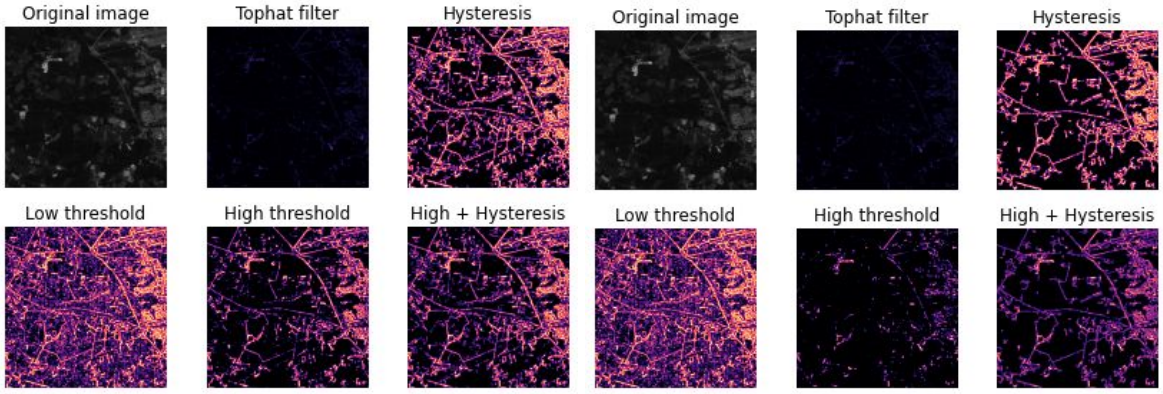
r = 1 *r = 3*



r=5 *r=10*

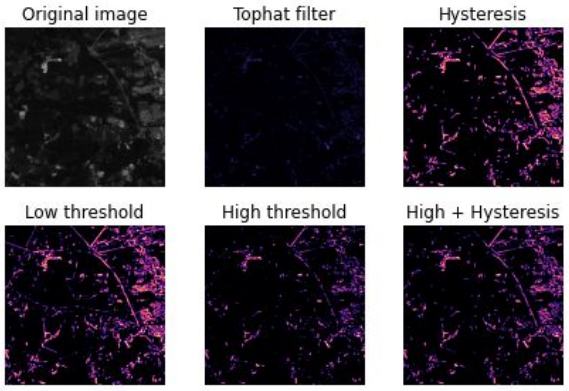
Plus r est grand plus on voit les ligne, or on voit également plus les autres parties de l'image.

<u>Les seuils:</u>



seuils 1 et 3 *seuil 1 et 10*

En augmentant le deuxième seuil on perd des lignes précises



seuils 5 et 10
en augmentant le premier seuil on perd les lignes principales

3 Segmentation par classification : K-moyennes

3.1 Image a niveaux de gris

image cells.tif après traitement par l'algorithme k-means

L'image est bien séparée en deux classes : les cellules et le fond. Si on avait voulu améliorer l'algorithme on aurait pu augmenter le n_init qui correspond au nombre de fois ou on change les centroids. Par défaut n_init = 10

En changeant quelques paramètres on obtient:

avec n_init = 100 *avec n_init = 1*

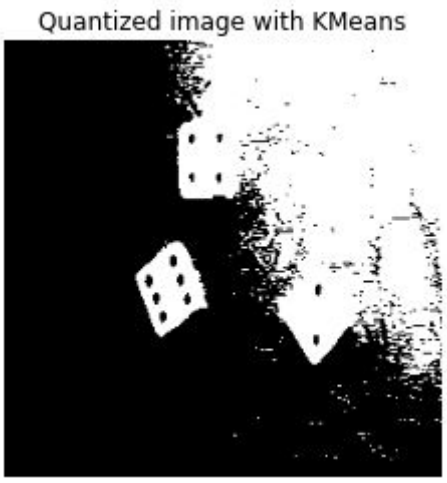
Finalement on n'a pas besoin d'augmenter le n_init car on obtient pareil pour 10 et pour 100. Par contre pour n_init = 1 on a un mauvais résultat, on va donc garder la valeur par défaut.

ici on a changé init = kmeans++ par init = random. On obtient le même résultat

init = random signifie que les centroids sont choisis aléatoirement
init = kmeans++ sélectionne les centroids de façon à accélerer la convergence

Ici je n'ai pas l'impression que ça change vraiment.

La classification est stable même avec init = 'random'
Ce n'est pas le cas quand on choisi random_state = 9 ou 10:
Pour 2 classes:



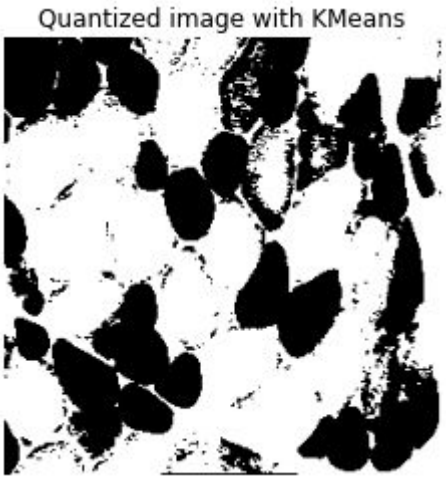
dice1: init = 'random', init = kmeans++ mais avec random_state = 0

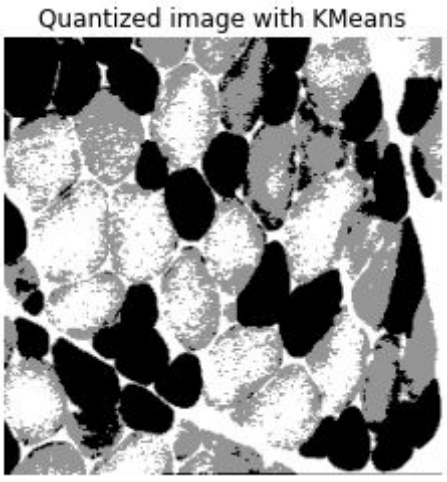
init = 'random' et random_state= 10

Le seul cas où on obtient pas la même répartition dans les classes c'est quand on active init = 'random' et random_state = 9 ou 10

random_state détermine la génération de nombres aléatoires pour l'initialisation des centroïdes.

Pour muscle.tif il est difficile d'isoler les cellules car elle n'ont pas le même niveau de gris.



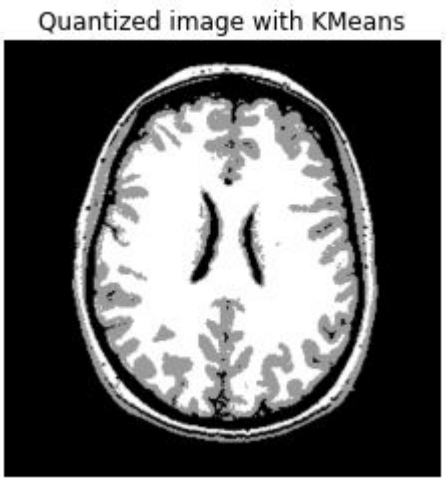


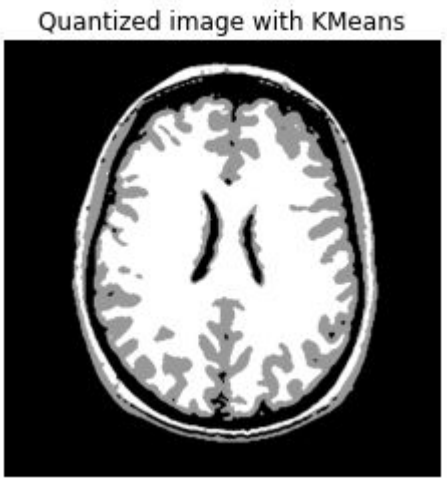
Avec 2 classes

Avec 3 classes

On ne distingue pas toutes les cellules, certaines sont prise dans la classe "fond" car elles ont un niveau de gris plus proche du fond que des autres cellules.

Filtrer permet de préparer à la segmentation puisque cela réduit le bruit et donc permet d'uniformiser les régions qui se ressemblent.

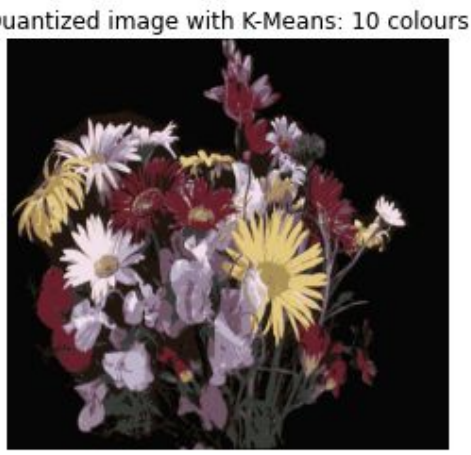




kmeans sur cerveau.tif 3 classes sans filtre *avec filtre médian*

Les régions sont plus uniformes après avoir filtré

3.2 Image en couleur

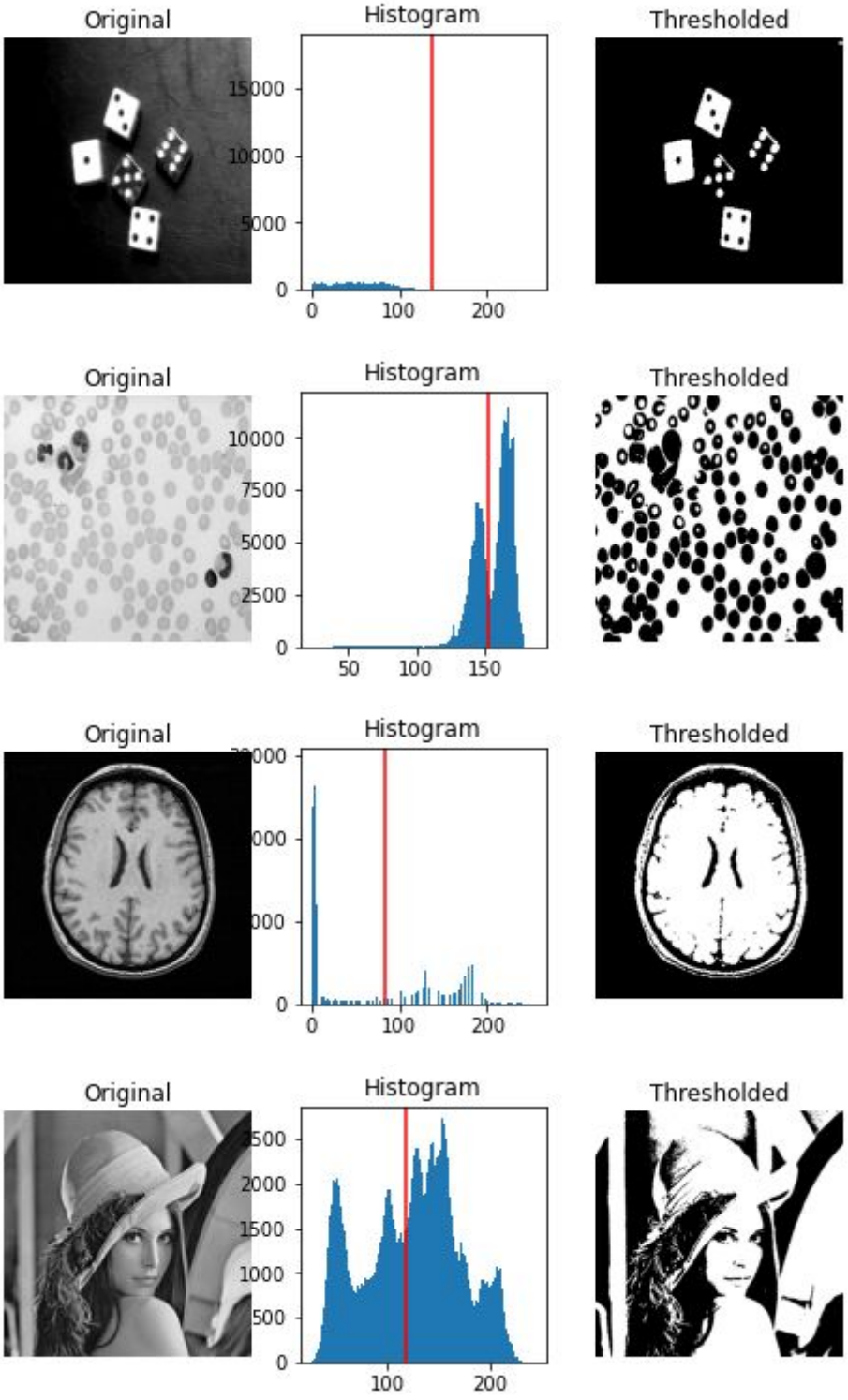


On remarque qu'après la segmentation on perd de la couleur et du relief

Avec 40 couleurs, l'image est très ressemblante mais avec 50 couleurs, l'image est presque la même :

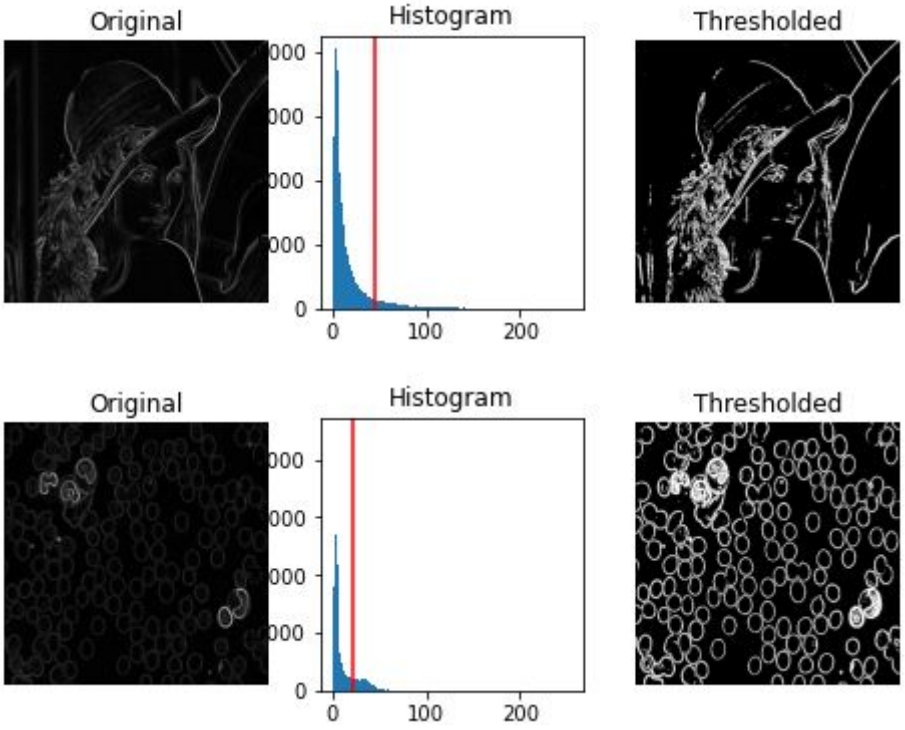
4 Seuillage automatique : Otsu

Ici on cherche à optimiser la valeur du seuil pour minimiser la variance intra-classe



Cela fonctionne relativement bien, on retrouve à chaque fois les informations principales de l'image surtout quand on veut segmenter et séparer deux parties de l'image (exemple : les cellules et le fond)

Seuillage sur norme du gradient:



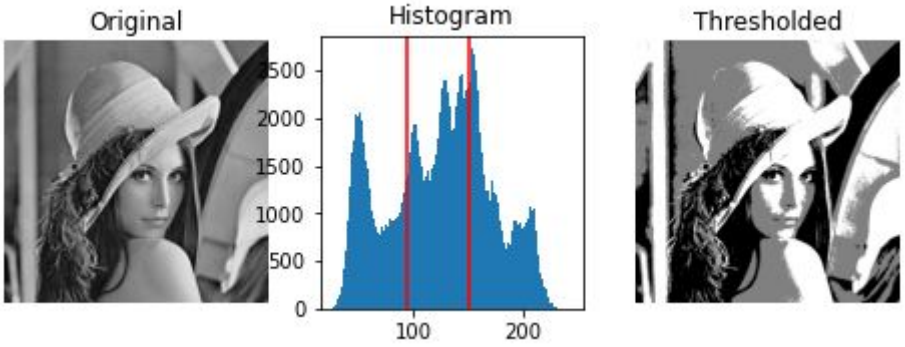
Cela fonctionne bien, cependant il il y a du bruit en plus. L'avantage c'est qu'on n'a pas à trouver manuellement le seuil, l'inconvénient c'est que le seuil n'est pas totalement optimal.

Seuillage à 2 seuils:

La partie qui a changé dans le code est:

```
    for t in range(256):
        for t2 in range(t,256):
            w0=0
            w1=0
            w2=0
            m0=0
            m1=0
            m2=0
            for i in range(t):
                w0=w0+h[i]
                m0=m0+i*h[i]
            if w0 > 0:
                m0=m0/w0

            for i in range(t,t2):
                w2=w2+h[i]
                m2=m2+i*h[i]
            if w2 > 0:
                m2=m2/w2

            for i in range(t2,256):
                w1=w1+h[i]
                m1=m1+i*h[i]
            if w1 > 0:
                m1=m1/w1

            k=w0*w2*(m0-m2)*(m0-m2) + w1*w2*(m1-m2)*(m1-m2) + w0*w1*(m0-m1)*(m0-m1)

            if k > maxk:
                maxk=k
                maxt=[t,t2]


    thresh=maxt

    return(thresh)
```

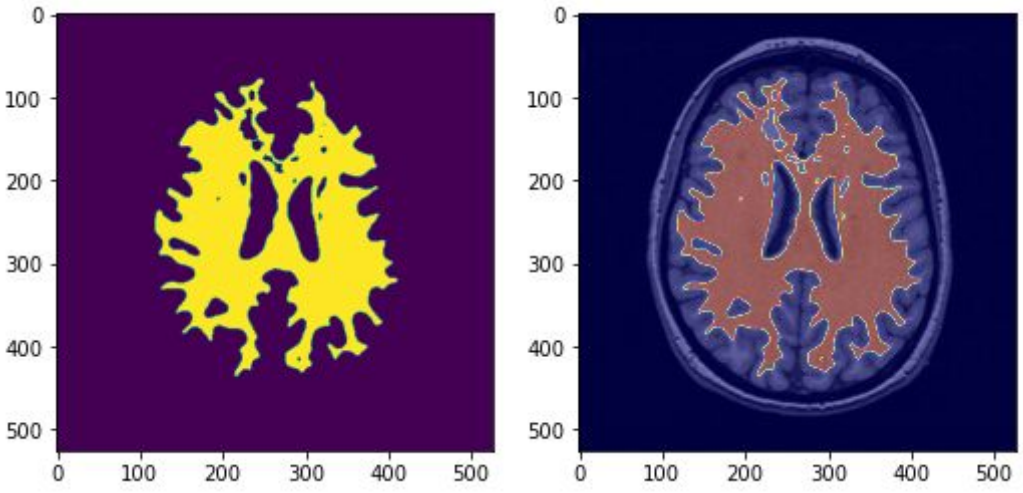
On obtient alors:



5 Croissance de régions

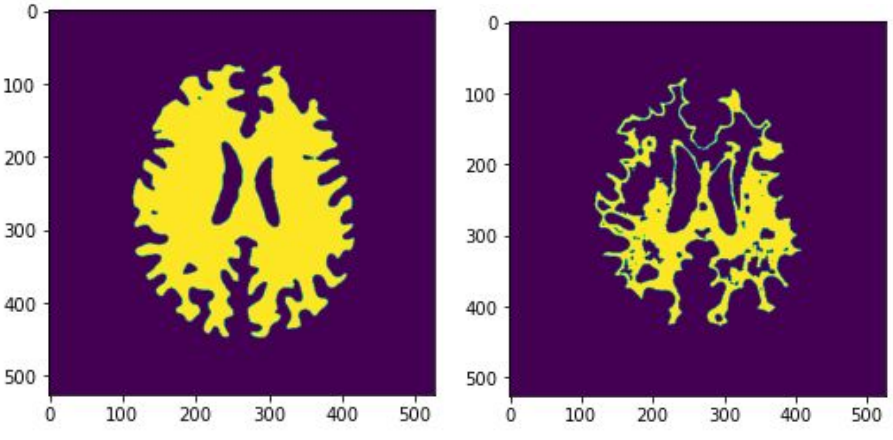
- Pour être ajouté à l'objet, le pixel doit avoir une moyenne au voisinage qui est proche du celle du centre du pixel de l'objet à un certain seuil près

Test:



x0=300 , y0=300 , tresh=2 , rayon= 5

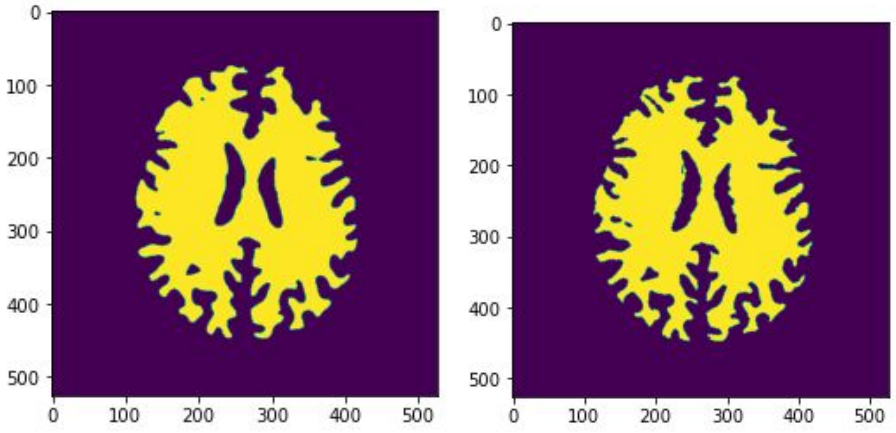
En faisant varier tresh:



tresh=5 *tresh=1*

Plus tresh est grand plus la zone sélectionnée sera étendue

Autres paramètres, le rayon:



rayon=5 *rayon =2*

La taille du rayon détermine la précision de la segmentation. Plus le rayon est petit, plus la dilatation sera précise.