

# Projet de IMA

---

## Waterpixels

---

Yuxuan Xi  
M1 Informatique IMA  
Sorbonne Université



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Prérequis</b>	<b>4</b>
<b>3</b>	<b>Algorithme de waterpixels</b>	<b>5</b>
3.1	Calcul du gradient de l'image . . . . .	5
3.2	Définition de cellules régulières sur les images, centrées sur les sommets d'une grille régulière . . . . .	8
3.3	Sélection d'un marqueur par cellule . . . . .	10
3.4	Régularisation spatiale du gradient à l'aide d'une fonction de distance . . . . .	18
3.5	Application de la transformation du watershed sur le gradient régularisé défini à l'étape 4 à partir des marqueurs définis à l'étape 2 . . . . .	21
<b>4</b>	<b>Critères d'évaluation</b>	<b>25</b>
4.1	Densité de contour . . . . .	25
4.2	Boundary - Recall . . . . .	26
<b>5</b>	<b>Résultat et Discussion</b>	<b>30</b>
5.1	Visualisation de c-waterpixels et m-waterpixels . . . . .	30
5.2	Densité de contours . . . . .	39
5.3	Boundary-Recall . . . . .	40
5.4	Densité de contours et Boundary-Recall . . . . .	41
5.5	Waterpixels et SLIC . . . . .	42
<b>6</b>	<b>Amélioration et perspectives</b>	<b>44</b>
6.1	Sélection du gradient d'image . . . . .	44
6.2	Sélection de la forme de la cellule initiale . . . . .	44
6.3	Sélection du critère d'évaluation . . . . .	44
6.4	Sélection de la méthode de segmentation . . . . .	44
6.5	Tester sur plus d'images . . . . .	44
<b>7</b>	<b>Conclusion</b>	<b>45</b>
<b>8</b>	<b>Référence</b>	<b>46</b>

## 1 Introduction

Mon sujet de recherche est une thèse intitulée Waterpixels. Waterpixels est une technique de segmentation d'image en tant que stratégie générale de génération de superpixels qui repose sur des transformations de watershed contrôlées par des étiquettes.

La méthode de génération de waterpixel se caractérise par les étapes suivantes :

1. Calcul du gradient de l'image
2. Définition de cellules régulières sur l'image, centrées sur les sommets d'une grille régulière
3. Sélection d'un marqueur par cellule
4. Régularisation spatiale du gradient à l'aide d'une fonction de distance
5. Application de la transformation de bassin versant sur le gradient régularisé défini à l'étape 4 à partir des marqueurs définis à l'étape 2.

Appliquer l'algorithme à l'image, et faire des observations visuelles en ajustant la taille de la cellule et la taille du paramètre de régularisation  $k$ .

Les performances de segmentation de l'algorithme sont testées en étudiant la densité de contour et le boundary-recall. Implémenter d'abord les algorithmes de densité de contour et de boundary rappel. Par les graphiques, en changeant la valeur du paramètre de la régularisation  $k$  et la taille de la cellule  $size$ , comparer et observer. De même pour la relation entre densité de contour et boundary rappel.

Au final, convertir la taille de la cellule dans l'image en nombre de superpixels, et les comparer avec l'algorithme SLIC la densité de contour et le boundary-recall.

J'implémente la thèse waterpixels, en utilisant le langage python dans Jupyter Notebook. Tous les codes comprennent la mise en oeuvre de l'algorithme, les tests, les visualisations et les réalisations de figures, dans le fichier waterpixels.ipynb. J'explique également en détail mes idées, les étapes de l'algorithme, les résultats de visualisations et les figures dans le rapport. Le fichier du rapport est waterpixels.pdf.

## 2 Prérequis

Les superpixels sont des régions résultant de la segmentation de bas niveau d'une image, souvent utilisées comme primitives pour une analyse plus approfondie. Il possède quatre propriétés : l'homogénéité, la connectivité et la partition, l'adhérence sur les contours de l'objet et la régularité.

Les superpixels sont plus pratiques pour l'analyse des images, mais la régularité et l'adhérence sur les contours sont quelque peu antithétiques, alors cherchez une solution. En plus des exigences de qualité de superpixel, l'efficacité de calcul est également importante, c'est pourquoi la méthode de complexité linéaire est utilisée. L'auteur a pensé à la transformation du bassin versant, le seul inconvénient est la sur-segmentation, mais pour la génération de superpixels, ce n'est pas un problème, car le nombre de superpixels peut être contrôlé.

Un procédé d'application de la transformée de bassin versant à la génération de superpixels est proposé. Les gradients de régularisation spatiale sont utilisés pour obtenir un compromis réglable entre la régularité des superpixels et l'adhérence sur les contours de l'objet. Cette méthode est évaluée quantitativement sur la base de données fractionnée de Berkeley et s'avère plus performante que la meilleure méthode de pointe en temps linéaire : le clustering itératif linéaire simple (SLIC). Les superpixels résultants sont appelés waterpixels.

### 3 Algorithme de waterpixels

#### 3.1 Calcul du gradient de l'image

Dans cette partie, il faut calculer le gradient de l'image.

Pour ici, je transforme de l'image couleur sur  $[0...255]^3$  en dimension  $N \times M \times 3$  à l'image en gris sur  $[0...255]^2$  en dimension  $N \times M$  en utilisant la bibliothèque **cv2** avec le paramètre **cv2.IMREAD\_GRAYSCALE** pour lire l'image en niveaux de gris.



FIGURE 1 – image couleur



FIGURE 2 – image en gris

Pour calculer le gradient de l'image, le premier étape est de choisir la méthode adapté. J'ai choisi le filtre de Sobel pour l'essayer et l'initialiser, comme le début de réaliser l'algorithme de waterpixels.

Deux noyaux de convolution du filtre de Sobel sont comme suivants :

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Pour faire l'opération de convolution, soient  $h(i, j)$  noyau de convolution avec la taille  $d \times d$  et  $f(i, j)$  image initiale. Le culcul est comme suivante :

$$f'(i, j) = (f * h)(i, j) = \sum_{n=-\frac{d-1}{2}}^{\frac{d-1}{2}} \sum_{m=-\frac{d-1}{2}}^{\frac{d-1}{2}} f(i-n, j-m)h(m, n)$$

On peut observer une image et ses 2 dérivées partielles. On constate que les variations horizontales (respectivement verticales) apparaissent dans la dérivée partielle  $\frac{\partial f}{\partial x}$  (respectivement  $\frac{\partial f}{\partial y}$  ).

Et on peut les définir comme suivantes :

$$G_x = \frac{\partial f}{\partial x} = f * S_x$$

$$G_y = \frac{\partial f}{\partial y} = f * S_y$$

De plus, on calcule le gradient de  $f$  :

$$\vec{\nabla}f = (G_x \quad G_y)^T = \left( \frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \right)^T$$

En même temps, l'amplitude du gradient est :

$$G = \|\vec{\nabla}f\| = \sqrt{(G_x)^2 + (G_y)^2} = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} = \sqrt{(f * S_x)^2 + (f * S_y)^2}$$

En cas général, avant de calculer la convolution du filtre de Sobel, il faut faire le padding de l'image pour traiter les bords de l'image. Pour ici, le noyeu de taille  $3 \times 3$ , donc la longueur et la largeur de l'image sont augmentées de 2 pixels, et les bords sont remplies par le 0 pixel.

L'algorithme **imagePad** est comme suivante :

---

**Algorithme 1 : imagePad(I)**

---

**Entrées :** une image représentée sous forme de matrice I  
**Sorties :** une matrice représentée le padding d'image img  
 $N \times M$  : la taille de l'image  
Initialiser une matrice de taille  $(N + 2) \times (M + 2)$   
**for**  $i$  in range( $N$ ) **do**  
  **for**  $j$  in range( $M$ ) **do**  
    |  $img[i + 1, j + 1] = I[i, j]$   
  **end**  
**end**

---

Ensuite, je fais le calcul du gradient. Et pour faire l'opération de convolution, j'utilise **convolve2d** dans la bibliothèque **scipy.signal**. En utilisant ce méthode, le paramètre **fillvalue** met le padding d'image avec valeur par défaut 0, le paramètre **mode = 'same'** signifie que la taille de la sortie est la même que celle de la matrice d'entrée.

---

**Algorithme 2 : SobelDetector(img)**

---

**Entrées :** une image représentée sous forme de matrice img  
**Sorties :** une matrice représentée le gradient de l'image d'entrée  
 $S_x = [[1, 0, -1], [2, 0, -2], [1, 0, -1]]$   
 $S_y = S_x^T$   
 $G_x = convolve2d(I, S_x, mode = 'same')$   
 $G_y = convolve2d(I, S_y, mode = 'same')$   
# Effectuer des opérations de convolution horizontale et verticale sur l'image  
respectivement, obtenu  $G_x$  et  $G_y$   
gradient =  $\sqrt{G_x^2 + G_y^2}$

---

L'image appliquée le filtre de Sobel est comme suivante :

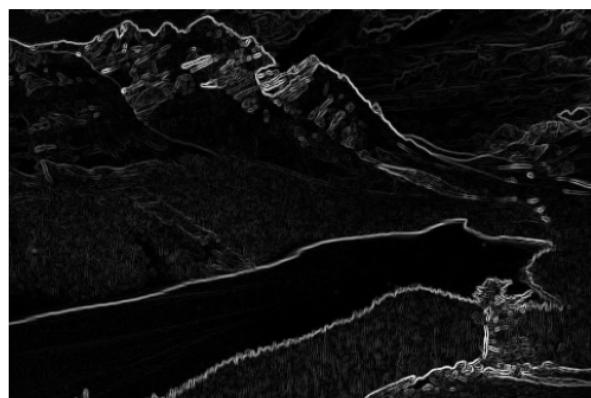


FIGURE 3 – image appliquée le filtre de Sobel

### 3.2 Définition de cellules régulières sur les images, centrées sur les sommets d'une grille régulière

Dans deuxième étape, il faut définir les cellules régulières sur les image, trouver les somment d'une grille régulière, après, bien trouver les pixels dans chaque cellule.

Tout d'abord, j'ai choisi des carrés comme cellules régulières. L'image séparée par carrés est comme suivante :

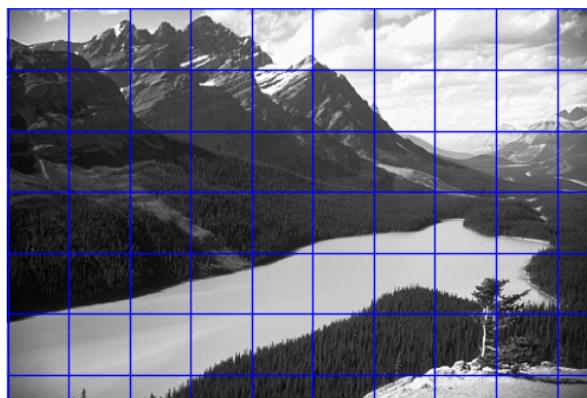


FIGURE 4 – image séparée par les carrés

L'image est en taille de  $N \times M$ ,  $size$  est la longueur du côté de cellules.

On peut calculer comme suivante :

$$N \div size = nb\_lig \cdots res\_lig$$

$$M \div size = nb\_col \cdots res\_col$$

Dans le dernier ligne, la largeur de cellules est  $res\_lig$ , donc il y a  $(nb\_lig + 1)$  de cellules, c'est  $\lceil \frac{N}{size} \rceil$  cellule par ligne.

Il en est de même pour la dernière colonne, la longueur de cellules est  $res\_col$ , donc il y a  $(nb\_col + 1)$  de cellules, c'est  $\lceil \frac{M}{size} \rceil$  cellule par colonne.

Selon ma méthode, lorsque la taille de la cellule n'est pas divisible par la longueur du côté, le plus à droite et le plus en bas ne sont pas des carrés, mais c'est aussi les cellules.

Donc il y a  $(nb\_lig + 1) \times (nb\_col + 1)$  de cellules dans l'image, c'est comme  $\lceil \frac{N}{size} \rceil \times \lceil \frac{M}{size} \rceil$  de cellules.

Ensuite, je fais le parcours, le sommet en haut à gauche est de  $(i \times size, j \times size)$ ,  $i \in \{0, \dots, nb\_lig\}$ ,  $j \in \{0, \dots, nb\_col\}$ , la fonction **rectangle** est comme suivante :

---

**Algorithme 3 : rectangle(img, size)**

---

**Entrées :** une image représentée sous forme de matrice et une taille de cellule  
**Sorties :** la liste rectangle qui contient toutes les coordonnées du sommet en haut à gauche des rectangles

$N \times M$  : la taille de l'image d'entrée

```

rectangle = []
nb_lig = [N / size] # nombre de cellules par ligne
nb_col = [M / size] # nombre de cellules par colonne
for i in range(nb_lig) do
    for j in range(nb_col) do
        Ajouter la coordonnée (i × size, j × size) dans la liste rectangle
        # (i × size, j × size) : le sommet en haut à gauche des rectangles
    end
end

```

---

Ensuite, je voudrais prendre tous les pixels dans les cellules. Donc je crée une fonction **cells** dans la base de la fonction **rectangle**, ci-dessous l'algorithme de **cells** :

---

**Algorithme 4 : cells(img, size)**

---

**Entrées :** une image représentée sous forme de matrice et une taille de cellule  
**Sorties :** Les pixels dans chaque cellule

```

liste_celle = []
sommet = rectangle(img, size) #la liste de sommets de cellules
for i dans sommet do
    pixels = img[i[0] : i[0] + size, i[1] : i[1] + size] # les pixels dans chaque cellule de
    # l'image d'entrée
    Ajouter pixels dans la liste liste_celle
end

```

---

Autre façon, on peut directement prendre tous les pixels dans les cellules sans appliquant la fonction **rectangle**.

---

**Algorithme 5 : cells\_bis(img, size)**

---

**Entrées :** une image représentée sous forme de matrice et une taille de cellule

**Sorties :** la liste de liste qui représente les pixels dans chaque cellule

$N \times M$  : la taille de l'image d'entrée

```

liste_celle = []
nb_lig =  $\lceil \frac{N}{size} \rceil$  # nombre de cellules par ligne
nb_col =  $\lceil \frac{M}{size} \rceil$  # nombre de cellules par colonne
for i in range(nb_lig) do
    for j in range(nb_col) do
        pixels = img[i : (i + 1) × size, j : (j + 1) × size] # les pixels dans chaque cellule
        de l'image d'entrée
        Ajouter pixels dans la liste liste_celle
    end
end

```

---

### 3.3 Sélection d'un marqueur par cellule

Dans le troisième étape, on cherche les marqueurs par cellule, c'est-à-dire que sur chaque cellule, on cherche le gradient minimal par un pixel ou quelques pixels qui sont un composant connex sur le gradient de l'image. Donc au premier, il faut trouver le composant connex. Ensuite, on cherche le gradient minimal pour un pixel ou quelques pixels, c'est alors les marqueurs.

Je crée une fonction **composant\_connex**. Un composant connexe est un ensemble de pixels qui sont connectés les uns aux autres par des pixels adjacents dans les quatre directions horizontale et verticale.

Sur cette fonction, nous parcourons chaque pixel de l'image en utilisant deux boucles for imbriquées pour accéder à chaque pixel une seule fois.

Si le pixel a déjà été visité ou s'il ne correspond pas à la valeur de référence, nous passons au pixel suivant. Sinon, une file d'attente est utilisée pour explorer tous les pixels qui lui sont connectés à l'aide d'un algorithme de recherche en largeur (BFS).

Nous ajoutons tous les pixels visités à un ensemble visité pour nous assurer de ne les visiter qu'une seule fois. À la fin de la traversée BFS, nous ajoutons tous les pixels que nous avons trouvés à un composant, que nous ajoutons à la liste des composants trouvés. Répétez ce processus pour chaque pixel non visité de l'image.

---

**Algorithme 6 : composant\_connex(img, value)**

---

**Entrées :** une image représentée sous forme de matrice (une liste de listes) et une valeur de référence value

**Sorties :** une liste de listes, chaque sous-liste contenant les coordonnées des pixels appartenant à un composant connexe de valeur value.

$N \times M$  : la taille de l'image d'entrée

composants = [ ] # pour stocker les composants connectés trouvés

visited = set( ) # pour enregistrer les coordonnées des pixels qui ont été visités

**for**  $i$  in range( $N$ ) **do**

**for**  $j$  in range( $M$ ) **do**

**si** ( $i, j$ ) est dans visited ou  $image[i,j] \neq value$  **alors**  
| continuer la boucle

**fin**

composant = [ ] # pour stocker les coordonnées de tous les pixels dans le bloc connecté actuel

queue =  $[(i, j)]$  # Définir une liste queue et ajouter les coordonnées de pixel actuelles ( $i, j$ ) à la file d'attente

**tant que** queue n'est pas vide **faire**

$x, y = queue.pop(0)$  # Prendre le premier élément de queue et nommer ses coordonnées x, y

**si** la coordonnée pixel ( $x, y$ ) est dans visited **alors**  
| Sauter la boucle en cours, continuer

**sinon**

visited.add( $(x, y)$ ) # Ajouter les coordonnées de pixel actuelles à visited  
composant.append( $(x, y)$ ) # Ajouter ses coordonnées à la liste

composant

neighbors =  $[(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)]$  # Obtenir les coordonnées des quatre pixels voisins du pixel courant

**for**  $xn, yn$  esr dans neighbors **do**

**si**  $xn \geq 0$  and  $xn < N$  and  $yn \geq 0$  and  $yn < M$  and  
 $image[(xn, yn)] == value$  # les pixels voisins se trouvent dans la plage de l'image et la valeur du pixel est égale à la valeur donnée  
**alors**

queue.append( $(xn, yn)$ ) # Ajouter les coordonnées des pixels voisins à la liste queue

**fin**

**end**

**fin**

$components.append(component)$  # Ajouter composant à composants

**fin**

**end**

**end**

---

Lors de la recherche de composants connectés, en plus de l'algorithme de recherche en largeur (BFS) d'abord utilisé dans le code ci-dessus, il en existe un autre qui peut être utilisé pour gérer la connectivité, en utilisant une structure de données union-find pour implémenter la recherche et la fusion de composants connectés.

L'algorithme analyse chaque pixel et ses voisins, met à jour l'étiquette du pixel (identification du composant) en une seule fois et maintient une table d'équivalence pour enregistrer la relation entre les composants.

Tout d'abord, nous implementons l'opération **find(parent, x)**, qui est utilisée pour trouver le noeud racine de l'élément **x**. **parent** est une liste utilisée pour stocker le noeud parent de chaque élément. La compression du chemin est effectuée pendant le processus de recherche pour améliorer l'efficacité de opérations de recherche ultérieures.

---

#### **Algorithme 7 : find(parent, x)**

---

**Entrées :** la liste parent et l'élément **x**  
**Sorties :** le noeud racine de **x** : parent[x]  
**si**  $\text{parent}[x] \neq x$  **alors**  
  |  $\text{parent}[x] = \text{find}(\text{parent}, \text{parent}[x])$   
**fin**

---

Ensuite, je implémente une fonction **union(parent, rank, x, y)**. Elle est utilisé pour fusionner deux collections, c'est-à-dire pour fusionner les collections où se trouvent les deux éléments.

**parent** est une liste pour stocker le noeud parent de chaque élément. **rank** est une liste utilisée pour enregistrer le rang de chaque ensemble. **x** et **y** sont les deux éléments à combiner.

Dans l'algorithme, nous comparons la taille de  $\text{rank}[\text{root\_x}]$  et  $\text{rank}[\text{root\_y}]$  pour déterminer quel ensemble a un rang supérieur, puis décidons comment effectuer l'opération de fusion,  $\text{rank}[\text{root\_x}]$  et  $\text{rank}[\text{root\_y}]$  sont les noeuds racines de l'élément **x** et **y**.

Chaque élément de la liste de classement correspond à une valeur de classement de l'ensemble. Initialement, chaque élément a une valeur de rang de 0. Au fur et à mesure que l'opération de fusion progresse, le rang de la collection peut augmenter. Lors de la fusion, nous voulons fusionner l'ensemble de plus petit rang dans l'ensemble de plus grand rang pour éviter un déséquilibre de l'arbre.

Si  $\text{rang} = k[\text{root\_x}] < \text{rank}[\text{root\_y}]$ , cela signifie que le rang de l'ensemble **root\_x** est inférieur au rang de l'ensemble **root\_y**. Dans l'opération de fusion, le noeud parent de **root\_x** est défini sur **root\_y** et l'ensemble de rang inférieur est fusionné avec l'ensemble de rang supérieur.

De même pour l'inversement. De plus, quand les deux sont en même, je choix de parenter **root\_y** à **root\_x** et d'augmenter le rang de **root\_x** de 1.

L'algorithme de **union** est comme suivante :

---

**Algorithme 8** : union(parent, rank, x, y)

---

**Entrées** : la liste parent, le rang de chaque ensemble rank, deux éléments x et y

**Sorties** : un ensemble

$root\_x = find(parent, x)$

$root\_y = find(parent, y)$

**si**  $root\_x \neq y$  **alors**

**si**  $rank[root\_x] < rank[root\_y]$  **alors**

        | parent[root\_x] = root\_y

**fin**

**sinon si**  $rank[root\_x] > rank[root\_y]$  **alors**

        | parent[root\_y] = root\_x

**fin**

**sinon**

        | parent[root\_y] = root\_x

        | rank[root\_x] += 1

**fin**

**fin**

---

La fonction **composant\_connex\_u** est similaire à la fonction **composant\_connex** fournie précédemment, mais elle utilise un ensemble de recherche d'union pour juger et marquer la connectivité, l'algorithme est comme suivante :

---

**Algorithm 9 : composant\_connex\_u(img, value)**

---

**Entrées :** une image représentée sous forme de matrice (une liste de listes) et une valeur de référence value

**Sorties :** une liste de listes, chaque sous-liste contenant les coordonnées des pixels appartenant à un composant connexe de valeur value.

$N \times M$  # le size de l'image d'entrée

composants = [ ] # pour stocker les composants connectés trouvés

parent = list(range( $N \times M$ )) # Pour stocker le noeud parent de chaque élément. Lors de l'initialisation, le parent de chaque élément est lui-même.

rank = [0]  $\times$  ( $N \times M$ ) # Initialement, le rang de chaque ensemble est fixé à 0.

**for**  $i$  in range( $N$ ) **do**

**for**  $j$  in range( $M$ ) **do**

**si**  $image[i, j] == value$  **alors**

$neighbors = [(i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1)]$

**for**  $xn, yn$  in  $neighbors$  **do**

**si**  $xn \geq 0$  and  $xn < N$  and  $yn \geq 0$  and  $yn < M$  and

$image[(xn, yn)] == value$  **alors**

$union(parent, rank, i \times M + j, xn \times M + yn)$

# Appelez la fonction d'union pour combiner le pixel actuel et les pixels adjacents dans le même ensemble connecté

**fin**

**end**

**fin**

**end**

**end**

$label\_dict = \{ \}$  # Pour collecter les mêmes coordonnées de pixel que l'étiquette de pixel.

**for**  $i$  in range( $N$ ) **do**

**for**  $j$  in range( $M$ ) **do**

**si**  $image[(i, j)] == value$  **alors**

$root = find(parent, i \times M + j)$

# Appelez la fonction de find pour trouver le noeud racine de l'étiquette de pixel actuelle et stockez le noeud racine dans le dictionnaire  $label\_dict$ , et la valeur correspondante est une liste

**si**  $root$  in  $label\_dict$  **alors**

$label\_dict[root].append((i, j))$

**fin**

**sinon**

$label\_dict[root] = [(i, j)]$

**fin**

**fin**

**end**

**end**

$components = list(label\_dict.values())$

---

Ensuite, je crée une fonction **min\_grad** qui cherche le plus grand composant connex du gradient minimal sur chaque cellule, c'est-à-dire que je prends le gradient de l'image, après je cherche le pixel minimal sur l'image du gradient, à la fin, je cherche le plus grand composant connex du pixel minimal de l'image de gradient.

D'abord, je cherche le plus petit pixel dans la matrice (la liste de liste) d'entrée qui représente les gradients de la cellule. Pour ici, je cherche le plus petit pixel par tous les colonnes, ensuite, je cherche le plus petit pixel sur la résultat d'avant, c'est alors le plus petit pixel dans la matrice d'entrée.

En utilisant la fonction **composant\_connex** pour trouver les composants connexes avec la valeur qui est le plus petit pixel dans la cellule.

Sur les composants connexes dans chaque cellule, on prend le plus grand composant connex.

L'algorithme **min\_grad** est comme suivante :

---

#### **Algorithme 10 : min\_grad(liste)**

---

**Entrées :** la matrice qui représente les gradient d'une cellule

**Sorties :** la liste de coordonnées du plus grand composant connex et le gradient minimal

```
liste_col = []
for col in liste do
    Ajouter le minimum de col dans liste_col
    # Par colonne, on cherche le pixel minimal
end
pixel_min = min(liste_col) # le pixel minimal de cette cellule (une matrice d'entrée)
com = composant_connex(liste, pixel_min) # trouver le composant connex avec la
    valeur (pixel_min) qui est le gradient minimal
marker = max(com, key=len) # permet de trier les composants connexes par longueur,
    obtenu le plus grand composant connex avec le pixel minimal.
```

---

Donc le plus grand composant connex du gradient minimal dans chaque cellule est un marqueur. La fonction **min\_grad** retourne le plus grand composant connex du gradient minimal sur une cellule.

Je voudrais planter une fonction, saisir l'image et la taille de la cellule, retourner l'image avec les marqueurs et les pixels sur marqueurs.

Donc il faut réoccuper les marqueurs sur tous les cellules. Je crée une fonction **marqueur** pour les réaliser, l'algorithme est comme suivant :

---

**Algorithme 11 :** marqueur(image,size)

**Entrées :** une image représentée sous forme de matrice img et une taille de cellule  
**Sorties :** une image modifiée avec des marqueurs aux emplacements des pixels ayant les gradients les plus faibles dans des cellules de taille.

$N \times M$  : la taille de l'image d'entrée

$img[:, :] = \infty$  # De même taille que l'image d'entrée

$img\_grad = SobelDetector(image)$  # Calculer le gradient de l'image d'entrée

$liste\_cell = cells(img\_grad, size)$  # Par la fonction cells, prendre toutes les listes de cellules.

list\_min\_position = [ ]

list\_min\_pixel = [ ]

**for**  $i$  dans liste\_cell **do**

    list\_min\_position.append(min\_grad(i)[0]) # Une liste représente toutes les positions de ce gradient minimal.

    list\_min\_pixel.append(min\_grad(i)[1]) # Une liste représente les gradients minimaux correspondants.

**end**

$nb\_ligne = \lceil \frac{N}{size} \rceil$ ,  $nb\_col = \lceil \frac{M}{size} \rceil$

$n = 0$  # indice de la liste de cellule

**for** ligne dans nb\_ligne **do**

**for** col dans nb\_col **do**

**for**  $(i, j)$  dans list\_min\_position[n] **do**

$img[i + ligne \times size, j + col \times size] = list\_min\_pixel[n]$

            # le marqueur en pixels de gradient minimal correspondant, et les autres en  $\infty$

**end**

$n = n + 1$

**end**

**end**

---

L'image avec les marqueurs est comme suivante, les marqueurs sont en pixels correspondant (le gradient minimal de chaque cellule) et les autres sont en  $\infty$ . Pour ici, je prends la taille de la cellule  $size = 50$ .

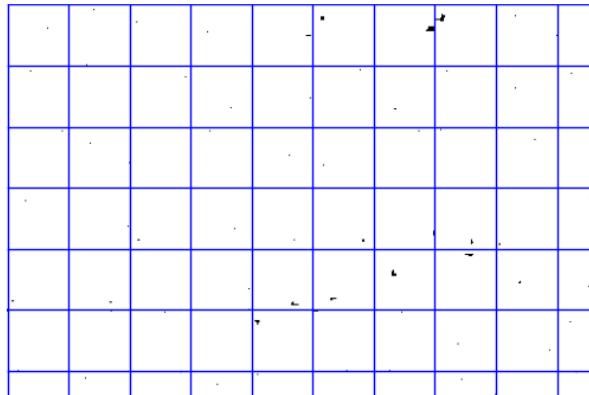


FIGURE 5 – image avec les marqueurs

Si le marqueur est affiché sur l'image couleur d'origine, ce sera comme indiqué dans la figure suivant, les marqueurs sont en rouge.

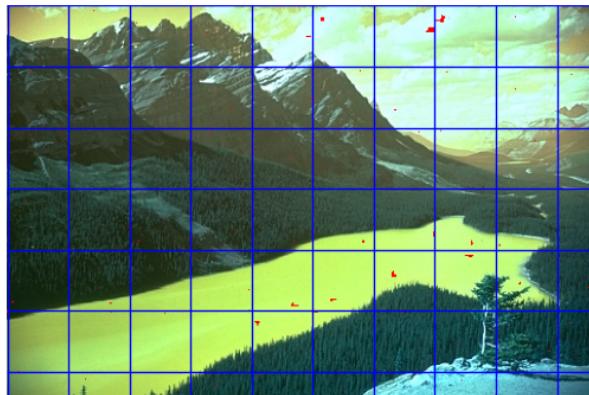


FIGURE 6 – image couleur d'origine avec les marqueurs

### 3.4 Régularisation spatiale du gradient à l'aide d'une fonction de distance

Dans quatrième partie, on définit d'abord la fonction de distance et fait la normalisation par  $\sigma$ , rendant la régularisation indépendante de la taille de la cellule choisie.  $\sigma$  est la taille de la cellule *size*. Ensuite, on effectue une régularisation spatiale sur le gradient.

Pour calculer la distance de la fonction  $d_Q$ , les méthodes sont comme suivantes :

Soient  $Q = \{q_i\}_{1 \leq i \leq N}$  un ensemble de  $N$  composantes connexes de l'image  $D$ ,  $p$  le pixel de l'image  $D$  et  $q_i$ ,  $\sigma$  la taille de cellule.

On étudie deux choix possibles du  $q_i$ . La première est de les choisir égales aux marqueurs :  $q_i = M_i$ . Les waterpixels résultants sont appelés **m-waterpixels**. La seconde consiste à les placer au centre des cellules :  $q_i = o_i$ , ce qui conduit à des **c-waterpixels**.

Donc la distance de la fonction est comme suivante :

$$d_Q(p) = \frac{2}{\sigma} \min_{i \in [1, N]} d(p, q_i), \forall p \in D$$

Après de calculer la distance de la fonction, la deuxième étape est de faire la régularisation spatiale du gradient comme suivante :

$$g\_reg = g + k \times d_Q$$

avec  $g$  le gradient de l'image,  $k$  le paramètres de la régularisation spatiale et  $d_Q$  la distance de la fonction.

Alors, je crée deux fonctions de la distance : **distance\_center** choisit le centre de cellule comme le marqueur, mais sur **distance\_marker**, le marqueur est qu'on a bien réalisé auparavant. Et deux fonction pour implémenter la régularisation spatiale : **sp\_regularizer\_centre** et **sp\_regularizer\_marker**.

#### c-waterpixels :

Pour calculer de la distance entre deux pixels, on peut appliquer la distance euclidienne :

$$d(A, B) = \sqrt{(X_B - X_A)^2 + (Y_B - Y_A)^2}$$

$(X_A, Y_A)$  est le pixel que je parcours,  $(X_B, Y_B)$  est le centre de la cellule.

Donc je peux calculer la distance  $d_Q$  par la fonction **distance\_center** comme suivante :

---

**Algorithme 12 :**  distance\_center(img,size)

---

**Entrées :** une image représentée sous forme de matrice img et une taille de cellule

**Sorties :** une matrice de même taille que l'image où chaque élément correspond à cette distance normalisée par la taille de la cellule.

$N \times M$  : la taille de l'image d'entrée

$grad = SobelDetector(img)$  # Calculer les gradient de l'image

$cell = np.zeros((size, size))$  # Initialisation du même size que la cellules

center = [size/2,size/2] # la centre pour chaque cellule

**for** i de 0 à size-1 **do**

**for** j de 0 à size-1 **do**

$cell[i, j] = \sqrt{(center[0] - i)^2 + (center[1] - j)^2}$  # Calculer la distance euclidienne de chaque pixel de cellule au point central

**end**

**end**

$B = np.matlib.repmat(cell, int(np.ceil(N/size)), int(np.ceil(M/size)))$  # Répéter la matrice cell dans chaque direction suffisamment de fois pour couvrir toute l'image.

Prendre  $B[0 : N, 0 : M]$ . # Découper cette matrice pour qu'elle ait la même taille que l'image en récupérant les N premières lignes et les M premières colonnes.

Multiplié par  $\frac{2}{\sigma}$

---

Dans ce cas, on calcule la régularisation spatiale du gradient. La fonction **sp\_regularizer\_centre** est comme suivante :

---

**Algorithme 13 :**  sp\_regularizer\_centre(image,size,k)

---

**Entrées :** une image représentée sous forme de matrice, une taille de cellule et un paramètre de régularisation k.

**Sorties :** la matrice régularisée de même taille que l'image d'entrée  
le gradient de l'image + la distance de la fonction  $\times k$

---

### m-waterpixels :

Au départ, j'ai trouvé la distance minimale en parcourant la distance entre chaque pixel de l'image et tous les marqueurs, mais c'est beaucoup de travail, très lent à réaliser.

J'ai donc utilisé le programme dynamique pour les calculer. Cette méthode s'appelle l'algorithme de transformation de distance.

Tout d'abord, on fait l'initialisation, si c'est dans le marqueur, c'est 0, sinon, c'est  $\infty$ . Après, pour appliquer un masque de la taille  $3 \times 3$ , on fait le padding de l'image en 0. À la fin, on crée une fonction **distance\_marker** pour calculer la distance  $d_Q$ .

Pour ici, je met le masque est  $\begin{bmatrix} 4 & 3 & 4 \\ 3 & 0 & 3 \\ 4 & 3 & 4 \end{bmatrix}$ .

La fonction de **distance\_marker** est comme suivante :

---

**Algorithm 14 : distance\_marker(image,size)**


---

**Entrées** : une image représentée sous forme de matrice, une taille de cellule.

**Sorties** : une matrice de même taille que l'image où chaque élément correspond à cette distance normalisée par la taille de la cellule

$m = [[4,3,4],[3,0,3],[4,3,4]]$  : le masque

$N \times M$  : la taille de l'image d'entrée

$d0 = initial(image) \#$  en appliquant la fonction initial

$d1 = imagePad(image) \#$  en appliquant la fonction imagePad

# Faire le parcours dans l'ordre

**for**  $i$  in  $range(N)$  **do**

**for**  $j$  in  $range(M)$  **do**

$d1[i, j] = min(d0[i, j], d1[i - 1, j] + 3, d1[i - 1, j - 1] + 4, d1[i, j - 1] + 3, d1[i, j + 1] + 4, d1[i - 1, j + 1] + 3, d1[i + 1, j - 1] + 3)$

**end**

**end**

$d2 = d1.copy() \#$  Dans la base de l'image  $d1$ , on fait le parcours dans l'ordre inverse

**for**  $i$  in  $range(N, 0, -1)$  **do**

**for**  $j$  in  $range(M, 0, -1)$  **do**

$d2[i, j] = min(d1[i, j], d2[i + 1, j] + 3, d2[i + 1, j + 1] + 4, d2[i, j + 1] + 3, d2[i, j - 1] + 4, d2[i + 1, j - 1] + 3, d2[i - 1, j + 1] + 3)$

**end**

**end**

$d_{min} = \frac{d2}{3}$

Retourner  $d_{min}[1 : N + 1, 1 : M + 1] \times \frac{2}{\sigma}$

---

Dans la fonction **distance\_marker**, j'ai appliqué la fonction **initial** pour faire l'initialisation, les pixels de marqueurs est en 0, les autres sont en  $\infty$ . C'est similaire que la fonction **marqueurs**, seulement de changer les pixels en marqueurs. L'algorithme est comme suivante :

---

**Algorithme 15 : initial(image,size)**

---

**Entrées :** une image représentée sous forme de matrice img et une taille de cellule

**Sorties :** une image modifiée avec des marqueurs des pixels 0, les autres sont en  $\infty$

$N \times M$  : le size de l'image d'entrée

$img\_grad = SobelDetector(image) \#$  Calculer le gradient de l'image d'entrée

$liste\_cell = cells(img\_grad, size) \#$  Par la fonction cells, prendre toutes les listes de cellules.

```

list_min_position = []
img[:, :] = infinity # De même taille que l'image d'entrée
for i dans liste_cell do
    list_min_position.append(min_grad(i)[0]) # Une liste représente toutes les
    positions de ce gradient minimal.
end
nb_ligne = [N / size], nb_col = [M / size]
n = 0 # indice de la liste de cellule
for ligne dans nb_ligne do
    for col dans nb_col do
        for (i, j) dans list_min_position[n] do
            img[i + ligne * size, j + col * size] = 0 # le marqueur en pixels 0, les autres
            en infinity
        end
        n = n + 1
    end
end

```

---

Ensuite, on fait la régularisation spatiale du gradient. C'est la fonction **sp\_regularizer\_marker** pour calculer.

---

**Algorithme 16 : sp\_regularizer\_marker(image,size,k)**

---

**Entrées :** une image représentée sous forme de matrice, une taille de cellule et un paramètre de régularisation k.

**Sorties :** la matrice régularisée de même taille que l'image d'entrée  
le gradient de l'image + la distance de la fonction  $\times k$

---

### 3.5 Application de la transformation du watershed sur le gradient régularisé défini à l'étape 4 à partir des marqueurs définis à l'étape 2

Dans la dernière partie, on applique de la transformation de watershed sur le gradient régularisé défini à l'étape 4 à partir des marqueurs définis à l'étape 2. Et on obtiens les c-waterpixels et m-waterpixels.

Les fonction de la réalisation **application\_center** et **application\_marker** sont comme suivantes :

---

**Algorithme 17 : application\_center(image,size,k)**


---

**Entrées** : une image représentée sous forme de matrice, une taille de cellule et un paramètre de régularisation spatiale k.

**Sorties** : la matrice des labels

*gradient = sp\_regularizer\_center(img, size, k)*

*markers = label(img, size)*

*labels = watershed(gradient, markers)*

# Appliquer la fonction watershed avec le gradient régularisé et les marqueurs.

---



---

**Algorithme 18 : application\_marker(image,size,k)**


---

**Entrées** : une image représentée sous forme de matrice, une taille de cellule et un paramètre de régularisation spatiale k.

**Sorties** : la matrice des labels

*gradient = sp\_regularizer\_marker(img, size, k)*

*markers = label(img, size)*

*labels = watershed(gradient, markers)*

# Appliquer la fonction watershed avec le gradient régularisé et les marqueurs.

---

Pour appliquer le méthode de **watershed** dans la bibliothèque **skimage.segmentation**, les pixels de marqueurs ne sont pas de pixel correspondant, il faut être dans l'ordre, le premier marqueur est de 1, incrémenté d'un. Donc il faut modifier un peu sur la fontion de réaliser les marqueurs. Les marqueurs sur la matrice de marqueurs doivent être dans l'ordre.

Donc j'ai créé une fonction **label** en modifiant un peu dans la base de la fonction **marqueur**. Pour la fonction **label**, on n'a pas besoin de pixels minimals de chaque cellule, à leur place se trouvent des indeces de marqueurs par cellules. Je obtiens une matrice avec les pixels de marqueurs dans l'ordre, et les autres en  $\infty$ .

La fonction **label** est comme suivante :

---

**Algorithme 19 :** label(image,size)

**Entrées :** une image représentée sous forme de matrice img et une taille de cellule  
**Sorties :** une image modifiée avec des marqueurs aux emplacements des pixels ayant les gradients les plus faibles dans des cellules de taille.

$N \times M$  : le size de l'image d'entrée

img\_grad = SobelDetector(image) # Calculer le gradient de l'image d'entrée

liste\_cell = cells(img\_grad,size) # Par la fonction cells, prendre toutes les listes de cellules.

list\_min\_position = []

**for** i dans liste\_cell **do**

| list\_min\_position.append(min\_grad(i)[0]) # Une liste représente toutes les positions de ce gradient minimal.

**end**

nb\_ligne =  $\lceil \frac{N}{size} \rceil$ , nb\_col =  $\lceil \frac{M}{size} \rceil$

img[:, :] =  $\infty$  # la même taille que l'image d'entrée

n = 0 # indice de la liste de cellule

nb = 0 # indice pour les marqueurs

**for** ligne dans nb\_ligne **do**

**for** col dans nb\_col **do**

**for** (i, j) dans list\_min\_position[n] **do**

| img[i + ligne \* size, j + col \* size] = nb

# le marqueur en pixels de gradient minimal correspondant, et les autres en

$\infty$

**end**

n = n + 1

nb = nb + 1

**end**

**end**

---

Le résultat de la fonction **application\_center** ou **application\_marker** est la matrice, pour chaque cellule (waterpixel), le pixel est l'indice de label. L'image de la matrice de label est comme suivante :

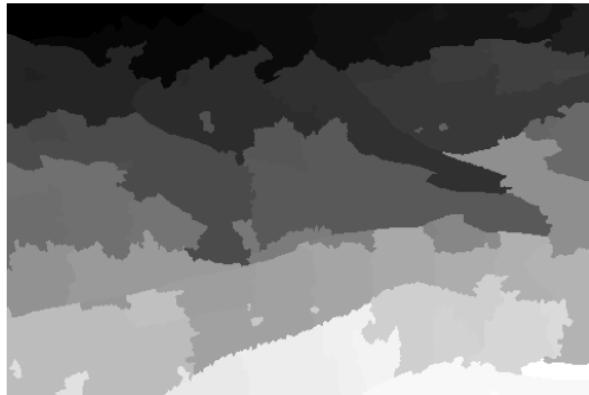


FIGURE 7 – image de label

Ensuite, j'applique la méthode **mark\_boundaries** dans la bibliothèque **skimage.segmentation** pour dessiner les contours de waterpixels. Le code est `mark_boundaries(image_color, labels)`, `image_color` est l'image couleur en initiale, `labels` est le résultat de la fonction **application\_center** et **application\_marker**.

De plus, l'image couleur en initiale avec les contours est comme suivants :



FIGURE 8 – image final

Pour ici, j'ai choisi size à 50, k à 20, et la distance entre les pixels et le centre de la cellule. Ceci n'est qu'un exemple, plus d'analyse suivra.

## 4 Critères d'évaluation

La densité de contour et le boundary-recall sont deux concepts et indicateurs différents dans le domaine de la segmentation d'image, qui décrivent différents aspects des performances de segmentation.

La densité de contour est une mesure utilisée pour décrire la densité sur le nombre de contours dans une image. Des valeurs de la densité de contour plus élevées indiquent des bords plus denses dans l'image, tandis que des valeurs plus basses indiquent des bords moins nombreux ou plus clairsemés.

D'autre part, le boundary-recall est une métrique utilisée pour évaluer les performances des algorithmes de segmentation d'image, qui mesure la capacité de l'algorithme à détecter correctement la frontière réelle. Il est calculé en fonction de la correspondance entre la contour de Ground-truth et le résultat de la segmentation.

### 4.1 Densité de contour

Soient  $S_c$  l'ensemble des pixels de contours,  $S_b$  l'ensemble des pixels de bordures et  $D$  l'ensemble de tous les pixels de l'image.

La densité de contour est :

$$CD = \frac{\frac{1}{2} \times |S_c| + |S_b|}{|D|}$$

$|\cdot|$  signifie le nombre d'éléments dans l'ensemble.

Soit  $N \times M$  la taille de l'image, on peut calculer le nombre des pixels de bordures :

$$|S_b| = 2 \times (M + N) - 4$$

Pour calculer le nombre des pixels sur les contours, j'applique la méthode **find\_boundaries** de **skimage.segmentation**.

Si c'est sur les contours, c'est True, sinon, c'est False. Je calcule le nombre de True. Mais les contours ont une largeur de deux pixels, il faut diviser 2. Donc c'est  $\frac{1}{2} \times |S_c|$ .

Je résume dans la fonction **contour\_densite\_center** et la fonction **contour\_densite\_marker** pour calculer la densité de contour. Les algorithmes sont comme suivantes :

---

**Algorithme 20 :** contour\_densite\_center(image, size, k)

---

**Entrées :** une image représentée sous forme de matrice image, une taille de cellule, le paramètre de la régularisation spatiale k

**Sorties :** la valeur de la densité de contour

$gradient = sp\_regularize\_center(image, size, k)$

$markers = label(image, size)$

$labels = watershed(gradient, markers)$

$boundaries = find\_boundaries(labels)$

Calculer la nombre de  $S_c$  : Calculer la somme de nombre de True dans la matrice de boundaries

Calculer la nombre de  $S_b$  :  $|S_b| = 2 \times (M + N) - 4$

Calculer la densité de contour CD :  $CD = \frac{\frac{1}{2} \times |S_c| + |S_b|}{|D|}$ .

---



---

**Algorithme 21 :** contour\_densite\_marker(image, size, k)

---

**Entrées :** une image représentée sous forme de matrice image, une taille de cellule, le paramètre de la régularisation spatiale k

**Sorties :** la valeur de la densité de contour

$gradient = sp\_regularize\_marker(image, size, k)$

$markers = label(image, size)$

$labels = watershed(gradient, markers)$

$boundaries = find\_boundaries(labels)$

Calculer la nombre de  $S_c$  : Calculer la somme de nombre de True dans la matrice de boundaries

Calculer la nombre de  $S_b$  :  $|S_b| = 2 \times (M + N) - 4$

Calculer la densité de contour CD :  $CD = \frac{\frac{1}{2} \times |S_c| + |S_b|}{|D|}$ .

---

## 4.2 Boundary - Recall

Afin de calculer Boundary-Recall, nous comparons ici les contours de l'images de waterpixels et les contours de l'image Ground-Truth.

L'image Ground-Truth correspondant aux contours des objets à segmenter, fournie dans la base de données de segmentation de Berkeley, est appelée GT. Parmi eux, l'image Ground-Truth représente les informations réelles sur les limites de l'objet, qui peuvent être utilisées comme référence pour le résultat de prédiction de l'algorithme.

Soient  $GT$  l'ensemble de contours de Ground-Truth,  $C$  l'ensemble de contours de waterpixels.

La méthode de calculer Boundary - Recall est comme suivante :

$$BR = \frac{|p \in GT, d(p, C) < 3|}{|GT|}$$

Tout d'abord, j'ai tiré toutes les coordonnées de contours de ground-truth et toutes les coordonnées de contours de waterpixels en deux liste. Et j'ai fait le parcours entre deux et deux pour trouver la distance inférieur que 3. Mais c'est trop long et moins efficace à réaliser. Donc j'utilise le programme dynamique.

Au premier, je fais l'inisialisation. Sur waterpixels, les pixels en contours sont en 0, les autres sont en  $\infty$ .

La distance minimale entre tous les pixels et les contours de waterpixels en appliquant la fonction **distance\_C**. J'ai créé la fonction **distance\_C** de même que la fonction **distance\_marker** sauf le dernier étape : multiplié par  $\frac{2}{size}$ , size est la taille de cellule.

J'ai une liste de toutes les coordonnées de contours de ground-truth, je fais le parcours, cherche la distance correspondante à 0, 1 et 2.

La fonction **distance\_C** est comme suivante :

---

**Algorithme 22 : distance\_C(image,size)**


---

**Entrées** : une image représentée sous forme de matrice, une taille de cellule.

**Sorties** : une matrice de même taille que l'image où chaque élément correspond à cette distance normalisée par la taille de la cellule

$m = [[4,3,4],[3,0,3],[4,3,4]]$  : le masque

$N \times M$  : la taille de l'image d'entrée

$d0 = initial(image)$  # en appliquant la fonction initial

$d1 = imagePad(image)$  # en appliquant la fonction imagePad

# Faire le parcours dans l'ordre

**for**  $i$  in range( $N$ ) **do**

**for**  $j$  in range( $M$ ) **do**

$d1[i, j] = \min(d0[i, j], d1[i - 1, j] + 3, d1[i - 1, j - 1] + 4, d1[i, j - 1] + 3, d1[i, j + 1] + 4, d1[i - 1, j + 1] + 3, d1[i + 1, j - 1] + 3)$

**end**

**end**

$d2 = d1.copy()$  # Dans la base de l'image d1, on fait le parcours dans l'ordre inverse

**for**  $i$  in range( $N, 0, -1$ ) **do**

**for**  $j$  in range( $M, 0, -1$ ) **do**

$d2[i, j] = \min(d1[i, j], d2[i + 1, j] + 3, d2[i + 1, j + 1] + 4, d2[i, j + 1] + 3, d2[i, j - 1] + 4, d2[i + 1, j - 1] + 3, d2[i - 1, j + 1] + 3)$

**end**

**end**

$d_{min} = \frac{d2}{3}$

Retourner  $d_{min}[1 : N + 1, 1 : M + 1]$

---

Pour prendre la liste de coordonnées sur les contours de GT, je parcours l'image dans la database, et si le pixel n'est pas en 0, je mets la coordonnées dans liste\_GT.

Dans la suite de l'algorithme **distance\_C**, on implémente la fonction **d\_GT\_C**, qui est comme suivante :

---

**Algorithme 23 : d\_GT\_C(boundaries, liste\_GT)**


---

**Entrées** : une image où les contours sont marqués en True et le reste de l'image est en False, et une liste de coordonnées (i,j) qui représentent les positions de contours de GT

**Sorties** : le nombre de pixels de chaque région d'intérêt qui se trouvent à différentes distances des contours

```

for (i,j) dans la liste_GT do
    # nombre de la distance minimale 0
    si ss[i,j]==0 alors
        | d0 = d0 + 1
    fin
    # nombre de la distance minimale 1
    sinon si ss[i,j]==2 alors
        | d1 = d1 + 1
    fin
    # nombre de la distance minimale 2
    sinon si ss[i,j]==2 alors
        | d2 = d2 + 1
    fin
    ...
end
```

---

Donc la fonction **BR** pour calculer Boundary-Recall est comme suivante :

---

**Algorithme 24 : BR(boundaries, liste\_GT) :**


---

**Entrées** : une image où les contours sont marqués en True et le reste de l'image est en False, et une liste de coordonnées (i,j) qui représentent les positions de contours de GT

**Sorties** : le boundary\_Recall

la somme de la nombre de distance 0, 1 et 2 dérive la nomble de liste\_GT

---

On obtient l'image de la distance entre GT et waterpixels de la manière suivante, pour ici, je prends la taille de cellule  $size = 50$ , le paramètres de la régularisation spatiale  $k = 20$  :

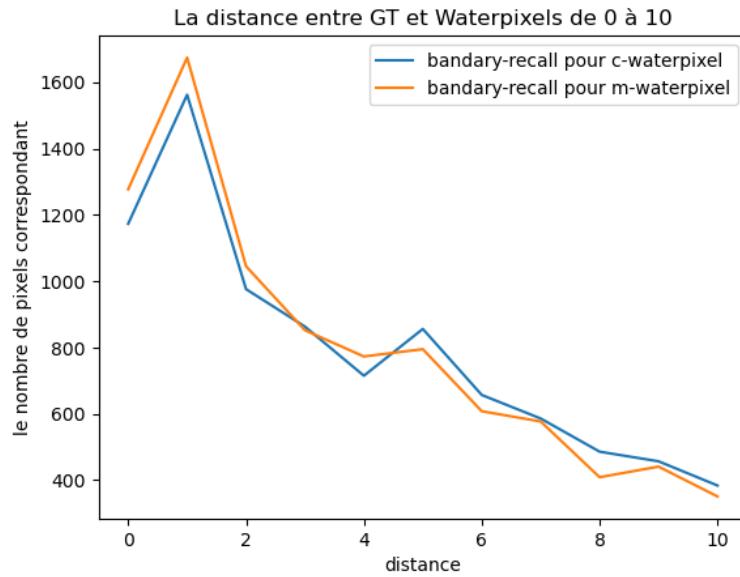


FIGURE 9 – distance entre GT et waterpixels

Les pixels avec une distance de 1 sont les plus nombreux, et à mesure que la distance augmente, l'image montre une tendance à la baisse.

Pour calculer le Boundary-Recall, on prend les trois premières distances, quand la distance est 0, 1 et 2.

## 5 Résultat et Discussion

### 5.1 Visualisation de c-waterpixels et m-waterpixels

**Voici huit images à comparer visuellement en modifiant la valeur du paramètre de la régularisation spatiale  $k$  et la taille de cellule  $size$  pour c-waterpixel et m-waterpixel.**

**1 :** **Figure-9** est la segmentation par c-waterpixels et **Figure-10** est la segmentation par m-waterpixels. Les deux figures sont sur le size de cellules fixé à 50 et le paramètre de régularisation spatiale  $k$  changé à 0, 10, 20, 50, 100, 150, 200, 250, 500, 1000.

Lorsque le paramètre de régularisation spatiale  $k$  vaut 0, la segmentation de waterpixels est pour le gradient de l'image, aucune régularisation du gradient n'est appliquée. Cela n'a rien à voir avec le choix de  $q_i$  en distance de la fonction. Ainsi, lorsque  $k = 0$ , les segmentation pour les c-waterpixels et les m-waterpixels sont les mêmes.

Au fur et à mesure que le paramètre de régularisation spatiale  $k$  augmente, on peut observer que la cellule segmentée se rapproche progressivement de la tessellation de Voronoi de l'ensemble  $q_i$  dans le domaine spatial. Donc pour le c-materpixel, la cellule ressemble plus à un carré et le m-materpixel est plus proche à un arc fermé. De plus, les m-waterpixels segmentés plus en ligne avec les limites de l'objet, et les c-waterpixels sont des superpixels plus réguliers.

En observant les changements du paramètre de la régularisation spatiale  $k$  pour segmenter les superpixels, j'ai constaté que lorsque  $k$  est compris entre 0 et 50, l'effet est le meilleur, nous l'affirons donc et continuons à observer.

**2 :** **Figure-11** est la segmentation par c-waterpixels et **Figure-12** est la segmentation par m-waterpixels. Les deux figures sont sur le size de cellules fixé à 50 et le paramètre de régularisation spatiale  $k$  changé à 5, 10, 15, 20, 25, 30, 35, 40, 45, 50.

**3 :** **Figure-13** est la segmentation par c-waterpixels et **Figure-14** est la segmentation par m-waterpixels. Les deux figures sont sur le paramètre de régularisation spatiale  $k$  fixé à 0 et le size de cellules changé à 10, 20, 30, 40, 50, 60, 70, 80, 90, 100.

**Figure-13** et **Figure-14** sont les même. Parce que quand  $k = 0$ , cela n'a rien à voir avec le choix de  $q_i$  en distance de la fonction. Dans ce cas, la segmentation est uniquement liée au size de la cellule, affectant la formation des marqueurs.

**4 :** **Figure-15** est la segmentation par c-waterpixels et **Figure-16** est la segmentation par m-waterpixels. Les deux figures sont sur le paramètre de régularisation  $k$  fixé à 20 et le size de cellules changé à 10, 20, 30, 40, 50, 60, 70, 80, 90, 100. J'ai choisi  $k = 20$ , parce que lorsque  $k$  est compris entre 0 et 50, l'effet est le meilleur.

Lorsque la taille de la cellule est très petite, en sur-segmentant. A mesure que la taille de la cellule augmente progressivement, les frontières les plus évidentes dans l'image de la vérité au sol sont distinguées, telles que : entre plusieurs sommets montagneux, rivières et montagnes, mais pour la segmentation des détails est insuffisante.

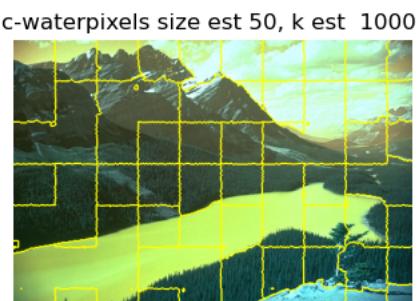
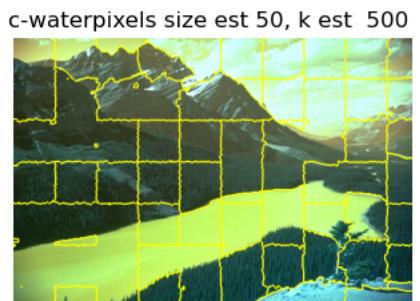
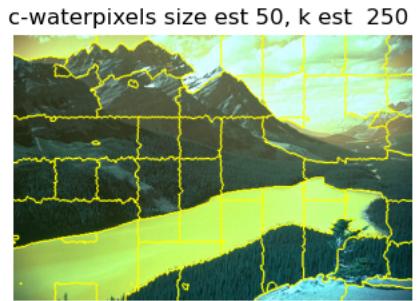
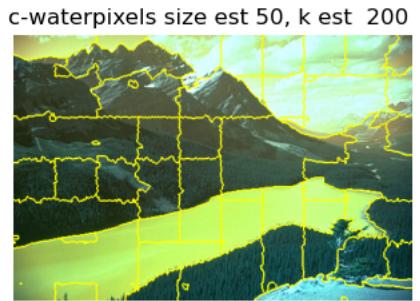
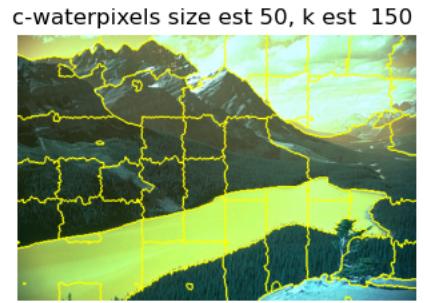
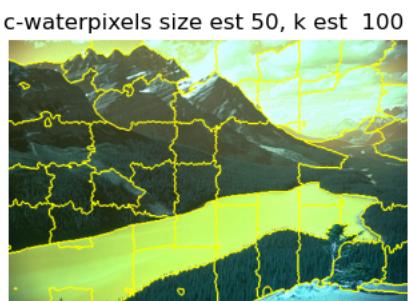
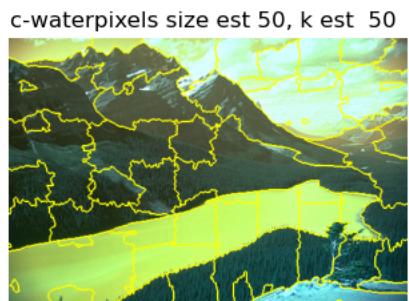
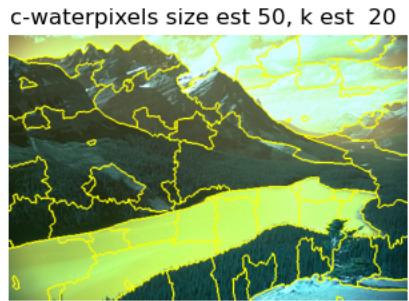
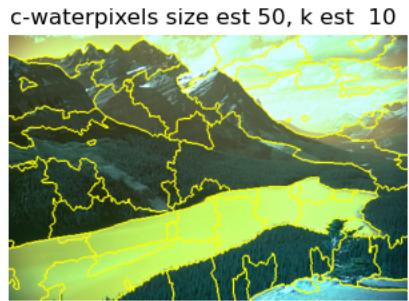
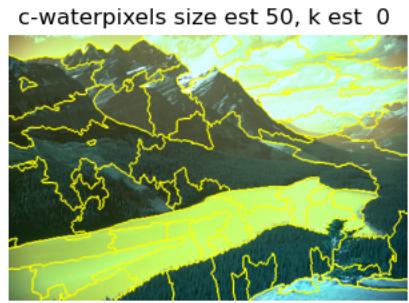


FIGURE 10 – c-waterpxiels : size = 50, k en [0,10,20,50,100,150,200,250,500,1000]

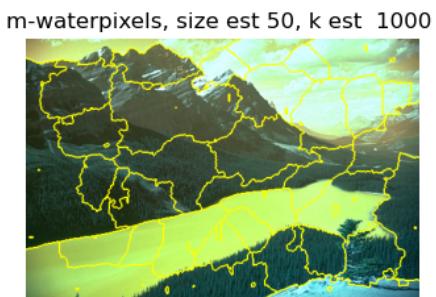
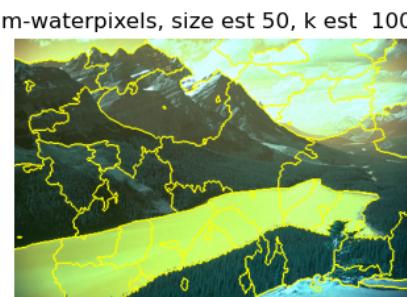
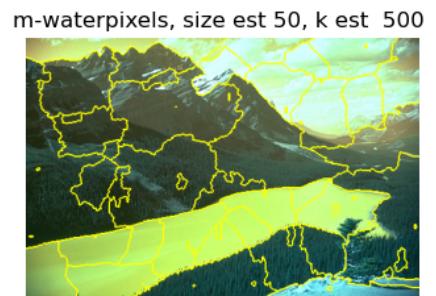
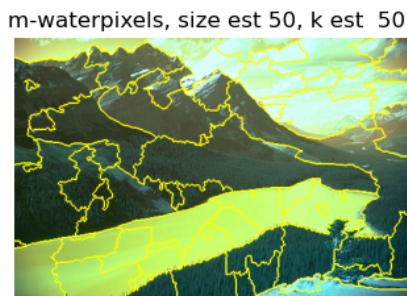
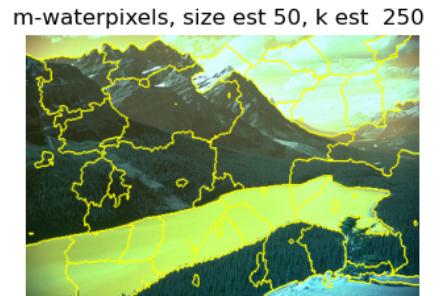
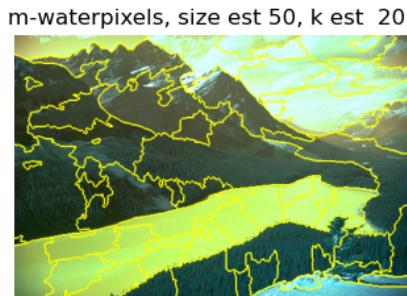
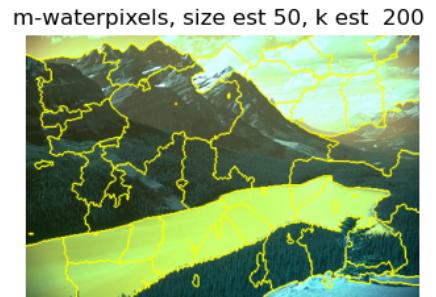
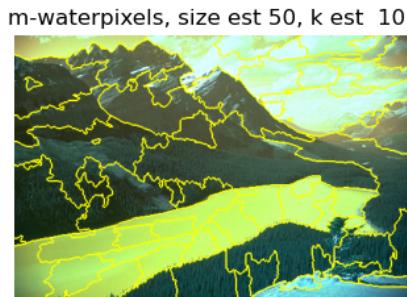
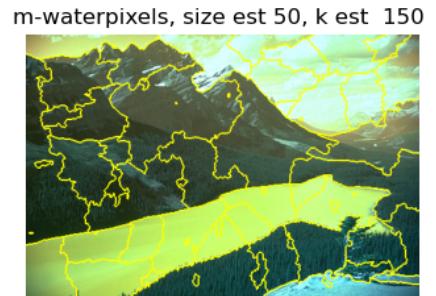
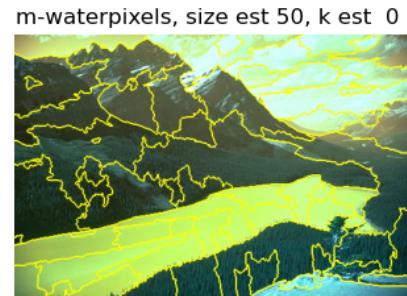


FIGURE 11 – m-waterpxiels : size = 50, k en [0,10,20,50,100,150,200,250,500,1000]

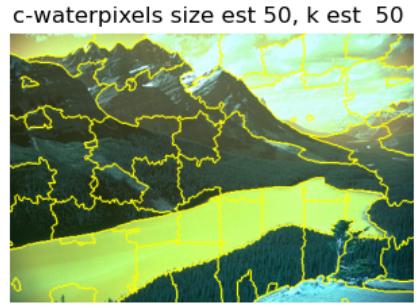
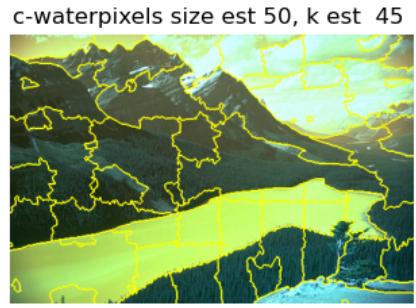
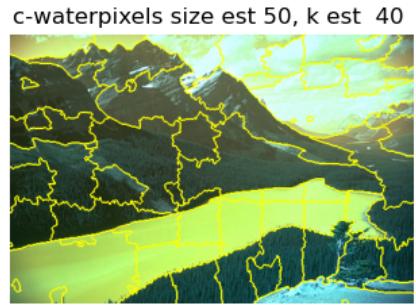
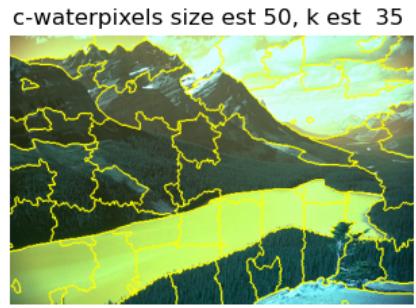
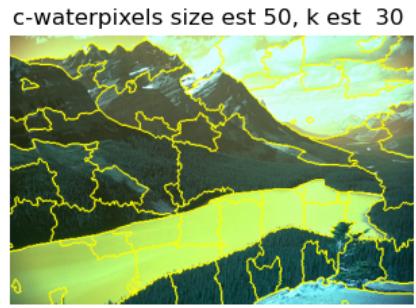
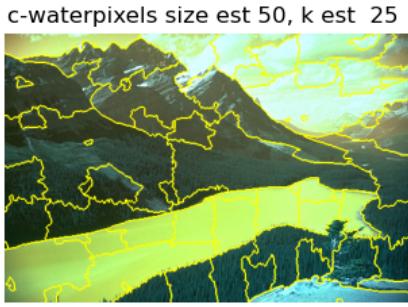
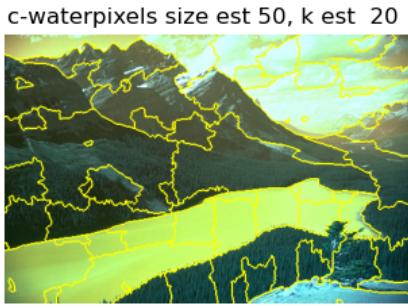
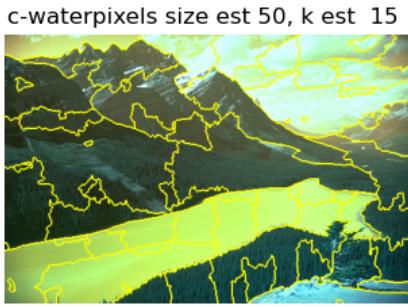
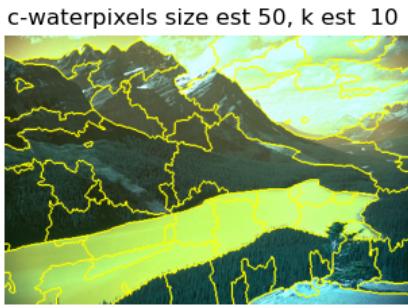
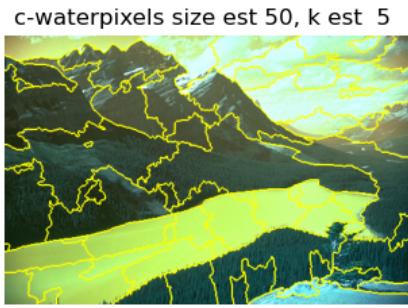


FIGURE 12 – m-waterpixels : size = 50, k en [5,10,15,20,25,30,35,40,45,50]

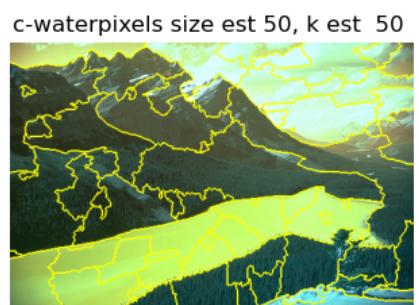
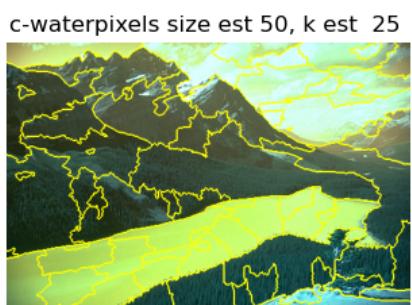
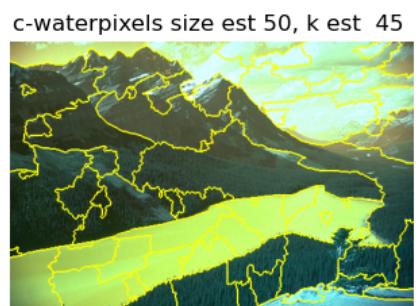
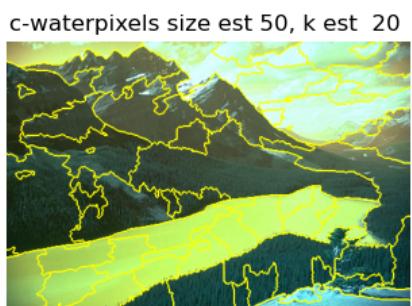
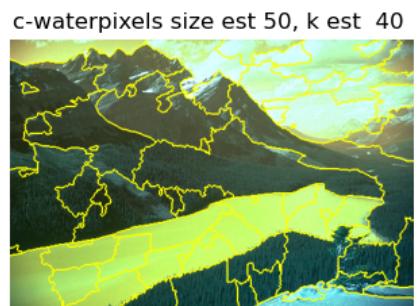
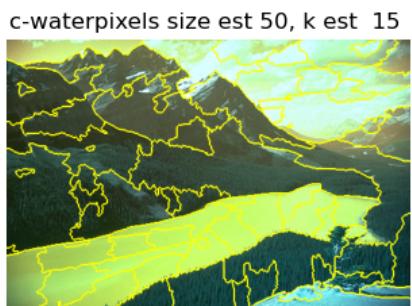
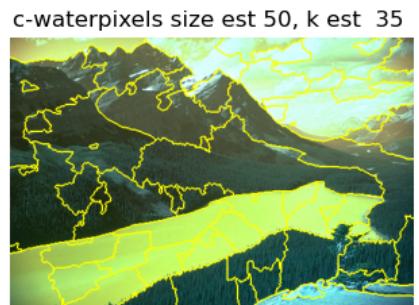
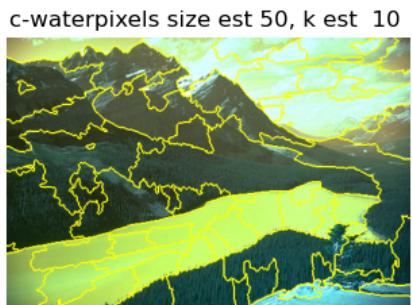
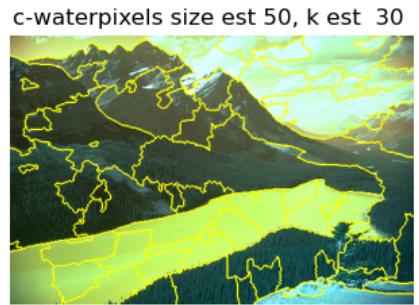
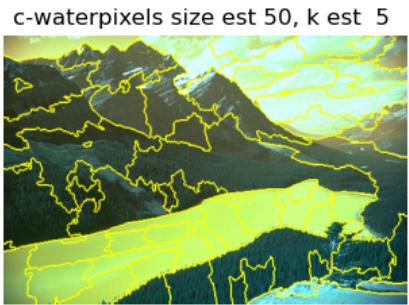


FIGURE 13 – c-waterpixels : k = 0, size en [5,10,15,20,25,30,35,40,45,50]

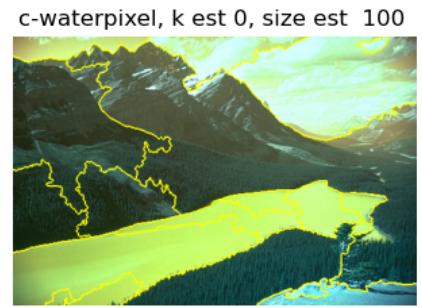
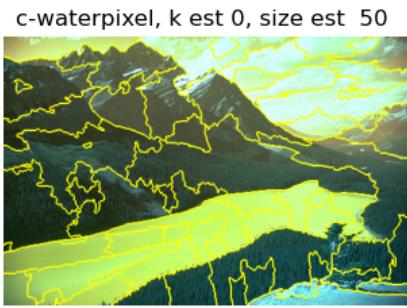
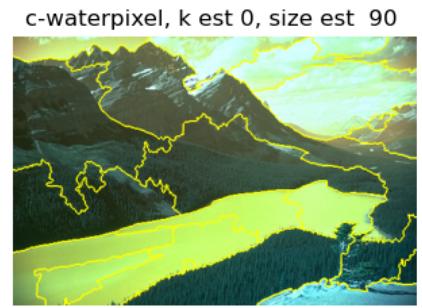
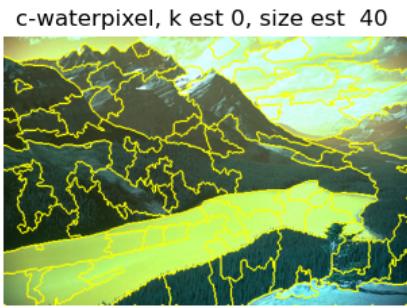
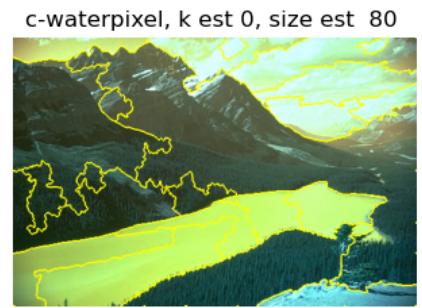
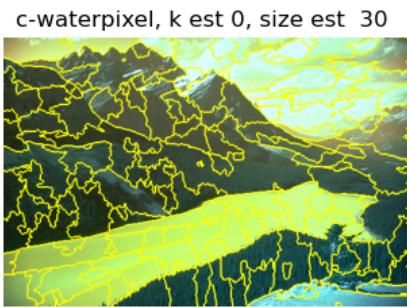
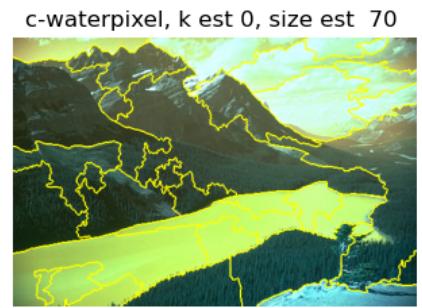
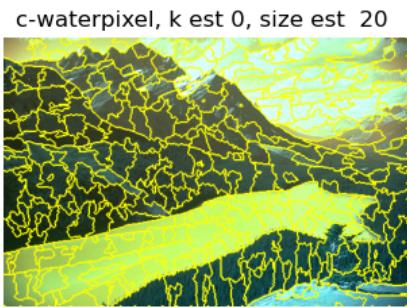
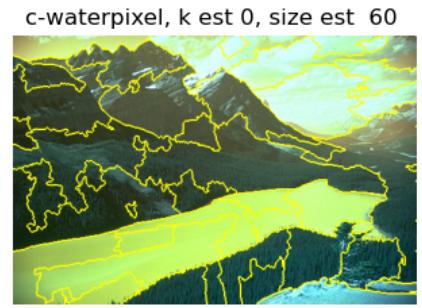
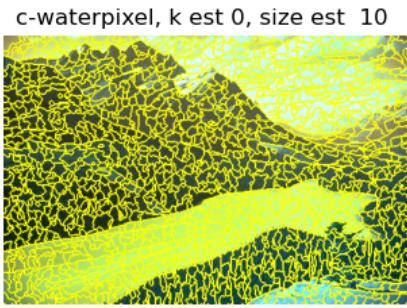


FIGURE 14 – c-waterpixels :  $k = 0$ , size en  $[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]$

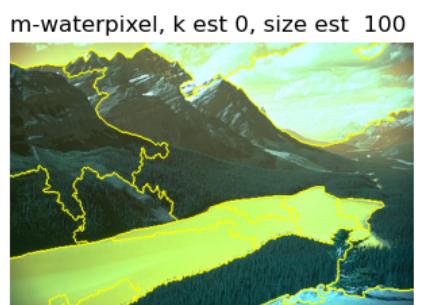
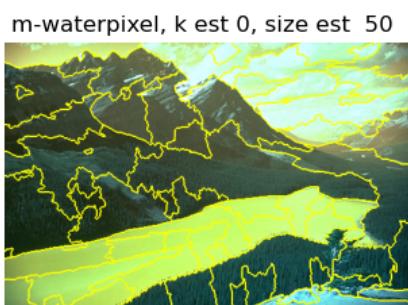
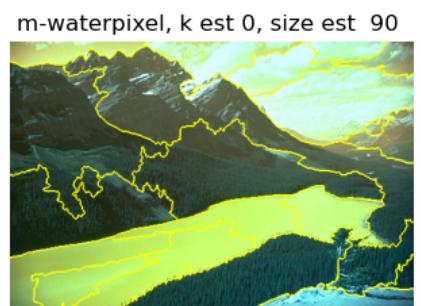
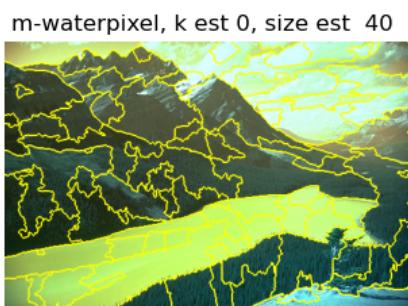
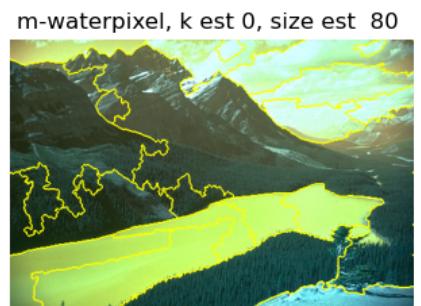
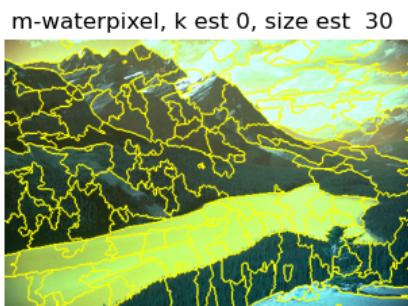
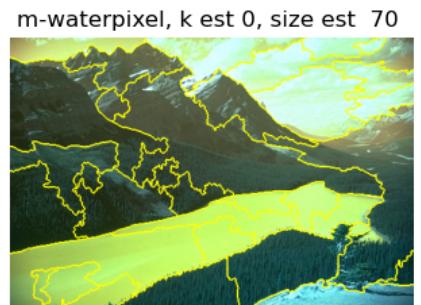
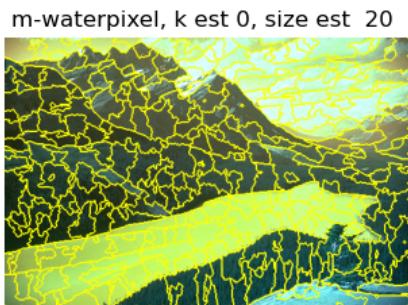
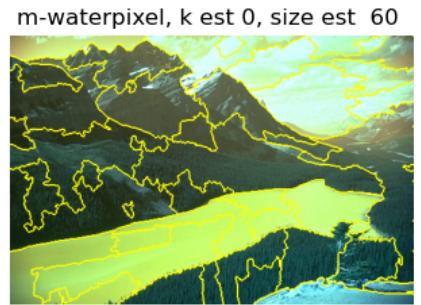
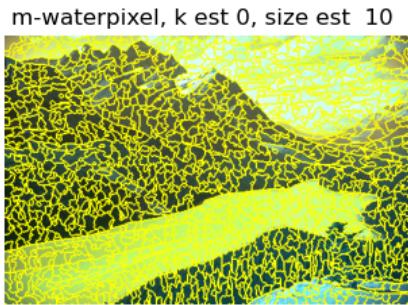


FIGURE 15 – m-waterpixels : k = 0, size en [10,20,30,40,50,60,70,80,90,100]

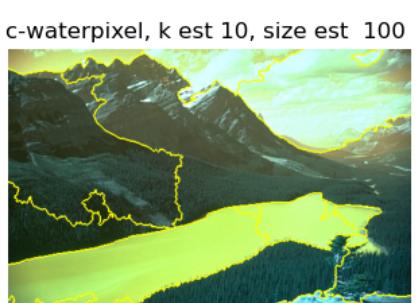
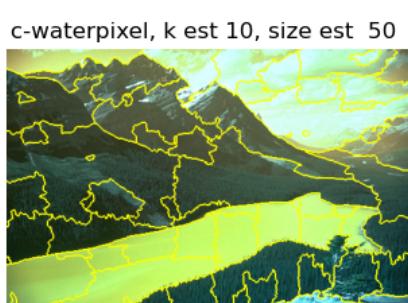
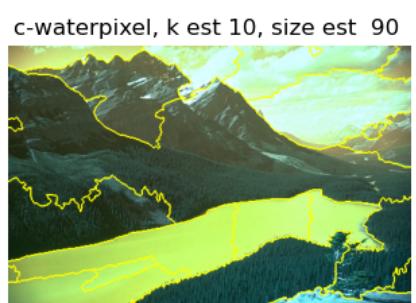
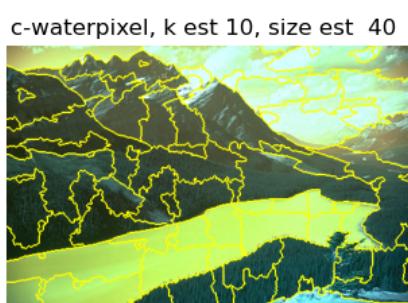
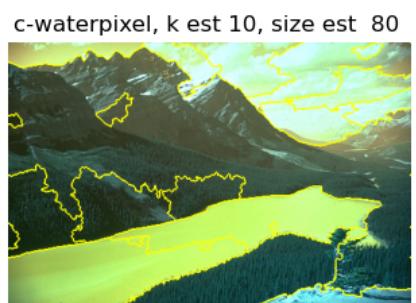
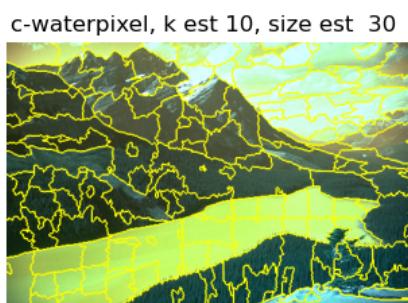
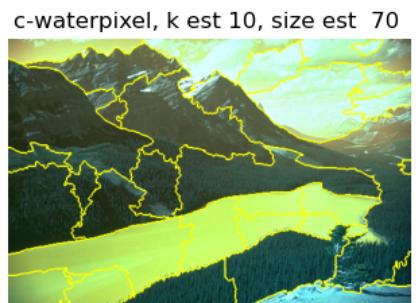
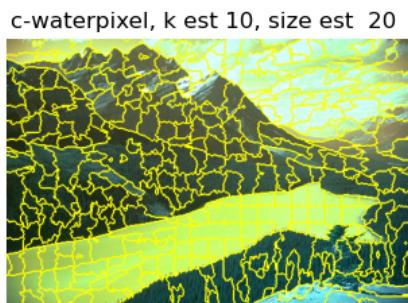
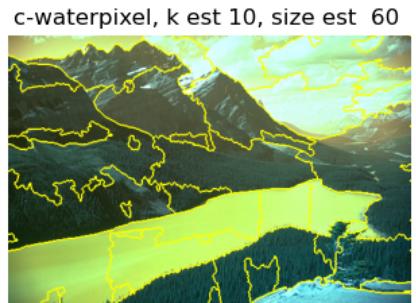
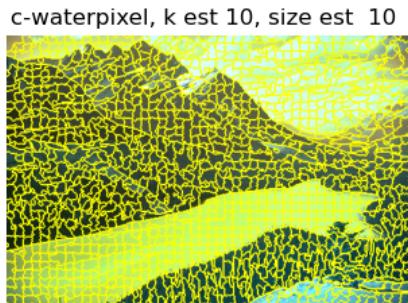


FIGURE 16 – c-waterpixels : k = 20, size en [10,20,30,40,50,60,70,80,90,100]

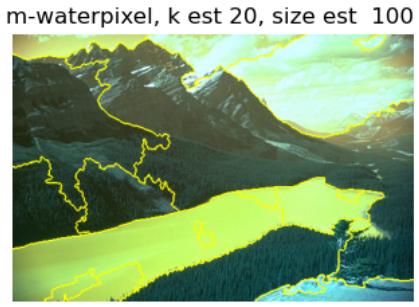
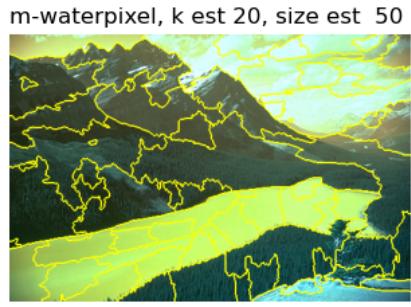
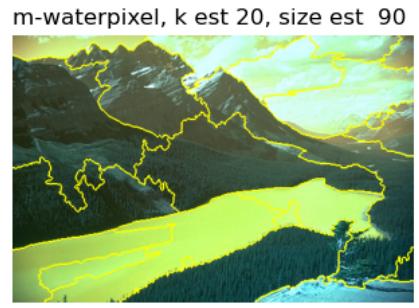
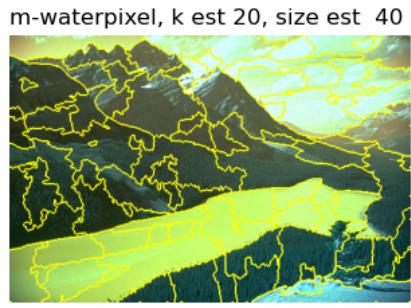
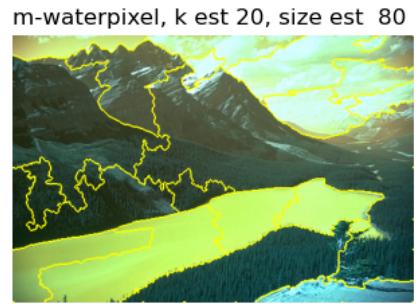
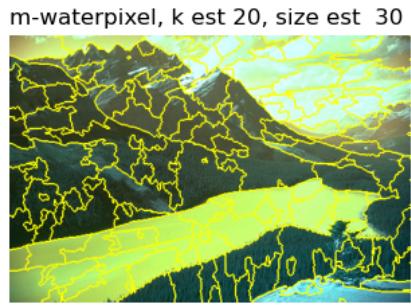
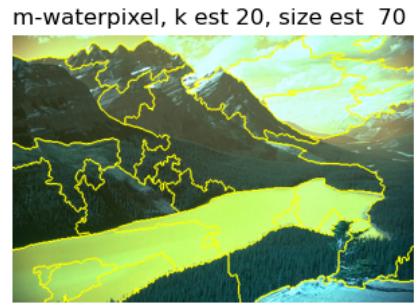
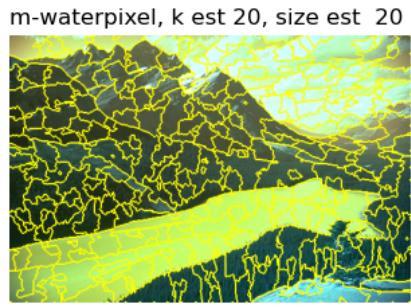
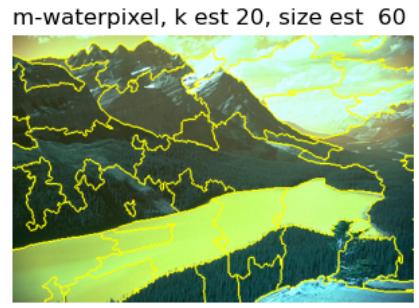
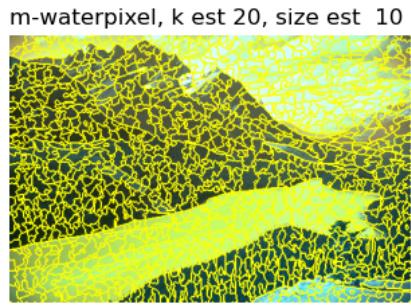
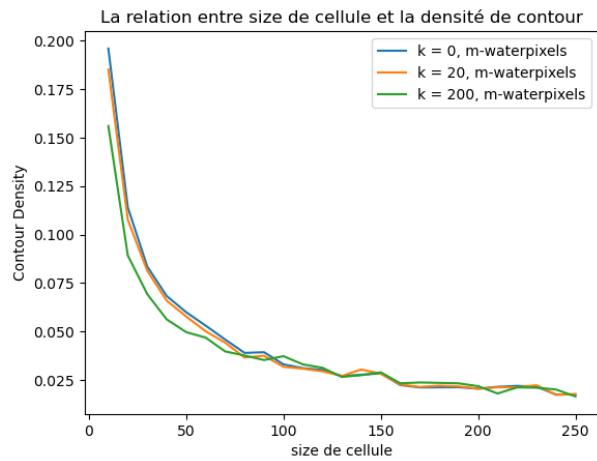
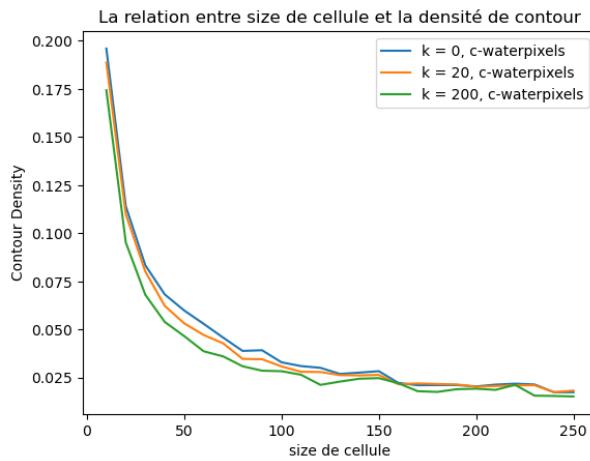


FIGURE 17 – m-waterpixels : k = 20, size en [10,20,30,40,50,60,70,80,90,100]

## 5.2 Densité de contours

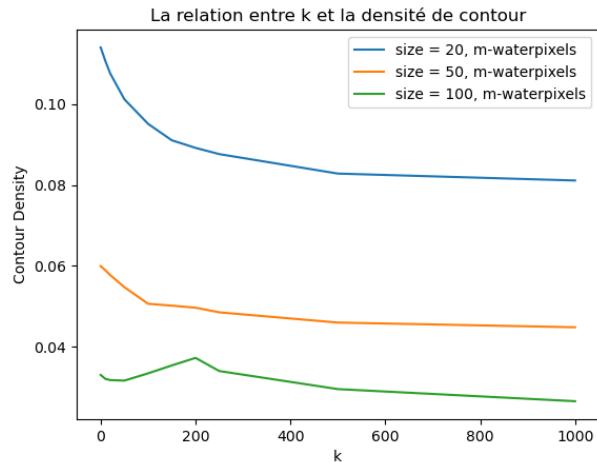
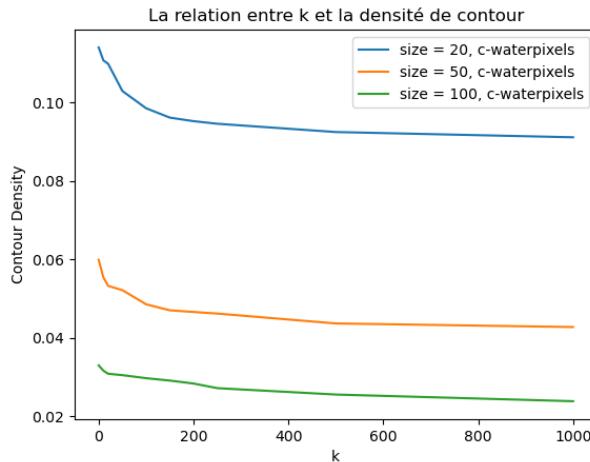
1. Tout d'abord, j'observe la relation entre la taille de la cellule et la densité de contour.

Au fur et à mesure que la taille de la cellule augmente, la densité de contour tend progressivement vers une valeur stable de 0,025.



2. Pour ici, j'observe la relation entre le paramètre de la régularisation spatiale  $k$  et la densité de contour.

À mesure que le paramètre de la régularisation  $k$  augmente, la densité de contour affiche une tendance à la baisse et tend à se stabiliser.



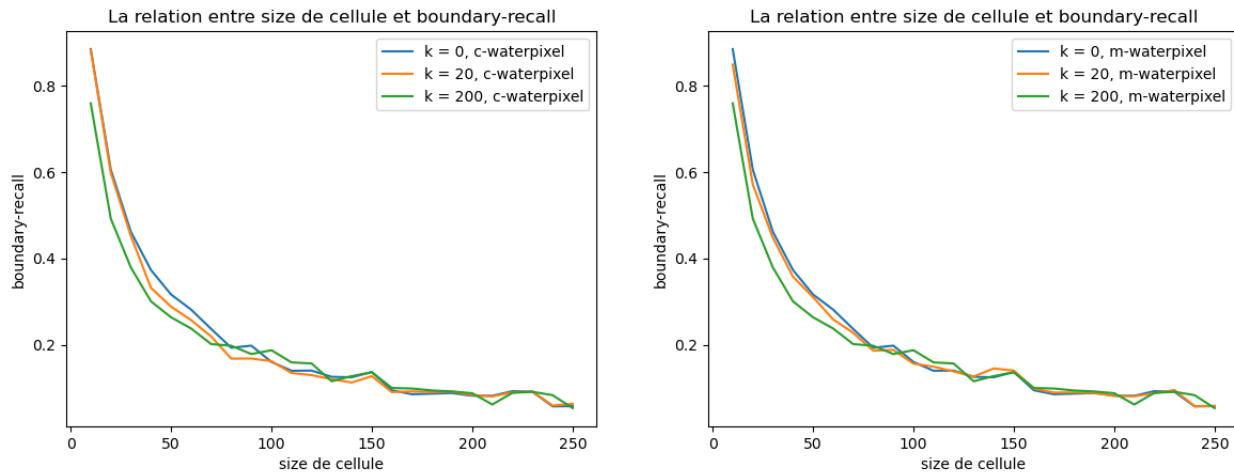
À travers les quatre figures ci-dessus, on peut conclure que lorsque le paramètre de la régularisation  $k$  est constant, à mesure que la taille de la cellule augmente, la densité de contour diminue progressivement et tend à être stable. Lorsque la taille de la cellule est constante, lorsque le paramètre de la régularisation spatiale  $k$  augmente, la densité de contour diminue progressivement et tend à se stabiliser.

Lorsque le paramètre de la régularisation  $k$  augmente ou que le size de la cellule augmente, les arêtes deviennent de plus en plus clairsemées. C'est-à-dire Des valeurs plus élevées indiquent des bords plus denses dans l'image, tandis que des valeurs plus basses indiquent des bords moins nombreux ou plus clairsemés.

### 5.3 Boundary-Recall

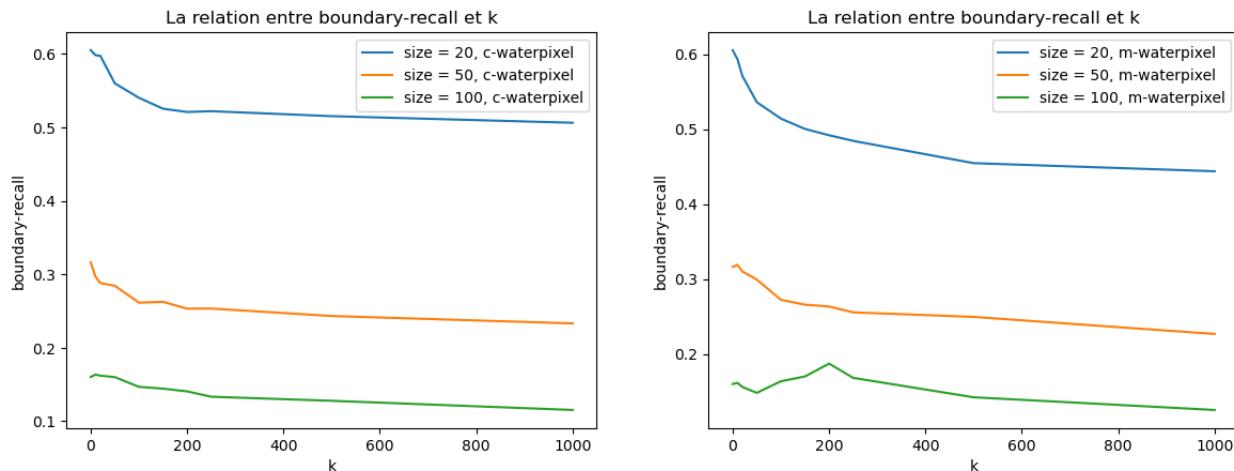
1. Premièrement, j'observe la relation entre la taille de la cellule et le boundary-recall. Pour les c-waterpixels et les m-waterpixels, le boundary-recall diminue progressivement à mesure que la taille de la cellule augmente. Lorsque la taille de la cellule est petite, la valeur du boundary-recall est plus élevée, mais cela peut entraîner une sur-segmentation.

Pour le paramètre de la régularisation spatiale  $k$ , je prends  $k = 0, 20, 200$ , dans ces deux figures, je n'ai pas vu une différence significative.



2. Pour ici, j'observe l'effet du paramètre de la régularisation spatiale  $k$  sur boundary-recall. J'ai vérifié les changements de boundary-recall des c-waterpixels et m-waterpixels lorsque la taille de la cellule est en 20,50,100.

Lorsque le paramètre de la régularisation spatiale  $k$  est compris entre 0 et 200, la diminution est évidente, puis elle tend à être plate.



De ces quatre chiffres, on peut conclure que lorsque le paramètre de la régularisation spatiale  $k$  passe de 0 à 200, le boundary-recall change fortement, la baisse est évidente, puis elle tend à être plate. Plus la taille de la cellule est grande, plus la valeur du boundary-recall est petite.

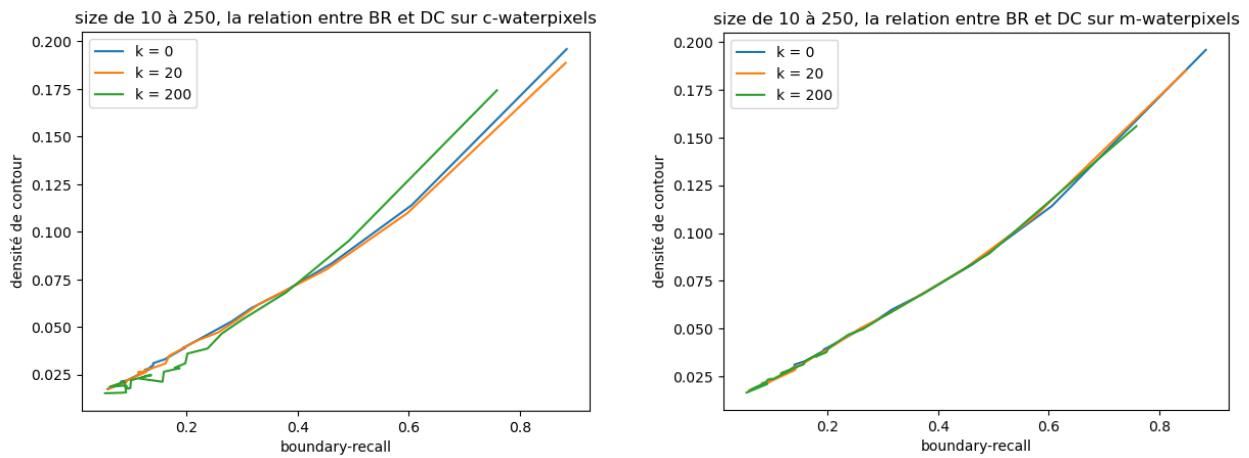
## 5.4 Densité de contours et Boundary-Recall

Tout d'abord, on observe la relation entre le boundary-recall et la densité de contour, lorsque la taille de la cellule est de 10 à 250, j'ai testé le paramètre de la régularisation spatiale  $k = 0, 20, 200$ , pour c-waterpixels et m-waterpixels.

A mesure que le boundary-recall augmente, la densité de contour augmente également, il existe donc une corrélation positive entre le boundary-recall et la densité de contour.

Pour c-waterpixels, quand  $k = 200$ , comparant avec  $k = 0$  et  $k = 20$  : premièrement, le taux de croissance est plus faible, puis dépasse les deux autres.

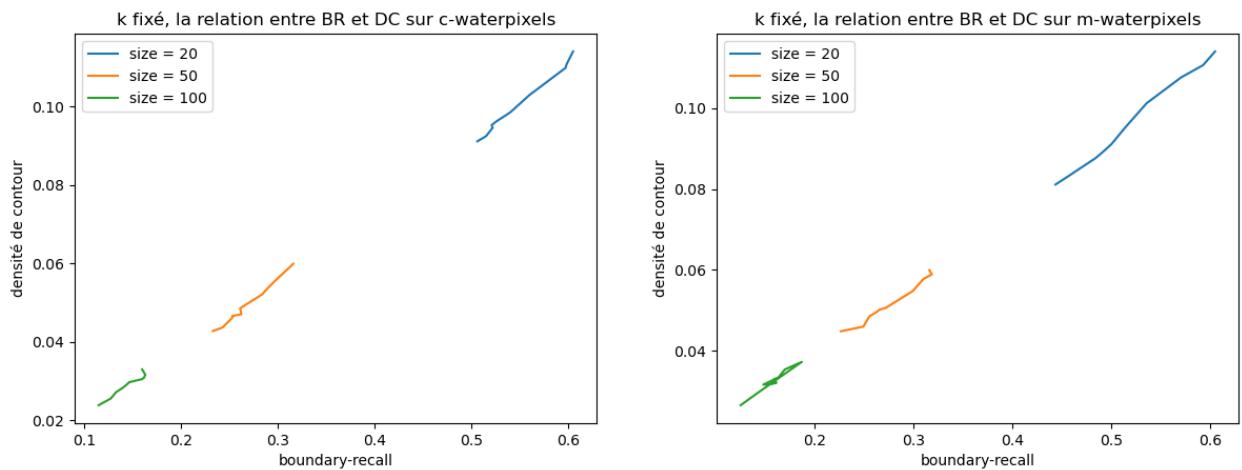
Pour m-waterpixels, pour  $k = 0, 20, 200$ , presque le même taux de croissance.



Puis, on observe la relation entre le boundary-recall et la densité de contour, lorsque le paramètre de la régularisation spatiale  $k$  est en  $[0, 10, 20, 50, 100, 150, 200, 250, 500, 1000]$ , la taille de la cellule est de  $size = 20, 50, 100$ , pour c-waterpixels et m-waterpixels.

On peut voir que pour des tailles de la cellule différentes, le boundary-recall et la densité de contour ont des plages de croissance différentes.

Plus la taille de la cellule grande, plus l'intervalle entre le boundary-recall et la densité de contour petit.



## 5.5 Waterpixels et SLIC

Dans cette partie, je voudrais comparer waterpixels et SLIC. Pour réaliser SLIC, j'applique **slic** dans la bibliothèque **skimage.segmentation** en changement le paramètre **n\_segments** qui est le nombre de superpixels.

Donc au début, il faut convertir la taille de la cellule au nombre de la cellule en initiale.

Je prends la taille de la cellule de 10 à 250, chaque pas est 10. Le code est comme suivante :

---

### Algorithme 25 :

```
size = [size for size in range(10, 251, 10)]
nb = []
N,M = img_gray.shape
for s in size do
    nb.append(np.ceil(N/s)*np.ceil(M/s))
end
```

---

Depuis, le résultat de la fonction **slic** est la matrice de segmentation, en appliquant la fonction **find\_boundaries**, on peut obtenir **boundaries**, le code est comme suivante :

---

### Algorithme 26 :

```
SLIC_seg(img,n_segments) :
    segments = slic(img, n_segments)
    boundaries = find_boundaries(segments)
    DC_liste = []
    BR_liste = []
    N × M la taille de l'image d'entrée
    for i in nb do
        boundaries = SLIC_seg(img_gray, i)
        sc = np.where(boundaries == True, 1, 0).sum()
        sp = (2 * (N + M) - 4)
        DC_liste.append((0.5 * sc + sp)/(N * M))
        BR_liste.append(BR(boundaries, liste_GT))
    end
```

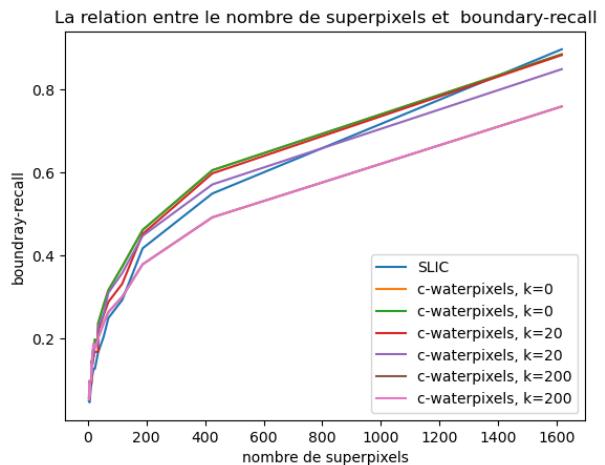
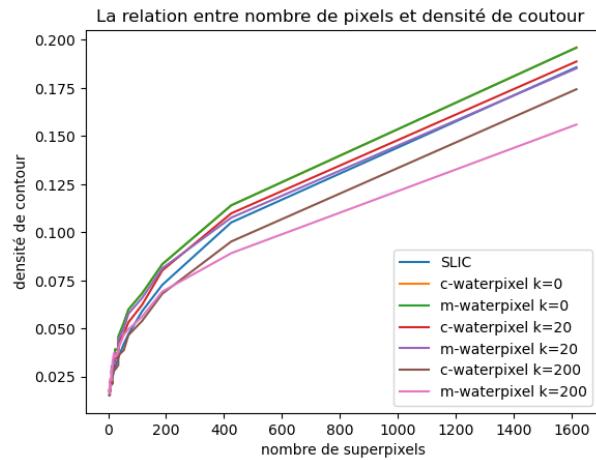
---

Donc, on obtins la densité de contours et le boundary-recall par SLIC.

Les deux images suivantes sont : la relation entre le nombre de superpixels et la densité de contours, et la relation entre le nombre de superpixels et le boundary-recall.

Pour la figure en gauche, on peut observer que, quand  $k = 0$  et  $k = 20$ , c-waterpixels et m-waterpixels ont la plus grande densité de contour que SLIC.

Pour la figure en gauche, on peut observer que, quand  $k = 0$  et  $k = 20$ , c-waterpixels et m-waterpixels ont le plus grand boundary-recall que SLIC, dans le cas où le nombre de superpixels est inférieur que 800. Et quand le nombre de superpixels est supérieur que 1300, SLIC a le plus grand boundary-recall. Mais il faut faire l'attention sur la sur-segmentation.



## 6 Amélioration et perspectives

Afin d'améliorer l'effet de la segmentation sur la base de mon expérience, on peut essayer les méthodes suivantes :

### 6.1 Sélection du gradient d'image

On peut améliorer le prétraitement des images et choisir de meilleures méthodes pour fournir des résultats de détection des bords précis. Dans mon expérience, j'ai choisi l'opérateur de Sobel. De plus, on peut essayer l'opérateur Prewitt, l'opérateur Roberts, l'opérateur Laplacien, la détection des contours Canny, etc. Identique à l'application de la méthode du filtre de soble.

### 6.2 Sélection de la forme de la cellule initiale

On peut expérimenter différentes formes de cellules initiales. Pour mon expérience, j'ai choisi des carrés. En plus de cela, on peut essayer des triangles, des hexagones, etc.

### 6.3 Sélection du critère d'évaluation

On peut choisir d'autres méthodes pour vérifier la précision des résultats de détection des contours. Dans mon expérience, j'ai essayé la densité de contour et le boundary-recall. En plus de cela, on peut également essayer le facteur d'inadéquation moyen  $MF$ .

### 6.4 Sélection de la méthode de segmentation

On peut choisir d'autres méthodes d'image de la segmentation par comparaison. Dans mon expérience, je compare avec la méthode de la segmentation SLIC. De plus, on peut également le comparer avec l'algorithme des watershed.

### 6.5 Tester sur plus d'images

Pour mon expérincence, j'ai appliqué l'algorithme de waterpixels sur une image dans la database de la segmentation Berkeley. On peut essayer plus d'images sur la database, et les tester.

## 7 Conclusion

Basé sur les connaissances de base d'images et l'application de bibliothèques de langage python, telles que matplotlib, cv2, numpy, scipy, skimage. J'ai implémenté l'algorithme de la segmentation de waterpixels à partir de la thèse.

Et l'algorithme de segmentation d'image est évalué par densité de contour et boundary-recall. De plus, comparé avec SLIC.

Un résultat relativement satisfaisant a été obtenu. Quand le paramètre de la régularisation spatical  $k$  est moins que environ 20, les waterpixels sont meilleurs que la segmentation SLIC car ils sont plus grands densité de contours et boundary-recall. Pour le choix de la taille de la cellule, il faut faire attention que lorsque la taille est petite, il y aura sur-segmentation, lorsque la taille est grande, les contours des détails n'apparaîtront pas.

J'ai une compréhension plus approfondie des bases de l'image, et je suis plus compétent dans la mise en œuvre d'algorithmes, ce qui est propice à l'étude continue du traitement d'image et de la découpe d'image.

## 8 Référence

- [1] Vaia Machairas, Matthieu Faessel, David Cardenas-Pera, Theodore Chabardes, Thomas Walter. (November 2015). Waterpixels. IEEE Transactions on Image Processing ( Volume : 24, Issue : 11)
- [2] Pablo Arbelaez, Charless Fowlkes, David Martin. (June, 2007).The Berkeley Segmentation Dataset and Benchmark. [https ://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/](https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/)
- [3] L'quipe de developpement de scikit-image. Comparison of segmentationand superpixel algorithms. [https ://scikit-image.org/docs/stable/auto\\_examples/segmentation/plot\\_segmentations.html](https://scikit-image.org/docs/stable/auto_examples/segmentation/plot_segmentations.html)
- [4] Darshita Jain. (2019). Superpixels and SLIC. [https ://darshita1405.medium.com/superpixels-and-slic-6b2d8a6e4f08](https://darshita1405.medium.com/superpixels-and-slic-6b2d8a6e4f08)
- [5] L'quipe de developpement de scikit-image. Markers for watershed transform. [https ://scikit-image.org/docs/stable/auto\\_examples/segmentation/plot\\_marked\\_watershed.html](https://scikit-image.org/docs/stable/auto_examples/segmentation/plot_marked_watershed.html)
- [6] Serge Beucher.(2010). IMAGE SEGMENTATION AND MATHEMATICAL MORPHOLOGY. [https ://people.cmm.minesparis.psl.eu/users/beucher/wtshed.html](https://people.cmm.minesparis.psl.eu/users/beucher/wtshed.html)