Noé Mosca et Baptiste Vibert

Rapport final de projet : Waterpixels

Sommaire

———

I Présentation du sujet

Notre projet consiste à implémenter, et montrer l'effet de différents paramètres d'une méthode de segmentation d'image en superpixels "waterpixels", adhérents aux contours des éléments de l'image, décrite dans l'article Waterpixels

Le sujet aborde notamment les différentes étapes pour implémenter un tel algorithme.

Les étapes sont :

- Calcul du gradient morphologique de l'image
- Définition de N cellules régulières de l'image (cellules carrées ou hexagonales) qui forme une grille

  Soit un ensemble de point $Q = \{q_i\}_{1 \leq i \leq N}$, on définit la distance

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

  avec σ le pas de a grille, et D l'ensemble des pixels de l'image. La distance utilisée d d dépend de la cellule choisie.
- On sélectionne un marqueur par cellule (centre de la cellule ou surface d'extinction la plus grande), l'ensemble des N marqueurs est $Q = \{q_i\}_{1 \leq i \leq N}$.
- On définit un nouveau gradient régularisé qui vaut $g_{reg} = g + k d_Q$
- On applique l'algorithme de watershed à l'image originale avec les marqueurs définis par les étapes précédentes.

II Les c-waterpixels

Pour l'implémentation des waterpixels, il faut dans un premier temps définir un ensemble de cellules.
Le cas le plus simple est celui des cellules carré, (nous étudierons aussi les cellules hexagonales).
Une fois ces cellules définies, il faut choisir un marqueur dans cette cellule. Intuitivement, on peut choisir le centre de chaque cellule, c'est le cas le plus simple que nous étudions ici.

a) Cellules carrées

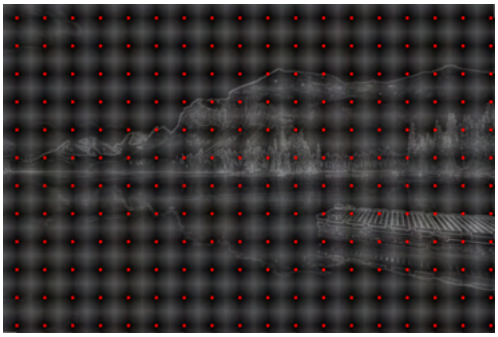
Pour les cellules carrées nous devons nous munir d'une norme telle que la boule unité soit un carré. C'est donc la norme infinie que l'on choisit ici :

$$x = (x_1, x_2),\ \|x\|_\infty = max(|x_1|, |x_2|).$$

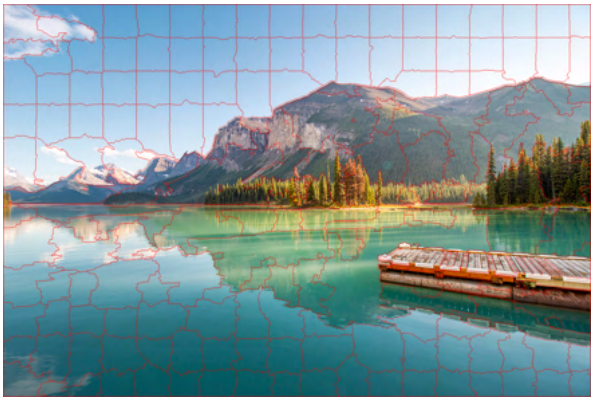
On part de l'image suivante :

- on applique ensuite le gradient régularisé avec k = 3 et un espacement de 85 pixels

- on ajoute les marqueurs



- on applique la transformation Watershed :



Les cellules carrées sont bien visibles dans les zones ou la couleur est uniformes et la plupart des contours importants sont marqués (même ceux du reflet dans l'eau), on observe néanmoins que certains contours sont espacés de l'image originale, la watershed manque de précision par endroits.

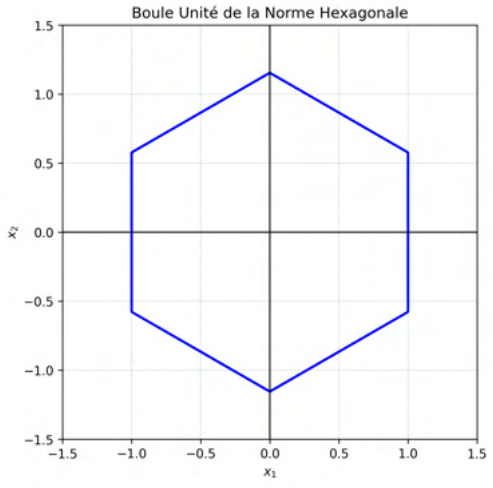
Les waterpixels avec des cellules carrées sont simples à implémenter, mais il est plus intéressant de travailler avec des cellules hexagonales qui épousent mieux les contours des objets.

b) Cellules hexagonales

L'implémentation des cellules hexagonales est similaire à celle des cellules carrées, mais la norme utilisée est différente. On utilise la norme hexagonale, pour laquelle la boule unité dans est un hexagone :

$$x = (x_1, x_2),\ \|x\|_{\text{hexa}} = \max\left(|x_1|, \left|\frac{\sqrt{3}x_2 + x_1}{2}\right|, \left|\frac{\sqrt{3}x_2 - x_1}{2}\right| \right)$$

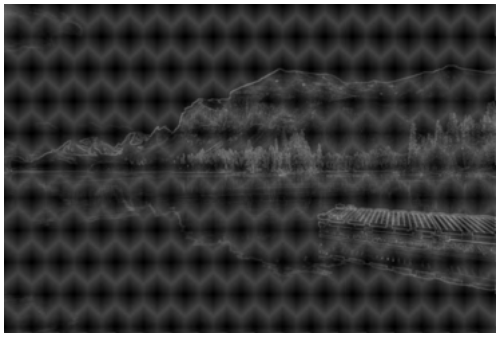
On obtient la boule unité suivante :



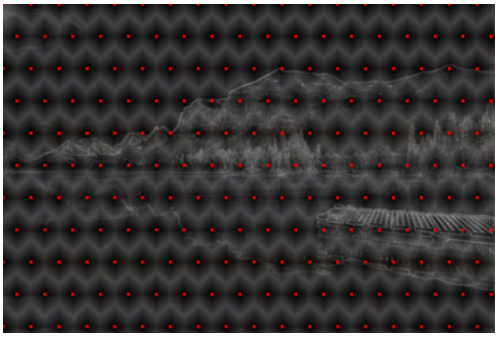
de largeur 2 et de hauteur $\frac{4}{\sqrt{3}}$.

En implémentant cela on obtient les mêmes étapes que précédemment avec pour différence les images intermédiaires suivantes :

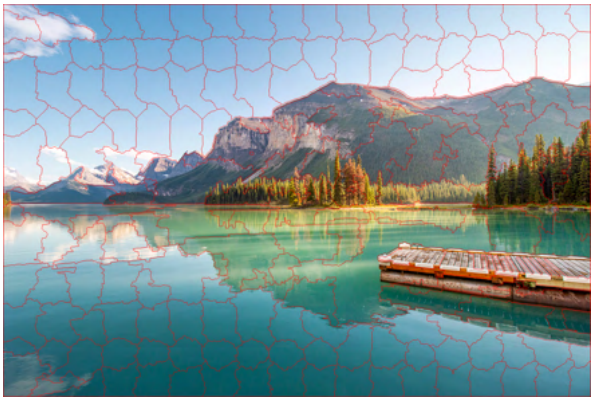
- gradient régularisé avec k = 3 et un espacement de 85 pixels



• avec les marqueurs :



• On obtient la transformation watershed finale :



Note importante :

Pour régulariser le gradient, il est nécessaire de calculer en chaque point sa distance à l'ensemble des centres des cellules. Dans le cas des cellules carrées, la norme infinie est utilisée. Dans le cas des cellules hexagonales, on se ramène à $\|.\|_\infty$.
En effectuant un simple calcul de minimum on obtiendrait une complexité en $n \times m$, où n est le nombre de pixels et m le nombre de waterpixels, et le temps de calcul résultant devient bien trop long. On a donc recours à la transformation en distance. La librairie **openCV** ne proposant que cette opération pour des fonctions de base bien définies, il a fallu la recoder. Nous avons choisi d'effectuer le calcul de manière approchée pour gagner en temps de calcul. L'algorithme consiste à initialiser la distance en chaque point à l'infini, puis à parcourir l'image dans toutes les directions, en affectant à chaque pixel le minimum entre sa valeur courante, et celle de ses voisins plus la distance du pixel considéré à ses voisins. Les distances à l'ensemble des centres de cellules se propagent donc de voisin en voisin pour couvrir toute l'image.

L'opération est bien en temps linéaire par rapport au nombre de pixels car elle consiste en plusieurs itérations sur l'ensemble des pixels ainsi que d'autres opérations moins coûteuses.

c) Évolution en fonction de k

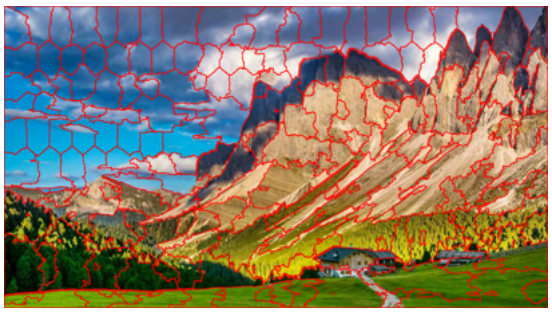
On cherche ici à déterminer comment se comporte les waterpixels lorsque l'on fait varier k.

Prenons ici le cas des cellules hexagonales.

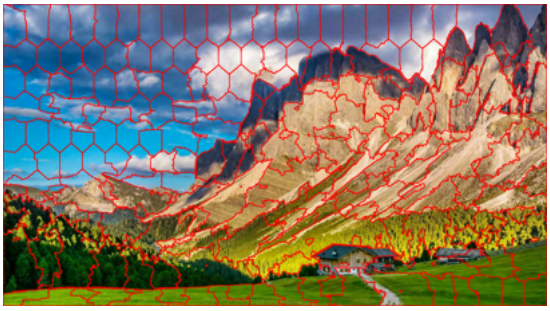
On fixe l'espacement à 85 pixels

k = 1

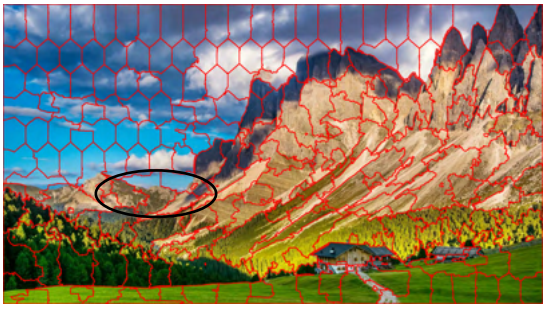
k = 3



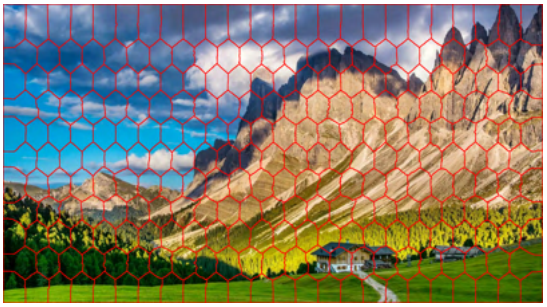
k = 6



k = 8 : *décollage des pixels dans le fond*



k = 200



La valeur de k a une influence sur la régularité des water-pixels. Plus k est faibles, plus les cellules sont adhérentes aux contours. Mais dans ce cas les cellules sont moins régulières, on observe des formes assez approximatives.

Pour une valeur de k intermédiaire comme k = 6 ou 7, on observe que les waters pixels épousent bien les contours principaux, et ceux sur des zones uniformes (comme le ciel sur l'image étudiée) prennent des formes régulières qui s'approchent d'hexagones.

A partir d'une certaine valeur de k on commence à perdre les contours principaux, on observe cette transition pour k = 10 ou les water pixels commencent à se détacher des contours des montagnes

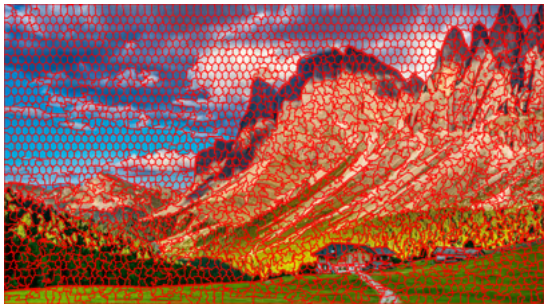
Pour k >> 10 (comme k = 200) les water pixels tendent vers un diagramme de Voronoï où l'image est pavée par des hexagones réguliers.

d) Évolution en fonction de l'espacement

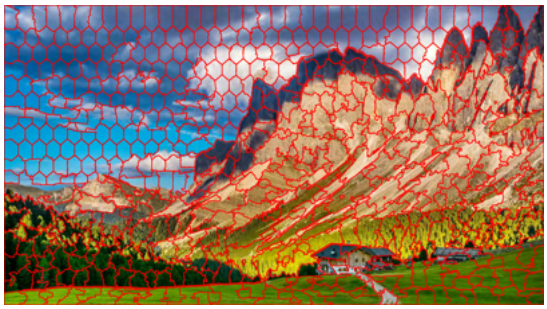
On a étudié précédemment l'influence du paramètre k sur les water pixels. On regarde ici pour un k fixé comment varie les water pixels en modifiant l'espacement des cellules. On garde la même image pour pouvoir comparer aux résultats précédent.

Fixons dans un premier temps k = 6, qui correspond à un juste milieu entre régularité des water pixels et bonne adhérence aux contours (du moins pour 85 pixels d'espacement).

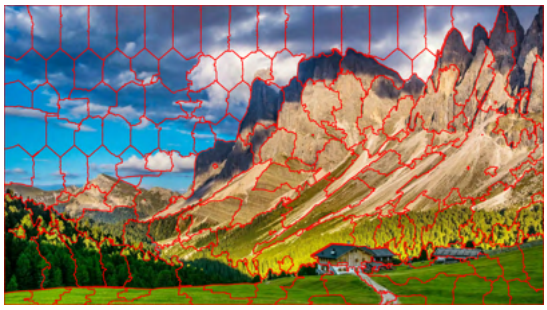
espacement = 25 pixels



espacement = 50 pixels



espacement = 100 pixels



espacement = 200 pixels

Pour k fixé, on remarque que pour un espacement très petit (ici 25 pixels) on a une multitudes de cellules hexagonales qui épousent tous les contours de l'image. Néanmoins il y a beaucoup trop de water pixels, ce qui rend la segmentation difficilement exploitable.

Pour un espacement intermédiaire de 100 pixels environ le nombre de watrpixels obtenu est correcte ils sont adhérents aux contours de l'image.

Pour un espacement trop grand (ici au dessus de 150 pixels), les waterpixels obtenus sont trop peu nombreux et se décollent des contours.

Le rapport de l'espacement sur la largeur de l'image idéal semble être de l'ordre de 1/10 environ pour la valeur de k donnée.

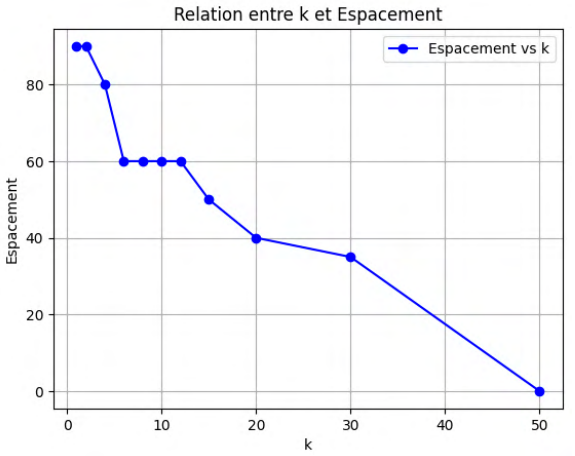
En conclusion comme pour le cas où k variait, il existe pour un k donné un espacement optimal pour les waterpixels. On peut donc définir pour chaque k une valeur de l'espacement telle que l'espacement est idéal.

e) Évolution de k et de l'espacement

Pour étudier le lien de l'évolution des waterpixels avec k et avec l'espacement on réalise une série de watershed pour k fixé en faisant varier l'espacement.
On répète cette opération pour plusieurs valeur de k.

Les valeurs de l'espacement sont celles à partir duquel le contour de la chaîne de montagne commence à se déformer.

Ce un critère très approximatif, mais qui vise simplement à montrer que plus k est élevé, plus on a besoin de cellules petite pour occuper tout les contours principaux.
On obtient les résultats suivants



Ainsi plus k augmente moins les water pixels sont adhérents aux contours de l'image. On peut donc tenter d'approcher les contours principaux en réduisant l'espacement. En effet pour les variations abruptes (comme celles qu'il peut y avoir sur l'image au niveau de la chaine de montagne), on peut approcher localement les contours pour découper les zones abruptes en plusieurs waterpixels qui vont ensemble dessiner un contour adhérent.
Néanmoins pour une valeur limite qui ici se situe aux alentours de 50, les water-pixels ne sont plus du tout assez adhérents aux contours, et la taille des cellules nécessaire tend vers 0 (ce qui n'a pas d'intérêt dans le cadre de ce projet).

f) Cellules carrées vs cellules hexagonales

Nous avons dans les parties précédentes implémenté des c-waterpixels avec des cellules carrés et des cellules hexagonales. Pourtant quel est la différence entre les 2, et laquelle est-il préférable d'implémenter ?

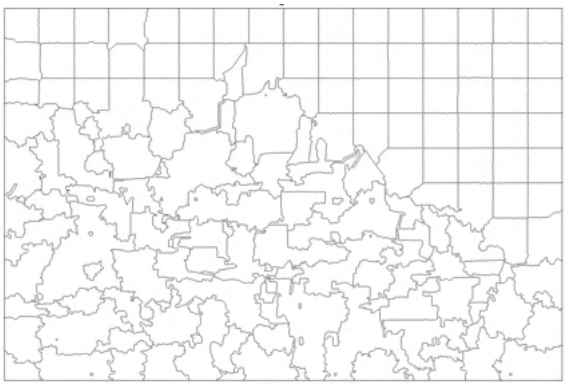
Pour cela on part de l'image suivante :

L'image fait 1200 × 800 pixels.

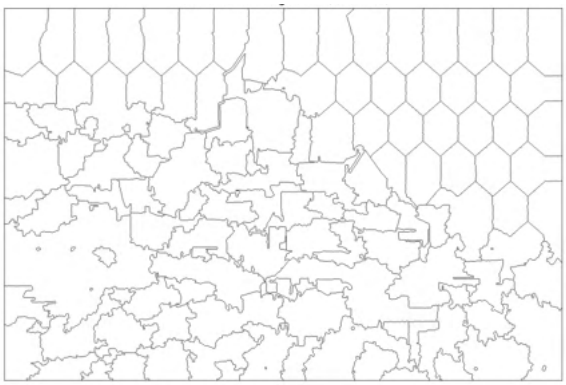
On applique notre algorithme pour former des waterpixels sur cette image avec des cellules carrées et des cellules hexagonales.

On prend k = 6 et un espacement de 75 pixels, soit un rapport espacement/largeur de 1/10 environ.

cellules carrées

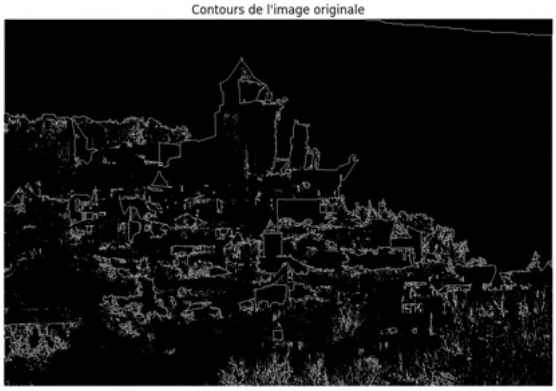


cellules hexagonales

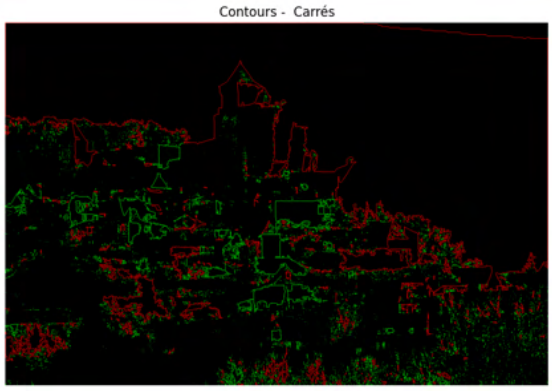


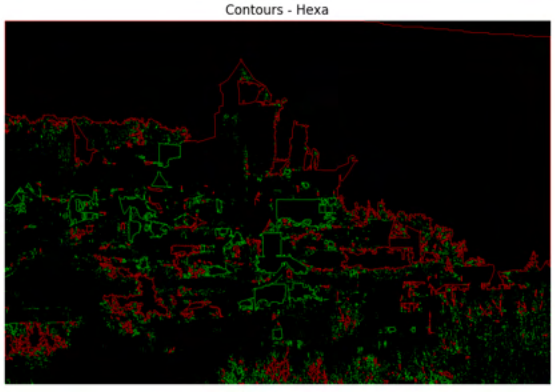
À partir de ces images il est difficile de déterminer quel type de cellule convient le mieux. Dans les 2 cas on reconnaît peu d'éléments de l'image originale, si ce n'est la forme générale ainsi que les contours de la tour principale et de que quelques maisons.

Pour savoir quel image correspond le mieux, on utilise la fonction **findContours** de **cv2**. Elle nous permet d'obtenir l'image suivante dont on récupère les contours :



On compare ensuite les contours des waterpixels avec ceux de la fonction **cv2.** Chaque pixel du contour des waterpixel qui correspond est coloré en vert.
On s'accorde que les pixels ne doivent pas correspondre exactement mais être à une distance de 1 pixel des contours réels.
On compte ensuite le nombre des pixels du contours que les waters pixels ont réussi à atteindre.





On obtient au finale que les contours des waterpixels matchent avec les contours de l'image avec une différence de 0.2% entre ceux issus des cellules carrés et ceux issus des cellules hexagonales.

Ce sont les cellules carrés qui ont une légère meilleure correspondance.

Pour pouvoir conclure il faudrait déterminer le maximum de correspondance avec les contours de l'image pour différentes valeurs de k et de l'espacement.

Il semble néanmoins que sur cette exemple les 2 types de cellules se valent. L'image choisie comporte aussi bien des contours droit que des contours arrondis. On peut penser que pour une images avec des contours droits (comme une photo avec des immeubles par exemple), serait mieux segmentée avec des waterpixels issus de cellules carrés, et inversement.

III Les m-waterpixels

Les m-waterpixels diffèrent un peu des c-waterpixels. La différence réside dans l'implémentation des marqueurs qui serviront après pour le watershed.

Dans le cas des c-waterpixels, on prenait simplement le centre des cellules. Pour les m-waterpixels on doit partir d'une surface. Pour chaque cellule le marqueur correspond à la plus grande surface d'extinction du gradient, c'est à dire à la plus grande zone où le gradient est minimal. Il peut arriver que ce minimum n'existe pas, mais cela est peu commun dans les image naturelles, donc nous n'étudions pas ce point.

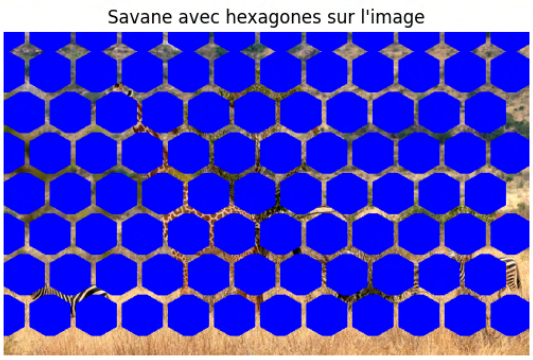
A partir de ces plus grandes zones d'extinction de gradient on extrait une nouvelle liste de marqueurs définie comme étant le barycentre de ces zones.

Pour implémenter cela en python nous avons procédé comme suit :

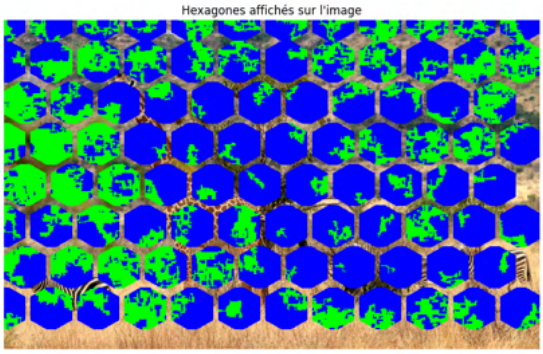
```
// 1. Définir les centres des cellules hexagonales dans l'image
centres = définir_centres_hexagonaux(image, espacement)

// 2. Initialiser une liste vide pour stocker les barycentres des zones calculées
barycentres = []

// 3. Définir un carré autour de (0, 0) avec une taille d'espacement suffisante pour contenir un hexagone
carré = extraire_carré(image, centre=(0, 0), largeur=espacement)

// 4. Initialiser une liste vide pour stocker les pixels de l'hexagone
hexagone_pixels = []

// 5. Pour chaque pixel dans le carré extrait autour de (0, 0)
pour chaque pixel dans carré :
   // a. Calculer la distance au centre (0, 0) en norme hexagonale
   distance_hex = calculer_distance_hexagonale(pixel, centre=(0, 0))

   // b. Si la distance est inférieure à l'espacement (taille de l'hexagone)
   si distance_hex < espacement :
      // Ajouter ce pixel à hexagone_pixels
      ajouter pixel à hexagone_pixels

// 6. Pour chaque centre (x, y) dans centres
pour chaque centre dans centres :
   // a. Traduire tous les pixels de l'hexagone par (x, y)
   hexagone_translaté = [pixel + (x, y) pour chaque pixel dans hexagone_pixels]

   // b. Calculer la zone de gradient minimal pour cette zone d'hexagone
   zone_min_grad = calculer_zone_minimal_gradient(image, hexagone_translaté)

   // c. Calculer le barycentre de la zone minimale (ou de l'hexagone si pas de gradient)
   barycentre = calculer_barycentre(zone_min_grad)

   // d. Ajouter le barycentre à la liste des barycentres
   ajouter barycentre à barycentres

// 7. Retourner la liste des barycentres
retourner barycentres
```

a) Implémentation des m-waterpixels

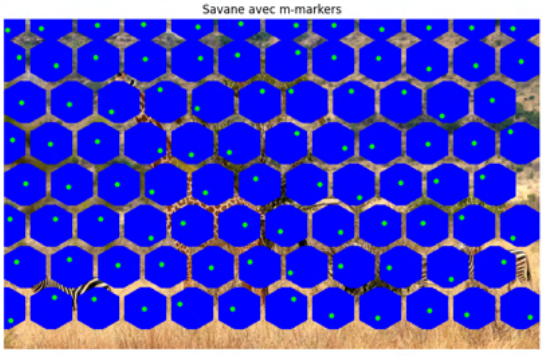
Partons de l'image suivante :



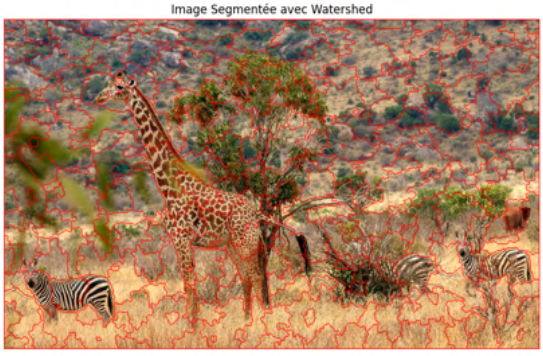
Pour chaque zone hexagone, on va chercher celle qui a la plus grande surface d'extinction. On obtient les surface suivantes :



En calculant le barycentre on obtient On applique



la transformation watershed :



On remarque que pour une faible valeur de k, les waterpixels sont adhérents au contours qui ne sont pas trop abruptes. Pour les zones qui varient plus brusquement, comme les buisson ou la tête de la girafe sur l'image, la segmentation est moins fiable.

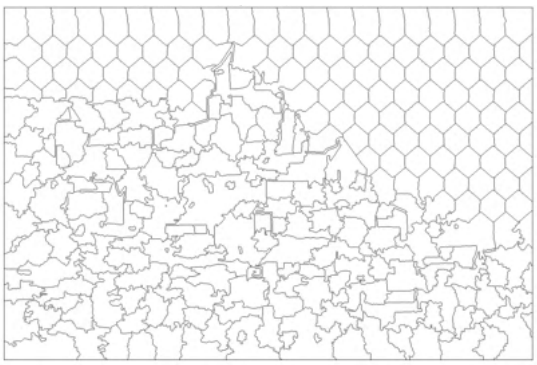
b) Évolution en fonction de k et de l'espacement

Pour l'évolution de des m-waterpixels avec k et avec l'espacement, on peut tirer les même conclusions que l'évolution des c-waterpixels avec k et avec l'espacement.

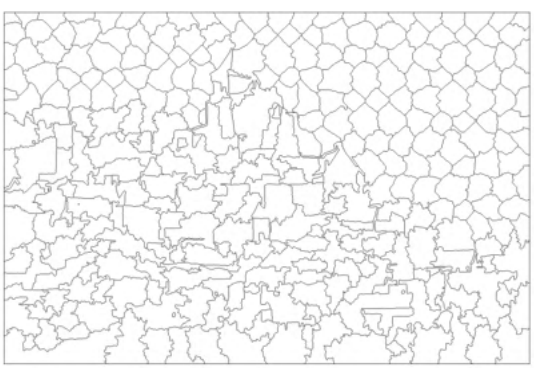
c) Comparaison des c-waterpixels et des m-waterpixels

La première chose qu'il peut être interessant de regarder et quel type de waterpixels permet le mieux d'adhérer aux contours d'un image pour des paramètres données.
On réalise le même procédé que pour la comparaison entre entre les cellules carrées et les cellules hexagonales pour les c-waterpixels (avec la même image). Ici k = 6 et l'espacement vaut 50 pixels.

Pour les c-waterpixels :



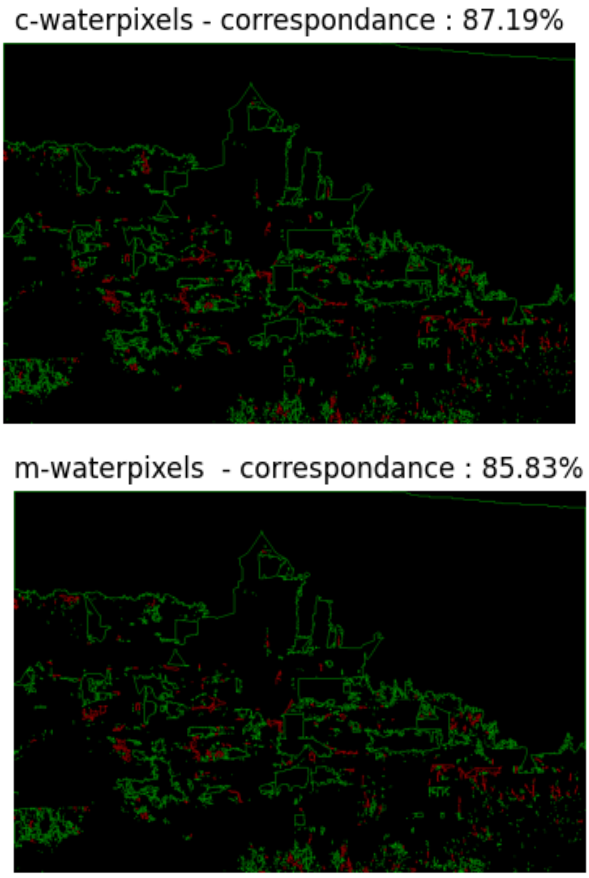
Pour les m-waterpixels :



On remarque que pour k fixé, les cellules des c-waterpixels sont beaucoup plus régulières que celles des m-waterpixels. Cela est du au fait que on force les marqueurs à être centrées dans les cellules, donc le watershed est plus régulier.

Aussi, lorsque l'on regarde les 2 images précédentes, il est difficile de déterminer quel type de cellule conduit à des meilleurs contours.

Nous avons utilisé une nouvelle fois la fonction **findContours** de **cv2,** et avons déterminé, pour une distance de 2 pixels, combien de contours réels de l'image sont à cette distance des contours des waterpixles.

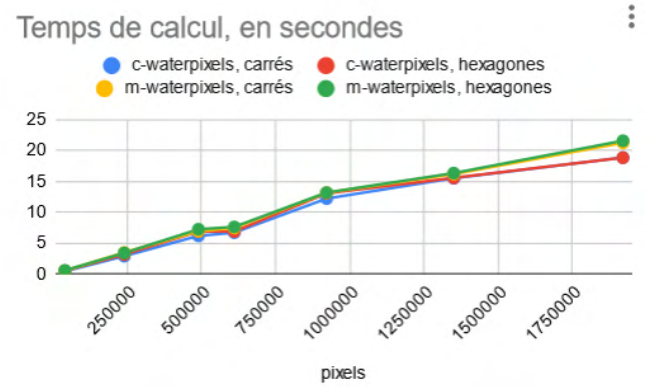


On obtient que 87,19% des contours réels sont approchés par les c-waterpixels à 2 pixels près contre 85,83% pour les m-waterpixels

Globalement les types de waterpixels donnent des résultat convaincant et assez proches (2% environ d'écart). Il est donc difficile à l'aide de ces résultats de trancher sur la supériorité d'une méthode sur l'autre.

d) Comparaison des temps de calcul

On mesure le temps de calcul de plusieurs séries de waterpixels :



Les tests de temps de calcul dont les résultats sont affichés ici ont été effectués pour des images en couleur, avec un espacement par défaut de 50 pixels. En pratique, les temps de calculs peuvent varier un peu selon les essais et les autres processus en cours d'exécution sur l'ordinateur utilisé. Il permet cependant de montrer plusieurs choses.

Tout d'abord, le temps de calcul semble effectivement être linéaire par rapport à la taille de l'image. On observe également que le temps de calcul pour les m-waterpixels est légèrement plus élevé, ce qui était attendu, car pour calculer les marqueurs dans ce cas, il est d'abord nécessaire de calculer la position des centres des cellules (qui sont les marqueurs utilisés pour les c-waterpixels !), avant d'effectuer le calcul de la plus grosse zone de gradient minimum dans chaque cellule. On observe également que les waterpixels carrés prennent très légèrement moins de temps que les waterpixels hexagonaux, ce qui était attendu car le calcul de la norme hexagonale est légèrement plus long que la norme infinie. En pratique, on observe que la majorité du temps de calcul est utilisée pour la régularisation du gradient.

IV Passage en couleur

La transformation watershed ne prend en compte que les images (ou gradients régularisés, dans le cas de notre projet) en niveau de gris. Pour les images en couleur, comme conseillé dans l'article, nous avons converti les images RGB dans l'espace HSV, plus rapproché de la vision humaine, puis moyenné les trois gradients morphologiques des trois coordonnées avant de régulariser le gradient. On note que l'opération de régularisation est linéaire, il est donc moins coûteux de procéder ainsi

V Conclusions

En conclusion dans ce projet nous avons étudié les différentes méthodes d'implémentations des waterpixels.
Nous avons travaillé sur les paramètres d'implémentation pour mettre en relief quelques propriétés de dépendance en ces paramètres (notamment comment obtenir de « bons » water pixels).
Néanmoins nous n'avons pas réussi à conclure quant à la pertinence d'un type de waterpixel sur un autre (m-waterpixels ou c-waterpixels).

Les méthodes et fonctions utilisées pour ce projet sont :

- des fonctions classiques de **num**-y et **pyplot** pour les calculs et affichages
- le gradient morphologique d'o**penCV** : - - **-cv2.getStructuringElement()**, **cv2.morphologyEx()**
- l'analyse en composantes connexes **cv2.connectedComponentsWithStats()**, utilisée dans la version optimisée du code python
- des fonction **d'openCV** pour le passage dans l'espace HSV cv2.**cvtColor**(), la séparation des coordonnées cv2.**split**(), et la renormalisation du gradient **cv2**.**normalize**()