

La segmentation d'image est une opération importante en traitement d'images et en vision par ordinateur qui vise à partitionner une image numérique en un ensemble d'éléments constituant l'image, distincts et significatifs (contours, régions, etc.). Elle permet de simplifier et/ou de changer la représentation d'une image en regroupant les pixels qui partagent des propriétés similaires (couleur, intensité, texture) en zones cohérentes, ou au contraire, en identifiant les frontières (contours) marquant une discontinuité entre ces zones. La binarisation est un cas particulier de segmentation où l'image n'est divisée qu'en deux classes (premier plan et arrière-plan). Le résultat final simplifié facilite la détection, l'analyse et l'interprétation de l'image pour des tâches ultérieures.

L'Objectif de ce TP est de développer en Python quelques techniques de base de segmentation d'image : Binarisation, détection des composantes connexes et segmentation en contours/régions.

Exercice 1 : Binarisation (seuillage) d'image et détection des composantes connexes

La binarisation d'image est une opération qui consiste à convertir une image en niveaux de gris (ou couleurs) en une image binaire, c'est-à-dire une image qui ne contient que deux valeurs de pixels possibles, généralement le noir pour le premier plan et le blanc pour l'arrière plan. Habituellement elle est utilisée pour séparer l'objet d'intérêt de l'arrière-plan de l'image. Aussi qualifiée de seuillage automatique d'image en déterminant un seuil optimal S (global ou local) qui garantit que tous les pixels de l'objet d'intérêt de l'image lui sont inférieurs. Cette séparation en deux classes présente plusieurs avantages essentiels dans le domaine du traitement et de l'analyse d'images (simplification des données, réduction de la taille des fichiers, séparation premier plan/arrière-plan, amélioration de l'efficacité et de la précision des tâches d'analyse d'image, comme la reconnaissance optique de caractères, réduction du bruit, optimisation des systèmes embarqués et/ou à ressources limitées).

On souhaite comprendre la différence fondamentale entre deux approches de binarisation d'image, très utilisées par les systèmes de vision intelligente : Binarisation globale et binarisation locale (adaptive).

Travail à faire :

- 1) Dans un nouveau fichier «*TP3_Exo1.py*», charger l'image «*Images/compteur.jpg*» dans *im1* en NG.
- 2) Utiliser la méthode de seuillage automatique globale d'Otsu d'OpenCV pour binariser *im1*, Stocker le résultat dans *im2* :
 - `otsu_threshold, im2 = cv2.threshold(im1, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU).`
 - Implémenter à la maison votre propre algorithme d'Otsu en utilisant les formules vues dans le cours.
- 3) Implémenter en Python votre propre fonction de seuillage local qui applique l'algorithme de Sauvola.
 - Nom de la fonction ***binSauvola(im, k=0.04, R=128, tailleFen=15)***,
 - Formule de seuillage de Sauvola :

$$T(x, y) = \mu(x, y) \cdot \left(1 + k \cdot \left(\frac{\sigma(x, y)}{R} - 1\right)\right)$$

k est un paramètre fixé à 0.04 et R est la dynamique de l'écart type, fixé à 128.

- Appliquer cette fonction sur *im1* et stocker le résultat sur *im3*, comparer ce résultat à *im2* issue de seuillage global, indiquer vos remarques sur la qualité de seuillage de chaque image.

- Analyser l'impact des paramètres k et de la fenêtre d'analyse ($k=0.1, 0.2, 0.5, 0.8, 1$ et $\text{tailleFen}=3, 5, 7, 9, 25$) sur le résultat de la méthode de Sauvola.
- 4) Détecter à l'aide d'OpenCV les composantes connexes et les rectangles englobants de chaque composante connexe. Que remarquer-vous ?

Exercice 2 : Segmentation d'image en contours

L'objectif de ce TP est de comparer les performances des détecteurs de contours classiques (Roberts, Prewitt, Sobel, Canny, Deriche) sur différentes images. Pour cela, créer un nouveau script Python sous le nom : « TP3_Exo2.py », importer cv2, numpy as np, charger une image en niveaux de gris (par exemple, "camera-man.jpg").

- 1) Au départ, on souhaite implémenter un détecteur de contours en trois étapes: calcul du gradient d'intensité (via les opérateurs Sobel, Roberts et Prewitt ci-dessous : ces opérateurs sont des détecteurs de première génération qui calculent une approximation discrète du gradient d'intensité de l'image), affinage des contours (via la suppression des non-maximum locaux) et seuillage par hystérésis pour avoir des contours binaires.

Gx	Gy	Gx	Gy	Gx	Gy
-1	0	1	1	2	1
-2	0	2	0	0	0
-1	0	1	-1	-2	-1
Sobel Mask					

Gx	Gy	Gx	Gy
1	0	0	1
0	-1	-1	0
Robert Mask			

Gx	Gy	Gx	Gy
-1	0	1	1
-1	0	1	0
-1	0	1	-1
Prewitt Mask			

- a) Appliquer les deux masques de Sobel (horizontal Gx et vertical Gy) par convolution pour obtenir les images de gradient correspondantes.
 - Calculer l'amplitude (norme) du gradient pour chaque pixel $G(x, y) = \sqrt{G_x^2 + G_y^2}$
 - Calculer l'orientation du gradient pour chaque pixel : $\theta(x, y) = \text{atan}(G_y / G_x)$
 - Visualiser les deux images : l'image d'amplitude de gradient (souvent bruitée et avec des contours épais) et l'image d'orientation de gradient.
- b) Suppression des non-maximum locaux (NMS), Cette étape affine les contours en ne conservant que les pixels qui sont des maximums locaux de l'amplitude du gradient dans la direction du gradient.
 - Quantifier l'orientation : Les orientations continues θ doivent être discrétisées en 4 directions ($0^\circ, 45^\circ, 90^\circ, 135^\circ$), correspondant aux directions des pixels voisins (horizontal, vertical, diagonales).
 - Parcourir l'image d'amplitude : Pour chaque pixel (x,y), examiner ses deux voisins dans la direction du gradient quantifiée : Si l'amplitude $G(x,y)$ est inférieure à l'amplitude de l'un de ses deux voisins, mettre son amplitude à zéro (suppression). Sinon, conserver son amplitude.
 - Afficher la nouvelle carte des gradients affinés (une image avec des contours d'une épaisseur d'un seul pixel).
- c) Seuillage par hystérésis : Appliquer deux seuils ($T_{\text{haut}}=70, T_{\text{bas}}=30$) pour classer les pixels en "contours forts", "contours faibles" et "fond". Conserver les contours faibles uniquement s'ils sont connectés à un contour fort dans un voisinage 3x3. Cette étape permettra de relier les segments de contour. Afficher le résultat.
- d) Mettre ce code dans une fonction personnelle « `sobel_edge_detection(im, sb=30, sh=70)` ».
- e) De même principe, créer les deux fonctions « `robert_edge_detection(im, sb=30, sh=70)` » et « `prewitt_edge_detection(im, sb=30, sh=70)` » permettant d'appliquer les opérateurs de Robert et Prewitt.
- f) Appliquer ces trois opérateurs sur plusieurs images, discuter de l'impact du bruit sur ces opérateurs et pourquoi un lissage (par exemple, un filtre gaussien) est souvent appliqué en comme prétraitement pour garantir une détection plus robustes au bruit.

- 2) L'algorithme de Canny est souvent considéré comme optimal car il intègre plusieurs étapes cruciales pour une détection de haute qualité. OpenCV fournit une fonction intégrée pour cela.
 - a) Utilisez la fonction `cv2.Canny()` sur l'image en niveaux de gris. Cette fonction prend en paramètres les seuils bas et haut pour le seuillage par hystérésis. Elle gère tout en interne, le lissage Gaussien (intégré), le calcul du gradient et de l'orientation (similaire à sobel), la suppression des non-maxima et le seuillage par hystérésis.
 - b) Expérimentation : Faites varier les seuils de `cv2.Canny()` et observez comment la connectivité et la quantité de contours changent.
- 3) Devoir à faire à la maison : Le filtre de Deriche (et Shen-Castan) est un filtre récursif optimal, souvent utilisé pour sa rapidité et sa bonne localisation des contours. Il n'existe pas de fonction intégrée dans OpenCV pour Deriche, il faudra donc utiliser une implémentation externe. Faites une brève recherche sur ce filtre et ses différences clés avec Canny (principalement l'utilisation de filtres récursifs). Comparez ses résultats par rapport à Canny en termes qualité de la segmentation et de rapidité de calcul.

Exercice 3 : Segmentation en régions (Croissance de régions)

Cet exercice permet de comprendre les concepts fondamentaux de la segmentation par région, qui n'est pas une fonction native d'OpenCV. L'algorithme de croissance de régions permet de segmenter des objets spécifiques dans une image. Cet algorithme est itératif et regroupe les pixels voisins en fonction d'un critère (prédicat) de similarité (homogénéité). Il nécessite de partir d'un point de départ (graine ou seed) et d'agréger les pixels voisins dont l'intensité est similaire.

Travail à faire :

- 1) Charger l'image en niveaux de gris « Cerveau_AVC1.JPG » dans `im1`. Appliquer un léger lissage gaussien « (`imb = cv2.GaussianBlur(im1, (5, 5), 4)`) » pour réduire le bruit en améliorant potentiellement la cohérence des régions. Stocker le résultat dans `imb`.
- 2) Définition du critère d'homogénéité : le plus simple est d'utiliser une différence de niveau de gris absolue entre un pixel voisin et la moyenne (ou l'intensité du premier pixel) de la région en cours de croissance. Définir un seuil de tolérance (*seuil*).
- 3) Créer la fonction « `region_growing(imb, seed_point = (260,140), seuil=15)` » qui : Initialise une image de sortie binaire (masque) de la même taille que l'image d'entrée. Utiliser une structure de données (en Python une liste ou deque de module standard `collections` suffira pour simuler une file d'attente FIFO ou une pile LIFO) pour stocker les pixels voisins à vérifier. Marquer le point de départ (seed) dans le masque et l'ajouter à la structure de données. Tant que la structure de données n'est pas vide, extrait un pixel de la structure. Vérifie ses 8 voisins (en s'assurant qu'ils restent dans les limites de l'image). Si un voisin n'a pas encore été traité et respecte le critère d'homogénéité (sa valeur de pixel est proche de la valeur initiale du seed ou de la moyenne actuelle de la région, selon la complexité souhaitée), marquez-le dans le masque et ajoutez-le à la structure de données. Tester cette fonction sur l'image de cerveau lissée précédemment. Afficher le résultat.
- 4) Créer une seconde fonction permettant de détecter automatiquement toutes les régions dans l'image : « `sequential_region_growing(imb, 15)` ». Le niveau de gris de chaque région doit être la moyenne des niveaux de gris de ses pixels. Tester cette fonction sur l'image de cerveau lissée `imb`.

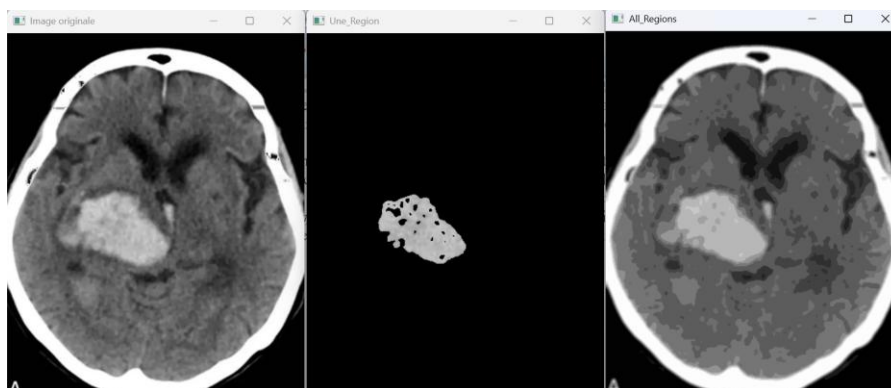


Figure 1. Résultats attendus