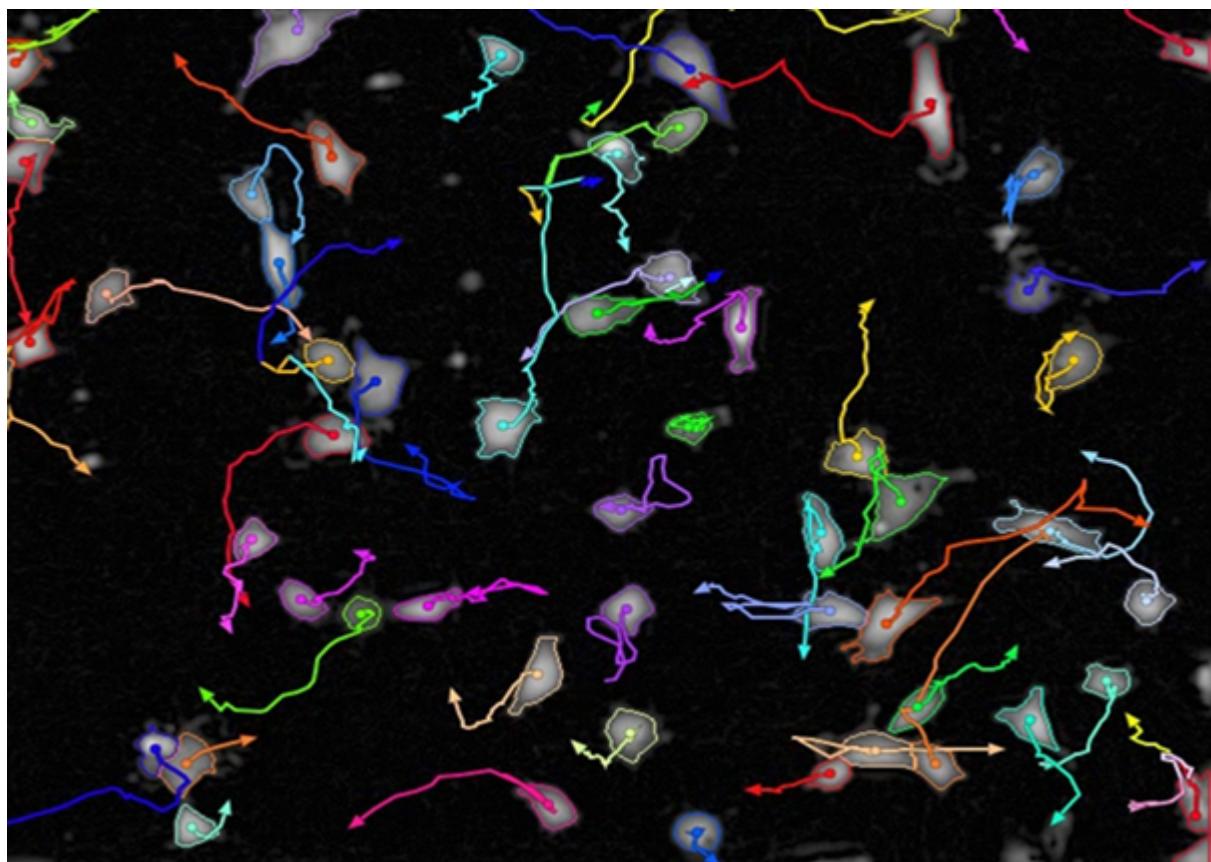


Retracer l'évolution de cellules dans le temps à partir d'images de microscopie de Live Cell Imaging



Auteurs :

Vivien Chambe
Ilaria Nissotti-Revel
Maïssane Bouchema
Judicaël Marion
Lilou Bouvier

Encadrants :

MoreHisto

Noemi Roggero <noemi.roggero@morehisto.ai>
Ludovic Bertsch <ludovic.bertsch@morehisto.ai>

Table des matières :

I - Rappel du sujet	p.3
II - Partie 1 : Segmentation et labellisation	p.4
III - Différentes méthodes segmentation	p.7
IV - Partie 2 : Assignation linéaire et tracking	p.9
V - Tests	p.12
VI - Résultats obtenus	p.15
VII - Expérience avec l'entreprise	p.16
VIII - Gestion du travail en groupe	p.17
IX - Annexe	p.18

I - Rappel du sujet :

Le but de ce projet est d'implémenter un code pouvant, à partir d'images de microscopie de Live Cell Imaging, retracer l'évolution des cellules dans le temps. On dispose d'images en 2D + T, c'est-à-dire des images en 2D prises au cours du temps, il s'agirait donc de trouver la correspondance de chaque cellule d'une image à l'autre et de tracer leurs trajectoires.

Visuellement, l'idée serait un code prenant en entrée toutes les images d'un puits et renvoyant les trajectoires de chaque cellule superposées à la toute première image. En plus de ce résultat visuel, on souhaiterait récupérer, par exemple dans un fichier excel, toutes les positions successives des cellules dans le temps et d'autres informations (nombre de cellules, aires etc.).

Pour mener à bien ce projet, nous avons décidé d'utiliser Python comme langage de programmation ainsi que des bibliothèques utiles dans de nombreuses applications telles que le traitement d'image, la mise en œuvre d'interfaces graphiques ou bien tout simplement les opérations mathématiques en général. Nous avons donc utilisé OpenCV, Numpy, Scipy et PyQt5. OpenCV sera utilisée principalement pour le traitement d'image, c'est-à-dire appliquer les filtres nécessaires sur les images afin de permettre la détection de nos cellules ; Numpy et Scipy, seront utilisées principalement pour les opérations mathématiques et finalement, PyQt5 sera utilisée pour tout ce qui concerne l'interface graphique.

Le projet était tout d'abord divisé en trois grandes parties. La partie segmentation qui consistait en la détection des cellules dans chaque image ainsi que leur labellisation, la partie assignation linéaire c'est-à-dire l'association des cellules d'une image à l'autre et le tracking des cellules basée sur la méthode d'assignation linéaire que nous verrons plus précisément dans la suite. Lors de la première semaine du projet (dédiée à la mise en place de ce dernier), nous avions planifié de faire la partie segmentation et assignation linéaire au cours de la première semaine d'avril et ensuite s'occuper du reste les semaines qui suivent, ce que nous n'avons finalement pas pu tenir. Nous avons donc fini par décider de les faire en même temps, lors de la deuxième semaine. Le projet sera donc finalement divisé en deux grandes parties soit la partie segmentation et la partie tracking et assignation linéaire.

II - Partie 1 : Segmentation et labellisation :

Dans notre planning de décembre, nous avions prévu de finir cette partie dans la première semaine ce qui a bien été fait à temps.

Tout d'abord, en observant les images, on remarque que les images des puits 1 et 2 sont grisâtres et que l'on distingue à peine les cellules. Pour régler ce problème, on normalise l'image avec la fonction normalize proposée par OpenCv ainsi que les paramètres 0, 255 et NORM_MINMAX. Ce dernier est proposé par OpenCv et permet de modifier la plage des valeurs d'intensité de l'image entre 0 et 255. On passe alors d'une image quasi illisible à une image où l'on peut déjà distinguer les cellules à l'œil nu (cf. Figure 1).

Exemple avec l'image “t000.tif” du puits 2:

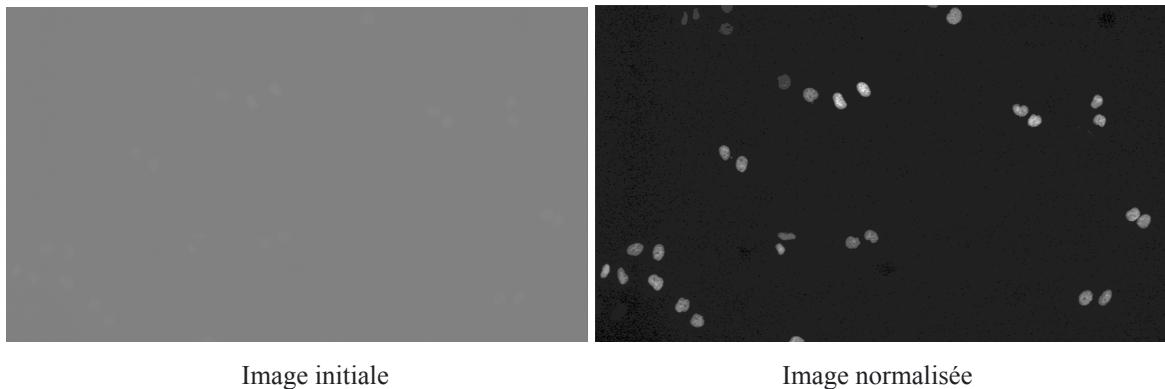


Figure 1: Normalisation de l'image puit02/t000.tif

La première étape consiste à « nettoyer » les images c'est-à-dire enlever le bruit qu'elles contiennent et donc faire en sorte de ne pouvoir détecter seulement les cellules. Cette opération relevant du traitement d'image, nous avons principalement utilisé les fonctions disponibles sur OpenCV. Premièrement, nous avons utilisé la fonction GaussianBlur. Cette dernière lisse l'image à savoir élimine le « bruit » (cf. Figure 2). Elle présente un paramètre important, la taille du noyau (« kernel size » en anglais), c'est ce qui lisse l'image (plus la taille du noyau est grande plus l'image est lissée). La taille optimale du noyau dépendant de l'image, nous avons décidé de créer une interface graphique afin que l'utilisateur puisse choisir la valeur puisqu'il est plus simple de le voir visuellement.

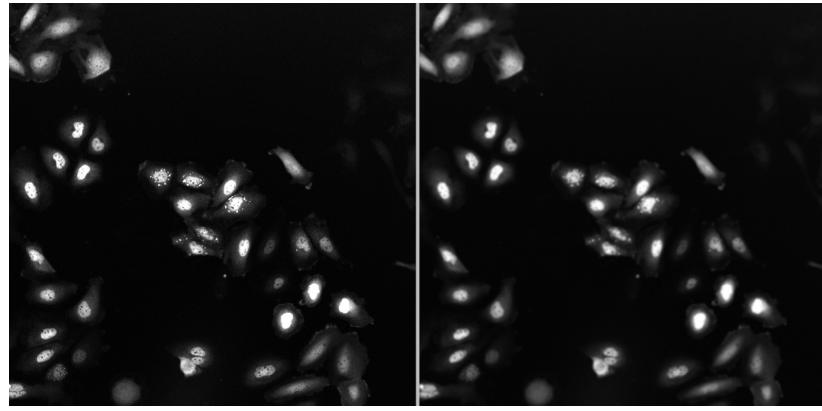


Figure 2: Application du GaussianBlur de l'image puit03/t000.tif

Ensuite, on utilise la fonction `threshold` afin de créer le masque binaire de notre image. Dans cette fonction, il y a plusieurs paramètres qui permettent de trouver un masque correct par rapport à l'image originale. Les deux plus importants sont deux entiers que l'on appellera seuil et val dans la suite. Le principe de `threshold` est de parcourir chaque pixel de l'image, si la valeur du pixel est supérieur à seuil, alors le pixel prend la valeur val. Notre but ici étant de créer un masque binaire, nous prendrons une valeur de 255 pour le second argument (val). Ainsi, toutes nos cellules apparaîtront blanches et l'arrière-plan noir (Figure 3). En ce qui concerne le choix de la valeur du seuil, on a utilisé le paramètre `THRESH_OTSU` proposé dans OpenCV qui indique à la fonction de déterminer une valeur de seuil global optimale à partir de l'histogramme de l'image. Ce dernier est utile seulement quand l'image a seulement deux valeurs (image bimodale), ou l'histogramme ne serait composé que de deux pixels. La valeur du seuil peut aussi être choisie arbitrairement. L'algorithme trouve alors la valeur du seuil optimale et le renvoie.

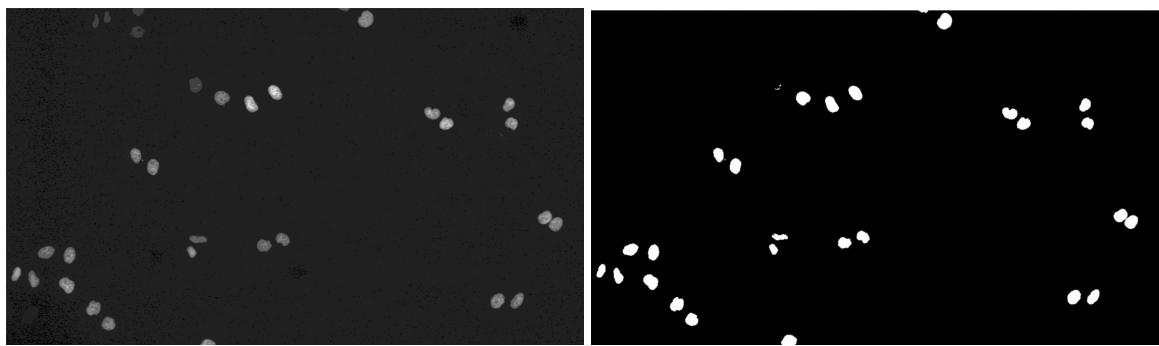


Figure 3: Application du threshold sur une image

OpenCV propose aussi une autre fonction, `adaptive threshold` (se traduisant par « seuil adaptatif » en français), que nous avions commencé à utiliser avant de passer à la fonction du `threshold` normal. La principale différence entre ces deux fonctions est que pour `threshold` normal la valeur du seuil est globale, c'est-à-dire qu'elle est la même pour tous les pixels de l'image tandis que pour `adaptive threshold` (le seuillage adaptatif) la valeur du seuil est

calculée pour des régions plus petites et, par conséquent, il y aura différentes valeurs de seuils. On peut, en quelque sorte, parler de valeur de seuil local. N'ayant constaté aucune différence significative entre les résultats des deux fonctions, nous avons finalement décidé d'utiliser le threshold simple.

D'autres fonctions, telles que la fonction d'erosion, dilation, closing et opening s'avèrent très intéressantes dans la création du masque le plus propre possible. La fonction erosion supprime tous les pixels aux limites des blobs (la partie extérieure des objets) en fonction de la taille du noyau. La fonction dilation fait l'opposé, elle ajoute des pixels aux limites de chaque objet. Opening est considéré comme l'érosion suivie par la dilatation, et donc est utile pour éliminer les détections qui relèvent du bruit. Finalement, closing est l'inverse de opening, c'est une dilatation suivie d'une érosion et elle est utile pour fermer des petits trous dans les objets. Comme pour le GaussianBlur, il n'y a pas toujours la même taille du noyau ou les mêmes valeurs pour chaque image. Nous avons donc, ici aussi, décidé de laisser l'utilisateur choisir les bonnes valeurs pour les paramètres de ces fonctions. Le point positif de cette méthode est que pour chaque groupe d'images ces valeurs restent plus ou moins les mêmes ce qui nous permet d'utiliser ces mêmes valeurs choisies par l'utilisateur pour le reste des images des mêmes cellules. L'utilisateur (Ici MoreHisto) ne devra donc pas choisir les meilleures valeurs pour toutes les images mais pour un puits seulement. Par ailleurs, ce principe s'avère être le même utilisé par l'entreprise MoreHisto.

Après avoir créé une image masque, on localise les cellules. Pour cela, nous utilisons la fonction proposée par OpenCv, appelée connectedComponentsWithStats (« composantes connectées avec Statistiques»). Il existe une fonction assez similaire (connectedComponents) mais cette dernière ne présente pas les mêmes valeurs de retour que nous estimons très utiles pour la partie de l'assignation linéaire. Ces dernières sont les suivantes:

nblabels : le nombre total de label c'est-à-dire le nombre d'identifiants soit au final le nombre total des cellules segmentées

labels : les valeurs des labels, donc les différentes valeurs des identifiants attribuées à chaque cellule, labels est une matrice de même taille que l'image et de valeur -1 aux bords, 0 pour le fond et un numéro pour chaque pixel de chaque cellule (distinct d'une cellule à l'autre bien évidemment).

stats : contient les « statistiques » des cellules segmentées et de leurs cadres englobant (« bounding box » en anglais) tels que (dans l'ordre) le pixel le plus haut de la cellule, le pixel le plus bas, la longueur du cadre englobant, sa largeur et finalement l'aire totale de la cellule.

centroids : renvoie un tableau contenant les coordonnées du centre de chaque cellule.

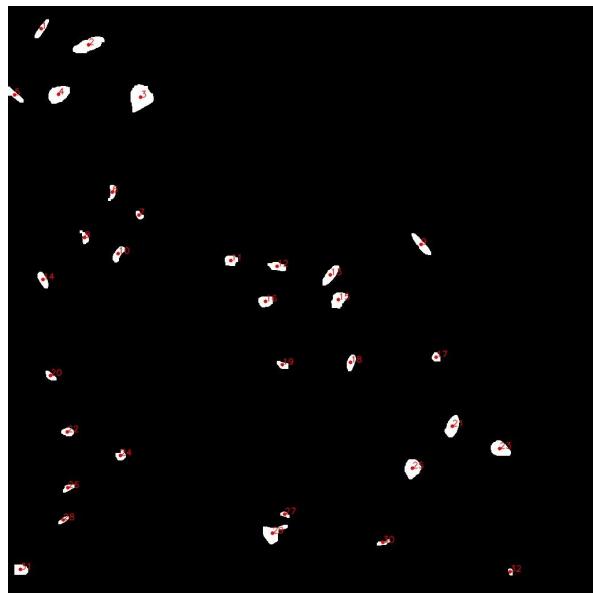


Image binarisé + labels

Figure 4: Affichage des labels de chaque cellules de l'image puit03/t000.tif

La segmentation nous a pris bien plus de temps que prévu, en effet, nous avons sous-estimé le temps de compréhension des fonctions et de leur utilisation. De plus, la mise en commun n'a pas été aussi simple que prévue. Mais finalement, cela a bien fonctionné car nous avons pu terminer une partie dans la première semaine, ce qui nous a permis de nous concentrer sur le reste à partir de la deuxième.

III - Explications différentes méthodes segmentation

Nous avons ensuite essayé d'améliorer la segmentation des cellules pour traiter les cas de cellules qui se touchent et qui se croisent. Pour cela nous avons essayé beaucoup de méthodes mais pour l'instant aucune n'a abouti.

Nous avons tout d'abord commencé par une fonction `detection_cents(img)`. Dans cette dernière, on commence par éroder l'image et récupérer ses propriétés avec `cv.connectedComponentsWithStats`. On regarde si le nombre de labels a changé si oui cela devrait signifier qu'il y a sur l'image non érodée deux cellules qui se touchent mais qui sont détectées comme une seule. Dans ce cas là, on parcourt la matrice `labels` correspondant à l'image non érodée et la matrice `labels` correspondant à la matrice érodée. Si le label est identique et n'a pas encore été rencontré, le nombre de centroïdes augmente de 1 et on ajoute ce label dans la liste des labels déjà vus. On remplace alors la liste de labels de l'image non érodée par celle érodée. Nous disposons alors d'une fonction permettant d'avoir le bon nombre de cellules de l'image mais nous n'avons pas réussi à l'utiliser pour modifier le premier labels afin qu'il correspondent au réel nombre de cellules sans supprimer les cellules qui auraient pu être supprimées avec l'érosion. De plus, le fait que le nombre de cellules soit identique avant et après l'érosion ne veut pas dire que ce sont les mêmes. En effet, il est possible qu'il y ait autant de cellules qui apparaissent et disparaissent, cela conserve le

nombre de cellules et donc vérifier l'égalité du nombre de labels n'est pas une bonne méthode.

Nous avons alors créé une fonction `detection_newcents(img,nb_it)` que l'on a implémenté trois fois.

Dans la première version, on cherche les centroïdes ayant même label dans l'image érodée pour ensuite pouvoir modifier les pixels correspondants dans l'image non érodée. On récupère les labels de l'image non érodée et de l'image érodée après `nb_it` itérations et les listes de labels correspondantes. Ensuite, on crée une liste de label déjà observée et une liste des "nouveaux" centroïdes. Pour chaque centroïde de l'image érodée si ce dernier correspond à un label déjà vu alors c'est un nouveau centroïde. Cela veut dire qu'en parcourant les pixels de même coordonnées que les centroïdes s'il y a deux centroïdes pour le même label alors l'image présente deux cellules qui se touchent. On obtient alors une liste contenant, supposément, tous les nouveaux centroïdes. Mais nous ne sommes pas parvenus à associer ces centroïdes avec ceux de l'image non érodée et donc n'avons pas réussi à bien labelliser nos cellules.

Dans la deuxième version, on commence également par éroder `nb_it` fois l'image et récupérer ses propriétés. On parcourt simultanément les matrices de labels de l'image érodée et de l'image non érodée. Si le pixel visité est de label 0 (valeur du fond) dans la matrice érodée alors il reste 0 dans celle non érodée et de même si le label est -1 (valeurs aux bords) dans la matrice érodée il reste inchangé dans la matrice non érodée. En ce qui concerne les zones où l'on détecte deux cellules qui se touchent, on n'utilise bien évidemment pas la même logique. Le principe est de calculer le milieu des deux nouveaux centroïdes détectés et de modifier la valeur du label au-delà de ce dernier. Visuellement parlant, cela donnerait la "cellule initiale" coupée en deux parties constituant les deux nouvelles cellules à présent distinguées par leurs nouveaux labels et centroïdes. Cette version n'a pas abouti non plus, en effet, les boucles étant bien trop longues et complexes (notamment dans la recherche des zones de même label) nous n'avons ni pu les tester ni réellement vérifier si ces dernières renvoient le bon résultat.

Pour l'avant dernière version, nous utilisons globalement le même principe que précédemment. En revanche, on décide de parcourir les zones de pixels de même labels à l'aide de la fonction `np.where` proposée dans Numpy qui renvoie les coordonnées des éléments respectant une condition passée en paramètre. Ici, on parcourrait avec un label actuel (`label_act`) les valeurs de labels et pour chaque valeur prise par `label_act` on appellerait `np.where(labels == label_act)` qui nous renverrait donc les coordonnées de chaque pixels de de label égale à `label_act`. Ensuite, même principe qu'au-dessus on cherche dans cette zone les centroids et si on en trouve deux on applique les modifications nécessaires. Cette version n'a pas non plus abouti pour exactement les mêmes raisons que précédemment. On peut en déduire que modifier seulement la boucle de recherche des zones de même label n'était pas suffisant.

Nous avons finalement essayé une dernière méthode consistant en la modification du label de l'image d'origine et la modification des pixels nécessaires afin de séparer les cellules. Pour cela, nous avons implémenté une fonction renvoyant les coordonnées de deux points appartenant à la perpendiculaire au milieu du segment formé par les deux centroïdes trouvés. Ensuite, il s'agissait de tracer le segment entre ces deux points en noir afin de séparer nos cellules. Cette version n'a finalement pas non plus abouti, en effet, en suivant les tracers de nos segments sur plusieurs images, nous avons pu remarquer un décalage de ce dernier par rapport aux cellules. Nous n'avons malheureusement pas eu le temps de déterminer d'où exactement venait le problème ni de le régler.

Après avoir essayé de traiter le cas des cellules qui se touchent on pourrait se demander comment traiter celui des cellules qui se croisent. Or, la seule différence entre ces deux cas est l'instant t où elles se "superposent" qui pose problème mais si on suppose qu'elles sont en mouvement constant, alors il existe un temps $t+i$ où les deux cellules se touchent seulement (juste avant de se séparer) et là notre méthode sera bien applicable mais on trackera la mauvaise cellule.

IV - Partie 2 : Assiguation linéaire et tracking :

Pour assigner les cellules d'une image à une autre nous allons devoir résoudre un problème d'assiguation linéaire aussi appelé LAP.

Description du LAP :

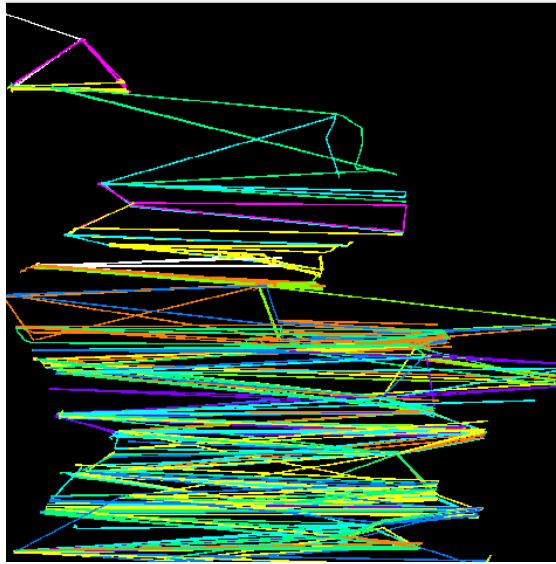
Le LAP est un problème d'optimisation combinatoire classique qui consiste à attribuer un certain nombre d'objets à un certain nombre de destinations, avec des coûts associés à chaque attribution.

Le but est de trouver l'affectation des objets aux destinations qui minimise le coût total de l'ensemble des affectations. Le problème peut être formulé mathématiquement sous forme de programme linéaire, où les variables de décision représentent les affectations possibles.

Le problème d'assiguation linéaire trouve des applications dans de nombreux domaines, tels que la planification de la production, l'affectation de personnel, la logistique, l'ordonnancement, etc.

Le problème d'assiguation linéaire peut être résolu efficacement en utilisant l'algorithme hongrois, également connu sous le nom de méthode de Munkres. Cet algorithme utilise des techniques de programmation dynamique pour trouver la solution optimale en temps polynomial, ce qui le rend très efficace même pour des problèmes avec un grand nombre d'objets et de destinations.

Nous avons donc utilisé dans une première version la fonction `linear_sum_assignment` de Scipy pour résoudre ce problème et nous obtenions une assiguation non conforme à nos attentes car nous ne pouvions pas tenir compte des cellules qui disparaissent et apparaissent.



Trajectoires avec méthode Hongroise

Figure 5

Cela venant du fait que cette méthode est efficace uniquement d'une image A vers B mais pas pour les enchaîner. Et n'ayant pas accès au code de Scipy nous n'avions aucun contrôle sur les paramètres ou l'implémentation.

Nous avons donc décidé de partir complètement sur autre chose en créant notre propre fonction.

Cette fonction repose sur l'utilisation d'une fonction de coût. On parcourt l'intégralité des cellules à l'instant t , et on donne une note de sa liaison à une cellule de l'image à $t+1$, on ne garde que la cellule qui la liaison la mieux notée, celle ci étant censé être celle qui est la plus apte à être notre cellule à $t+1$.

On a tout d'abord pensé à une note entre 0 et 1 où plus la note tend vers 1 plus l'association des deux cellules est bonne. Puis on est finalement partis vers une note strictement positive tendant vers $+\infty$, car cette idée semblait plus faisable. Sachant que l'on veut minimiser la différence entre deux cellules, en distance et en surface, à des temps différents, il nous faut une fonction qui, quand on lui donne des paramètres tendant vers 0, tend vers l'infini. Nous sommes donc partis vers une somme de fonction inverse pour notre calcul de coût car celui-ci correspond parfaitement à ce dont on avait besoin.

$$costv1(d, s) = \frac{1.5}{d} + \frac{1}{s} \mid costv2(d, s) = \frac{2}{d} \text{ si } s < 10, \frac{1}{d} \text{ sinon}$$

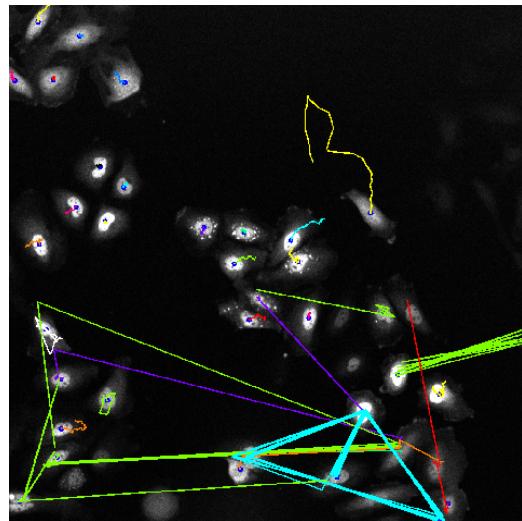
avec d : la distance entre les centroïdes et s : la différence entre les deux surfaces des cellules

Ces modèles sont naïfs et imprécis, et il existe probablement des cas dans lesquels la note attribuée ne donnera pas ce que nous attendons, une cellule se déplaçant vite vers une cellule qui fait pile la même surface, elle sera très probablement assignée à cette dite cellule au lieu d'être assignée à celle prévue.

Nous avons aussi ajouté la gestion d'un rayon d'intérêt, réglable à souhait par l'utilisateur en fonction du puits choisi, celui-ci permettant de gagner en efficacité en sautant les étapes de comparaisons avec des cellules trop distantes.

Auparavant, la fonction renvoyait les liens entre chaque image à l'instant t , autrement dit on renvoyait une liste de doublet représentant quelle cellule allait avec laquelle, le souci c'est que c'était assez dur à utiliser. On renvoie toujours cette liste mais elle n'est plus vraiment utile au bon fonctionnement de notre programme principal.

Le problème que nous avons rencontré était que nous ne mettions pas à jour les IDs des cellules, nous comparons donc les bonnes cellules mais nous assignons les mauvaises.



Trajectoires sans mise à jour des IDs

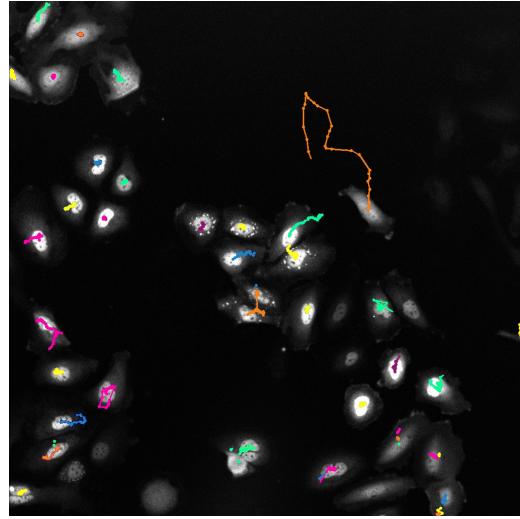
Figure 6

Le nouveau fonctionnement change donc les IDs des cellules en fonction de l'ID de la première cellule historiquement traquée, rendant l'assignation plus simple dans le main, mais forçant du coup à créer un nouvel ID à chaque nouvelle cellule traquée.

Nous avons ainsi pu récupérer les bons IDs et ainsi les assigner à un dictionnaire où chaque clé est un ID et sa valeur est une liste de cellules qui sont les cellules successives dans les différentes images.

Nous avons ensuite pu réaliser des tests en faisant varier le poids des différents paramètres et du rayon de recherche de la fonction.

Pour le puits 3 nous avons trouvé qu'un rayon de 50 permettait un résultat plus que satisfaisant (Figure 7)



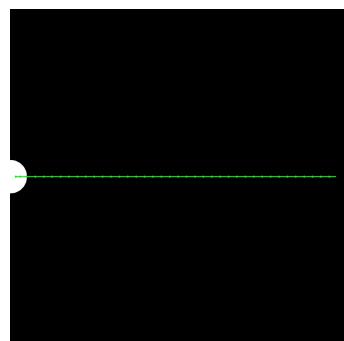
Trajectoires finales avec mise à jour des IDs
Figure 7

Après quelques réflexions, nous nous sommes rendus compte que de nombreux paramètres manquaient pour avoir une fonction de coût qui nous permette de différencier à tous les coups les cellules. Tout d'abord, le programme n'a aucune idée de la vitesse à laquelle les images ont été enregistrées, et celui-ci est pourtant primordial à la différenciation des cellules. En effet, une petite différence avec une fréquence d'images haute est beaucoup plus significative qu'une à une fréquence d'images basse. Il devient alors difficile de créer une fonction de coût cohérente avec le set d'images qu'on nous propose car dans un puits, les constantes choisies arbitrairement peuvent marcher de la meilleure des façons, et ensuite donner des résultats très éloignés de ceux attendus dans une autre série d'images.

V - Tests :

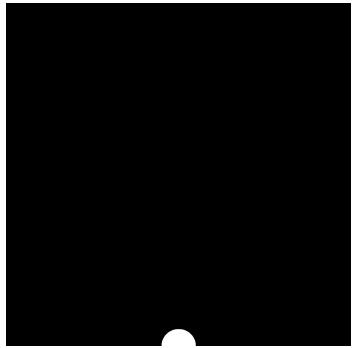
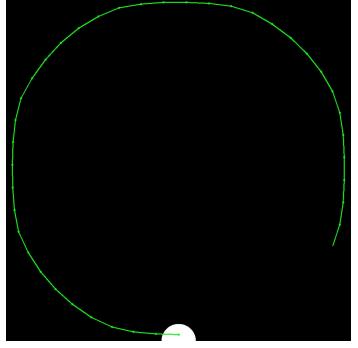
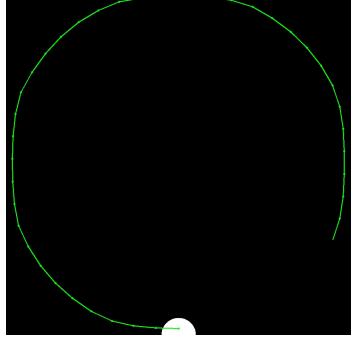
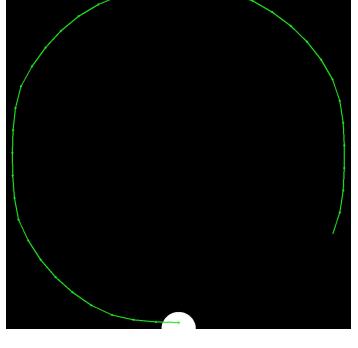
Pour ce genre de problèmes il est compliqué d'avoir des tests automatisés pour savoir si nos résultats sont les bons, le nombre de cellules détectées correctement ou encore si tout a bien été tracké.

Nous avons donc créé un puits de test de trente images, à partir d'un script python, qui simule le déplacement d'une cellule en ligne droite.



Test sur une trajectoire rectiligne
Figure 8

Dans un deuxième temps nous avons fait un parcours en cercle en modifiant les valeurs du rayon.

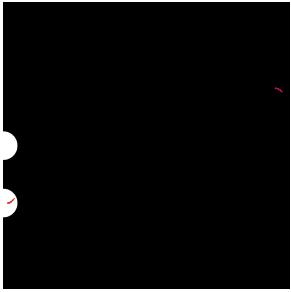
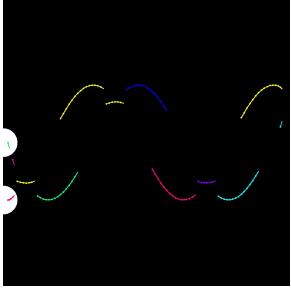
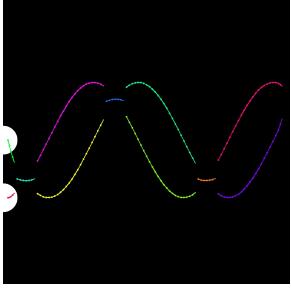
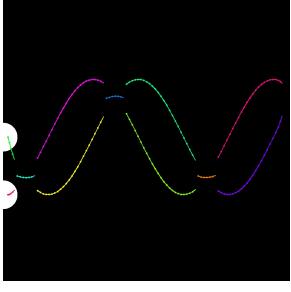
<u>Rayon</u>	<u>Résultat</u>
60	
70	
80	
90	

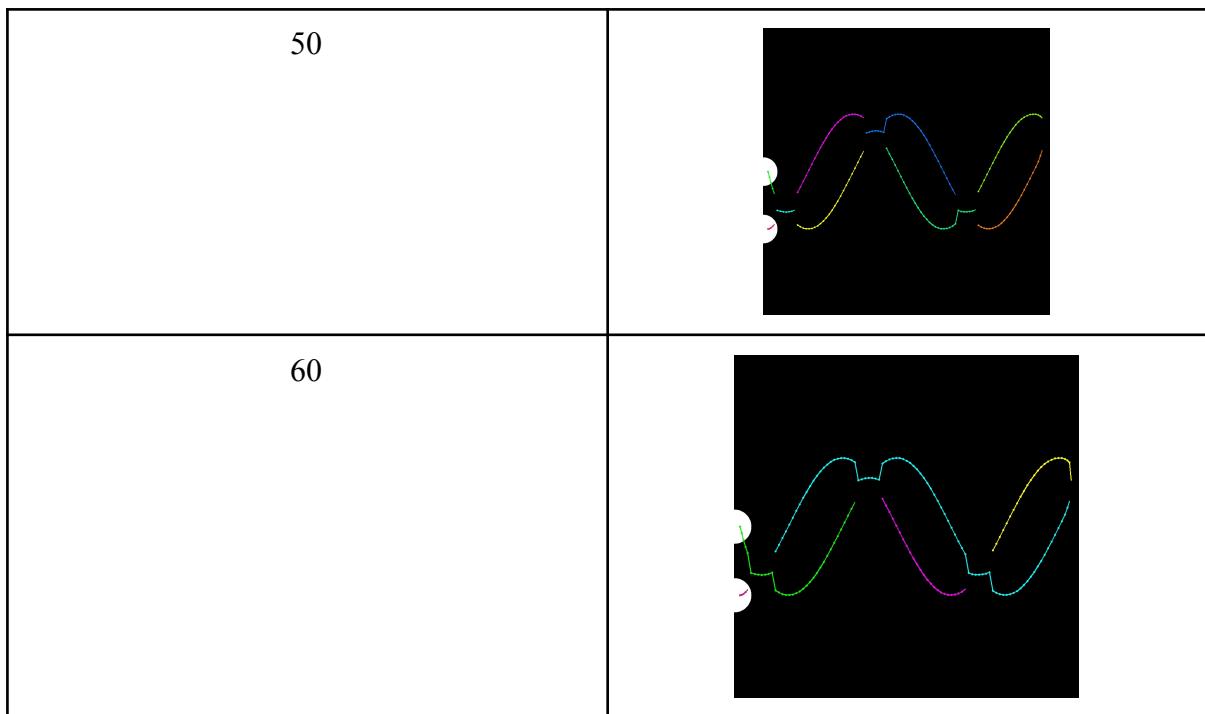
Trajectoire (en vert) d'une cellule se déplaçant en cercle pour un rayon de recherche allant de 60 à 90

Table 9

On peut observer que pour un rayon supérieur à 70 le résultat ne change pas même à 1000 (soit la taille de l'image) car il n'y a qu'une seule cellule.

Pour le dernier test nous avons mis 2 cellules qui suivent deux sinusoïdes.

<u>Rayon</u>	<u>Résultats</u>
10	
20	
30	
40	



Trajectoires de deux cellules suivant un cosinus et un sinus pour un rayon de recherche allant de 10 à 60
Table 10

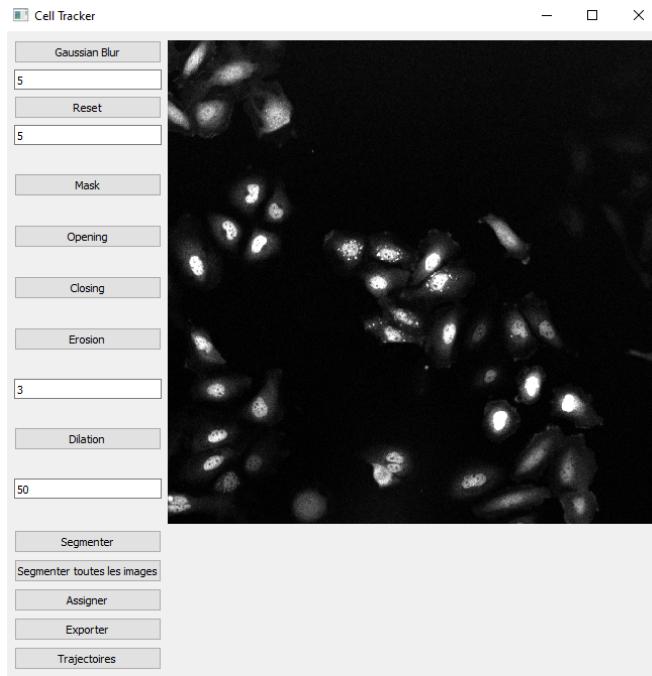
On peut observer les cas limites de notre implémentation ici lorsque les cellules se croisent, nous détectons à chaque fois de nouvelles cellules.

VI - Résultats obtenus :

Afin de pouvoir utiliser toutes les fonctionnalités implémentées plus simplement nous avons réalisé une interface graphique à l'aide de la librairies PyQt, cela nous permet de régler les paramètres de segmentation ainsi que de tracking nous pouvons aussi réaliser le tracking et l'exportation des données de cette façon.

Pour plus d'efficacité il est également possible de réaliser le traitement d'un jeu de données en "headless" c'est-à-dire sans interface graphique. C'est cette partie qui est utilisée pour les tests en faisant varier le rayon entre plusieurs valeurs et ainsi obtenir les tracés de la partie Tests.

Lors du tracé des trajectoires nous enregistrons également la totalité du traitement sous forme de GIF afin de pouvoir vérifier le parcours des cellules et leur précision. Les résultats des segmentations successives sont enregistrées dans le dossier "thresholds" et le GIF des trajectoires dans le dossier "trajectoires".



Interface graphique
Figure 11

VII - Expérience avec l'entreprise :

Tout au long de ce projet, nous avons eu des visioconférences avec nos encadrants de l'entreprise. Pendant la semaine de préparation en décembre nous avions deux encadrants, mais pendant le mois d'avril nous avions une nouvelle encadrante. Nos encadrants travaillent déjà avec OpenCV et Python, ils ont donc vraiment pu nous aider à trouver des idées par rapport aux problèmes rencontrés et nous donner des pistes. Ils ont pu nous dire exactement vers quoi nous diriger et les attendus de ce projet afin notamment de ne pas perdre trop de temps sur la segmentation.

Pendant le mois d'avril nous avons eu deux visioconférences dans la première semaine et deux dans la deuxième avec une conférence en présentiel à l'entreprise. C'était très intéressant d'aller visiter l'entreprise et de voir leur lieu de travail. Pendant la conférence à l'entreprise on a pu bien montrer le travail réalisé et d'expliquer les problèmes que nous avons rencontré. La discussion à propos de ces problèmes nous a permis de trouver la source du problème tous ensemble et de le résoudre l'après-midi même. En tout, on a eu une très bonne expérience avec l'entreprise.

VIII - Gestion de travail en groupe :

Pendant la première semaine nous avons rencontré quelques difficultés pour bien commencer l'implémentation du projet. Les deux premiers jours ont été dédiés aux recherches sur les bibliothèques utiles pour notre projet ainsi que les fonctions qu'on pourrait utiliser, mais l'implémentation du code n'a commencé qu'à partir du mercredi. C'était difficile de travailler en groupe au début car la première partie était difficile à séparer en cinq tâches différentes. Nous avons décidé d'attribuer une étape de la segmentation chacun, mais mettre ces fonctions ensemble a pris plus de temps que prévu. Après cette première semaine, nous avons décidé de diviser le travail en plusieurs groupes.

A partir de la deuxième semaine nous avons fait deux groupes de deux. Chaque groupe de deux avait une tâche à faire (une fonctionnalité du code), et la dernière personne a commencé à écrire le rapport et la documentation pour ne pas devoir le faire au dernier moment. C'est la première fois que nous travaillons sur un projet avec plus de deux personnes, et nous avons trouvé que cette méthode était la meilleure et la plus efficace.

Par rapport à la planification des tâches créée en décembre, nous l'avons assez bien suivie, mais il y a juste eu un changement. Etant donné que nous avons commencé l'implémentation du code au milieu de la première semaine, nous n'avons pas eu le temps de faire l'assignation linéaire dans la première semaine. Finalement ce retard n'a pas posé problème car cela nous a permis de nous consacrer au tracking dès le retour des vacances.

VI - Annexe :

Notre projet sur Github : <https://github.com/neiviv-dev/Cell-Segmentation-Tracking>

Bibliographie :

- Hideya Aragaki, Katsunori Ogoh, ..., 2022, Scientific Reports, Vol.12, “**LIM Tracker: a software package for cell tracking and analysis with advanced interactivity**”, <https://www.nature.com/articles/s41598-022-06269-6>
- Chentao Wen, Takuya Miura, ..., 2021, eLife, Vol.10, “**3DeeCellTracker, a deep learning-based pipeline for segmenting and tracking cells in 3D time lapse images**”, <https://elifesciences.org/articles/59187>
- Sami Nadif, cours-gratuit, “**Tuto Python & OpenCV : traitement d’images - Tutoriel Python**”,
<https://www.cours-gratuit.com/tutoriel-python/tutoriel-python-les-bases-de-traitement-dimages-en-python-opencv#:~:text=OpenCV%20est%20l%27une%20des,Elle%20poss%C3%A8de%20plusieurs%20algorithmes%20optimis%C3%A9s>
- Manav Narula, Delft Stack, 2022, “**Fonction OpenCV Sobel()**”,
<https://www.delftstack.com/fr/howto/python/opencv-sobel/>
- Anon, “**OpenCV, c’est quoi ?**”,
<https://www.axopen.com/blog/2019/09/open-cv-cest-quoi/>
- #include<>, 2019, Medium, “**Scikit-image VS OpenCV**”,
<https://medium.com/@hashinclude/scikit-image-vs-opencv-a2ce6e9b23d1>
- Anon, docs.opencv, “**OpenCV: Morphological Transformations**”,
https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html
- Anon, kongakura, “**Tutoriel Opencv Python - Traitement d’images - Vision par ordinateur - Kongakura**”,
https://www.kongakura.fr/article/OpenCV_Python_Tutoriel
- Anon, LearnOpenCv, 2021, “**Contour Detection using OpenCV (Python/C++)**”,
<https://learnopencv.com/contour-detection-using-opencv-python-c/>

