Rapport de stage - DUT Informatique - 2ème Année

Développement Java de NucleusJ et intégration d'OMERO

IUT de Clermont-Ferrand
Département Informatique

Année universitaire 2020-2021

Présenté par **: Alexandre RONGIER**

J’autorise la diffusion de
mon rapport sur
l’intranet de l’IUT

Remerciements

Je remercie l'ensemble des personnes qui ont contribué au bon déroulement de mon stage et qui m'ont aidé lors de la rédaction de ce rapport.

Je voudrais ensuite remercier le GReD et plus particulièrement Christophe TATOUT pour m'avoir accueilli au sein de son équipe. Je remercie grandement mon encadrante, Sophie DESSET pour m'avoir suivi, aidé et conseillé tout au long du stage.

Merci à Tristan Dubos et à Pierre POUCHIN, avec qui j'ai collaboré, pour m'avoir guidé et m'avoir souvent fourni une aide précieuse.

Merci également à Nadia GOUÉ pour les informations qu'elle m'a données sur le Mésocentre Clermont-Auvergne et pour m'avoir suivi tout le long de mon stage.

Je tiens aussi à remercier l'IUT Informatique de Clermont-Ferrand pour m'avoir aidé à chercher et à trouver ce stage. Enfin je remercie Carine Simon pour avoir veillé au bon déroulement de mon stage.

I. Sommaire

Introduction

L'imagerie par microscopie s'est beaucoup développée en biologie depuis les années 2000 avec l'arrivée de nouvelles technologies qui permettent de reconstituer des images d'échantillons en 3 dimensions. Parallèlement, l'analyse des images, ainsi que leur traitement statistique, s'est également beaucoup répandue et nécessite toujours plus de ressources de stockage et de calcul.

Les laboratoires de biologie utilisent des logiciels pour analyser des images de microscopie. Un des logiciels les plus utilisés est le logiciel ImageJ. Le GReD a développé en 2014 un plugin***** de ce logiciel, NucleusJ, destiné à l'analyse des noyaux de cellules. En 2020, une nouvelle version de ce plugin a été développée pour automatiser le processus et lui permettre de traiter de grandes quantités de données.

Pour faciliter la sauvegarde et la gestion des données, les fichiers d'images sont stockés sur un serveur mutualisé du mésocentre Clermont-Auvergne via une API***** open source* appelée OMERO. La bibliothèque Simple-OMERO-Client a été développée au laboratoire pour permettre d'effectuer les calculs sur les images directement à partir de cette API.

C'est dans ce contexte que j'ai effectué mon stage afin de participer au développement du plugin NucleusJ ainsi qu'à la bibliothèque Simple-OMERO-Client utilisée pour lancer des calculs à partir d'images stockées sur OMERO.

Le stage s'est effectué dans l'institut GReD à la faculté de médecine en collaboration avec un ingénieur du GReD, Pierre POUCHIN et d'un doctorant en bio-informatique, Tristan DUBOS.

Plusieurs objectifs étaient attendus : ajout de fonctionnalités au plugin NucleusJ, optimisation de traitements, mise en place de tests fonctionnels, harmonisation des fonctionnalités du plugin sur les environnements utilisés ou encore correction de certains problèmes qui peuvent survenir dans le cadre du développement de NucleusJ et de l'intégration d'OMERO.

Je présenterai d'abord le GReD et l'équipe que j'ai intégré, puis je parlerai du contexte, des différents outils utilisés pendant mon stage ainsi que des objectifs fixés. Ensuite j'exposerai le travail accompli en commençant par l'harmonisation et l'ajout de fonctionnalités. Après cela je montrerais la résolution d'un bug, les tests que j'ai réalisés et pour finir l'optimisation des performances du logiciel.

I. Présentation du GReD

1. Le GReD

Le GReD (désormais iGReD) est un institut de recherche en biologie-santé localisé à Clermont-Ferrand. Il rassemble depuis 2008 des équipes de recherche reconnues dans les domaines de la Génétique, de la Reproduction et du Développement. Les trois tutelles de cette Unité Mixte de Recherche (UMR) sont le CNRS (UMR6293), l'INSERM (U1103) et l'Université Clermont Auvergne (UCA).

Les questions biologiques posées visent à comprendre comment se développe un organisme vivant et comment, par suite d'un programme génétique et épigénétique bien orchestré, des dérégulations peuvent malgré tout survenir et entraîner des pathologies variées.

L'unité se compose de quatorze équipes réparties en trois axes de recherche qui sont :

- Axe 1 : Dynamique du génome et contrôle épigénétique ;
- Axe 2 : Reproduction et développement normal et pathologique ;
- Axe 3 : Endocrinologie, signalisation et cancer.

2. Équipe intégrée

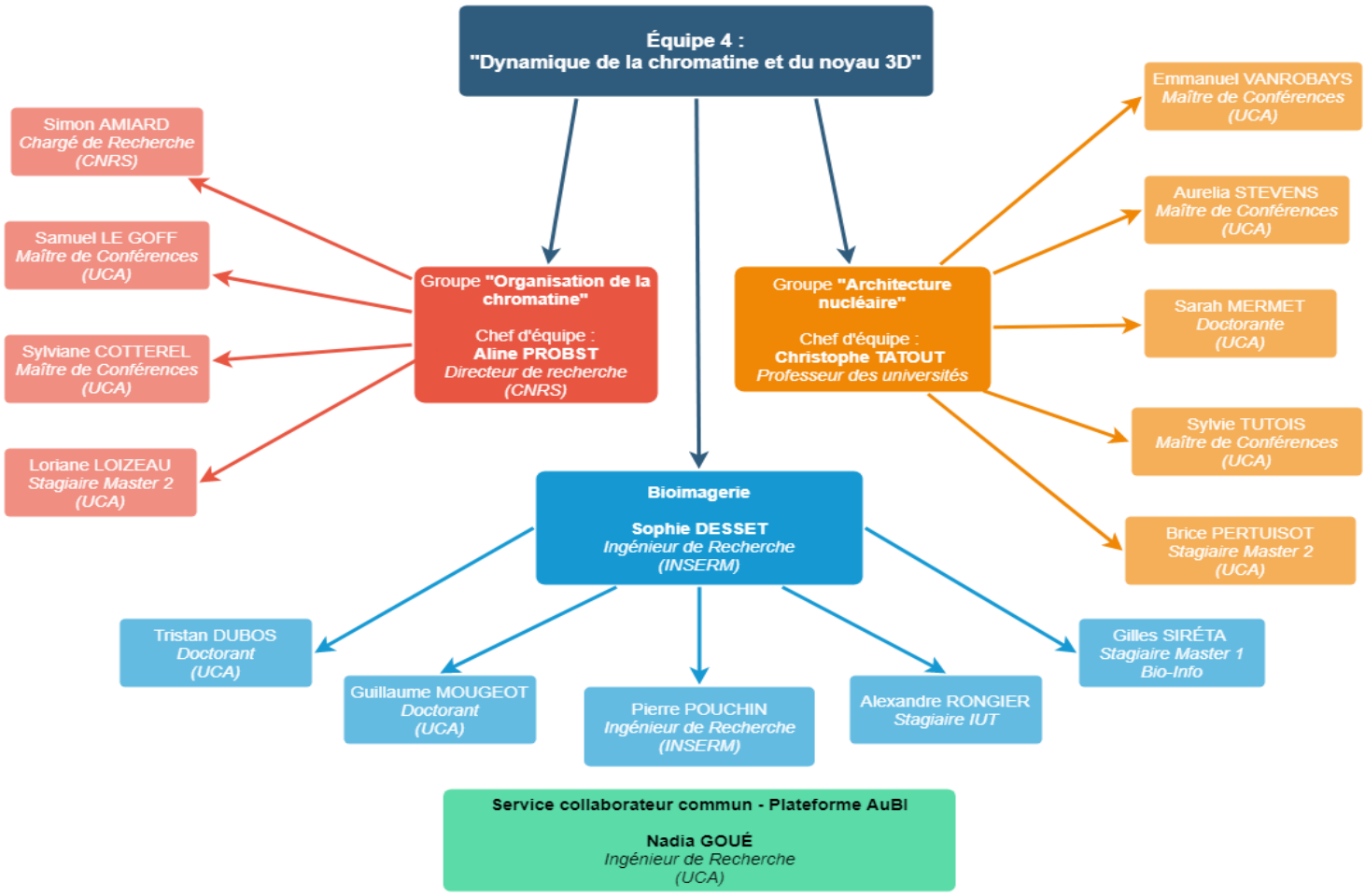

Figure 1 : Organigramme de l'équipe CODED. Au sein de l'équipe CODED, mon stage s'inscrit principalement dans le groupe « bio-imagerie 3D » du GReD. Les personnes dans les 2 autres groupes de l'équipe sont des utilisateurs

du plugin sur lequel j'ai travaillé. Le groupe bio-imagerie collabore avec une structure mutualisée de l'UCA, la plateforme Auvergne Bioinformatique au Mésocentre.

J'ai effectué mon stage dans l'équipe 3 nommée CODED "Chromatin and 3D nuclear Dynamics", dirigée par Aline Probst et Christophe Tatout et dont l'organigramme est présenté en **Erreur ! Argument de commutateur inconnu.**. Cette équipe de l'axe 1 étudie l'impact de la morphologie nucléaire (forme/ taille des noyaux dans les cellules) et de l'organisation de la chromatine (fibre d'ADN empaquetée par des protéines, majoritairement des histones) sur la régulation des gènes. Il s'agit de mécanismes dits "épigénétiques" parce qu'ils modulent la régulation des gènes sans modifier la séquence de l'ADN. Pour leurs recherches, les biologistes généticiens ont recours à des mutations dans les protéines histones ou dans les protéines de l'enveloppe nucléaire, et ils font des comparaisons entre les situations mutées et "sauvages", c'est-à-dire sans mutation.

Parmi les techniques utilisées dans l'équipe, une grande part des études concerne l'imagerie par microscopie confocale, c'est-à-dire l'observation des noyaux en trois dimensions. Les images obtenues doivent ensuite être analysées par des logiciels dédiés afin de calculer les différents paramètres décrivant la morphologie et l'organisation des noyaux. Les populations de noyaux mutants et sauvages sont comparées par des outils statistiques ce qui nécessite de produire puis de traiter un grand nombre de données. Il est important pour l'équipe d'automatiser toutes les étapes du processus et d'augmenter ses ressources de calculs pour produire rapidement des résultats robustes.

3. Contexte biologique

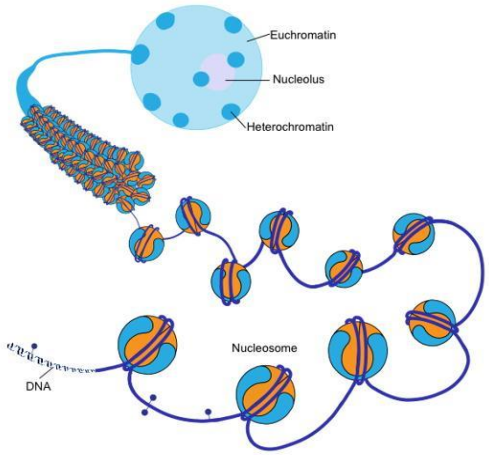

Figure 2 : Noyau (cercle bleu) dans lequel on retrouve la chromatine, un complexe où l'ADN (violet) est enroulé autour des histones pour former les nucléosomes en bleu et orange (structure perlée). En rose, le nucléole, un compartiment sans ADN. En bleu foncé, la chromatine, très compactée, prend le nom d'hétérochromatine qui est structurée en « chromocentres » dans la plante modèle Arabidopsis Thaliana.*

L'information génétique chez les êtres vivants est contenue dans l'ADN qui se trouve lui-même dans le noyau de chaque cellule chez les eucaryotes. L'ADN ne s'y trouve pas nu mais complexé à des protéines qu'on appelle histones. Le complexe formé par l'ADN et les histones est la chromatine. Un chromosome est une unité moléculaire de chromatine qui contient une seule double hélice d'ADN, sauf dans la phase précédant la division cellulaire où il en contient deux : il adopte alors la forme bien connue qui sert souvent à le représenter, en X. Lorsque la cellule n'est pas en phase de division le chromosome n'a pas de structure macroscopique bien définie, on le représente comme une pelote de laine débobinée. Malgré cette impression d'arrangement aléatoire, le noyau des cellules est très organisé. On sait par exemple que chaque chromosome débobiné occupe toujours la même place relative dans une cellule et que les chromosomes voisins sont toujours les mêmes.

Pour former la chromatine, la double hélice s'enroule autour d'un complexe de 8 histones pour former ce qu'on appelle un nucléosome, comme illustré sur la Figure 2, et les nucléosomes sont plus ou moins rapprochés les uns des autres, donnant à la chromatine un état plus ou moins condensé :

- L'hétérochromatine est plus compactée et contient des séquences répétées comme les centromères. Elle se localise majoritairement en périphérie nucléaire. Chez la plante Arabidopsis thaliana, l'organisme modèle étudié dans l'équipe, elle est organisée en « chromocentres » au sein du noyau. Les chromocentres ont la forme de spots intenses lorsqu'on colore la double hélice car il s'agit des zones les plus denses (Figure 3).

- L'euchromatine est moins condensée et comprend des gènes qui s'expriment dans la cellule.

La morphologie du noyau et la structure de la chromatine sont à la base de mécanismes épigénétiques qui ont un impact sur l'expression des gènes. C'est ce qui intéresse l'équipe CODED du GReD : quel est le lien entre l'architecture nucléaire et la régulation des gènes ? L'étude des mécanismes épigénétiques par l'imagerie nécessite de produire des images en trois dimensions et de les analyser. La microscopie confocale permet d'obtenir ces images 3D en capturant plusieurs plans focaux et en les assemblant afin de former une pile d'images. Ceci permet d'avoir une représentation en profondeur de l'objet observé.

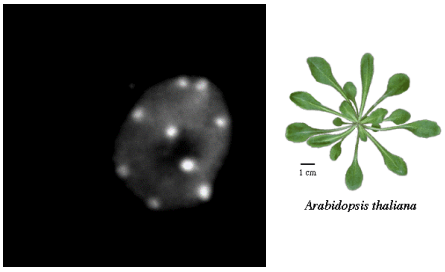

Figure 3 : Image obtenue à partir de la projection d'une pile d'images d'un noyau type (à gauche) de la plante Arabidopsis thaliana (à droite). Les spots intenses à l'intérieur du noyau sont les chromocentres. La partie grise est l'euchromatine

Les analyses nécessaires dans les études de l'équipe exigent une reproductibilité pour obtenir des données sur un grands nombre de noyaux. Pour répondre à ce besoin, des logiciels interopérables sont utilisés par les chercheurs et c'est dans ce cadre que s'inscrit la mission de mon stage.

II. Présentation du stage

Les missions de mon stage tournent autour de 2 principaux outils utilisés par les biologistes du GReD. Le premier est le logiciel Fiji et le second est le logiciel OMERO.

1. Outils

a) Fiji

Fiji, dont le nom est un acronyme récursif pour "Fiji Is Just ImageJ", est une version "étendue" de ImageJ, un logiciel de traitement d'images développé en Java par les NIH américains (National Institutes of Health). Fiji est livré avec un certain nombre de plugins, permet d'en télécharger d'autres directement depuis l'interface et offre la possibilité d'effectuer des mises à jour automatiques, des dépendances comme des plugins (Figure 4).

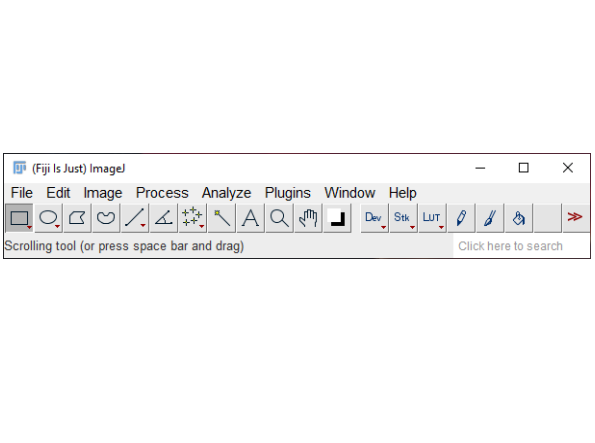

Figure 4 : Interface de Fiji avec une image 3D grand champ ouverte.

Cette distribution d'ImageJ est particulièrement populaire pour l'analyse d'images en biologie. Fiji peut notamment ouvrir des fichiers dans de nombreux formats propriétaires, imposés par les fabricants de microscopes, grâce à la bibliothèque Bio-Formats qui est quasiment exhaustive.

b) OMERO

OMERO est un logiciel client-serveur open source* développé par le consortium OME (Open Microscopy Environment), au même titre que Bio-Formats, et financé par l'Union Européenne ainsi que le Royaume-Uni. Cet outil permet d'organiser, de visualiser et d'annoter des images dans de nombreux formats et il est utilisé par le laboratoire GReD pour la gestion des images de microscopie acquises par les chercheurs (une autre instance est disponible au mésocentre Auvergne sur la plateforme Auvergne-Bioinformatique, d'où mes interactions avec le mésocentre pendant le stage). Ces données sont stockées sur un serveur, gérées par le logiciel OMERO.server et sont accessibles de plusieurs manières en passant par :

- l'interface web OMERO.web (client web), illustrée sur la **Erreur ! Argument de commutateur inconnu.** ;
- le logiciel OMERO insight (client lourd) ;
- les API (Application Programming Interface) OMERO mises à disposition dans différents langages comme Python, Java, C++ ou encore R.

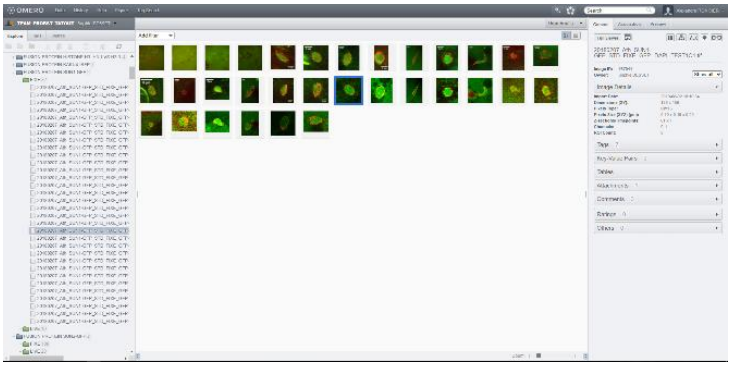

Figure 5 : Interface web d'OMERO (à gauche les conteneurs : Datasets et Projets, à droite, les métadonnées liées à l'image sélectionnée et au centre la liste des images du dataset actuel)

L'intérêt principal d'OMERO est d'organiser les fichiers d'images (contenant l'image et ses métadonnées) dans le serveur grâce à des conteneurs (*Project* et *Dataset*) et de pouvoir annoter les images pour connaître facilement les informations correspondant aux données acquises et aux manipulations effectuées.

c) Le plugin NucleusJ

NucleusJ est un plugin* open source* du logiciel Fiji qui ajoute des fonctionnalités permettant d'effectuer des analyses sur la morphologie du noyau ainsi que sur l'organisation de la chromatine au sein de celui-ci.

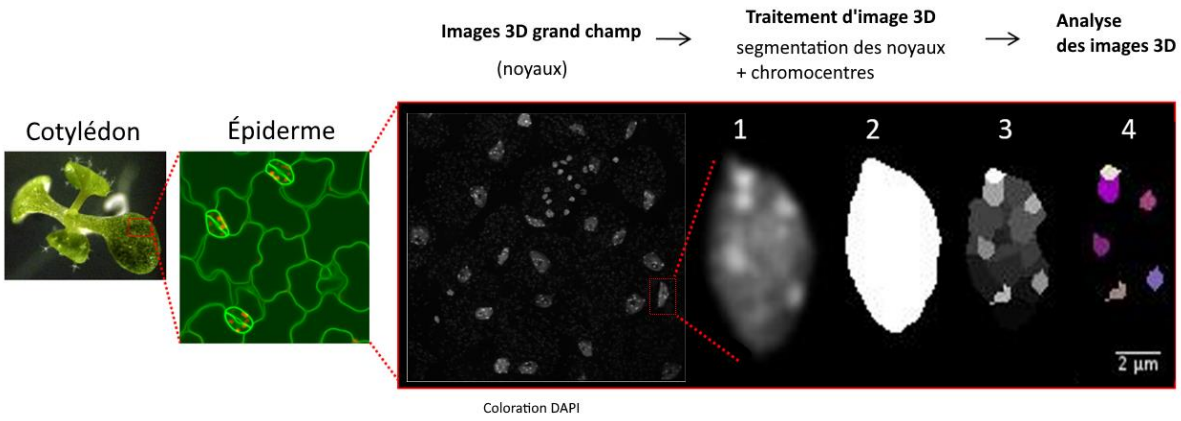

Figure 6 : Schéma représentant le cycle d'utilisation de NucleusJ. Ici on observe les noyaux des cellules de l'épiderme de cotylédon dans une plantule (à gauche). L'image grand champ obtenue grâce à la coloration DAPI (colorant se fixant sur l'ADN pour le mettre en évidence par fluorescence) permet de visualiser les noyaux. NucleusJ traite les images grands champ en 4 étapes : (1) individualisation des noyaux dans des fichiers (2) segmentation d'image* pour pouvoir analyser leur morphologie (3) segmentation de leurs chromocentres pour analyser leur organisation. La dernière étape est un seuillage* manuel (étape 4).*

L'utilisation du plugin peut être résumée en 3 étapes (Figure 6**) :**

- La première étape est l'extraction, à partir d'images « grand champ » en 3D, de régions d'intérêt contenant chacune un noyau, au moyen de boites en 3D (autocrop) : cela produit une image 3D par noyau détecté.
- La deuxième étape est la segmentation des noyaux dans ces images, qui permet d'obtenir des paramètres sur la morphologie du noyau. Elle utilise un algorithme de seuillage qui est une version modifiée de la méthode Otsu*****.
- La dernière étape est la segmentation des chromocentres qui utilise les étapes précédentes pour délimiter les zones d'intensité à l'intérieur du noyau et obtenir des informations sur le nombre de chromocentres, le volume d'hétérochromatine, etc. Cette étape comprend une segmentation par le logiciel qui prédéfinit des zones d'intérêt en utilisant un algorithme de watershed***** et un seuillage manuel qui en précise les contours.

Une phase de l'analyse des chromocentres est manuelle et nécessite l'intervention de l'utilisateur pour obtenir des résultats. Néanmoins, un projet appelé NodeJ est mené en parallèle au sein du GReD et vise à automatiser l'analyse des chromocentres.

Le plugin est principalement développé, depuis 2018, par Tristan Dubos, doctorant en bio-informatique au sein de l'équipe. NucleusJ2 (NJ2) permet la détection, le découpage et la segmentation des noyaux sur une grande quantité d'images (Dubos T et al, 2020).

2. Environnement logiciel

a) GitLab

Dans le cadre du développement de NucleusJ, nous utilisons GitLab, un système de gestion de maintenance collaboratif libre basé sur Git, lui-même un logiciel de gestion de versions largement répandu. Ce dernier permet surtout de conserver d'anciennes versions du code ou bien des modifications sans nécessairement changer le code en utilisant un système de branches. GitLab permet entre autres le stockage, le suivi des

bugs, l'intégration continue* et la livraison continue. NucleusJ2 est disponible sur GitLab en intégration continue (https://gitlab.com/DesTristus/NucleusJ2.0).

b) OMERO & NucleusJ

Afin de pouvoir traiter les images stockées sur OMERO avec NucleusJ, Rémi VALARCHER, un stagiaire ingénieur, a développé en Java la bibliothèque "Simple-OMERO-Client" qui permet de faciliter l'usage de l'API Java d'OMERO, notamment dans le cadre du développement de NucleusJ.

La bibliothèque met à disposition des « wrappers » permettant d'encapsuler les différents éléments accessibles sur OMERO comme des datasets, des projets, etc.

NucleusJ utilise cette bibliothèque pour se connecter à OMERO et ainsi faire des traitements sur les images stockées sur le serveur.

c) Java

NucleusJ est développé en utilisant le langage de programmation Java, un langage de programmation orientée objet détenu qui est maintenu par la société Oracle.

Java est compilé pour pouvoir être exécuté par une machine virtuelle* Java (JVM). L'utilisation d'une JVM permet au langage de fonctionner sur la plupart des plateformes du moment qu'elle y est disponible.

Java met à disposition le format de fichier JAR (Java Archive), qui peut contenir un programme exécutable Java. Néanmoins dans la plupart des cas il contient une bibliothèque Java où les fichiers sont rassemblés et compressés. Cette bibliothèque est destinée à être intégrée à un autre projet ou logiciel. On retrouve dans les JAR un manifeste qui informe sur la version et l'auteur du code mais il peut aussi donner des informations sur la compilation dans le cas d'un exécutable.

De plus, Fiji possède un répertoire « plugins » dans lequel sont regroupés les fichiers JAR correspondants au extensions logicielles dont il dispose. Ainsi pour ajouter une extension il suffit d'insérer dans ce répertoire un fichier JAR respectant un format de plugin propre à Fiji.

d) Maven

Apache Maven est un outil de gestion et d'automatisation de production des projets logiciels Java en général. Il permet par exemple de gérer les dépendances entre deux logiciels tels que ImageJ et son plugin NucleusJ, qui utilise la bibliothèque ImageJ pour certains traitements d'images.

Un projet Maven est configuré par un POM (Project Object Model) qui contient les informations nécessaires à Maven pour traiter le projet. Ce POM est matérialisé par le fichier pom.xml qui doit être présent à la racine du projet. On retrouve dans ce fichier les caractéristiques classiques d'un projet (nom, description, version, …) mais aussi les dépendances à d'autres projets, des informations concernant la structure du projet, etc. Maven inclut également plusieurs commandes qui sont organisées selon un cycle de vie, de la compilation jusqu'au déploiement, en passant par les tests.

3. Objectifs

Ma mission lors de ce stage était de développer en Java de nouvelles fonctionnalités pour NucleusJ v2 et de simplifier son l'utilisation pour les utilisateurs. Les tâches que j'ai effectuées ont été établies au fur et à mesure de mon stage et donnaient suite aux retours de mon encadrante Sophie Desset, qui utilise le plugin NucleusJ pour ses études.

On peut regrouper les tâches demandées en 3 parties : ajout de fonctionnalités, tests et optimisations :

- **Harmonisation et ajouts de fonctionnalités :**

  NucleusJ possédait plusieurs fonctionnalités qui complètent les 4 étapes d'utilisation de la Figure 6 mais leur utilisation pouvait être limitée à un environnement, par exemple certaines étaient uniquement disponibles en ligne de commande. Il était donc nécessaire d'harmoniser l'utilisation du plugin en ajoutant ou modifiant les fonctionnalités qui pouvaient varier selon que l'on utilise le plugin en ligne de commande, sur OMERO, en interface graphique intégrée à Fiji, etc.

- **Tests :**

  Lors du développement il est important de prévenir les bugs qui peuvent apparaître pour les résoudre le plus rapidement possible et garantir qu'il n'y a pas de retour en arrière en vérifiant par exemple la validité des résultats. Dans le cadre de NucleusJ, il était nécessaire d'ajouter des tests automatisés afin d'améliorer la stabilité et la qualité du plugin dans le temps.

- **Optimisations :**

  Certaines étapes de NucleusJ demandaient à être optimisées. En effet, certains traitements prenaient un temps considérable pouvant s'étendre sur plusieurs jours, il fallait donc le réduire.

III. Développement de NucleusJ et intégration d'OMERO

1. Harmonisation et ajout de fonctionnalités

a) Modes d'utilisation de NucleusJ

Je suis principalement intervenu sur 2 cas d'utilisation de NucleusJ qui sont visibles sur le diagramme de cas d'utilisation de la Figure 7, l'autocrop et la segmentation de noyaux, pour lesquels j'ai harmonisé l'utilisation avec Omero en ligne de commande et par l'interface.

J'ai aussi corrigé les fichiers de résultats obtenus lors de l'autocrop afin qu'ils suivent le même format et donnent les mêmes informations dans tous les modes d'utilisation.

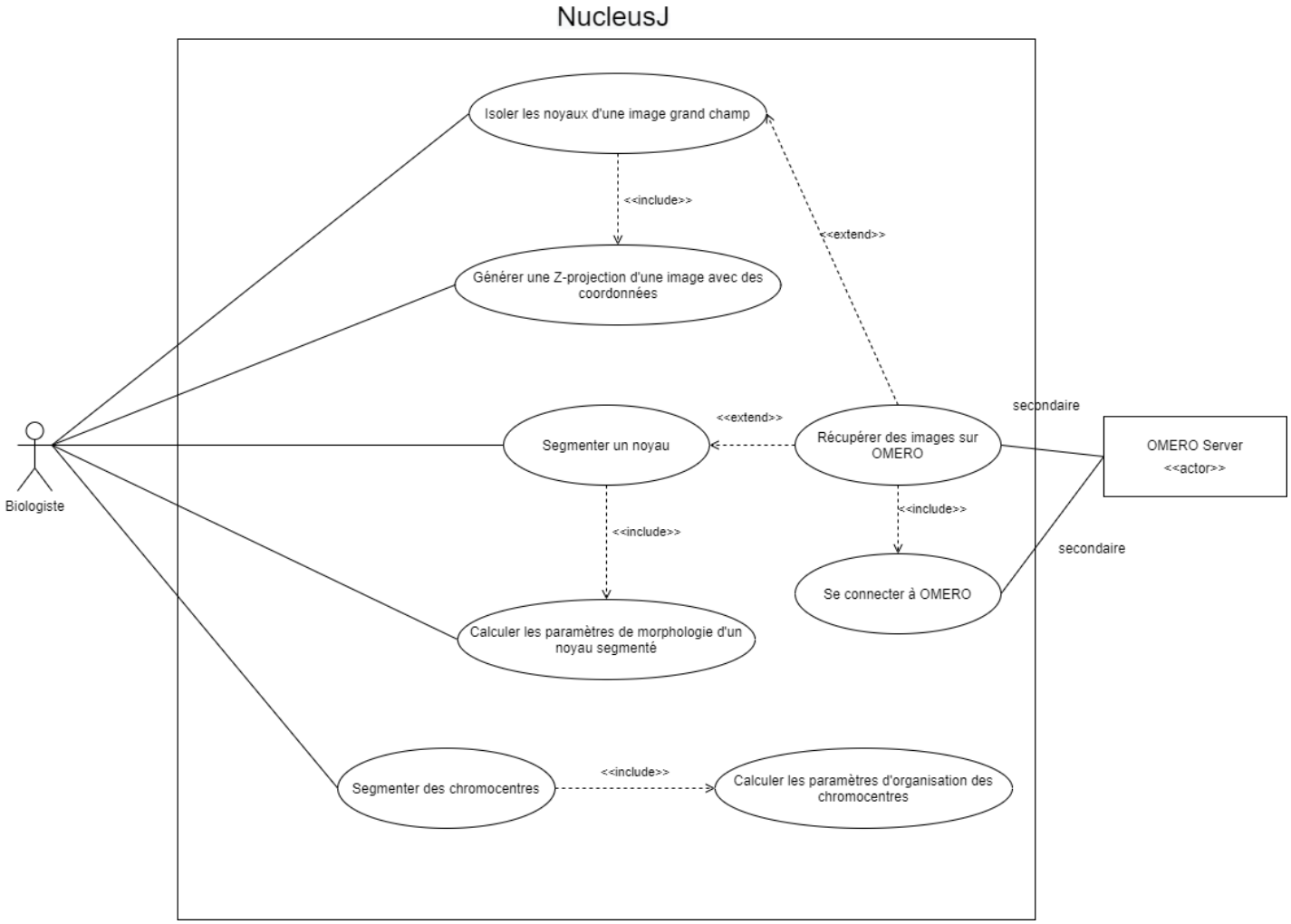

Figure 7 : Diagramme de cas d'utilisation de NucleusJ

Description du diagramme :

- ***Isoler les noyaux d'une image grand champ :*** *Ce cas d'utilisation correspond à* ***l'autocrop*** *qui détecte les coordonnées des noyaux dans une ou plusieurs images 3Ds afin d'obtenir des fichiers séparés pour chaque noyau. De plus lors de ce traitement on génère une Z-Projection (*Figure 10*) qui résume l'opération en utilisant les coordonnées trouvées (cf. « Générer une Z-Projection d'une image avec coordonnées »).*

L'utilisateur peut choisir d'utiliser des images stockées sur OMERO (cf. « Récupérer des images sur OMERO »).

- ***Générer une Z-Projection d'une image avec coordonnées :*** *À partie de coordonnées déjà connues, l'utilisateur peut générer une projection faisant office de résumé de tous les noyaux présents dans l'image.*
- ***Segmenter un noyau :*** *L'utilisateur peut, à partir d'une ou plusieurs images de noyaux isolés, obtenir un masque (une image binaire*) pour reproduire le volume des noyaux en 3D. En plus de ça, les paramètres de morphologie des noyaux sont calculés (cf. « Calculer les paramètres de morphologie d'un noyau segmenté »). L'utilisateur peut choisir d'utiliser des images stockées sur OMERO (cf. « Récupérer des images sur OMERO »).*
- ***Calculer les paramètres de morphologie d'un noyau segmenté :*** *L'utilisateur peut obtenir des paramètres calculés à partir d'un noyau segmenté (Ex : Sphéricité, Élongation, Volume...etc.).*
- ***Segmenter des chromocentres :*** *L'utilisateur peut segmenter les chromocentres au sein d'un noyau. Cette fonctionnalité calcul aussi des paramètres sur l'organisation de la chromatine du noyau (cf. « Calculer les paramètres d'organisation des chromocentres »).*
- ***Calculer les paramètres d'organisation des chromocentres :*** *On peut obtenir des paramètres calculés à partir d'une segmentation de chromocentres (Ex : Volume d'hétérochromatine...etc.).*
- ***Récupérer des images sur OMERO :*** *Ce cas d'utilisation permet à l'utilisateur de télécharger des images issues d'OMERO pour effectuer des traitements. Ce cas d'utilisation nécessite à l'utilisateur de se connecter pour qu'il puisse récupérer ses images (cf. « Se connecter à OMERO »).*
- ***Se connecter à OMERO :*** *L'utilisateur peut se connecter à son compte (avec entre autres son mot de passe et son login) sur un serveur OMERO (en passant l'adresse et le port).*

NucleusJ peut être utilisé de différentes manières par les biologistes.

- D'une part le plugin est utilisable en ligne de commande. Pour cela il faut exécuter le fichier JAR généré par une commande de type :

```
java -jar NucleusJ_2-2.0.0.jar -a autocrop -in ./input-nuclei -out ./output -c ./autocrop.config
```

Le paramètre précédé par -a indique le type de traitement à effectuer, l'Autocrop dans le cas présent, -in indique le chemin d'entrée et -out le chemin de sortie. Il est également possible de préciser un fichier de configuration avec le paramètre -c, le cas échéant.

Pour traiter des images issues d'OMERO la commande est différente :

```
java -jar NucleusJ_2-2.0.0.jar -ome -a segmentation -in dataset/78856 -out 9689 -ho omero.gred-clermont.fr -port 4064 -u demo -g 553
```

Le paramètre -ome indique que l'on souhaite utiliser OMERO, dans ce cas-là -in indique le type de conteneur et son ID et -out l'ID du projet de sortie. Il faut aussi donner l'adresse du serveur OMERO (-ho) et le port (-port). De plus, il est nécessaire d'entrer le nom d'utilisateur après -u ainsi que l'ID du groupe auquel il appartient dans la base OMERO après -g.

À l'exécution le mot de passe de l'utilisateur est demandé avant de lancer le traitement.

L'utilisation en ligne de commande n'est pas intuitive, notamment pour des biologistes, c'est pourquoi un script Bash destiné à simplifier l'utilisation du fichier JAR a été écrit. Il permet de générer étape par étape la commande précédente.

À noter que le GReD possède des serveurs de stockage et de calcul appelés Wario, Bowser, Yoshi, Luigi et Mario. Ces serveurs tournent sur un système d'exploitation GNU/Linux (Debian) ce qui leur permet d'exécuter des scripts Bash pour ensuite traiter de grandes quantités de données.

- D'autre part, NucleusJ s'intègre à Fiji, et est disponible dans l'onglet « Plugin » une fois que le fichier JAR a été placé dans le dossier « plugins » de Fiji (Figure 8).

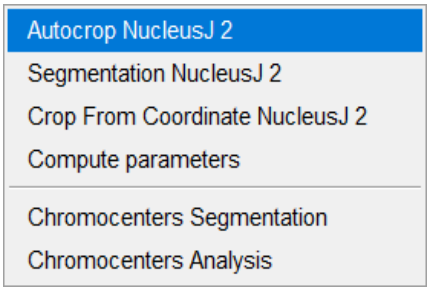

Figure 8: Interface du menu global NucleusJ disponible depuis Fiji. Les fonctionnalités principales sont « Autocrop » et la « Segmentation ».

« Crop From Coordinate » permet de recouper les noyaux 3D d'une images grand champ sur un autre canal (une autre coloration de la même image) sans avoir à recalculer un seuil puisque l'on sait où se situent les noyaux)

« Compute parameters » permet de recalculer les paramètres d'une image de noyau déjà segmentée.

« Crop From coordinate » et « Compute parameters » sont des fonctionnalités supplémentaires de NucleusJ qui ne sont pas disponibles quand on travaille avec OMERO. NucleusJ étant un projet collaboratif sur lequel plusieurs personnes ont déjà travaillé, il possède de nombreuses fonctionnalités qui ne sont pas harmonisées autour des différents modes d'utilisation disponibles. De plus, cet aspect collaboratif du projet est visible au sein même du code puisqu'il n'y avait pas vraiment de règles d'uniformisation du code totalement respectée avant mon arrivée.

b) Réalisation des interfaces pour OMERO

L'interface a été réalisée en utilisant la bibliothèque graphique Swing qui était déjà utilisée pour réaliser les versions précédentes de l'interface. Elle est aussi indépendante de la plateforme et permet donc de faire des interfaces similaires partout. Enfin, Swing, dans sa version standard, ne nécessite pas d'ajouter de dépendances supplémentaires au projet Maven.

Dans le cas d'un plugin destiné à être utilisable facilement par des biologistes, ce type d'interface est largement suffisant.

L'utilisation d'une interface graphique permet avant tout de simplifier l'utilisation du plugin et de le rendre accessible au plus grand nombre. Par exemple l'interface permet de mettre en évidence les champs et les options possibles pour un traitement d'image.

J'ai réalisé, en restructurant le code précédent, les interfaces pour lancer la segmentation et l'autocrop sur une ou plusieurs images issues d'OMERO, comme illustré sur la Figure 9.

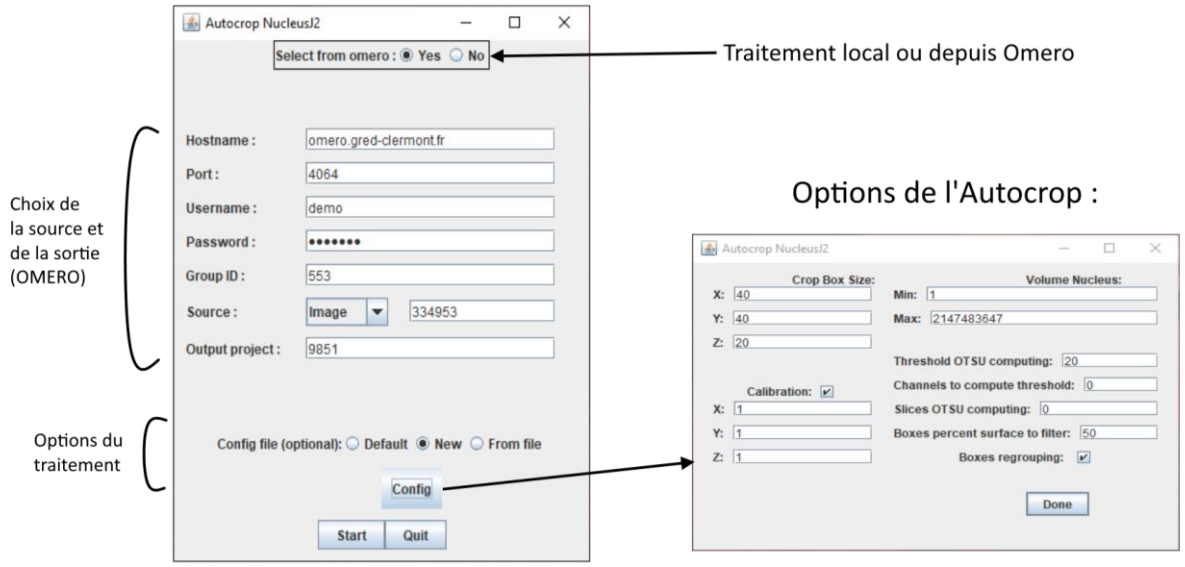

Figure 9 : Interface Fiji de lancement de l'Autocrop sur OMERO de NucleusJ2.

De même, j'ai adapté la fenêtre permettant la segmentation pour l'intégration d'OMERO. Elle se présente de la même manière mais les options de traitement sont différentes de celles de l'autocrop.

Le développement de l'interface m'a principalement permis de comprendre la structure du projet NucleusJ en abordant directement les points d'entrée du plugin, c'est-à-dire l'interface et la ligne de commande.

L'utilisation en ligne de commande se fait à partir d'une classe Main qui prend les paramètres vus précédemment comme un programme Java basique.

Lorsque le plugin est intégré au logiciel Fiji, le fonctionnement est différent. Fiji utilise un fichier « *plugins.config* » qui répertorie les fonctionnalités du plugin. Ce fichier définit le menu global du plugin accessible dans l'onglet « Plugins » de Fiji comme sur la Figure 8 et contient des lignes telles que :

```
Plugins>NucleusJ2.0, "Autocrop NucleusJ 2", gred.nucleus.plugins.Autocrop_
```

Ce qui signifie que l'entrée « Autocrop NucleusJ2 » (Accessible dans « Plugins>NucleusJ2.0 ») appelle la classe *gred.nucleus.plugins.Autocrop_* du fichier JAR. Cette classe implémente l'interface *PlugIn* du package *ij.plugin* qui définit une manière de lancer la fonctionnalité ce qui permet à Fiji d'utiliser la classe comme souhaité.

Il était d'autant plus pertinent pour moi de commencer par voir les points d'entrée du projet puisque celui-ci contient une quinzaine de packages différents et compte autour de 100 classes différentes.

c) Harmonisation des fichiers d'analyse

L'autocrop génère un ensemble de dossiers et de fichiers exploitables :

- un dossier « nuclei » contenant tous les noyaux 3D rognés de l'image grand champ au format TIF (Tagged Image File Format) ;
- un dossier « coordinates » contenant un fichier texte pour chaque image grand champ traitée qui répertorie les coordonnées des boîtes 3D (régions d'intérêt) contenant chaque noyau détecté ;

- un dossier « zprojection » qui contient les z-projections de chaque pile d'images grand champ initiale (Figure 10).

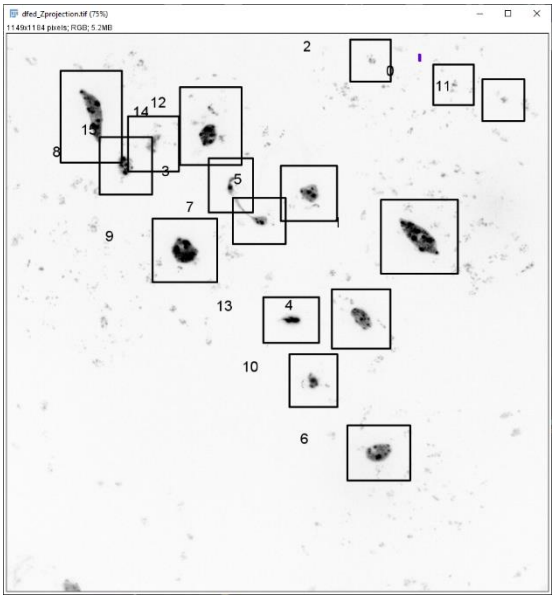

Figure 10 : Exemple d'une Z-Projection obtenue après un autocrop à partir d'une pile d'images. Tous les noyaux ne sont pas délimités (non détectés ou incomplets en z) et certains ne sont pas corrects (incomplets en xy) mais l'autocrop parvient à extraire les noyaux exploitables. A noter que la LUT (Look Up Table) est inversée par rapport aux images grand champ dont le fond est noir.

La Z-Projection est une sorte de résumé de l'autocrop car il s'agit d'une image 2D où chaque point correspond au point le plus intense parmi les points aux mêmes coordonnées de chaque niveau de profondeur de l'image 3D grand champ.

En plus de ces dossiers un fichier CSV récapitulatif est généré : il indique les paramètres de départ du traitement et reprend chaque image grand champ en indiquant entre autres le nombre de noyaux trouvés et le seuil calculé pour trouver les noyaux avec la méthode d'Otsu (algorithme de seuillage).

Sur OMERO, les dossiers deviennent des datasets et les fichiers de résultats sont attachés à des conteneurs. Par exemple, le fichier des résultats globaux est rattaché au *projet* ou au *dataset* où se trouvent les images en entrée.

J'ai procédé à des corrections pour harmoniser la génération de ces fichiers. Je me suis assuré que les fichiers soient bien enregistrés et disposent des bonnes informations dans tous les cas. J'ai aussi pris en compte les cas où le traitement est effectué sur une seule image ou lorsqu'il est effectué sur un ensemble d'images pour harmoniser les résultats.

La partie d'harmonisation sur laquelle j'ai travaillé m'a permis de simplifier l'utilisation de NucleusJ et d'homogénéiser les résultats. D'autre part j'ai pu appréhender le projet dans sa globalité pour pouvoir réaliser les tâches futures qui demandent une plus grande compréhension de la structure et du fonctionnement de NucleusJ.

2. Tests

J'ai dédié une partie de mon stage à une phase de développement de tests qui ont pour but de faciliter la détection des bugs susceptibles d'apparaître lors du développement de NucleusJ.

Cette partie de mon stage n'était pas initialement prévue et ce besoin est survenu après l'apparition d'un bug.

a) Gestion d'un bug

Le déclenchement de cette problématique dans mon stage a eu lieu lorsque Sophie, la principale utilisatrice de NucleusJ au sein du GReD, a fait part d'un problème de résultats. En effet, des résultats obtenus lors de l'autocrop, mais également dans certains cas pour la segmentation, différaient lorsque l'on utilisait NucleusJ depuis Fiji ou lorsqu'on l'utilisait en ligne de commande. De plus, les résultats obtenus dans un cas n'étaient pas vraiment satisfaisants et donc n'étaient pas exploitables à cause du problème.

Ce problème était d'autant plus surprenant qu'aucun changement de fond du code n'avait eu lieu avant l'apparition du problème.

En comparant les fichiers résultats, une différence apparaissait au niveau du seuil calculé avec la méthode Otsu. Ce seuil est utilisé à 2 reprises, lors de l'autocrop pour différencier les noyaux en 3D et les rogner et lors de la segmentation pour délimiter les contours du noyau sur l'image rognée. Dans les 2 cas la bibliothèque ImageJ est utilisée. C'est donc cette piste qui fut choisie pour trouver l'origine du bug.

Le code de NucleusJ lui-même n'étant pas le problème, nous avons déduit que le problème venait des dépendances du plugin. ImageJ est fréquemment mis à jour et le plugin NucleusJ utilise ImageJ lors de ces traitements. Il est donc possible que certaines fonctionnalités changent lorsque les bibliothèques d'ImageJ sont mises à jour. Néanmoins, dans un projet Maven, le pom.xml définit la dépendance à une version fixe d'ImageJ (Figure 11).

```
<dependencies>
    <!--...-->
    <!-- https://mvnrepository.com/artifact/net.imagej/ij -->
    <dependency>
        <groupId>net.imagej</groupId>
        <artifactId>ij</artifactId>
        <version>1.51n</version>
    </dependency>
    <!--...-->
</dependencies>
```

Figure 11 : Partie du fichier pom.xml définissant la dépendance à la version 1.51n de ImageJ qui était la version définie lors de mon arrivée.

La différence était donc vraisemblablement causée par le mode d'utilisation de NucleusJ :

- dans le cas où on lance NucleusJ en ligne de commande, la version d'ImageJ utilisée est celle définie dans le pom.xml.
- en revanche, lorsqu'on lance NucleusJ à travers Fiji, c'est ce dernier qui gère la dépendance à ImageJ et définit la version utilisée, cette dernière pouvant être différente de celle du pom.xml.

À la suite de ce constat, il apparaissait qu'une mise à jour de la version d'ImageJ (qu'intègre Fiji) ait amené un changement dans les résultats.

Pour vérifier cette hypothèse, j'ai réalisé différents tests manuels. J'ai d'abord vérifié que Fiji imposait bien sa version d'ImageJ en faisant varier les versions dans le pom.xml et en lançant le plugin depuis Fiji puis en

ligne de commande (CLI). Les résultats en ligne de commande variaient selon la version mais sur Fiji aucun résultat ne changeait puisque la version était fixe et imposée par Fiji.

Les facteurs qui nous intéressaient étaient :

- La version d'ImageJ, puisqu'il s'agissait de la source du problème.
- Le type de l'image. En effet, le plugin utilise ImageJ si nécessaire lorsque l'image en entrée est au format 16 bits pour la convertir au format 8 bits avant le traitement (*Figure 12*)

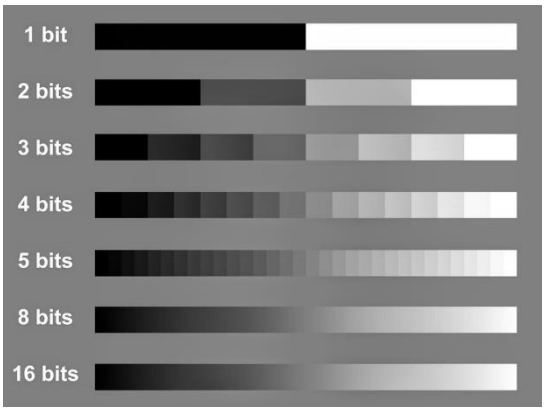

Figure 12 : Valeurs d'intensité possibles en fonction du format de l'image.

En 16 bits chaque pixel peut avoir 2^{16} soit 65 536 valeurs différentes en convertissant l'image en 8 bits on passe à 2^{8} soit 256 valeurs différentes. Cette réduction permet de réduire drastiquement le temps pris par les traitements suivant la conversion, notamment dans le cadre d'un seuillage comme ici pour l'autocrop.

```
find . -type f -exec ./is_result.sh {} \; |grep autocrop |grep cli |grep img-1

./1.51n/cli/16bits/img-1/autocrop      > seuil =  31
./1.51n/cli/8bits/img-1/autocrop       > seuil =  31
./1.53c/cli/16bits/img-1/autocrop      > seuil =  27
./1.53c/cli/8bits/img-1/autocrop       > seuil =  27
./1.53i/cli/16bits/img-1/autocrop      > seuil =  65
./1.53i/cli/8bits/img-1/autocrop       > seuil =  27
```

Figure 13 : Résultats de tests manuels sur une même image avec l'autocrop. J'ai réalisé un script Bash (appelé par la commande « find » ci-dessus) pour plus facilement lire l'ensemble des tests en récupérant les valeurs de seuil directement dans les fichiers CSV.

Constater ces différences (Figure 13) permettait de chercher plus précisément où était utilisé ImageJ et pourquoi la version utilisée par Fiji ne donnait pas de résultats satisfaisants.

Les différences observées étaient en fait liées à des évolutions d'ImageJ : entre la version 1.51n, qui est la version définie dans le pom.xml, et la version 1.53c, une partie d'ImageJ chargée d'augmenter le contraste d'image en utilisant un étirement d'histogramme a été cassée.

Un histogramme permet de représenter la répartition des valeurs de pixel d'une image. Pour augmenter le contraste on peut modifier cette répartition en « étirant » l'histogramme ce qui permettra d'augmenter le contraste d'une image pour mettre en évidence certaines parties de l'image, c'est ce que fait un étirement d'histogramme.

Ce problème a été résolu dans la version 1.53i mais notre code n'était plus adapté à la nouvelle version. Le code a dû être adapté pour effectuer correctement l'étirement d'histogramme sur l'intégralité de l'image 3D.

De plus, il a fallu définir la plage d'affichage de l'image car une autre partie d'ImageJ chargée de convertir l'image en 8 bits avait changé et convertissait l'image selon les paramètres d'affichage utilisés par le logiciel, sans prendre en compte les nouvelles valeurs maximales et minimales pendant la conversion. La plage

d'affichage fut définie en prenant l'intensité maximale et l'intensité minimale des pixels de l'ensemble de la pile d'images.

b) Mise en place de tests

Le problème qui est apparu a soulevé une problématique importante : comment s'assurer ou vérifier que les mises à jour de NucleusJ ou bien de ses dépendances n'altèrent pas les résultats ?

Ma mission suivante était de mettre en place des tests chargés de vérifier la validité des résultats après les mises à jour. Cet objectif découlait directement du bug précédent puisqu'il s'agissait de réaliser des tests fonctionnels. Contrairement aux tests unitaires qui permettent de tester des parties très restreintes du code, les tests fonctionnels eux sont justement destinés à vérifier qu'une fonctionnalité fait bien ce qu'on attend d'elle et, dans notre cas, donne des résultats satisfaisants.

Les tests sont importants, notamment dans le cadre de l'intégration continue, or les outils déjà utilisés dans le projet permettaient d'en mettre facilement en place.

c) JUnit & Maven

JUnit est un framework* qui permet, entre autres, de faire des tests unitaires en Java. Néanmoins, dans notre cas nous l'avons utilisé pour réaliser des tests fonctionnels.

Maven permet d'organiser le cycle de vie de l'application en regroupant un ensemble de commandes, ce qui signifie qu'au lancement d'une commande, il exécutera celles qui la précède dans le cycle. On peut alors définir quels tests utiliser à l'étape « Test » du cycle qui se situe après la compilation.

Après avoir ajouté une dépendance à la bibliothèque JUnit dans le pom.xml j'ai pu commencer à mettre en place mes tests.

d) Principes des tests

Les tests ont été réalisés en 2 temps. Il fallait d'abord trouver un ensemble d'images 3D pour lesquelles nous disposions d'un résultat stable et satisfaisant après le traitement de l'image par NucleusJ. Ensuite, il fallait organiser les « cibles ». Ainsi j'ai regroupé dans un dossier « input » les images sources, dans un dossier « target » les résultats attendus pour ces images et j'ai préparé un dossier « output » pour contenir les résultats obtenus lors du test.

J'ai créé des classes pour tester les fonctionnalités clés de NucleusJ : l'autocrop et la segmentation.

Dans les 2 cas le fonctionnement est le même : d'abord j'effectue le traitement sur les images de test, puis j'utilise un ensemble de méthodes que j'ai développées pour récupérer les résultats attendus et les résultats ciblés. Après avoir obtenu les fichiers CSV de résultats ou les images de sortie, je peux les vérifier. Pour ce faire, j'ai créé une structure chargée de représenter et de stocker le résultat d'un autocrop et d'une segmentation ainsi qu'un ensemble de méthodes pour m'assurer de la validité de ces résultats en acceptant une marge d'erreur dans chacun des cas.

i. **Autocrop :**

Pour valider l'autocrop, je vérifie d'abord le nombre de noyaux récupérés présents dans le fichier CSV en acceptant une certaine marge d'erreur pour tolérer les changements mineurs de résultat.

Je vérifie ensuite les coordonnées des boîtes 3D contenant les noyaux : pour ce faire j'ai créé des méthodes permettant de connaître le pourcentage de chevauchement de 2 boîtes 3D.

Enfin, je considère le résultat comme valide si le volume de l'intersection des 2 boîtes est au moins supérieur à 80 % du volume de chacune des boîtes (Figure 14).

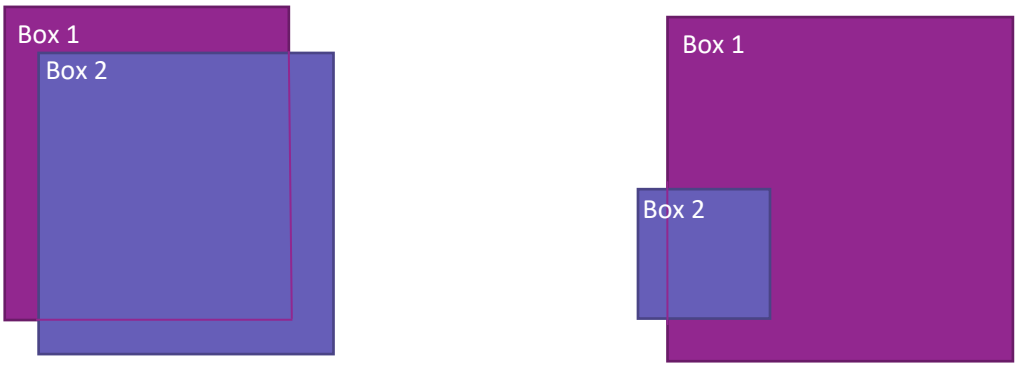

Figure 14 : Exemple de cas de figure pour la vérification des rognages :
À gauche, le résultat sera accepté comme les 2 boîtes sont chevauchés au moins à 80 %.
À droite, en revanche, seule la box 2 est chevauchée à plus de 80% le résultat ne sera pas validé.
C'est ce principe qui, étendu à la 3D, est utilisé pour les tests fonctionnels sur l'autocrop.

ii. **Segmentation :**

Pour vérifier la segmentation des images il faut directement utiliser les images binaires qui servent de masque aux noyaux. Ces images 3D n'ayant que 2 valeurs possibles blanc ou noir, il est facile de comparer les pixels entre les 2 images. La méthode que j'ai utilisée demande de prendre la valeur absolue de la soustraction des 2 images pour obtenir une image où chaque pixel blanc correspond à un pixel où les 2 images diffèrent. Je peux ensuite utiliser cette image différentielle en comptant l'ensemble des pixels blancs. Si le pourcentage de pixel différents est inférieur à 5 %, je considère le résultat comme valide (Figure 15).

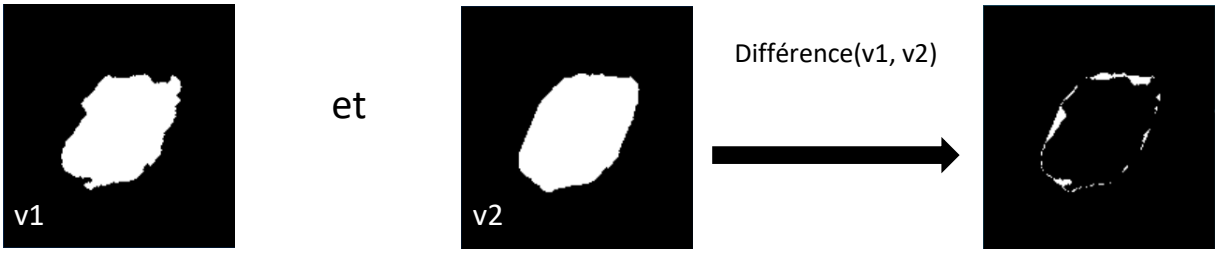

Figure 15: Exemple de soustraction de 2 versions de masques de noyaux. Chaque pixel blanc du résultat indique bien une différence.

3. Optimisations

J'ai consacré une partie de mon stage à l'optimisation de NucleusJ. Certains traitements prenaient, à mon arrivée, beaucoup de temps. À titre d'exemple, lors de mon stage une segmentation fut lancée sur un grand nombre de noyaux et a pris plusieurs semaines pour aboutir. Il était donc nécessaire de réduire ce temps pour obtenir des résultats plus rapidement. De plus, le temps de traitement augmente avec la taille des images. NucleusJ peut être amené à traiter des noyaux de cellules animales de la même manière qu'il traite les noyaux de la plante Arabidopsis Thaliana. Ainsi un traitement sur des noyaux de cellules animales prendra beaucoup plus de temps qu'un traitement sur des cellules d'Arabidopsis Thaliana car les images sont plus grosses, le calcul se faisant sur l'ensemble les pixels.

a) Optimisation du calcul d'enveloppe convexe

La segmentation d'un noyau est un processus composé de 2 étapes : dans un premier temps les pixels sont classés en utilisant un seuil calculé avec la méthode d'Otsu comme pour l'autocrop. Cette étape permet d'obtenir un masque 3D du noyau mais l'intensité n'est pas uniforme dans le noyau : il s'y trouve notamment une grande zone sombre qui correspond à un sous compartiment du noyau appelé le nucléole***** (Figure 3). Ces zones d'intensité plus faible créent des trous dans les masques c'est pourquoi il est nécessaire d'effectuer un autre traitement après la première segmentation afin de les combler.

Pour représenter correctement le noyau à partir du noyau « troué » un algorithme d'enveloppe convexe est utilisé (en 3D). L'enveloppe convexe correspond à l'ensemble le plus petit qui contient tous les pixels blancs de l'image. Son principe est comparable à celui d'une zone délimitée par un élastique qui englobe tous les points en se contractant au maximum.

L'algorithme mis en place pour faire cette opération est l'algorithme de « gift wrapping » (littéralement « emballage de cadeau ») aussi appelé « Marche de Jarvis ».

Ce calcul de l'enveloppe convexe était le calcul le plus long de tous les traitements effectués dans NucleusJ, c'est donc cette partie que j'ai d'abord cherchée à optimiser.

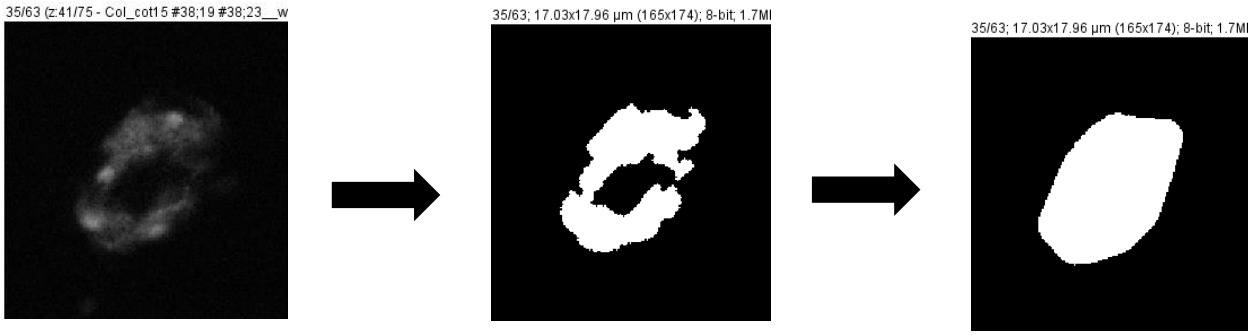

Figure 16 : Différentes étapes de la segmentation visibles sur le plan 35 d'une image de noyau. Le nucléole (au centre du noyau) créer un trou dans la l'image segmentée par la méthode Otsu et certains bords plus sombre entraînent des trous dans l'enveloppe du noyau. Le gift wrapping résout ces problèmes.

Pour le type de noyau de la

Figure *16*, le temps pris par la segmentation est d'environ **1 seconde par noyau** alors que le gift wrapping prend entre **10 et 15 minutes par noyau.**

i. La marche de Jarvis

Le principe de la marche de Jarvis est simple : prendre un point initial qui est à une des extrémités (le plus en haut, en bas, à gauche ou à droite) puis parcourir tous les points restants en trouvant à chaque fois le point suivant qui est celui le plus incliné vers « la gauche » c'est-à-dire le point dont l'angle polaire et le plus petit par rapport au point précédent. On ajoute le point trouvé à l'ensemble de l'enveloppe convexe et on s'arrête lorsque l'on retrouve le point initial. Ce principe est illustré sur la Figure 17.

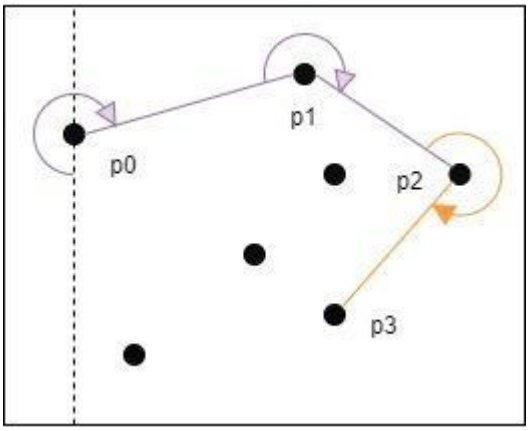

Figure 17 : Exemple d'une étape de l'algorithme de la marche de Jarvis.

L'angle initial est l'angle le plus à gauche (p0).

Ici l'angle formé avec p3 est l'angle polaire minimal c'est donc p3 qui va être ajouter à l'enveloppe convexe.

L'algorithme est utilisé en 3D en appliquant la marche de Jarvis à chaque plan de chaque paire d'axes, c'est-à-dire chaque plan de XY, de XZ et de YZ. Ensuite, les résultats sont combinés pour reconstruire une enveloppe convexe en 3 dimensions.

La lenteur de l'algorithme vient du fait qu'il traite, pour chaque plan de l'image, un grand nombre de points (tous les points de la bordure du noyau) pour lesquels il calcule à chaque fois un angle. Un rayon de recherche de point est défini au préalable pour éviter de tous les parcourir à chaque sommet.

ii. Première idée : utilisation d'un plugin externe

La première idée pour réduire le temps de calcul était d'utiliser un autre plugin de Fiji qui permettait de calculer les points extrêmes d'une enveloppe convexe.

Ce plugin open source appelé « 3D Convex Hull » permettait d'obtenir l'ensemble des sommets extrêmes du noyau en 3D de manière quasi-instantanée. L'idée était d'utiliser cette fonction du plugin entre la segmentation Otsu et le calcul de l'enveloppe convexe pour réduire le nombre de points parcourus et ainsi réduire le temps de calcul (*Figure 18*).

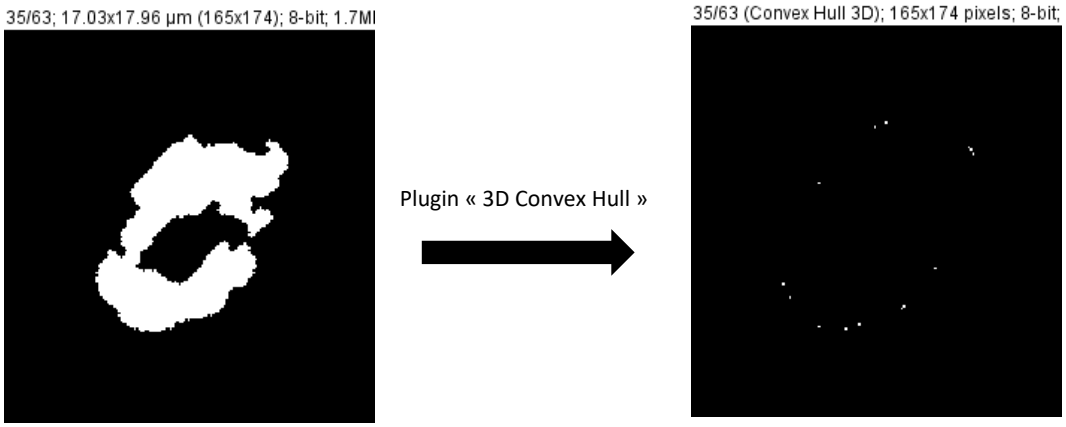

Figure 18 : Exemple de résultat de l'utilisation du plugin « 3D Convex Hull » sur une image segmentée par Otsu. On obtient bien des sommets à partir desquels on pourrait représenter l'enveloppe convexe du noyau.

J'ai testé cette idée en adaptant l'algorithme de gift wrapping. J'ai dû faire 3 principaux changements pour la mettre en pratique :

- La **phase de recherche des pixels en bordure**, c'est-à-dire des noyaux en contact avec des pixels noirs, qui était utilisée pour récupérer un ensemble de point sur lesquels appliquer la marche de Jarvis a dû être enlevée. En effet, ne restant plus que les points des sommets ils suffit de sélectionner l'ensemble des points du plan.
- La **gestion des composantes connexes** a été négligée. L'algorithme précédent détectait les îlots de points isolés du reste du noyau et selon leur taille choisissait de les intégrer ou non (voir Figure 19).
- Le **rayon limite de recherche** a été enlevé puisque les sommets peuvent être très éloignés entre eux. Cela a pour impact d'augmenter le temps de calcul mais la quantité de sommets étant limitée, le changement reste intéressant pour réduire le temps de calcul.

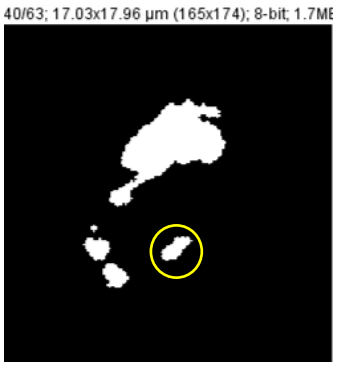

Figure 19 : Une coupe de noyau d'une image segmentée par la méthode Otsu avec des composantes connexes (exemple entouré en jaune).

Après ces changements j'ai pu lancer un calcul d'enveloppe convexe sur un ensemble de sommets calculés avec le plugin 3D Convex Hull. Le temps pris par le traitement sur un noyau était réduit (entre 3-5 minutes) néanmoins les résultats n'étaient pas convaincants (Figure 20).

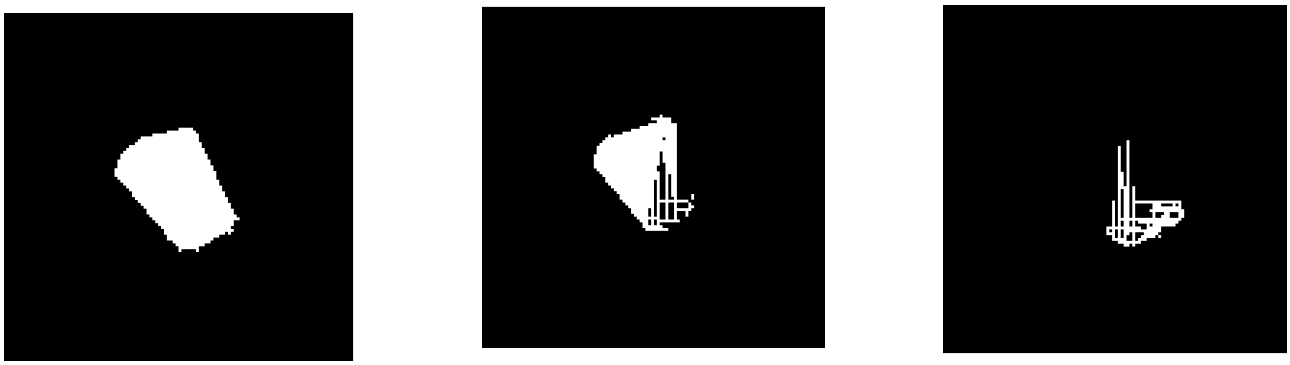

Figure 20 : Exemple de coupes d'une image segmentée avec la méthode utilisant 3D Convex Hull. Certaines profondeurs de l'image laissaient apparaître des trous dans l'enveloppe convexe. Ces trous apparaissaient après la phase d'assemblage des 3 paires de dimensions XY, XZ et YZ, certainement car l'algorithme ne parvenait pas à reconstruire une enveloppe convexe 3D avec la nouvelle méthode qui utilisait seulement quelques points pour interpoler le reste de l'enveloppe du noyau.

Cette première idée posait des problèmes supplémentaires et, en plus de ça, nécessitait l'ajout et l'utilisation du code source de l'intégralité d'un plugin extérieur (une dizaine de fichiers). J'ai donc décidé de trouver une solution alternative au problème de lenteur du gift wrapping.

iii. Deuxième idée : parcours de Graham

Je me suis intéressé à la nature de l'algorithme de calcul d'enveloppe convexe en lui-même. En effet, l'algorithme de gift wrapping a été implémenté au début du développement de NucleusJ, néanmoins, en faisant des recherches, on trouve facilement d'autres algorithmes de calcul d'enveloppe convexe reconnus.

Si on s'intéresse à la complexité algorithmique de la marche de Jarvis on trouve une complexité de $O(nh)$ où n est l'ensemble total de points et h l'ensemble des points de l'enveloppe convexe résultat (ce qui implique que le temps pris par l'algorithme dépend du résultat).

Quand on parle de la complexité d'un algorithme, on cherche généralement à étudier le temps pris par celui-ci dans « le pire des cas ». On peut alors obtenir une fonction de la taille d'entrée (par exemple le nombre de pixels d'un noyau) pour représenter le temps pris par un algorithme. L'intérêt est d'estimer la rapidité d'un algorithme indépendamment des facteurs matériels.

Afin de comparer et de trouver une meilleure alternative, je me suis intéressé à un autre algorithme : le parcours de Graham.

Dans cet algorithme, on cherche d'abord le point le plus en bas à gauche qui servira de point initial. L'étape suivante est le tri de l'ensemble des points en fonction de l'angle qu'ils forment avec le point initial et l'axe des abscisses. On parcourt ensuite les points dans l'ordre, comme illustré sur la Figure 21.

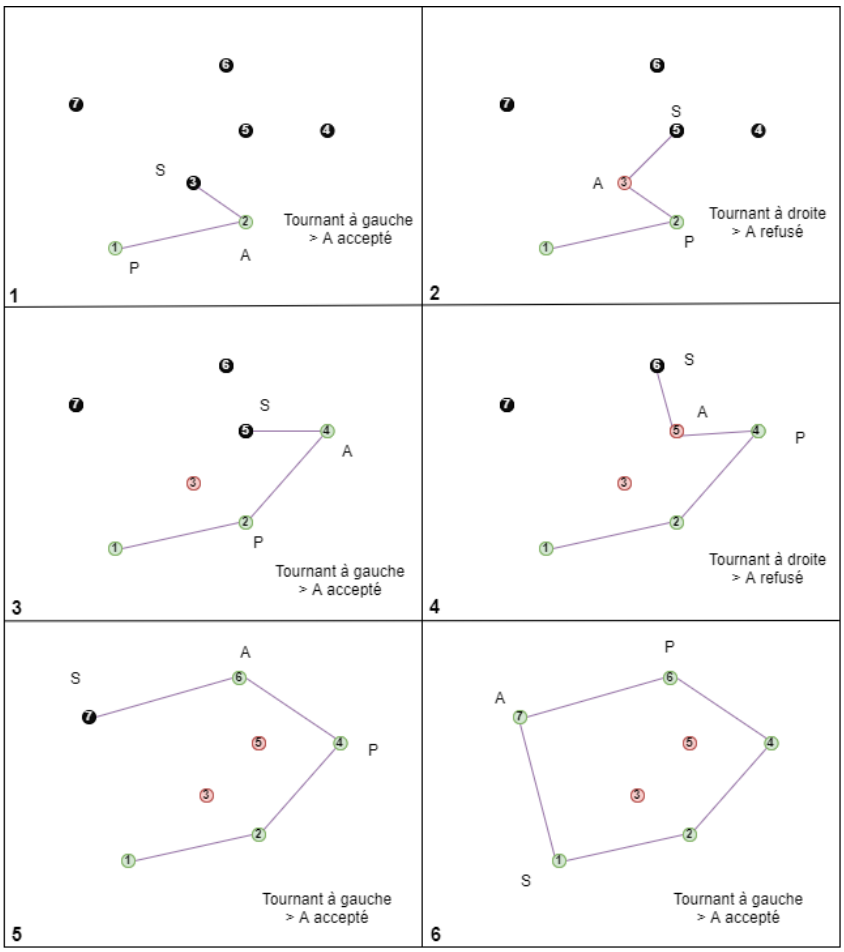

Figure 21 : Visualisation du parcours de Graham.

Tous les points sont numérotés (triés) en fonction de l'angle formés avec le point initial (le 1) et l'axe horizontal.

On parcourt l'ensemble trié en gardant toujours 3 points tout en regardant l'angle qu'ils forment. Ces points sont : le précédent (P), l'actuel (A) et le suivant (S).

Si l'angle forme un tournant à droite (étapes 2 et 4) alors on refuse le point courant.

Si l'angle forme un tournant à gauche (étapes 1, 3, 5 et 6) alors on accepte le point courant.

On s'arrête une fois qu'on retrouve le point initial.

La complexité de cet algorithme est liée principalement au tri initial des points selon l'angle qu'ils forment avec la droite horizontale passant par le point initial. Les étapes de recherche du point initial et de l'acception ou du refus des points ayant toutes les 2 une complexité de *O(n)* où *n* est le nombre de points, c'est le tri utilisé qui définit la complexité de l'algorithme. Le tri possède une complexité de *O(nlog(n))*. Ainsi l'algorithme de du parcours de Graham à une complexité *O(nlog(n))* contre *O(nh)* pour la marche de Jarvis.

Le gift-wrapping est plus rapide que le parcours de Graham uniquement lorsque le nombre *h* de points composant l'enveloppe convexe finale est inférieur à *log(n)*.

Il existe un algorithme de calcul d'enveloppe convexe appelé l'algorithme de Chan qui mélange la marche de Jarvis et le parcours de Graham et permet d'obtenir une complexité de O(nlog(h)) surpassant ainsi les 2 autres algorithmes. Je n'ai pas cherché à l'implémenter car cet algorithme est plus compliqué à mettre en place et ces implémentations souvent beaucoup plus complexes que pour le parcours de Graham.

J'ai décidé d'essayer d'utiliser le parcours de Graham à la place de la marche de Jarvis. Pour faire cela sans perdre trop de temps j'ai cherché une implémentation Java open source et donc librement utilisable et modifiable. J'ai trouvé une implémentation simple qui correspondait à ces critères (https://github.com/bkiers/GrahamScan), puisque sous une licence MIT*, sur GitHub (un système de gestion de maintenance collaboratif similaire à GitLab).

Cette implémentation faisait autour de 200 lignes (une unique classe), était particulièrement compréhensible et bien documentée. De plus, elle utilisait pour le tri des points qui est la partie la plus coûteuse du parcours de Graham, la collection java *TreeMap* et implémentait de manière interne un arbre de recherche binaire* dans lequel étaient insérés, et donc automatiquement triés, les points. La complexité d'ajout dans un arbre est de *O(log(n))* où *n* est le nombre de points dans l'arbre, comme nous insérons

chaque point cela revient à un algorithme de tri avec une complexité de *O(nlog(n))*. Cette façon de faire à l'avantage d'être concise en utilisant les fonctionnalités Java.

J'ai adapté l'algorithme à notre projet et aux objets que nous manipulons mais j'ai gardé l'idée de base de notre algorithme qui consistait à assembler les plans de chaque dimension après avoir appliqué l'algorithme sur chacun d'entre eux.

Le résultat fut très satisfaisant puisque, sur les noyaux testés, le temps de calcul est passé d'environ **10-15 minutes** par noyau à environ **0.2 secondes**.

iv. Vérification de l'amélioration

J'ai donc intégré cette méthode à la place du gift-wrapping pour constater la différence entre les 2 algorithmes.

Pour vérifier la similitude des résultats j'ai procédé de 2 manières différentes mais complémentaires que j'ai appliquées sur un ensemble de 654 noyaux. J'ai d'abord créé des tests permettant de comparer les volumes pour le masque du noyau obtenus après une segmentation.

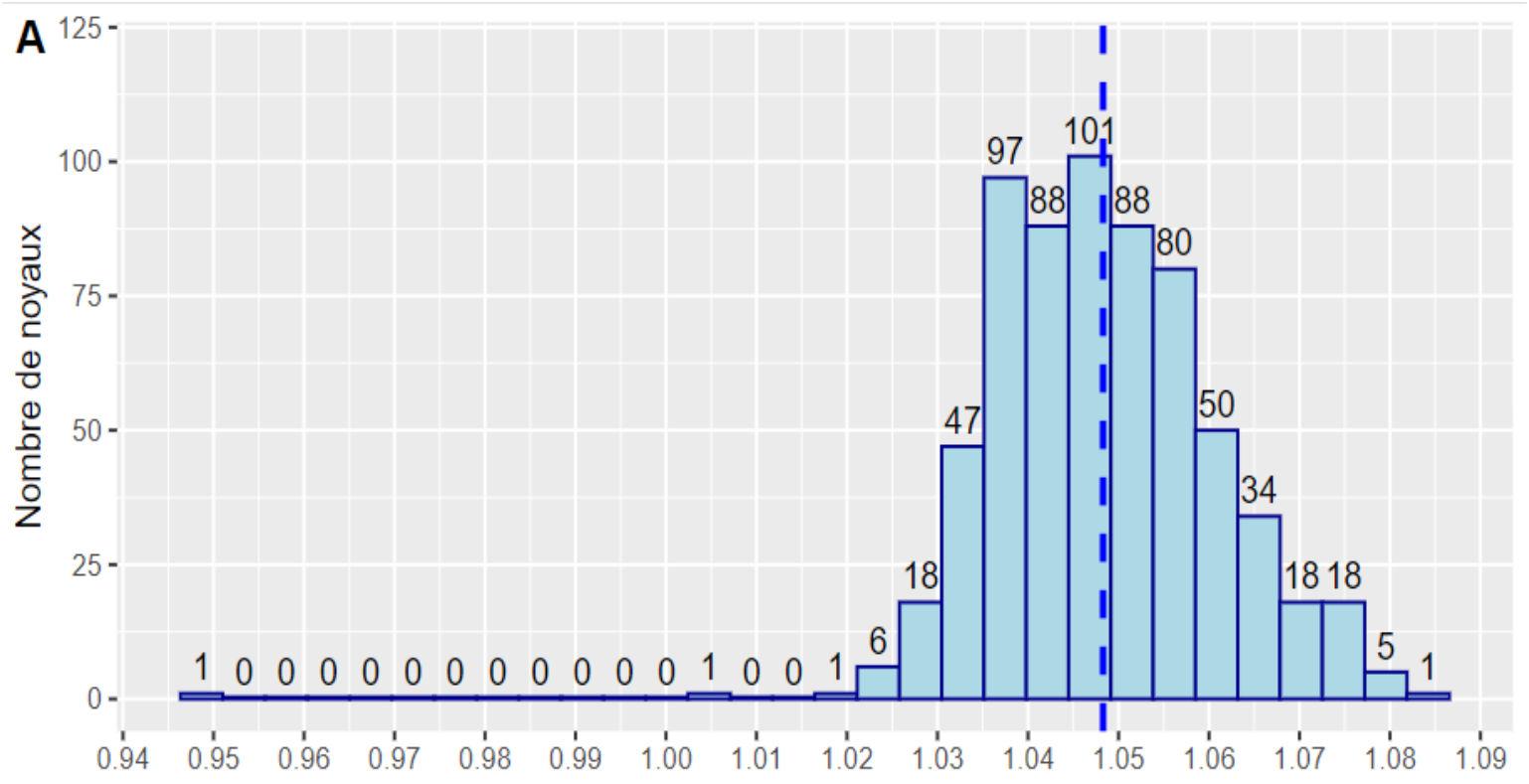

Figure 22 : Répartition du rapport entre le volume obtenu avec le gift wrapping et le volume obtenu avec le parcours de Graham (axe des abscisses). Les ratios sont majoritairement regroupés autour de 1.05 au niveau de la médiane donc on retrouve une différence de volume d'environ 5 % qui reste négligeable.

J'ai également utilisé les tests fonctionnels que j'avais réalisés auparavant pour comparer les différences exactes entre les images segmentées (Figure 23).

Image	Volume Gift/Volume Graham	Pourcentage de différence
2018051_1527146281.346_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_Q2_2_C0.tif	1.0340	4.41 %
2018051_1527150440.986_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_R1_1_C0.tif	1.0321	6.07 %
2018051_1527149304.236_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_M3_28_C0.tif	1.0530	6.18 %
2018051_1527153427.186_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_S3_20_C0.tif	1.0424	6.52 %
2018051_1527154284.426_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_W3_7_C0.tif	1.0614	7.10 %
2018051_1527149020.746_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_M2_2_C0.tif	1.0643	7.64 %
2018051_1527153998.156_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_W2_21_C0.tif	1.0432	5.11 %
2018051_1527147448.346_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_V3_15_C0.tif	1.0429	5.87 %
2018051_1527152002.426_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_O1_5_C0.tif	1.0453	6.32 %
2018051_1527156162.66_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_T1_4_C0.tif	1.0388	5.69 %
2018051_1527148737.316_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_M1_10_C0.tif	1.0528	6.86 %
2018051_1527149304.236_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_M3_6_C0.tif	1.0649	6.96 %
2018051_1527147448.346_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_V3_16_C0.tif	1.0466	5.43 %
2018051_1527153712.216_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_W1_1_C0.tif	1.0855	10.36 %
2018051_1527156162.66_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_T1_19_C0.tif	1.0754	8.30 %
2018051_1527155312.146_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_P1_0_C0.tif	1.0649	8.25 %
2018051_1527149587.586_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_N1_7_C0.tif	1.0496	6.15 %
2018051_1527145991.696_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_Q1_1_C0.tif	1.0551	6.27 %
2018051_1527146281.346_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_Q2_5_C0.tif	1.3813	38.14 %
2018051_1527149587.586_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_N1_3_C0.tif	1.0800	8.72 %

Figure 23 : Tableau de résultats des tests de différences de résultat entre les 2 algorithmes sur 20 noyaux parmi les 654 noyaux.

Si on s'intéresse plus précisément aux résultats obtenus avec les 2 méthodes, ils sont présentés dans la Figure 23. La 1ère colonne correspond au nom du fichier contenant le noyau. La 2ème contient le rapport entre le volume du noyau segmenté par gift wrapping et le noyau segmenté par le parcours de Graham. La 3ème colonne contient, elle, le pourcentage de différence entre les deux noyaux qui a été calculée grâce à la méthode vue dans la partie sur les tests fonctionnels.

En ce qui concerne le rapport entre les volumes, on trouve des valeurs légèrement au-dessus de 1. Cela signifie que les noyaux ont des volumes très similaires même si les noyaux segmentés par gift-wrapping sont légèrement plus volumineux. On constate qu'il n'y a aucune valeur avec plus de 10% de différence de volume.

De plus, la majorité des noyaux possède moins de 10 % de différence entre les 2 versions de la segmentation. En effet, on trouve environ 90 noyaux qui ont plus de 10% de différence. On peut alors étudier les raisons de cette différence en examinant visuellement les noyaux en question (Figure 24).

Par exemple pour la ligne :

2018051_1527146281.346_Ath_Col0--KAKU4-wt--CRWN1-wt--CRWN4-wt_Cot_J13_STD_FIXE_H258_Q2_5_C0.tif	1.3813	38.14 %

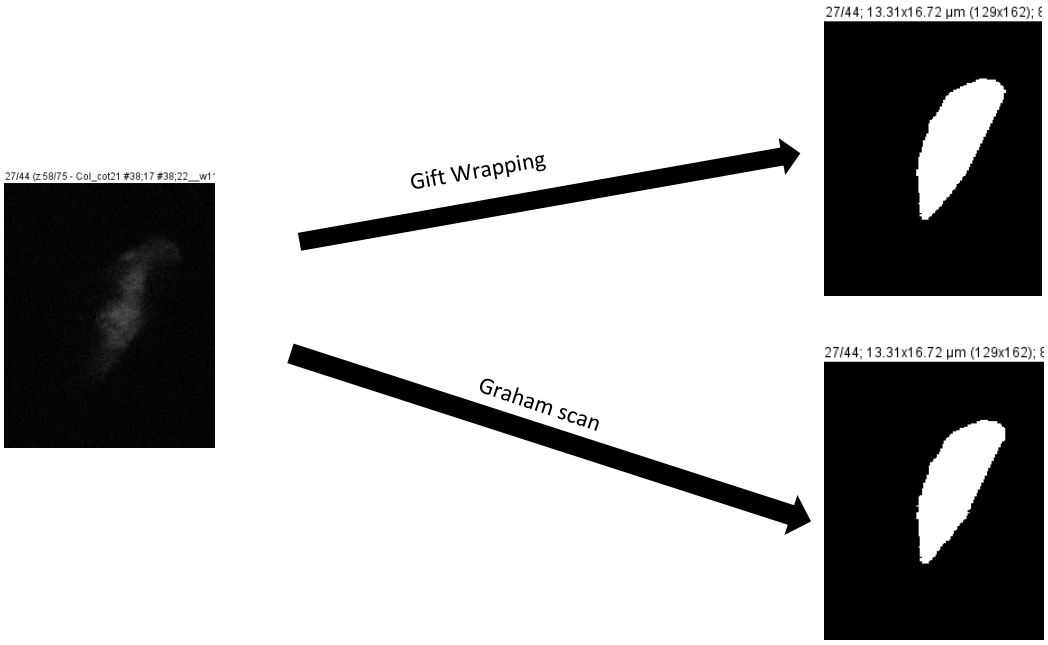

Figure 24 : Résultat du gift wrapping et du graham scan pour la même coupe d'un noyau.
Le noyau brut à gauche est un noyau particulier, il est relativement peu visible : on devine difficilement ses bordures.

On peut considérer que les résultats sont tous les deux convenables car d'une part les différences sont minimes et proviennent essentiellement de quelques différences de pixels sur chaque coupe du noyau comme on peut le voir sur la Figure 24. D'autre part, pour les petits noyaux une différence de quelques pixels peut représenter un grand pourcentage du volume c'est pourquoi on trouve des noyaux avec plus de 10% de différence entre les 2 versions.

Après avoir exposé les résultats aux autres membres du projet, la nouvelle méthode a été adoptée car on ne peut pas dire avec certitude lequel des 2 algorithmes est le plus proche de la réalité. Par exemples, les différentes coupes en profondeur peuvent ne pas être assez rapprochées pour laisser voir une invagination du noyau. Néanmoins ces cas d'incertitudes sont négligeables car on cherche alors simplement à obtenir une enveloppe convexe pour étudier la disposition globale de la chromatine dans les cellules.

Ce changement d'algorithme est un changement majeur permettant de gagner un temps de calcul considérable et donc une nouvelle version du plugin NucleusJ a été délivrée après mes réalisations.

b) Parallélisation

J'ai ensuite adopté une nouvelle approche pour optimiser les traitements de NucleusJ car je me suis intéressé à la parallélisation de l'autocrop et de la segmentation.

À mon arrivée, tous les traitements de NucleusJ étaient réalisés de manière séquentielle. Lorsqu'on lançait une segmentation, le programme traitait les images les unes après les autres. Mon but fut donc de paralléliser les traitements pour pouvoir traiter les images en même temps et ainsi réduire le temps de calcul global en utilisant plusieurs cœurs du processeur.

Un "thread" désigne une file d'exécution qui réalise des opérations séquentiellement et qui peut être lancé en parallèle d'une autre file d'exécution.

Pour mettre en place la parallélisation j'ai utilisé la gestion des threads mise à disposition par le langage Java. En effet, le langage fournit un ensemble de classes et de méthodes permettant de manipuler les threads et de paralléliser facilement du code pour utiliser plusieurs processeurs.

Cette optimisation est d'autant plus pertinente que le GReD dispose de serveurs de calcul avec un grand nombre de processeurs disponibles. À titre d'exemple, Bowser, le serveur sur lequel j'ai effectué la plupart de mes tests, possède 64 cœurs ce qui signifie qu'il est techniquement possible d'avoir 64 threads lancés et efficaces simultanément.

i. Paralléliser à quel niveau ?

Lorsque l'on cherche à paralléliser du code il faut prendre en compte le coût de la création, de la gestion et de la coordination des threads. Il est plus rentable de créer des threads qui travaillent longtemps que de gérer régulièrement des petites tâches. De même, plus il y a de mémoire partagée, plus il y aura une queue pour écrire dedans quand c'est nécessaire.

Inversement, si on avait des considérations de taille de mémoire à ne pas dépasser impérativement, il faudrait peut-être paralléliser dans des boucles internes : traiter n images avec n threads implique de stocker n images en mémoire, tandis que traiter 1 image avec n threads en espérant la traiter n fois plus vite n'augmente pas la quantité de mémoire utilisée.

Nous avons choisi de paralléliser en divisant les threads en fonction des fichiers pour traiter n images avec n threads, cela pour plusieurs raisons :

- C'est plus facile à programmer puisqu'il n'est pas nécessaire de diviser le traitement d'une unique image en plusieurs étapes indépendantes, les fichiers d'images étant déjà indépendants entre eux.
- C'est efficace car on utilise des longues tâches avec peu de variables communes et que l'on traite généralement un nombre suffisant d'images.
- C'est acceptable au niveau de la mémoire. Chaque image ne fait pas une taille démesurée et les ordinateurs performants auront en général une RAM adaptée au traitement du nombre d'images qu'ils peuvent faire simultanément.

ii. Utilisation des threads Java

La première parallélisation de la segmentation fut mise en place en 1 ligne de code supplémentaire.

Il s'agissait d'utiliser *Java Parallel Streams* qui est une fonctionnalité apparue à la version 8 de Java et qui permet d'augmenter les performances grâce aux threads. La principale contrainte dans l'utilisation des threads Java est le fait qu'on ne puisse pas contrôler l'ordre d'exécution des threads. Dans le cas de NucleusJ les images sont indépendantes les unes des autres donc elles peuvent être traitées dans n'importe quel ordre.

L'ajout initial ressemblait à cela :

```
directoryInput.listFiles().parallelStream().forEach(v -> { /* Process current file */ });
```

La méthode *directoryInput.listFiles()* retourne une liste de fichiers qui est ensuite traitée parallèlement en utilisant *parallelStream* pour chacun des fichiers (*forEach*).

Néanmoins, cette méthode seule ne suffit pas car il faut prendre en compte la « thread-safety » du code.

La thread-safety correspond à la capacité du code à être exécuté simultanément par plusieurs threads dans le même espace mémoire. C'est une notion importante de la programmation concurrente et elle nécessite

la mise en place de structures et de solutions plus ou moins complexes qui vise à éviter des erreurs que l'on appelle « race conditions* » ou « situation de concurrence ».

Dans le cadre de la segmentation, une variable commune était manipulée par tous les threads : il s'agissait du fichier de résultat global, qui est censé être rempli au fur et à mesure de l'exécution. Il est représenté par une chaîne de caractères qui est construite au fil de l'exécution.

Il fallait trouver une solution pour éviter que plusieurs threads n'accèdent à cette variable en même temps. De plus, il est préférable que le traitement répété d'un même dossier donne un fichier de résultat toujours ordonné de la même façon, or avec la parallélisation cela n'est pas garanti : il était donc nécessaire d'intervenir pour trier les résultats.

Pour résoudre ces 2 problèmes j'ai utilisé un autre objet Java appartenant au package *java.util.concurrent* : il s'agit d'une collection permettant de stocker des données et d'y accéder depuis plusieurs threads de manière sécurisé grâce à un système géré par Java.

Cette collection est une table de hachage ce qui permet de donner une clé et une valeur à chaque élément de la collection. Dans notre cas la valeur est la ligne à ajouter au fichier d'information et la clé est soit le nom du fichier soit l'ID de l'image sur OMERO, ceci permet ensuite de remplir le fichier dans l'ordre en récupérant après le traitement chaque ligne à partir de sa clé. On garde alors le même ordre dans le fichier de traitement à chaque exécution.

iii. ThreadPool & ExecutorService

Java Parallel Streams utilise un *ExecutorService*. Il s'agit d'une interface qui fournit des méthodes pour gérer l'exécution de tâches sur plusieurs threads.

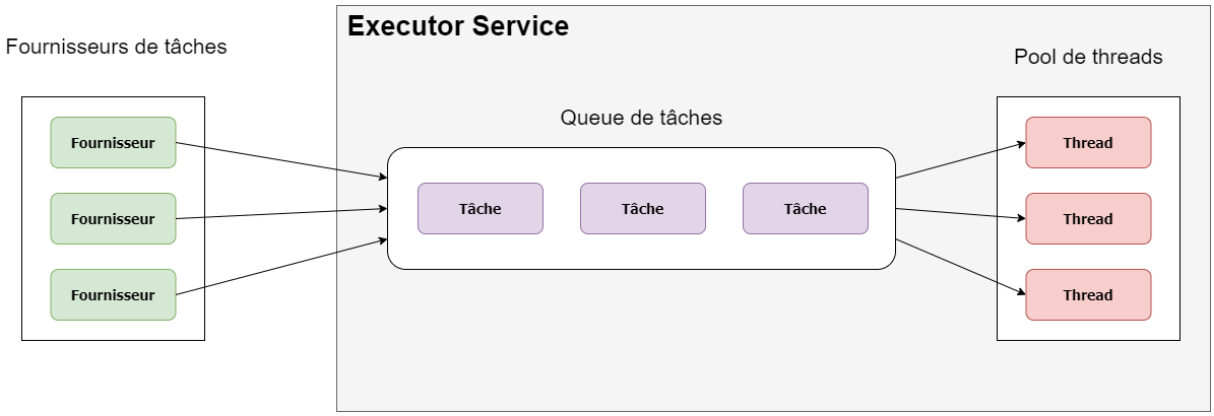

Figure 25 : Fonctionnement d'un ExecutorService. Un ExecutorService reçoit des tâches qui lui sont soumises. En Java ces tâches peuvent être représentées par des objets « Runnable ». (Ou « Callable » et qui fonctionnent comme Runnable mais permettent en plus de retourner une valeur et de lancer une exception si besoin).

Ces tâches sont stockées dans une queue qui est utilisée pour répartir les tâches sur les threads disponibles dans le « pool » de threads, ou les conserver le temps qu'un thread se libère (Figure 25).

Il existe différentes implémentations *d'ExecutorService*. L'implémentation de celui utilisé par la méthode *parallelStream*() est un *ForkJoinPool* dont le principe est de diviser une grosse tâche en sous-tâches (fork) sur plusieurs threads puis de fusionner les résultats une fois obtenus (join).

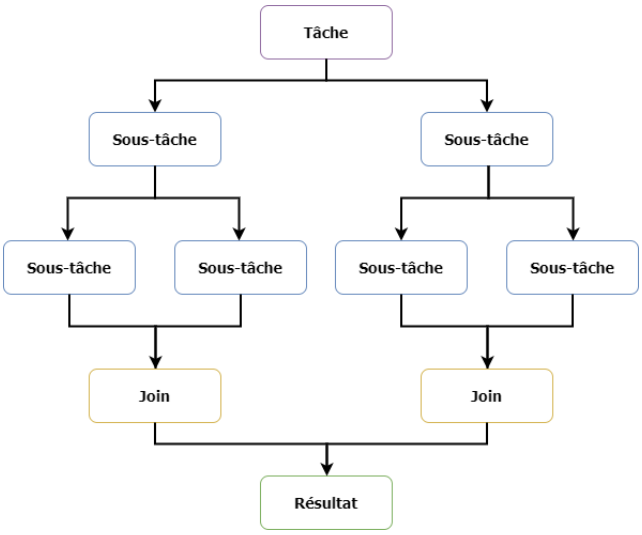

Figure 26 : Schématisation du fonctionnement du ForkJoinPool. On retrouve les 2 étapes clés du ForkJoinPool, le fork c'est-à-dire la division des tâches (en bleu) et le join qui refusionne les sous-tâches (en jaune).

Cette implémentation fonctionne en divisant une tâche (Figure 26) or dans le cas des traitements de NucleusJ nous souhaitons traiter des images indépendantes en même temps et il n'y a pas lieu de vouloir fusionner le résultat de ces images car elles sont simplement enregistrées après le traitement.

De plus, on ne peut pas choisir le nombre de threads à utiliser avec cette façon de faire et cela pourrait s'avérer utile notamment lorsqu'on lancera des traitements sur un serveur de calcul.

J'ai donc opté pour une autre solution qui nécessitait plus de lignes de codes. J'ai utilisé une implémentation *d'ExecutorService* appelée *FixedThreadPoolExecutor*. Cette implémentation nécessite uniquement 1 paramètre qui est le nombre de threads à utiliser. Son principe est de garder un nombre de threads fixe pendant toute l'exécution : c'est une des versions les plus simples d'un *ExecutorService* et elle est adaptée pour la plupart des cas où on souhaite paralléliser des tâches indépendantes les unes des autres.

Il existe d'autres implémentations d'ExecutorService en plus de ForkJoinPoolExecutor et de FixedThreadPoolExecutor. Beaucoup font varier des paramètres comme la taille du pool de thread, la taille maximum lorsque des threads peuvent être créer en cours d'exécution ou encore le temps pendant lequel garder un thread en vie, lorsqu'il n'est plus utilisé, pour qu'il puisse être récupéré et réutilisé.

iv. Résultats du multithreading

Après avoir mis en place le multithreading sur les images pendant une segmentation, j'ai effectué des tests pour comparer le gain de performance. J'ai écrit des tests que j'ai lancés en utilisant un script Bash sur le serveur Bowser. De plus, afin de pouvoir me déconnecter du serveur sans stopper mes tests j'ai utilisé un outil appelé Screen qui est un multiplexeur de terminaux. Il me permettait de me détacher d'un terminal sans le fermer afin d'y revenir plus tard quand l'exécution serait terminée : cela m'a été très utile car mes tests ont pris plusieurs heures à s'exécuter.

Les tests écrits remplissaient un fichier CSV (Figure 27) au fur et à mesure afin que je puisse ensuite extraire et visualiser les données plus facilement :

Noyaux	1 thread	4 threads	8 threads	16 threads
30 noyaux	149 s	56 s	46 s	40 s
z60 noyaux	289 s	88 s	66 s	53 s
200 noyaux	1110 s	325 s	195 s	152 s
600 noyaux	3350 s	1110 s	509 s	351 s

Figure 27 : Tableau du temps pris par la segmentation en fonction du nombre de noyaux traités et du nombre de threads utilisés.

Pour obtenir le tableau de la Figure 27 mes tests ont pris environ 2 heures. Afin de voir plus clairement les données, j'ai construit un graphique relatant l'évolution du temps de traitement :

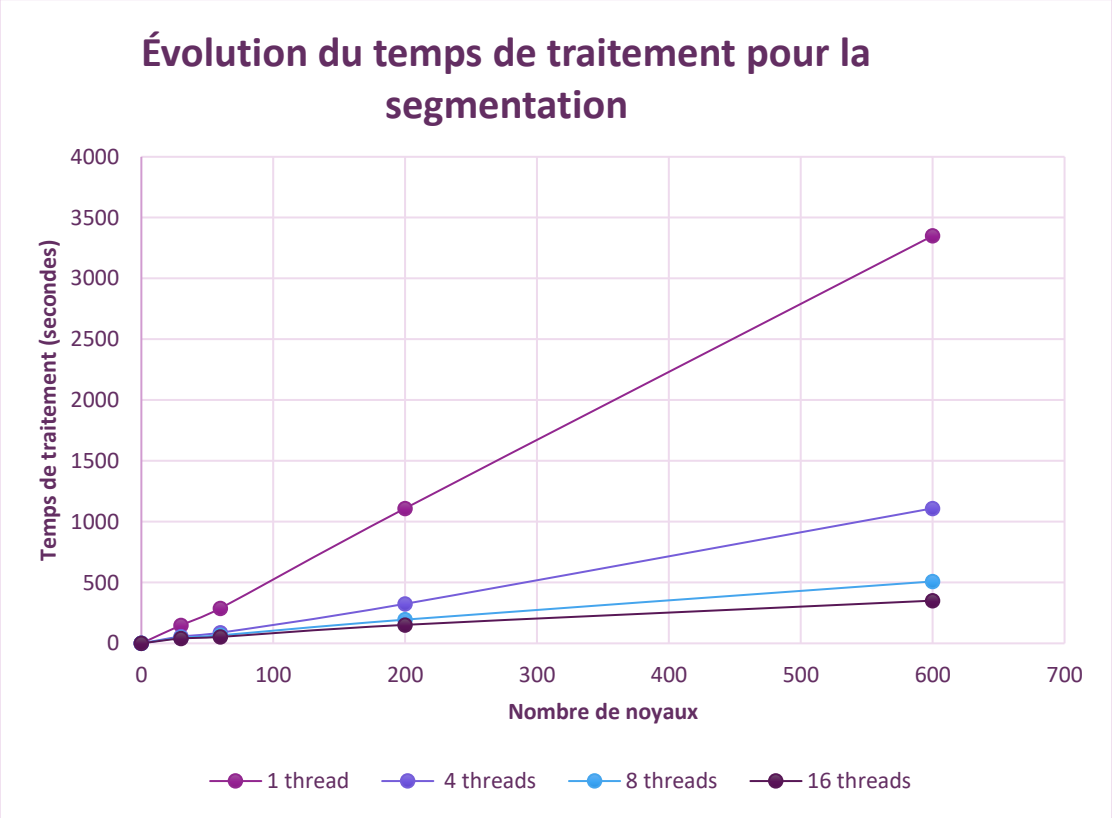

Figure 28 : Courbe de l'évolution du temps de traitement en fonction du nombre de noyaux pour chaque taille de pool de threads testée.

On observe très clairement sur la Figure 28 que le temps baisse considérablement lorsque l'on augmente le nombre de threads mais cette réduction est particulièrement visible avec un grand nombre de noyaux.

Cela s'explique par le fait que plus le nombre de threads est élevé plus le coût de création et de synchronisation de threads est élevé et donc la réduction du temps sur de petits nombres de noyaux est moins visible. De plus, notre parallélisation étant au niveau du traitement de chaque fichier, lorsqu'il y a seulement quelques fichiers avoir plus de threads que nécessaires n'est pas vraiment utile. Si je souhaite traiter 10 fichiers avec 16 threads il y aura 6 threads inactifs.

v. Parallélisation avec OMERO

La parallélisation des traitements sur des images issues d'OMERO met en lumière d'autres problématiques. Lorsque l'on utilise OMERO il faut prendre en considération le temps de récupération de l'image et le temps de mise en mémoire. Lorsque l'on parallélise le téléchargement d'images on risque de saturer la bande passante et donc d'influer sur les performances globales du programme. Pour éviter cela, il faut donc créer 2 pools de threads mis bout à bout, c'est-à-dire que les threads de l'un *ExecutorService* sont également des fournisseurs de tâches pour l'autre. Le premier *ExecutorService* est chargé de télécharger les images et ne contient qu'un seul thread. Le second est chargé de répartir les images obtenues sur les threads disponibles pour pouvoir les traiter.

vi. Problèmes d'importation

Dans le cadre de la parallélisation avec OMERO un problème est apparu qui permettait de mettre en avant un danger supplémentaire lié au multithreading. Comme mentionné précédemment, NucleusJ utilise une bibliothèque développée au sein du GReD qui a pour objectif de faciliter la connexion et l'utilisation d'OMERO.

Lorsque l'on parallélise, les images résultantes sont uploadées de manière simultanée. Après quelques tests j'ai remarqué que les images segmentées ne se trouvaient pas toujours dans le bon *dataset*. Certaines images segmentées uniquement avec la méthode Otsu se retrouvaient dans le *dataset* des images segmentées par l'algorithme de calcul d'enveloppe convexe et inversement.

Après vérification, j'ai établi que les paramètres d'importation passés dans le code de NucleusJ n'étaient pas en cause et que c'est un élément de la bibliothèque Simple-OMERO-Client qui n'était pas thread-safe.

La cause de ce bug était un problème de gestion de la mémoire simple mais qui a était mis en évidence par la parallélisation. Afin de se connecter, la bibliothèque Simple-OMERO-Client utilise l'API d'OMERO pour construire un objet de type *ImportConfig* qui représente la configuration nécessaire à l'importation de données. Cet objet contient entre autres le nom de l'utilisateur, l'adresse du serveur mais surtout l'identifiant et le type du conteneur cible c'est-à-dire l'un des 2 *datasets*. La référence de l'objet étant passée à plusieurs threads il y avait un problème de modification concurrente. Ainsi on peut visualiser la situation grâce au schéma mémoire de la Figure 29.

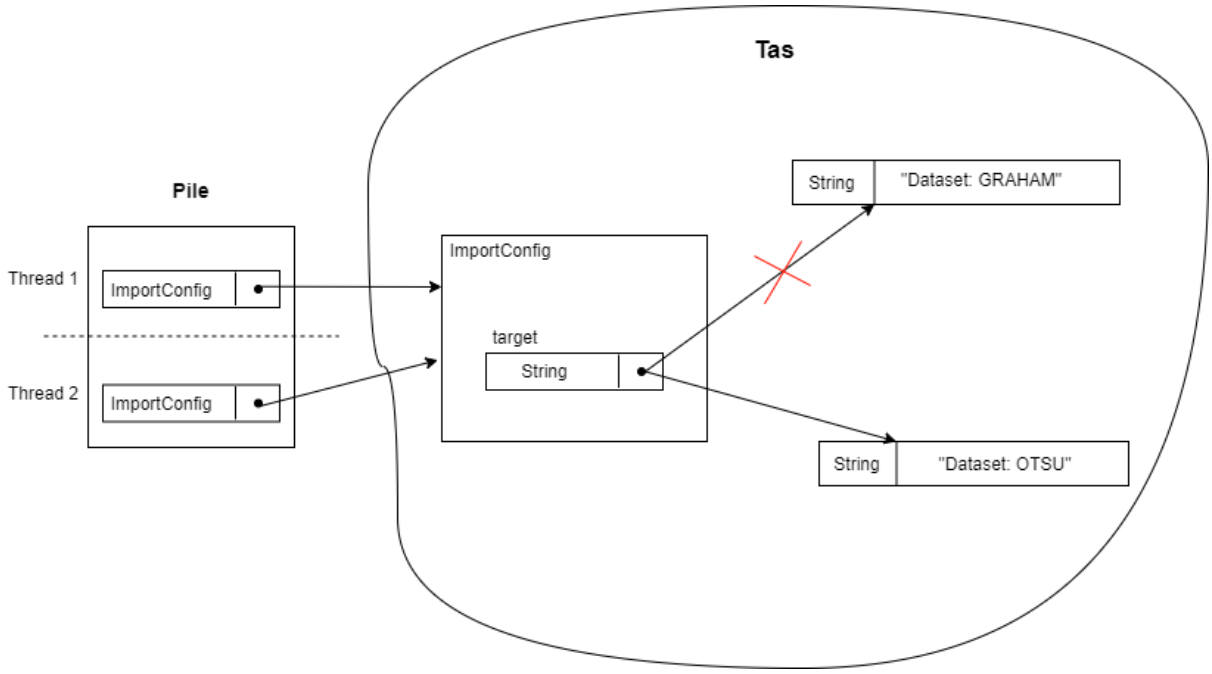

Figure 29 : Schéma de l'état de la mémoire lorsque 2 threads utilisent une copie simple d'un ImportConfig.

Sur la pile, chaque thread possède une référence vers l'objet de configuration d'importation : il est donc possible qu'un thread modifie la valeur de destination pendant qu'un autre thread lit la valeur, d'où les exportations qui semblaient se tromper de *dataset* sur OMERO.

Pour corriger ce problème il a donc été nécessaire de modifier le code de la bibliothèque Simple OMERO Client afin de faire une copie « profonde » de l'objet de configuration pour éviter que 2 threads accèdent et modifie le même objet en mémoire.

Ce bug a permis de voir l'impact que peut avoir la parallélisation lorsque le code sur lequel on l'utilise n'est pas avéré « thread-safe ».

vii. Optimisation autocrop

Pour finir l'optimisation j'ai également parallélisé l'autocrop en utilisant la même méthode que pour la segmentation que ce soit en local ou en utilisant OMERO.

L'autocrop après avoir calculé un seuil et trouvé les coordonnées des boxes contenant les noyaux va réaliser un rognage pour extraire chaque noyau et le copier dans un fichier isolé.

C'est cette dernière étape qui était la plus longue car elle nécessitait pour chaque noyau trouvé de recharger l'image avec les méthodes de la bibliothèque d'ImageJ afin de ne pas modifier l'image initiale. Pour éviter de perdre ce temps de réouverture, ImageJ fournit des méthodes pour dupliquer puis rogner l'image rapidement, ce léger changement a permis de gagner un temps considérable pour l'autocrop.

viii. Ajout de l'option pour les utilisateurs

Pour finir, j'ai ajouté des options permettant de changer le nombre de threads à utiliser lors d'une segmentation ou d'un autocrop. Cet ajout fut rapide car j'avais déjà compris le fonctionnement des points d'entrée de l'application et les différents appels aux méthodes.

Pour la ligne de commande (Figure 30) j'ai utilisé la structure déjà en place qui utilise la bibliothèque *apache.commons* plus précisément le package *apache.commons.cli*, qui fournit des outils pour créer facilement des lignes de commande avec des options (voir le lancement de NucleusJ partie III.a) »)

```
-th,--threads <arg>    Number of threads used to split image processing
                       (do not exceed the number of available CPUs  (=12
                       CPUs))
                       Default : 4 threads for several images (otherwise
                       1 thread for single image processing)
```

Figure 30 : Capture d'écran du résultat lors de l'exécution du fichier JAR en ligne de commande avec la commande -h pour obtenir de l'aide. Cette partie affiche la nouvelle option de nombre de thread et comment l'utiliser et conseil de ne pas dépasser le nombre de processeurs de l'ordinateurs tout en affichant quel est ce nombre (ici il y a 12 processeurs logiques).

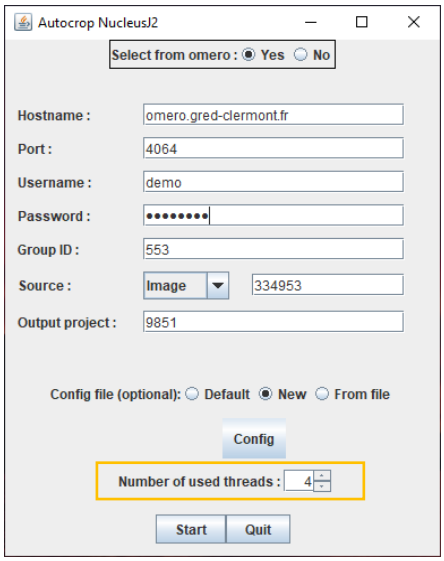

Figure 31 : Capture d'écran de l'interface graphique permettant de définir entre autres, le nombre de threads à utiliser (ici encadré en orange). Par défaut 4 threads sont sélectionnés sauf si la machine a moins de 4 processeurs dans ce cas le nombre de processeurs est présélectionner. De plus, le nombre de threads ne peut pas être inférieur à 1 ou supérieur au nombre de processeurs disponibles.

J'avais également restructuré l'interface de la Figure 31 pour pouvoir facilement ajouter un élément graphique sans avoir à restructurer à nouveau l'ensemble. J'ai donc facilement ajouté la même option.

IV. Bilan technique et prolongement

Lors de mon stage j'ai participé au développement du plugin NucleusJ et à son intégration avec OMERO.

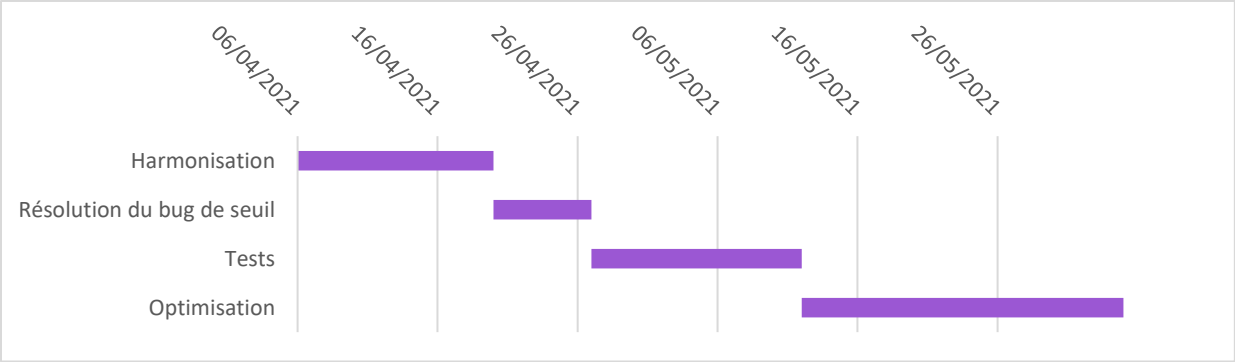

Figure 32 : Répartition des tâches durant le stage. Le temps pris par la période d'harmonisation comprend aussi le temps d'adaptation au projet et aux outils.

Comme on peut le voir sur la Figure 32, j'ai d'abord harmonisé les fonctionnalités du plugin en ajoutant à chaque mode d'utilisation les mêmes fonctionnalités. Il est maintenant possible de traiter une ou plusieurs images en local comme sur OMERO, que ce soit depuis la ligne de commande ou depuis Fiji, et d'obtenir des résultats similaires dans tous ces cas de figure pour l'autocrop et la segmentation des noyaux.

Pour donner suite à la résolution d'un bug, j'ai mis en place des tests fonctionnels pour permettre de vérifier la validité des résultats lors de mises à jour. Ces tests intégrés avec Maven jouent également un rôle dans l'intégration continue puisqu'ils permettent de détecter la potentielle régression du code ou l'apparition d'un problème au cours du développement de NucleusJ.

Enfin, la partie d'optimisation de mon stage a permis de réduire le temps pris par les traitements de NucleusJ. J'ai changé l'algorithme de calcul d'enveloppe convexe utilisé par l'algorithme du parcours de Graham. Cette première amélioration a permis de réduire le temps des segmentations qui prenait plusieurs jours auparavant à seulement quelques heures. Plus globalement, en parallélisant l'autocrop et la segmentation des noyaux j'ai pu encore réduire le temps de calcul global en permettant à l'utilisateur de choisir, en ligne de commande, et sur l'interface, le nombre de threads à utiliser pour diviser le temps de calcul notamment lorsque l'on traite une grande quantité d'images.

Il reste encore beaucoup de travail lié à NucleusJ, voici les principales tâches en perspective :

- NodeJ est un projet annexe à NucleusJ développé au sein du GReD et qui a pour but **l'analyse automatique des chromocentres** d'un noyau. Actuellement cette analyse nécessite une étape manuelle c'est pourquoi il serait intéressant de fusionner les 2 projets.
- En utilisant le serveur du mésocentre, il serait intéressant **d'analyser la performance** des calculs grâce aux outils mis à dispositions par l'entreprise Siliceum avec qui une collaboration est prévue.
- Encore beaucoup de petits **ajouts et corrections** permettraient d'améliorer l'utilisation du plugin comme : l'amélioration de certains affichages de Z-Projection, la gestion de plus de messages d'erreurs, la capacité de réaliser un autocrop à partir de coordonnées sur un autre canal de l'image (une autre coloration lors de l'acquisition) sans avoir à lancer le « vrai » autocrop sur OMERO, l'ajout d'un menu pour gérer les images FISH (fluorescence in situ hybridization) ou encore le traitement d'images en 2D…
- Enfin d'autres solutions sont explorées pour améliorer les traitements de NucleusJ, la plus importante est la piste de **l'intelligence artificielle** et plus précisément de l'utilisation du machine learning pour la segmentation et l'analyse des noyaux. Guillaume MOUGEOT, un doctorant, est actuellement en train de s'intéresser à ces nouvelles méthodes.

Conclusion

Ce stage au sein de l'institut GReD m'a permis de découvrir le monde de la recherche en biologie. Étant intéressé par les sciences, j'ai aimé appréhender les problématiques étudiées par les équipes notamment en assistant aux réunions ainsi qu'aux différentes présentations qui ont eu lieu et auxquelles j'ai moi-même participé pour présenter mon travail. J'ai donc beaucoup apprécié le côté bi-disciplinaire de mon stage.

Cette première expérience professionnelle m'a permis de mettre en pratique les connaissances acquises pendant le DUT notamment autour de la programmation orientée objet en Java, mais il m'a également permis d'utiliser des outils importants dans un cadre réel comme Maven ou encore GitLab dont l'utilisation est essentielle pour le projet collaboratif puisque j'ai effectué tout mon travail sur plusieurs branches avant de fusionner ces dernières avec le reste de NucleusJ. J'ai donc acquis des connaissances et compétences qui pourront s'avérer utiles dans le futur.

Enfin, au-delà des aspects scientifiques et techniques très enrichissants de mon stage, j'ai aussi travaillé dans un cadre particulièrement agréable où j'ai pu collaborer avec un doctorant en bio-informatique et des ingénieurs de recherche de domaines différents qui ont pu m'aider et me guider tout au long du stage. J'ai donc beaucoup appris en termes de travail en équipe et de collaboration.

Plus globalement l'ensemble des personnes travaillant à l'institut ainsi que les stagiaires qui étaient présents lors de mon stage ont rendu mon expérience très plaisante et m'ont aidé à m'intégrer facilement au sein du GReD.

Résumé en anglais

As a part of my studies in computer science at the IUT, I did an internship in a research center in Clermont-Ferrand called the GReD. This center, supported by the University Clermont Auvergne, the CNRS and the INSERM, studies genetics, reproduction, development as well as their deregulations. I joined team 3 which studies chromatin and nuclear dynamics in 3D.

In living cells, and especially in the cells of Arabidopsis Thaliana which is the plant used as a model by the team, the DNA is organized in the nucleus in a set of molecules, we call this whole “chromatin”. As the lab is interested in the organization of chromatin, a software called NucleusJ has been developed by the GReD. This software is a plugin meant to be used as a part of a bigger and well-known software in the field of biology called ImageJ used for image processing. Likewise, another tool called OMERO is used by biologists to gather and organize microscopic images in 2D or in 3D on a remote server on which you can annotate the images and use them for your studies. To use these services, a Java library was made to ease the use of the OMERO API and easily connect, export, and import images, this library is used by NucleusJ to handle images from OMERO.

I engaged in the Java development of NucleusJ in collaboration with a research engineer and a PhD student in bioinformatics. My first task was to harmonize the different ways of using NucleusJ, indeed it is possible to launch image processing from both command line and from an interface and in both cases, you can process local images or OMERO images. I made sure that the 2 main features for nuclear analysis, autocrop (which detect and crop a box around the nucleus found in 3D wide field images) and segmentation (create a mask for a single raw nucleus), were available in all cases. I also verified that we obtained similar results files.

As a follow-up to a bug that appeared due to an update of ImageJ, which is a dependency of the project, I worked on functional tests. These tests check the similarity of two results using several methods. Thanks to tools already in place, Maven and GitLab, the tests were integrated in continuous integration and now can detect potential code regression or the emergence of new bugs.

Lastly, I took part in the optimization of NucleusJ by first changing the algorithm used to compute a convex hull and which is used to get a better segmented nucleus. I changed the “gift wrapping” algorithm to the Graham scan algorithm to reduce the complexity and thus the time taken. I was able to further improve the performances by using multiple threads, and thus multiple processors, to process several images simultaneously. I also added an option to allow the user to define how many cores he wants NucleusJ to use. After my change, the overall computation time which took several days, even weeks sometimes, was reduced to only a few hours for a given image processing.

Lexique

Image grand champ : Désigne une image qui contient plusieurs objets à observer.

Segmentation d'image : La segmentation d'une image en traitement d'image consiste à rassembler des pixels entre eux en suivant certains critères définis. Ils forment alors des régions de l'image, par exemple cela peut permettre de séparer des objets du fond

Seuillage : Le seuillage d'image est une technique simple de binarisation d'image, elle consiste à transformer une image en niveau de gris en une image dont les valeurs de pixels ne peuvent avoir que la valeur 1 ou 0. On parle alors d'une image binaire ou image en noir et blanc.

Méthode Otsu : Algorithme de seuil qui calcule le seuil optimal qui sépare les 2 classes afin que leur variance intra-classe soit minimale.

Watershed : La segmentation par watershed ou « ligne de partage des eaux » en français désigne des méthodes de segmentation qui considèrent une image à niveaux de gris comme un relief topographique, dont on simule l'inondation.

Plugin : Un logiciel conçu pour être ajoutée à un autre logiciel afin d'apporter à ce dernier de nouvelles fonctionnalités

Machine virtuelle : Une machine virtuelle est un outil permettant de simuler le comportement d'un appareil informatique. On peut créer une machine virtuelle en définissant précisément les ressources matérielles allouées grâce à un émulateur.

Logiciel open source : Logiciel qui permet la libre redistribution, rend le libre l'accès au code source et autorise les modifications et l'utilisation du code.

L'intégration continue (CI) : un ensemble de pratiques qui consistent à tester et à consigner immédiatement tout changement fréquent et isolé lors de la fusion avec un référentiel de code afin d'éviter toute apparition de bug ou régression (retour en arrière du code).

Framework : Un framework désigne un ensemble d'outils et de composants logiciels qui est souvent sous la forme d'une bibliothèque logicielle.

Image binaire : Image dont les pixels ne peuvent avoir que 2 valeurs différentes (blanc et noir)

API : En informatique, une interface de programmation d'application ou interface de programmation applicative est un ensemble normalisé de classes, de méthodes, de fonctions et de constantes qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels.

Licence MIT : Une licence permissive courte et simple avec des conditions n'exigeant que la préservation des avis de droit d'auteur et de licence. Les œuvres sous licence, les modifications et les œuvres plus importantes peuvent être distribuées sous des conditions différentes et sans code source.

Chromocentres : Fragment de chromosome composé d'hétérochromatine, qui est une condensation de la chromatine. Il correspond à la zone appelée centromère lorsque l'ADN est compacté sous forme de chromosomes.

Nucléole : Nom du plus gros sous-compartiment du noyau cellulaire où est réalisée la transcription des ARN ribosomiques.

Arbre de recherche binaire : Une structure de données basée sur des nœuds où chaque nœud contient une valeur et deux sous-arbres, le gauche et le droit. Il permet des opérations rapides pour rechercher un élément et insérer ou supprimer un élément.

Race conditions : Une race condition ou (« situation de compétition ») se produit lorsque deux threads ou plus peuvent accéder à des données partagées et tentent de les modifier en même temps. Étant donné que l'algorithme de planification des threads peut permuter entre les threads à tout moment, vous ne connaissez pas l'ordre dans lequel les threads tenteront d'accéder aux données partagées.

Bibliographie et webographie

Allan, C., Burel, J.-M., Moore, J., Blackburn, C., Linkert, M., Loynton, S., MacDonald, D., Moore, W.J., Neves, C., Patterson, A., et al. (2012). OMERO: flexible, model-driven data management for experimental biology. Nature Methods *9*, 245–253.

Dubos T, Poulet A, Gonthier-Gueret C, Mougeot G, Vanrobays E, Li Y, Tutois S, Pery E, Chausse F, Probst AV, Tatout C, Desset S. Automated 3D bio-imaging analysis of nuclear organization by NucleusJ 2.0. Nucleus. 2020 Dec;11(1):315-329

Poulet, A., Arganda-Carreras, I., Legland, D., Probst, A.V., Andrey, P., and Tatout, C. (2015). NucleusJ: an ImageJ plugin for quantifying 3D images of interphase nuclei. Bioinformatics *31*, 1144–1146.

Schneider, C.A., Rasband, W.S., and Eliceiri, K.W. (2012). NIH Image to ImageJ: 25 years of image analysis. Nature Methods *9*, 671.

Sheets KG, Bokkyoo J, Zhou Y, Winkler J, Zhu M, Petasis N, Gordon WC, Bazan NG (2011). Topical Neuroprotectin D1 Attenuates Experimental CNV and Induces Activated Microglia Redistribution. Invest Ophthalmol Vis Sci. ARVO Meeting Abstracts April 22, 2011 52:5470 (3D Convex Hull, https://imagej.nih.gov/ij/plugins/3d-convex-hull/index.html)

Implémentation du parcours de Graham : https://github.com/bkiers/GrahamScan