MASTER EN SCIENCES INFORMATIQUES

Système d'aide au traitement d'image

Guelder, Laurent; Wilvers, Karl

Award date:
2019

Awarding institution:
Universite de Namur

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

Système d'aide au traitement d'image

Karl Wilvers
Laurent Guelder

RUEGRANDGAGNAGE,21 ▮ B-5000NAMUR(BELGIUM)

Université de Namur
Faculté d'informatique
Année académique 2018–2019

Système d'aide au traitement d'image

Karl Wilvers
Laurent Guelder

Promoteur : _____(Signature pour approbation du dépôt - REE art. 40)
Jean-Marie JACQUET

Confidentialité du mémoire :

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

Ce mémoire tente d'établir un système d'aide au traitement d'image numérique. Les images à traiter viennent de la Faculté de médecine de l'université de Namur, elles représentent des boîtes de Pétri dans lesquelles sont contenues des cellules cancéreuses. Les traitements des images visent à dénombrer les cellules dans les images.

Sept chapitres décrivent les filtres et méthodes de traitement d'image les plus populaires. Une version de ces filtres est implémentée en C# et une seconde version est implémentée avec l'API OpenCV. Des tests ont été réalisés pour détecter des cellules et leur noyau mais ils n'ont pas été concluants.

Un chapitre est consacré à la recherche d'une relation entre le nombre de cellules connues dans un ensemble d'images et la superficie des pixels mauves. La régression linéaire simple et linéaire multiple sont utilisées sur un set de données obtenues à l'aide des filtres précédemment implémentés. Le set de données est trop petit pour obtenir une relation concluante.

La dernière partie est une analyse d'un système d'aide à la décision complet. Une application client-serveur qui permettrait de créer un flux de travail, de détecter des cellules et de nourrir un réseau de neurones à convolution.

This thesis attempts to establish a support system in digital image processing. The images to be processed come from the Faculty of Medicine of the University of Namur. They represent Petri dishes in which cancer cells are contained. Image processing aims to count the cells in the images.

Seven chapters describe the most popular filters and image processing methods. A first version of these filters is implemented in C# and a second version is implemented with the OpenCV API. Tests have been carried out to detect cells and their nuclei, but they have not been conclusive.

A chapter is devoted to the search of a relationship between the number of known cells in an image set and the size of the purple pixels. Single and multiple linear regressions are used on a set of data obtained by means of previously implemented filters. The dataset is too small to obtain a relationship.

The last part is an analysis of a complete decision support system. It is a client-server application that would allow to create a workflow, to detect cells and to feed a convolutional neural network.

Avant-propos

Nous voudrions adresser nos remerciements aux personnes qui nous ont aidés dans la réalisation de ce mémoire de master.

Tout d'abord, nous voudrions remercier le Professeur Jaquet de la Faculté d'informatique à l'université de Namur, notre promoteur qui nous a offert l'opportunité de travailler sur ce sujet. Le Professeur Jaquet nous a soutenu et aiguillé tout au long de notre rédaction.

Ensuite, nous remercions le professeur Gillet de la Faculté de Médecine et Mademoiselle Marie Fourrez pour nous avoir reçus et nous avoir fourni le matériel nécessaire pour nos recherches.

Nous remercions également Mademoiselle Florence Coulonval et Monsieur Clément Gaye pour leur aide inestimable à la relecture et à la correction de notre mémoire.

Nous souhaitons faire un petit clin d'œil à Monsieur Jean-Michel Bechet pour ses précieux conseils sur les outils à utiliser.

Je voudrais personnellement remercier ma compagne Mélanie et mes trois enfants Margot, Alice et Charly pour avoir fait preuve de patiente durant ces trois longues dernières années.

Table des matières

Glossaire

Connexité : La connexité d'un kernel est le nombre de pixels du voisinage

Gradient : En mathématique, le gradient est un vecteur indiquant comment la valeur d'une fonction de plusieurs variables varie quand ces variables varient. C'est une généralisation de la notion de dérivée d'une fonction d'une seule variable.

Discrétisation : (Statistiques) Action de classification raisonnée de données. La discrétisation consiste à la fois au découpage de données en classes homogènes et en la justification mathématique de cette classification et du nombre de classes retenues.

Gradient discret : Correspond à un taux de variation calculable, et ce, même si l'image est discontinue.

Variance : Moyenne des carrés des écarts à la moyenne.

Fonction affine : Fonction obtenue par addition et multiplication de la variable par des constantes.

Coordonnée à l'origine : La valeur de Y d'une fonction affine lorsque X vaut 0.

Chapitre 1
Introduction

Quotidiennement, de nombreuses équipes s'affairent à trouver de nouvelles méthodes pour dépister les cancers, trouver des traitements ou en étudier le comportement. Dans le cadre de ce mémoire, il s'agira de se pencher sur l'imagerie numérique appliquée à l'imagerie médicale, et plus spécialement sur le dénombrement de cellules cancéreuses.

L'image médicale est connue dans ses formes les plus répandues telles que l'imagerie par résonance magnétique (IRM), la radiographie par rayons X, la tomographie par émission de positons (TEP ou PET scan), par ultrasons (échographie) ou par rayon lumineux (optique). Dans ce dernier cas, l'une des méthodes consiste à placer sous un microscope une plaquette contenant les tissus à analyser. C'est sur cette dernière technique d'imagerie que reposera l'ensemble de ce travail.

L'imagerie numérique peut mettre en évidence certaines propriétés de l'image grâce aux filtres et aux algorithmes appropriés. Si ce n'est sa qualité, aucune caractéristique d'une image, couleur ou niveaux de gris, profondeur de couleur de 8 ou de 16 bits, ou encore son format, ne peut empêcher son analyse.

Ces dernières années, l'imagerie médicale a pris un nouvel élan grâce à l'essor des techniques d'intelligence artificielle. Ces nouvelles technologies permettent, si ce n'est détecter un problème, tout du moins d'attirer l'attention sur des zones que le spécialiste aurait pu manquer à l'œil nu.

Le problème

Dans le cadre de recherche sur les cellules cancéreuses, l'équipe du Professeur Gillet de la Faculté de Médecine de l'Université de Namur a été confrontée à un problème de taille, dénombrer les cellules cancéreuses d'une culture dans un ensemble de boîtes de Pétri. Après avoir réalisé une culture dans une boîte de Pétri, les cellules cancéreuses doivent être dénombrées. Lors de cette campagne de tests, trois chercheurs ont passé trois semaines, à temps complet, à morceler, à compter et à consigner manuellement le nombre de cellules présentes.

La méthode est simple, chaque image prise par le microscope est divisée en images plus petites. Ces images sont analysées individuellement. Chaque cellule cancéreuse est repérée grâce à un marqueur dans un éditeur d'images puis comptée par le spécialiste. Le dénombrement manuel a une bonne précision, mais a l'énorme défaut d'être chronophage. Cette perte de temps a tendance à décourager les chercheurs à lancer de nouvelles campagnes de tests.

L'équipe du professeur Gillet a tenté de résoudre le problème à l'aide de ImageJ, un logiciel de traitement d'image, mais sans atteindre de résultat. C'est difficile pour un non initié de

réaliser une détection de motif dans une image, même avec les bons outils. La forme et la couleur de ces cellules semblent reconnaissables, et après ? Des logiciels standard, tel que ImageJ, paraissent difficilement exploitables par une équipe scientifique médicale.

Les questions ?

Sur base de ce problème, est-il possible de trouver un système d'aide au traitement d'images numériques afin de quantifier le nombre de cellules cancéreuses d'un cliché ?

Est-il possible de réaliser un outil permettant d'assister l'équipe de la Faculté de médecine ?

Est-il possible d'écrire un outil plus généraliste, permettant de dénombrer d'autres types de cellules, dans d'autres conditions, d'autres couleurs ?

Un outil qui pourrait être paramétré pour tenir compte de ces différentes caractéristiques ?

La méthode

Le traitement d'image numérique est vieux de plus de 40 ans maintenant, la quantité de matière sur le sujet est considérable. Les méthodes, algorithmes et techniques, sont très nombreuses. En tant que néophyte, deux approches sont possibles, l'une basée sur l'intuition qui mènera à évaluer une seule et même voie, et l'autre, basée sur une vision globale du sujet. C'est cette dernière approche que nous utiliserons.

La première partie de ce mémoire est un état de l'art sur le traitement d'image numérique. Bien entendu, le sujet étant très vaste, nous nous restreindrons aux méthodes et algorithmes pouvant être utiles au sujet qui nous préoccupe. L'idée première de cette réalisation est la maitrise du sujet, et la possibilité d'utiliser un large éventail d'outils.

La seconde partie sera une mise en œuvre de différentes techniques.

Une première mise en œuvre d'une séquence de filtres devrait permettre d'identifier les cellules et leurs noyaux. Dans la foulée, l'assemblage de ces différents filtres donnera naissance à un système expert qui permettra de régler précisément les paramètres des filtres et d'optimiser la détection.

La deuxième idée consiste à produire un modèle de régression linéaire mettant en relation un nombre de cellules et une quantité de pixel.

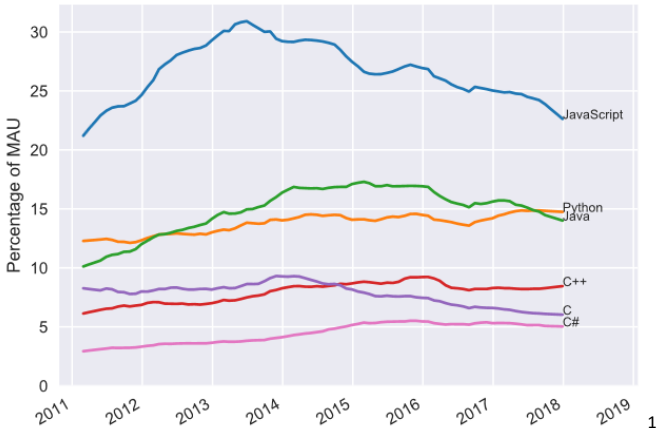
Pour des raisons pratiques et arbitraires, le code est réalisé avec C# et la librairie OpenCV. Les alternatives sont nombreuses. Les plus connues sont C++, Python et Java.

Le C++ est la solution la plus performante, les librairies OpenCV sont codées en C++ natif et compilées avec des options d'optimisation mémoire et processeur. Nous n'avons pas retenu cette option parce que le C++ n'est plus très populaire et son enseignement n'est plus généralisé. Dans le cadre de nos recherches, la performance n'est pas prioritaire, nous traitons des images simples et non du flux vidéo.

Le python aurait été la solution optimale, en effet, ce langage est l'étoile montante. La librairie OpenCV est portée en Python. De plus ce langage est grandement utilisé dans le cadre du deep

learning. Le seul problème avec Python c'est le temps d'apprentissage et le temps c'est ce qu'il nous manque. Dans un futur plus ou moins proche, nous avons le projet de porter le fruit de notre travail dans ce langage.

Enfin, la troisième alternative est Java. Bien que répandus dans le milieu académique, nous n'avons vu aucun avantage à l'utiliser. En effet, depuis l'émergence de .Net Core, Java n'a plus l'exclusivité du multiplateforme. De plus, les outils gratuits de développement Java ne font pas le poids face à Visual Studio Enterprise. Java est un c-like, adapter le code demanderais tout au plus une demi-journée de travail. Notons que l'utilisation de java est en plein déclin.



[1]

Figure 1 : Utilisation des langages sur GitHub

Les sources de l'implémentation des filtres, des tests unitaires et de l'interface utilisateur ont été déposé sur un repository GitHub public et peuvent être trouvées à l'adresse suivante : https://github.com/kwilvers/MasterImageProcessing/tree/master/CancerCellDetection. Elles sont libres de droit et pourrait être utilisées pour poursuivre les recherches.

Nous faisons face à un grand défi tant l'inconnue est grande, autant dans le domaine d'activité qu'est la biologie et l'étude des cellules que dans le domaine technique qui est la vision par ordinateur. Pour nous, être humain, voir est naturel et identifier des objets ne demande qu'une fraction de seconde. Mais pour un ordinateur, c'est plus compliqué, il faut lui apprendre en très peu de temps ce que nous mettons une vie à apprendre.

Notre premier grand défi sera d'apprendre un maximum de choses sur la vision par ordinateur, tout ce qui est digne d'intérêt et qui a été énoncé ces trente dernières années. Le deuxième défi sera de développer un modèle d'identification le plus performant possible.

1.1 La problématique des images

Les images utilisées dans ce document ont été obtenues auprès de l'équipe du professeur Gillet. Quelques 250 images ont été fournies ainsi que le dénombrement des cellules de 50 d'entre elles.

---

[1] https://www.developpez.com/actu/185087/Quels-sont-les-langages-de-programmation-les-plus-utilises-par-les-developpeurs-Une-analyse-des-evenements-publics-sur-GitHub/

La taille des images est variée, les largeurs et hauteurs vont de ± 14.000 pixels à ± 17.000 pixels.

Les images ne sont pas de très bonne qualité et différents problèmes peuvent y apparaître :

- Certaines zones peuvent paraître nettes et d'autres floues ;
- Certaines images souffrent de problèmes d'alignement de reconstruction, comme un panorama mal assemblé ;
- Des taches ou crasses peuvent être visibles ;
- La boîte de Pétri et son contour occupent plus de 45% de la superficie de l'image, ce qui implique un temps de traitement presque deux fois plus grand que nécessaire ;
- Une ombre portée est visible autour du bord de la boîte de Pétri.

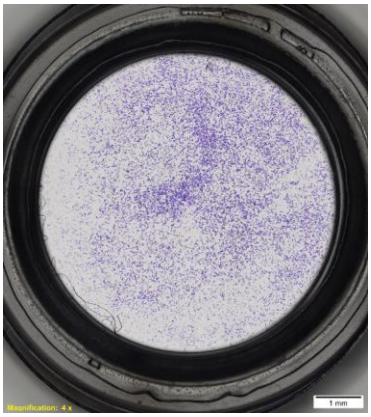


Figure 2 : image d'origine

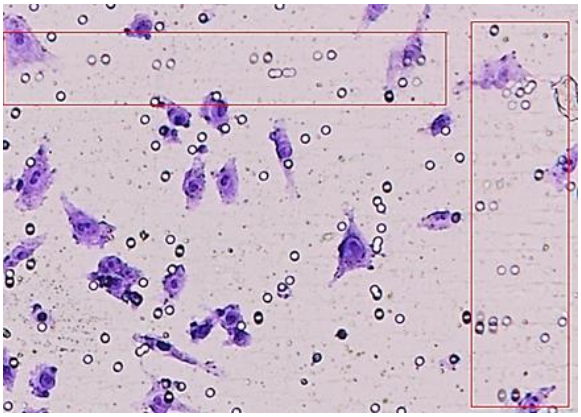


Figure 3 : flou et problème d'alignement

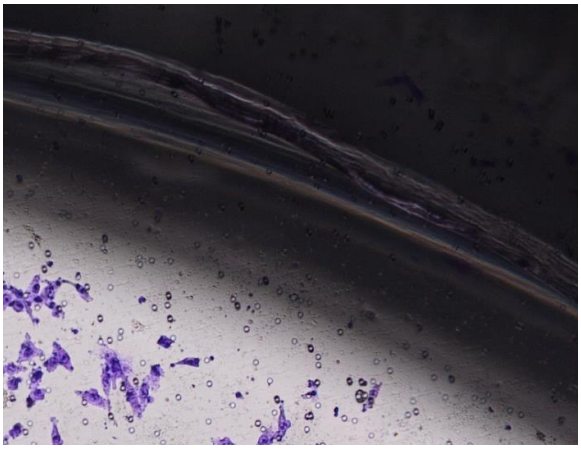


Figure 4 : Ombre portée du bord de la boîte de Pétri

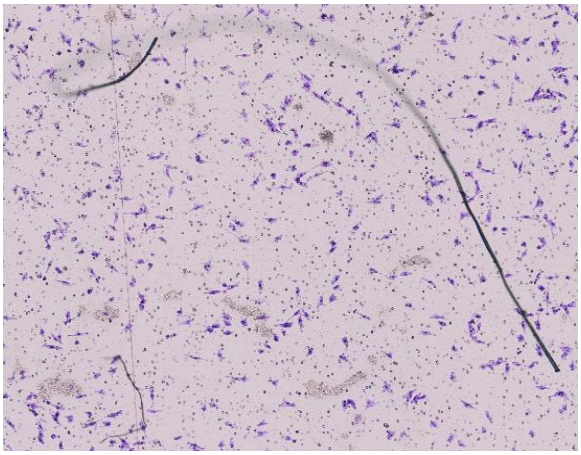


Figure 5 : Crasse sur l'échantillon

1.2 Une cellule

Il est très difficile d'identifier et de dénombrer les cellules au sein d'une zone colorée. C'est sans doute le point le plus complexe du problème : comment décrire une cellule pour permettre au système de les repérer ? Le cytoplasme est mauve clair et le noyau est mauve foncé. Plusieurs types d'amas sont identifiable. Il y a les cellules simples, les cellules en cours de division, les cellules ayant « deux » noyaux juste après la division et les cellules en cours de migration de la face inférieure vers la face supérieure au travers d'un pore

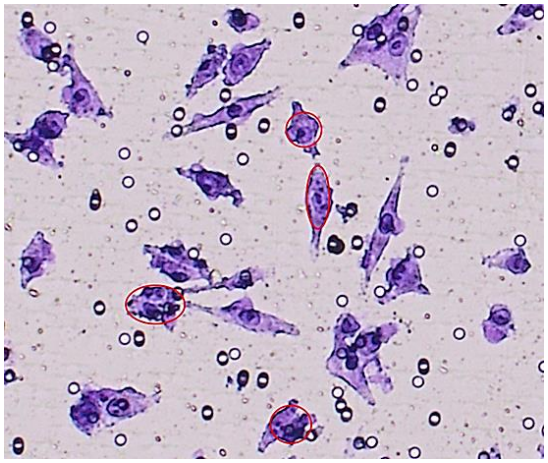


Figure 6 : partie d'une image d'identification de cellules

1.3 Vers une solution idéale ?

La première chose qui vient à l'esprit lorsque l'on se rend compte du temps perdu par les équipes médicales à analyser des images : comment peut-on les aider pour gagner du temps. Comme tout travail, il peut être modélisé au travers d'un workflow, qui dans le cas de l'équipe du professeur Gillet comprendrait les étapes suivantes :

1. Prendre une photo à l'aide d'un microscope électronique ;
2. Récupérer le fichier, le découper en plusieurs images plus petites ;
3. Analyser l'image dans un logiciel, et y apposer des marqueurs sur les zones déjà traitées ;
4. Enumérer toutes les cellules qui composent une zone et en faire la somme du nombre de cellules cancéreuses.

Excepté la première étape, qui n'apporterait pas grand-chose, toutes ces étapes peuvent-être simplifiées voire automatisées.

1.3.1 Système d'aide à la décision

Un système expert devrait permettre de créer et configurer des flux de travail comportant un ensemble de traitements numériques configurables. Le flux de travail en cours de configuration doit pouvoir être testé sur de petites parties d'une image test afin d'en visualiser le résultat.

Le système pourrait charger les nouvelles images, exécuter un ensemble de traitements et identifier les cellules détectées. L'expert pourrait ensuite confirmer ou infirmer les résultats obtenus à l'aide d'un parcours visuel et de marqueurs de couleur. Le nombre de marqueurs représenterait le nombre de cellules dénombrées.

L'analyse de l'image par des yeux experts pourrait être soumise à l'apprentissage (deep learning), et ainsi, après un certain nombre d'images traitées, être capable de mâcher le travail de l'équipe médicale. Chaque zone pourrait alors être associée à un nombre de cellules détectées ainsi qu'à un degré d'incertitude. Le déplacement d'une zone à l'autre pourrait alors être dirigé par ce degré d'incertitude, en exposant à l'expert les zones les moins certaines

en premier lieu.

1.3.2 Mise en mémoire tampon de l'image

Un système idéal pourrait charger l'image petit à petit et se déplacer de la dernière zone traitée, vers la première non traitée (en supposant que le logiciel soit capable de détecter ces zones). Il suffirait alors d'avoir en mémoire la zone affichée à l'écran et éventuellement les données nécessaires au déplacement vers les zones contigües. Une image miniature pourrait indiquer où se situe la zone affichée par rapport à l'image globale et ainsi potentiellement permettre un déplacement rapide vers une autre zone.

1.3.3 Réutilisabilité

Dans une optique de réutilisabilité et pour ne pas se cantonner au problème de dénombrement de cellules cancéreuses de l'équipe du professeur Gillet, ce système idéal doit pouvoir définir simplement ce qui est recherché sur l'image, par exemple en décrivant mathématiquement les zones et leur contenu, par le biais de la programmation fonctionnelle, d'un DSL, où même d'un outil graphique de type workflow.

1.3.4 Travail en équipe

Permettre à plusieurs personnes de travailler sur la même image (c'est d'ailleurs peut-être la raison de la découpe en plusieurs images) semble également faire partie des qualités de ce système idéal. Une architecture client-serveur qui permettrait de synchroniser l'avancée du traitement entre les clients concernés pourrait répondre à ce besoin.

1.3.5 Petit à petit…

Tout ceci est bien beau, mais probablement trop ambitieux dans le cadre d'un mémoire. Rien n'empêche bien sûr de creuser quelques-unes de ces idées tout en étant guidé par cette vision idéale.

1.4 Conclusion

La tâche qui s'étend devant nous est considérable. Les compétences à acquérir sont importantes. Tout d'abord, il faut se pencher sur la théorie du traitement d'image numérique, viendra ensuite l'apprentissage de la librairie OpenCv. Et enfin si le temps nous le permet, la réalisation d'un système d'aide à la décision.

Nous n'oublierons pas notre première intuition qui est de comparer les surfaces colorées avec le nombre de cellules présentes afin d'en déterminer une relation.

Chapitre 2
L'imagerie numérique

La détection de motifs ou de caractéristiques dans une image numérique est sensiblement identique quel que soit le domaine d'application. Nous traiterons dans ce chapitre des couleurs de l'image ou de son absence de couleur et de ses caractéristiques principales permettant de la traiter.

D'un point de vue abstrait, nous pouvons définir une image par un ensemble de points de couleur contigus visant à former des motifs, le tout représentant un objet, une situation allant de l'infiniment grand à l'infiniment petit.

2.1 Généralités

La représentation informatique d'une image implique une discrétisation des données, c'est-à-dire un échantillonnage. Cette discrétisation est encodée sous forme de pixels, d'un maillage carré et d'un espace colorimétrique. Ci-dessous, une ligne droite et un zoom permettant de visualiser la discrétisation.

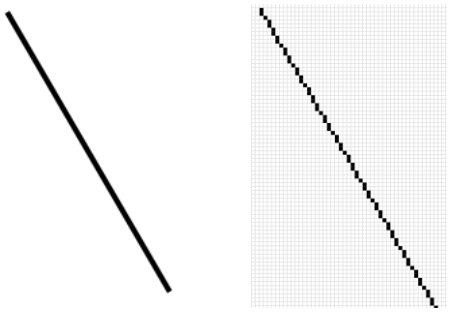


Figure 7 : Discrétisation d'une ligne droite sur un maillage carré

Une image est composée de pixels. Les pixels sont les composants d'une image étant généralement représentée par 256 niveaux de gris ou par 16 millions de couleurs encodées dans un triplet de valeurs (RGB –Red, Green et Blue).

Une image comporte N pixels de large sur M pixels de haut formant ainsi une matrice rectangulaire.

2.2 Les espaces de couleur

Les espaces de couleur sont représentés par des triplets d'informations. On distingue essentiellement les espaces suivants : RGB, SRGB, CMYN, HSB. Il en existe d'autres mais sont souvent utilisés pour des applications particulières.

Il existe des méthodes pour passer d'un espace à l'autre avec un risque de perte de la couleur d'origine, mais cela peut s'avérer utile dans le traitement du signal informatique. Par exemple, éclaircir une couleur est plus facile à réaliser si l'on travaille sur la composante de brillance de l'espace HSB que sur l'espace RGB qui demande de modifier chaque composante de manière symétrique sans pour autant obtenir un résultat optimal.

2.2.1 L'espace RGB

Cette représentation décompose la couleur en un triplet rouge, vert et bleu (Red Green Blue). Il s'agit d'une synthèse additive, c'est-à-dire une addition des trois couleurs avec une saturation maximum qui donne du blanc. Le noir de cet espace est une absence de couleur, toutes les composantes sont égales à zéro.

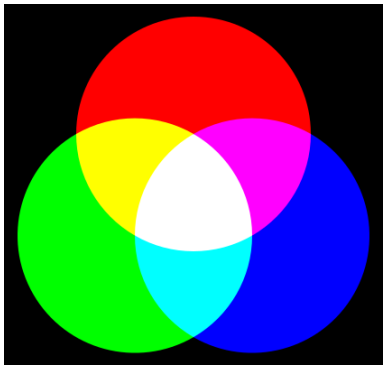


Figure 8 : Roue chromatique RGB

2.2.2 L'espace SRGB (short RGB)

SRGB est une représentation identique au RGB si ce n'est qu'il permet de normaliser la représentation pour les moniteurs, ceux-ci ne pouvant afficher l'entièreté de l'espace RBG. La différence de couverture de l'espace CIEXYZ entre le RGB et le SRGB se situe dans la saturation des cyans. Cet espace permet de ne pas devoir effectuer une conversion lors de l'affichage.

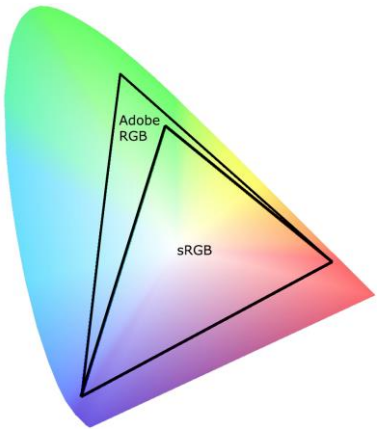


Figure 9 : Espace de couleurs RGB et sRGB

2.2.3 L'espace CMY

Cette représentation est utilisée dans les domaines de l'imprimerie et utilise une synthèse soustractive. Le triplet cyan, magenta et jaune (cyan, magenta, yellow) permet d'obtenir le

spectre de couleur ou le blanc est une absence de couleur et le noir est une combinaison des trois composantes. Il est fréquent de l'accompagner d'une composante noire, permettant ainsi d'obtenir, lors de l'impression, des noirs profonds.

Figure 10 : Roue chromatique CMY

2.2.4 L'espace HSB (HSV)

Dans ce cas-ci, la couleur est décomposée en teinte, saturation et brillance (Hue, Saturation, Brightness). La teinte H est représentée par une valeur d'angle reprise sur le cercle chromatique où le rouge est à 0°, le jaune à 60°, le vert à 120°, le cyan à 180°, le bleu à 240° et le magenta à 320°. La saturation S est la pureté, l'intensité de dissolution dans la lumière blanche, et s'exprime de 0 à 100% (0% est gris). La brillance B, où lumière, permet de déterminer si une couleur est sombre et s'exprime également de 0 à 100% (0% est noir).

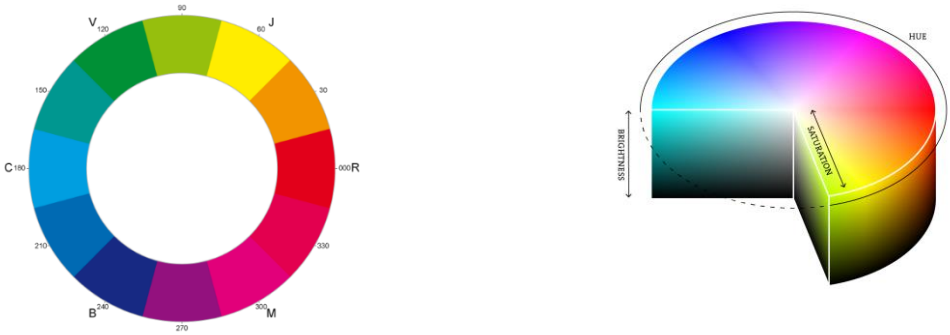


Figure 11 : Espace HSB

Il est possible de convertir une couleur RGB en couleur HSB. Le format RGB est le plus fréquemment utilisé, notamment lors de traitement d'image numérique. Le format HSV peut être utilisé dans le cadre de traitement ou filtre particulier. Un traitement pourrait avoir besoin d'isoler une couleur ou une gamme de couleurs. Le format RGB ne permet pas de traiter facilement la composante de teintes et donc d'isoler l'une d'entre elles. Il peut donc être utile de passer du format RGB au format HSB. La conversion n'est pas linéaire et se fait différemment pour chaque composante.

$$\min_c = Minimum(R, G, B)$$

$$\max_c = Maximum(R, G, B)$$

$$\Delta = \max_c - \min_c$$

$$H = \begin{cases} 0 & si\ \Delta = 0 \\ \frac{1}{6}\ \frac{G-B}{\Delta} & si\ \max_c = R \\ \frac{1}{6}\ \frac{B-R}{\Delta} + \frac{1}{3} & si\ \max_c = G \\ \frac{1}{6}\ \frac{R-G}{\Delta} + \frac{2}{3} & si\ \max_c = B \end{cases}$$

$$S = \begin{cases} 0 & si\ \max_c\ est\ 0 \\ \frac{\Delta}{\max_c} & sinon \end{cases}$$

$$B = \max_c$$

Exemple : pour une couleur dont la valeur RGB est (200,147,233), la valeur HSB est :

Min = 147, Max = 233 donne Δ=233-147 = 86
H = (max = B) → 1/6 * (200-147)/86 + 2/3 = 0.76937 * 360 = 277°
S = 86/233 * 100 = 37 %
B = 233/255 * 100 = 91%
RGB(200, 147, 233) = HSB(277, 37, 91)

2.2.5 Les niveaux de gris

Les niveaux de gris est un espace particulier qui est caractérisé par l'absence de couleur, ou plutôt, une quantité équivalente des trois composantes de couleur. Cet espace est atteint en convertissant un espace couleur en échelle de gris. Il est encodé sur 8 bits soit un octet, offrant 256 niveaux de gris (0 pour le noir et 255 pour le blanc).

Il faut distinguer deux aspects.

Le premier : une image en niveaux de gris peut être stockée sur disque à l'aide d'un octet par pixel.

Le second : les pixels affichés à l'écran doivent être composé des trois composantes RGB afin d'allumer le pixel du moniteur. C'est pour cette raison que les trois composantes ont la même valeur.

Encodage : Les différents espaces sont encodés sur un ensemble d'octets.

Les espaces RGB et SRBG sont encodé sur 4 octets dont trois représentent le niveau de chacune des couleurs, de 0 (noir) à 255 (blanc). Le dernier octet peut être utilisé pour définir un niveau de transparence, communément appelé canal alpha (Alpha RGB) dont les valeurs possibles vont de 0 (invisible) à 255 (opaque).

L'espace CMY est également encodé sur quatre octets dont trois seulement sont utile, de 0

(blanc) à 255 (noir).

L'espace HSB encode la teinte sur deux octets dont les valeurs de 0 à 360 représentent l'angle sur le cercle chromatique. Les saturations et brillance sont, chacune des deux, encodées sur un octet dont chaque valeur représente environ 0.4 % de la plage.

L'espace gris est encodé sur un octet. La valeur de 0 à 255 va du noir au blanc en passant par les nuances de gris.

2.3 La matrice

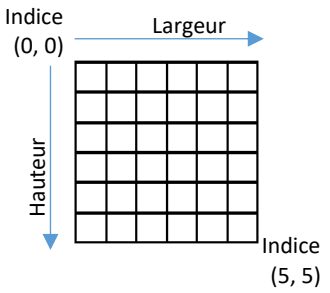
Les images numériques sont caractérisées par une matrice dont le maillage est généralement quadrillé. Il en existe d'autres mais ils sont réservés à des applications spécifiques et ne seront pas exposés ici.

La matrice de l'image a une hauteur *H* et une largeur *L*. Le nombre de pixels *Nb* de l'image correspond à la hauteur multipliée par la largeur.

$$Nb = H * L$$

La taille de la mémoire utilisée correspond au nombre de pixels multiplié par l'encodage de l'espace colorimétrique. Pour un encodage RGB, pour lequel chaque pixel est encodé sur 4 bytes, la taille de la mémoire *Mem* utilisée est de $Mem = Nb * 4$.

La matrice est composée de pixels et chacun d'eux peut être accédé par son l'indice. L'indice (0, 0) de la matrice est dans le coin supérieur gauche. L'indice des colonnes évolue de gauche à droite et celui des lignes de haut en bas par pas de 1.



Nb = 5 * 5 = 25 pixels
Mem = 25 * 4 = 100 bytes

2.4 En mémoire et sur disque

Sur disque, une image est enregistrée comme un vecteur, un tableau à une dimension dont les octets ne sont pas nécessairement contigus. Plus l'image est importante, plus le risque de fragmentation sur le disque est important.

En mémoire, une image est lue et stockée, par défaut, dans un vecteur, ici aussi un tableau à une dimension. Néanmoins, il est possible de charger l'image dans un tableau à deux dimensions dont la hauteur et la largeur sont identiques au nombre de lignes et de colonnes de pixels de l'image ? Cette dernière façon de faire demande une transformation du vecteur

en tableau à deux dimensions.

OpenCv utilise la méthode du tableau à deux dimensions autrement appelé matrice. La méthode matricielle offre une meilleure lisibilité du code, la maintenance en est facilitée. Lors de l'implémentation personnalisée des filtres la méthode de chargement standard dans un vecteur a été utilisée pour des raisons de performance. L'accès au pointeur du vecteur en C# améliore grandement le temps de réponse. De plus l'implémentation personnelle des filtres n'étant pas destiné à perdurer, il n'est pas nécessaire de le rendre pérenne.

2.5 Conclusion

La manipulation d'une image numérique est rendue relativement simple grâce aux fonctions du framework .Net ou de OpenCv. Qu'il s'agisse du chargement d'une image PNG ou d'une image JPG, il n'y a pas de différence, tout est encapsulé. De même, l'espace de couleur utilisé par défaut est RGB, l'espace le plus simple à manipuler et à comprendre.

Dès lors que l'image a été chargée, les pixels sont facilement manipulables et la matrice peut être parcourue aussi facilement grâce aux propriétés de ligne et colonne. Pour la librairie OpenCv, la gestion de la mémoire est optimisée et la représentation est sous forme de matrice dont le fonctionnement est très intuitif.

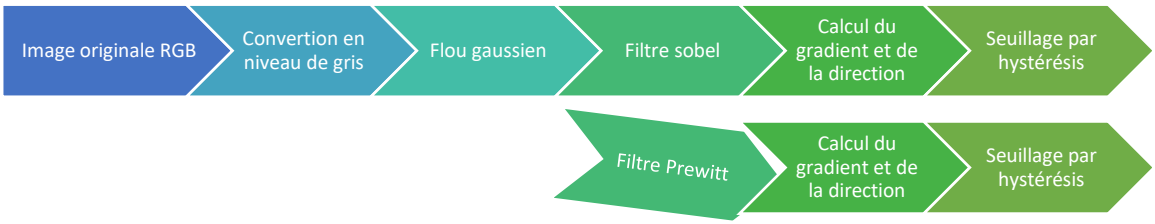
Chapitre 3
Généralités

Nous savons maintenant ce qu'est une image numérique, comment elle est représentée en mémoire et qu'elle évolue au sein d'un espace de couleur. Cette nouvelle escale va nous permettre d'aborder nos premiers traitements d'image, à l'aide d'une des méthodes de traitement les plus populaires : la convolution.

Avant d'emprunter ces petits chemins sinueux, revenons un instant sur le but de tout ceci. Dans le cadre du système d'aide à la décision, l'intention est de permettre d'empiler une série de fonctions de traitement paramétrées dont l'image d'entrée d'une fonction est l'image de sortie de la fonction précédente. Une telle méthode permet d'affiner le traitement et permet au système expert d'être adapté au besoin.

Pour illustrer cette idée, imaginons que chacune des étapes du filtre de Canny soit codée au sein d'une fonction individuelle et indépendante. Il nous serait alors possible d'empiler ces mêmes fonctions afin de réaliser le filtre de Canny. Et si nous décidions d'ajouter avant la conversion en niveaux de gris une fonction de correction gamma ou d'augmentation de contraste ? Et si à la place d'un filtre de Sobel, nous voulions appliquer un filtre de Prewitt ? Dans une méthode de programmation tout en un, nous serions vite limités. Avec une méthode découpée au plus petit traitement, nous pouvons réaliser tout ce qui nous fait envie.

Le workflow traditionnel de la fonction de détection de Canny pourrait comporter une branche permettant d'appliquer deux filtres de détection différents et ainsi obtenir deux résultats à l'aide d'une même première partie.



Pour chaque étape du workflow, le système doit pouvoir permettre d'adapter les paramètres des filtres et le type de filtre à utiliser. La simple étape de conversion en niveaux de gris pourrait utiliser l'un des cinq filtres du chapitre 7 dont au moins un est paramétrable.

Afin de parvenir à un degré de paramétrisation suffisant, le système expert disposera de cinq méthodes de conversion en niveaux de gris, cinq filtres de lissage, douze filtres de détection de contour et une dizaine d'autres filtres, soit plus de 3000 combinaisons, sans compter les paramètres de ces filtres. Chaque filtre sera décrit et implémenté tout au long de nos déambulations.

3.1 Filtrage

Le filtrage simple d'une image consiste à appliquer une fonction *f* à chacun des pixels de telle sorte que l'ensemble des pixels de l'image subit une transformation sur une ou plusieurs de ses composantes de couleur ou sur une de ses propriétés. L'entrée de ce filtrage est un pixel et une fonction de filtrage. La sortie est le pixel auquel la fonction a été appliquée. On compte parmi eux le filtre inverse, la correction de contraste ou encore la correction gamma. La plupart de ces filtres sont assez simples à implémenter. Il est inutile de parcourir les lignes ou colonnes, il suffit de parcourir le vecteur de pixels de façon linéaire et d'appliquer la transformation sur chacun des pixels.

Le filtrage d'une image *i* par convolution consiste à convoluer cette image telle une fonction avec une fonction *k*, le kernel, dont le résultat est appelé réponse impulsionnelle du filtre. Pour ce type de filtrage, l'entrée est le pixel courant, le voisinage défini par le kernel et le kernel lui-même. La sortie est le produit de convolution appliqué au pixel courant.

Les filtres peuvent être catégorisés en quatre groupes : passe-bas, passe-haut, passe-bande et coupe bande.

Le filtre passe-bas atténue fortement les hautes fréquences d'un spectre pour ne laisser passer que les basses fréquences. Il diminue le bruit et atténue les détails d'une image. Il y a une apparition d'un flou plus ou moins prononcé. Le filtre atténue les détails tels que les contours. Ce type de filtre est souvent utilisé en pré-traitement dans la recherche de contour. Il permet d'aplatir des lignes et lisser des courbes. Les filtres moyenneur, gaussien et médian font partie des filtres passe-bas.

Le filtre passe-haut atténue fortement les basses fréquences d'un spectre pour ne laisser passer que les hautes fréquences. Il améliore le contraste mais augmente le bruit dans l'image. Il appuie les contours. Le filtre passe-haut favorise les hautes fréquences, les changements francs. Les filtres de Sobel, Prewit et Laplace font entre autres partie des filtres passe-haut.

Le filtre passe-bande ne laisse passer que les fréquences d'un spectre compris entre deux bornes. Un filtre passe-bande peut servir à ne conserver qu'une gamme de couleurs, par exemple, le mauve entourant les cellules.

Pour être complet, le filtre coupe-bande supprime les fréquences comprisses entre deux bornes. Un tel filtre pourrait permettre de ne conserver que les hautes lumières et les noirs en appliquant une valeur de 0 sur le canal alpha d'une image traitée et affichée en ARGB.

3.2 Fréquences et filtres

Le concept de fréquences en imagerie numérique caractérise le nombre de modifications sur un petit nombre de pixels. Les hautes fréquences dans une image peuvent être représentées par une texture tant la modification des hauteurs des pixels est fréquente.

Les basses fréquences quant à elles sont caractérisées par des modifications lentes des valeurs des pixels, sur une plage comprenant de nombreux pixels.

Dans l'image de notre padawan ci-dessous, l'ovale bleu caractérise des basses fréquences, le gris foncé est stable, il y a peu de changement de teinte. L'ovale orange caractérise les hautes fréquences, la teinte des cheveux change fréquemment.

Figure 12 : Hautes et basses fréquences

Un filtre passe-bas va atténuer et filtrer les hautes fréquences pour ne laisser que les basses fréquences. En d'autres termes, un filtre passe-bas appliqué à notre padawan atténuerait les détails dans les cheveux.

A l'inverse, un filtre passe-haut va atténuer les basses fréquences pour ne laisser que les hautes fréquences. Le filtre passe-haut va amplifier les détails de l'arrière-plan, et va faire ressortir la structure du bois de la table.

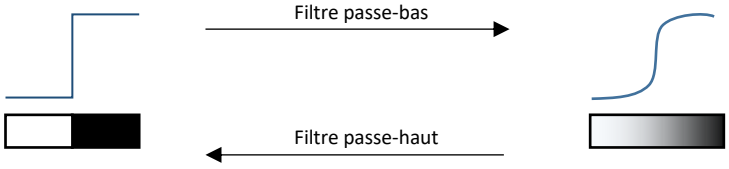


Figure 13 : Transition filtre passe bas - filtre passe haut

En complément, un filtre passe-bande ne conservera qu'une gamme de fréquence délimité par un seuil bas et un seuil haut. Un tel filtre pourrait filtrer les très hautes fréquences ainsi que les très basses fréquences.

Un filtre coupe bande est moins répandu, il ne conservera que les basses et hautes fréquences, celles situées en dehors d'une bande délimitée par un seuil haut et bas.

3.3 Convolution

L'opération mathématique nommée convolution peut être utilisée pour filtrer des images. De nombreux filtres (Sobel, flou gaussien, …) peuvent être effectués grâce au produit de convolution. La convolution est une fonction permettant l'utilisation de filtre linéaire ou non-linéaire. Une implémentation de base prenant en paramètre la définition du filtre autrement appelé le kernel permettra de multiplier facilement le nombre de filtres du système expert.

Appliqué à une image I_1 de dimension finie et à un kernel K de dimension 3 x 3, les pixels de l'image I2 obtenus par convolution de I1 par K ont pour valeur[2] :

$$I2(i,j) = \sum_{k=0}^{2}\sum_{l=0}^{2} I1(i+1-k, j+1-l)K(k,l)$$

L'œil averti aura remarqué qu'un pixel au tour de l'image n'est pas traité, nous y reviendrons au point Le problème du rembourrage ou padding.

Plus simplement, la convolution consiste à parcourir les pixels d'une image d'entrée et à les modifier en fonction du voisinage du pixel en cours et du kernel de convolution. Si i(x,y) est une image et que K est le kernel de convolution, alors g(x,y) est le produit de convolution de i(x,y) par K.

$$g(x,y) = \mathrm{i(x,y)} * \mathrm{K}$$

3.3.1 Le kernel

Le kernel, ou filtre, est une matrice de taille impaire et symétrique. Le kernel est défini par le pixel central à traiter et de son voisinage. Les tailles de kernel peuvent se présenter en 3x3 soit un pixel central et 8 voisins, en 5x5, soit un pixel central et 24 voisins ou en 7x7, soit un pixel central et 48 voisins, et ainsi de suite.

Par exemple, pour une détection de contours verticaux, le kernel est défini par :

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Le kernel accorde de l'importance au voisin de gauche du pixel, aucune aux pixels centraux et une importance négative au pixel de droite.

Dans ce document, les kernels ont des valeurs entières et pourront contenir un coefficient multiplicateur qui sera inscrit à droite du kernel. Ce coefficient devra être multiplié au résultat d'une convolution. Ce coefficient peut être soit une addition des valeurs de la matrice, soit une valeur constante, soit une valeur nulle ne modifiant pas le résultat obtenu.

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} * 1/16$$

3.3.2 La valeur de sortie du pixel traité

La valeur du pixel traité est la somme des produits des voisins multipliée par la valeur du kernel

[2] Voir l'ouvrage : Introduction au traitement d'image

correspondante. Pour le kernel ci-dessus, les 9 valeurs du kernel sont multipliées par les 9 pixels de l'image composée du pixel central et de ses 8 voisins.

5	6	4	2	3	1
8	**9**	2	0	5	6
4	7	3	6	4	4
8	2	3	4	6	8
9	5	1	3	8	5
8	5	3	1	2	5

*

1	0	-1
1	0	-1
1	0	-1

=

8			

La valeur de sortie du pixel de l'image d'entrée est obtenue par la somme des produits suivants :

5x1 + 8x1 + 4x1 + 6x0 + 9x0 + 7x0 + 4x-1 + 2x-1 + 3x-1 = 8

Si le coefficient du kernel n'est pas neutre, il doit être appliqué.

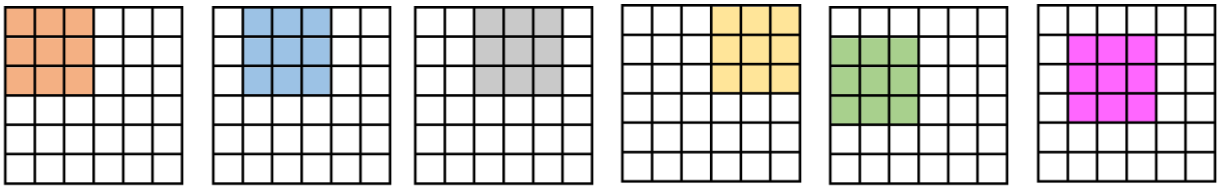
$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} * 1/16$$

5x1 + 8x2 + 4x1 + 6x2 + 9x4 + 7x2 + 4x1 + 2x2 + 3x1 = 98

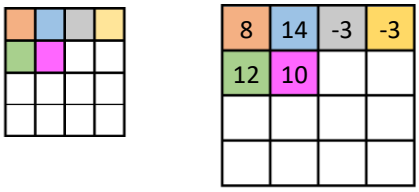
98 / 16 = 6.125

3.3.3 Le parcourt linéaire du kernel

Par convention, le parcourt se fait de gauche à droite et de haut en bas, comme illustré ci-dessous. L'ordre dans lequel le parcours s'effectue n'a aucune importance puisque l'image d'entrée n'est pas altérée et le résultat est inséré dans une nouvelle image.



Pour reprendre l'exemple précédent, le parcours des six premiers pixels donne les valeurs suivantes :



8	14	-3	-3
12	10		

3.3.4 La taille de l'image de sortie

Ci-dessus, il est constatable que la taille de l'image résultante de la convolution est inférieure à la taille de l'image d'entrée. En effet, on calcule la valeur du pixel central sur base du masque du kernel. Le kernel ne peut pas être appliqué au pixel se trouvant en indice (0 ,0), sinon celui-ci dépasserait de l'image d'entrée. Intuitivement, la taille de sortie est amputée dans chaque direction du nombre de pixels entourant le pixel central du kernel. Pour un kernel de 3x3, un pixel est amputé tout autour de l'image et pour un kernel de 7x7, trois pixels sont supprimés.

La taille de sortie d'une image d'entrée de 10x20 et d'un kernel de 5x5, deux pixels à gauche et à droite ainsi que deux pixels en haut et en bas sont tronqués. La largeur est obtenue par 10-2-2 = 6 et la hauteur 20-2-2 = 16.

La taille de l'image de sortie pour une image d'entrée de taille M x N et d'un kernel de taille F x F, peut être obtenue à l'aide de la formule suivante :

$$(M - F + 1)\ x\ (N - F + 1)$$

Appliqué à l'exemple ci-dessus, dont M = 10, N = 20 et F = 5, la taille de l'image de sortie est obtenue par :

$$(10 - 5 + 1)\ x\ (20 - 5 + 1) = 6 \text{ x } 16$$

3.3.5 Le problème du rembourrage ou padding

Le problème de la convolution montre que l'image de sortie résultante diminue à chaque convolution. Si de nombreux filtres sont appliqués à une image au sein d'un même workflow, la taille de l'image de sortie diminue à chaque itération. Pour des images de grandes tailles, le problème peut ne pas être significatif, mais pour de plus petites images, il y a un risque de perte d'informations dans le contour de l'image.

Pour une image dont M et N=150 et pour laquelle cinq filtres dont F = 9 sont appliqués nous obtenons la réduction suivante :

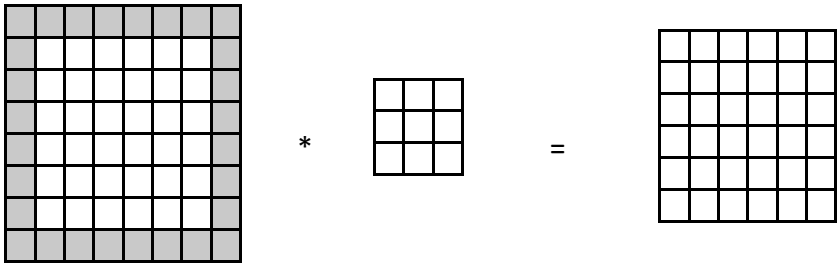
$$(150 - 9 + 1) = 142 \ \rightarrow\ (142 - 9 + 1) = 134 \ \rightarrow\ (134 - 9 + 1) = 126$$
$$\rightarrow\ (126 - 9 + 1) = 117 \ \rightarrow\ (117 - 9 + 1) = 109$$

Le résultat final diffère de +-20 pixels tout autour de l'image. Qu'en est-il si le processus tente de détecter les contours et qu'une information essentielle se trouve en périphérie de l'image ?

Un autre problème résultant de ce troncage implique que le pixel se trouvant en indice (0, 0) ne compte que pour une seule itération de la convolution, tandis que le pixel se trouvant en (2, 2) prend part à 9 itérations, et donc sa valeur influence l'ensemble de ses 8 voisins.

La solution à ce problème peut être résolu grâce au padding, c'est-à-dire l'ajout d'un rembourrage tout autour de l'image d'entrée. Intuitivement, pour un kernel de 3 x 3, il a été observé que l'image de sortie était réduite de 1 pixel tout autour de l'image. Pour une image de sortie de même taille, il faudrait par conséquent ajouter un pixel tout autour de l'image d'entrée, le padding P serait égal à un.

Une image dont M et N = 6, P = 1 et F = 3, la nouvelle taille d'entrée est 6 +1 +1 = 8. Après le padding, la taille de l'image de sortie est (8 – 3 + 1) = 6 donc 6 x 6, la même taille que l'image d'entrée. Par convention les valeurs du padding sont initialisées à zéro afin d'éviter que le padding n'aie d'influence sur le résultat final.



La taille de sortie est obtenue en ajoutant deux fois la valeur du padding. La formule :

$$(M + 2P - F + 1)\ x\ (N + 2P - F + 1)$$

Dans notre cas, la valeur de M et N de l'image de sortie : (6 + 2x1 – 3 +1) = 6

Sur base de la taille du kernel, il est possible d'en déduire la taille du padding à utiliser pour obtenir une image de sortie de taille identique.

$$si\ (M + 2P - F + 1)\ = M$$

$$Alors\ (\cancel{M} + 2P - F + 1)\ = \cancel{M}$$

$$\frac{F - 1}{2} = P$$

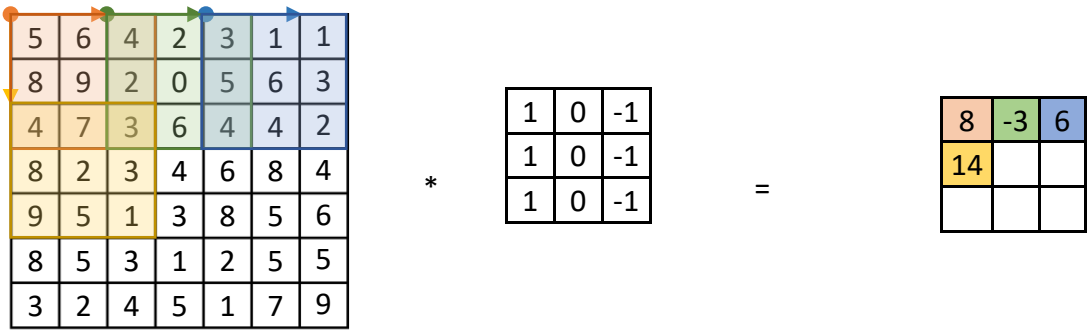
Pour un kernel de F = 9, nous obtenons un padding de $\frac{9-1}{2} = 4$.

Une convolution sera « valide » si elle ne comporte pas de padding et « identique » si l'image d'entrée comporte un padding permettant d'obtenir une taille d'image de sortie identique à la taille d'image d'entrée.

3.3.6 La foulée ou stride

La foulée est utilisée pour contrôler les dimensions de l'image de sortie, afin de sous échantillonner une image pour diminuer le volume de sortie et ainsi la hauteur et la largeur.

Lors d'une convolution traditionnelle, la foulée S est égale à 1, le kernel est déplacé de 1 pixel à la fois et le volume de sortie est identique au volume d'entrée à la valeur du padding près. Si la foulée est de 2 à N, le kernel se déplace de 2 à N pixels à la fois. Il en résulte un volume de sortie moins important, soit moins de hauteur et moins de largeur. La foulée est à la fois horizontale et verticale.



La dernière foulée horizontale de l'exemple ci-dessus ne permet pas d'inclure entièrement le kernel, elle doit être ignorée. La foulée suivante effectue une foulée verticale de taille S à partir de l'extrême gauche.

La taille de sortie de l'image peut être obtenue en divisant les dimensions d'origine par la valeur de la foulée S. La formule précédente du padding peut être adaptée pour tenir compte de la foulée.

$$\frac{M + 2p - F}{S} + 1 \; x \; \frac{N + 2p - F}{S} + 1$$

Appliqué à l'exemple ci-dessus où M et N = 7, F = 3, P = 0 et S = 2, nous obtenons les dimensions suivantes :

$$M \; et \; N = \frac{7 + 0 - 3}{2} + 1 = \frac{4}{2} + 1 = \; 3$$

Dans un autre exemple où M et N = 8, F = 3, P = 0 et S = 2, nous obtenons les dimensions fractionnées et erronées suivantes :

$$M \; et \; N = \frac{8 + 0 - 3}{2} + 1 = \frac{5}{2} + 1 = \; 3.5$$

En effet, il est impossible de créer un demi-pixel, il faut arrondir la dimension vers le bas, obtenir la valeur plancher. La formule peut être adaptée :

$$\left\lfloor \frac{M + 2p - F}{S} + 1 \right\rfloor \; x \; \left\lfloor \frac{N + 2p - F}{S} + 1 \right\rfloor$$

3.3.7 Un petit bout de code

Nous retrouvons ici un bout de code permettant de réaliser la convolution d'une image en niveaux de gris par un kernel unique. Le parcourt se fait sur le vecteur, ce qui explique le calcul des offsets. Il n'y a pas d'implémentation à l'aide de matrice de OpenCV puisque la convolution est intégrée à l'API. En annexe 14.1, vous trouverez une version plus complète de l'implémentation.

Implémentation personnalisée de la convolution d'une image en niveaux de gris
`//Chaque lignes de l'image`

```
for (int rowIndex = padding; rowIndex < height - padding; rowIndex++)
{
    //Chaque colonnes de l'image
    for (int colIndex = padding; colIndex < width - padding; colIndex++)
    {
        //Calcul de l'offset du pixel courant
        byteOffset = rowIndex * stride + colIndex * 3;

        var k = kernel.Kernel;
        var factor = kernel.Factor;
        gray = 0;

        //Chaque lignes du kernel
        for (int filterRowIndex = -padding; filterRowIndex <= padding; filterRowIndex++)
        {
            //Chaque colonnes du kernel
            for (int filterLineIndex = -padding; filterLineIndex <= padding; filterLineIndex++)
            {
                //Calcul de l'offset du pixel voisin
                calcOffset = byteOffset +
                             (filterLineIndex * 3) +
                             (filterRowIndex * stride);

                //La somme des produits
                gray += (double) (pixelBuffer[calcOffset]) *
                        k[filterRowIndex + padding, filterLineIndex + padding];

            }
        }

        //Calcul la valeur absolue ou la valeur multiplié par le facteur
        gray = filter.ForceAbsoluteValue
            ? Math.Abs(factor * gray)
            : factor * gray;
        //Borne de 0 à 255 niveaux de gris
        gray = gray > 255 ? 255 : gray < 0 ? 0 : gray;

        //Attribue la valeur du pixel à l'image de sortie
        resultBuffer[byteOffset] = (byte) gray;
    }
}
```

3.4 Conclusion

Les filtres simples sont très faciles à implémenter. Le nombre d'itérations est de l'ordre de N (le nombre de pixels), l'implémentation est assez performante que ce soit l'implémentation personnalisée ou la version OpenCv. La partie la plus complexe des filtres simples est la fonction *f*, l'application de *f* à une ou plusieurs des composantes de couleur. Par exemple, la fonction inverse est une simple soustraction tandis que la fonction gamma est l'exponentiel d'une division.

Les filtres de convolution sont beaucoup plus complexes à réaliser mais aussi plus riches dans leur possibilité. Il n'y a pas de limite dans le nombre de filtres possibles, la taille est variable et les combinaisons de valeurs sont presque illimitées. Le résultat des pixels peut lui aussi différer selon le nombre de kernels utilisés ou selon le résultat désiré.

Pour les deux types de filtres, implémenter une classe de base permet d'accélérer le développement et de faciliter l'ajout de nouveaux filtres. L'implémentation de la convolution personnelle n'implémente pas le stride. Il ne nous semblait pas opportun de lui consacrer trop de temps sachant que nous utiliserions la librairie OpenCv. Dans cette implémentation, l'avantage est de lui donner une liste de kernels, et l'algorithme choisit la meilleure méthode

pour calculer la valeur de sortie. Avec la librairie OpenCv, il nous appartient d'extraire la bonne valeur de l'ensemble des matrices de sortie. Heureusement, la librairie regorge de méthodes pour manipuler les matrices.

Chapitre 4
Filtre passe-bas de lissage

Nous avons abordé les généralités, il est maintenant temps de perdre un peu de hauteur pour descendre en contre bas. Plongeons maintenant dans nos premiers filtres et particulièrement dans ces filtres magiques qui permettent de supprimer le bruit. Je me souviens de la première fois ou j'ai appliqué un filtre permettant de débruiter une prise de vue à 6400 iso et mon étonnement de pouvoir récupérer des informations de couleur, de pouvoir améliorer une photo qui me semblait perdue… Dans quelques instants, vous connaîtrez les trucs du magicien, mais n'allons pas trop vite.

Le filtre passe-bas peut diminuer le bruit et/ou atténuer les détails d'une image. Une image sur laquelle a été appliqué un filtre passe-bas aura l'aire plus floue. Dans le cas qui nous occupe, le filtre passe-bas sera utilisé pour supprimer le bruit lors de la détection d'objet. Le bruit pouvant donner lieu à des faux positifs.

4.1 Bruit

Le bruit d'une image numérique est une distorsion de la réalité, qu'elle soit colorimétrique ou géométrique. Deux types de bruit peuvent être présents, le bruit de chrominance qui agit sur la composante colorée du pixel et qui est identifiable par des taches de couleurs aléatoires, et le bruit de luminance qui agit sur la composante lumineuse du pixel et qui se caractérise par des taches plus claires ou plus foncées.

Le bruit peut avoir différentes origines :

- Bruit électromagnétique dû à l'environnement (ex. IRM) ;
- Bruit poivre et sel dû à des pixels morts du capteur, des crasses sur le capteur ou sur la lentille et est souvent nommé bruit impulsionnel ;
- Bruit thermique dû à la surchauffe du capteur, notamment lors de temps de pose prolongé ;
- Bruit de lecture des données du capteur et de sa profondeur de couleur ;
- Bruit induit par la sensibilité ISO du capteur ;
- Bruit de flouté dû à la profondeur de champs (focalisation) ou de bougé lors de la prise de vue ;
- Bruit dû aux aberrations chromatique et sphérique induit par la fabrication des lentilles ;
- Bruit gaussien ajouté en vue de réaliser des tests, il est uniforme à l'environnement, aléatoire et a une distribution gaussienne ;
- Dans notre cas, nous ajouterons le bruit panoramique qui est induit par l'assemblage de plusieurs clichés pour n'en former qu'un seul.

Schématiquement, le bruit peut être visible de différentes façons :

4.2 Filtre moyenneur

Le filtre moyenneur est un filtre passe-bas et il permet de supprimer du bruit en ajoutant du flou. Le principe est de remplacer le pixel par la moyenne pondérée du pixel et de ses voisins. Il existe différents filtres, dont la connexité est de 4 ou 8 voire plus.

$$M_4(x,y) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} * 1/5 \qquad M_8(x,y) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} * 1/9$$

$$M_{24}(x,y) = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} * 1/25 \qquad M_{49}(x,y) = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} *$$

$$1/49$$

Remplacer un pixel par la moyenne de ses voisins permet de faire disparaitre les transitions brutales d'intensité des pixels. Le bruit est caractérisé par cette transition brutale, il sera donc éliminé en même temps que les contours et bords qui eux aussi ont cette caractéristique.

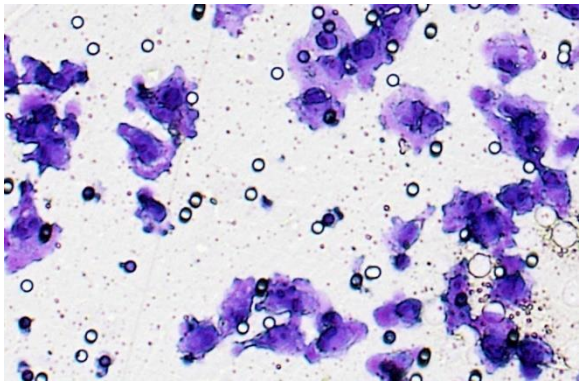


Figure 14 : Originale couleur

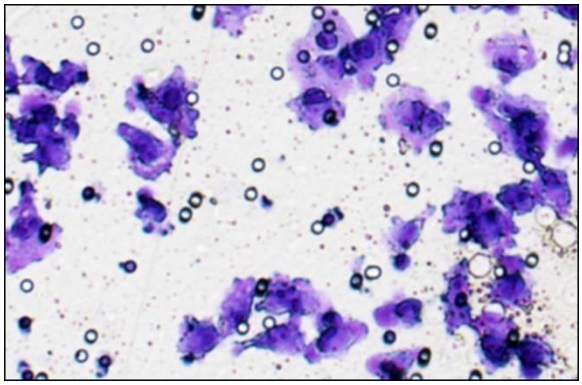


Figure 15 : Filtre moyenneur connexité 48 taille 7x7

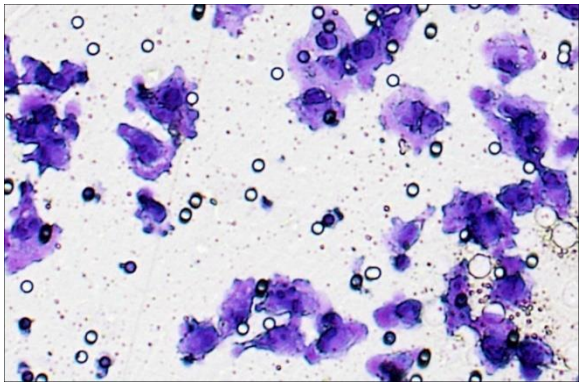


Figure 16 : Filtre moyenneur connexité 4 taille 3x3

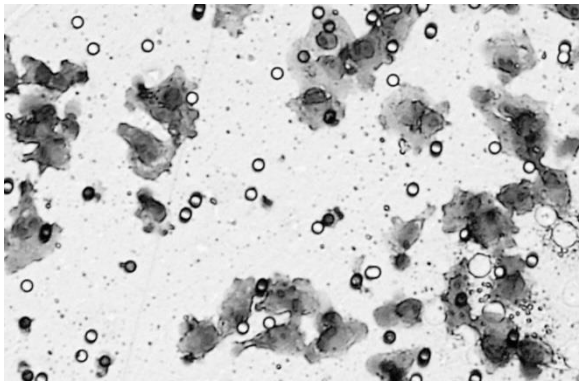


Figure 17 : Originale niveaux de gris

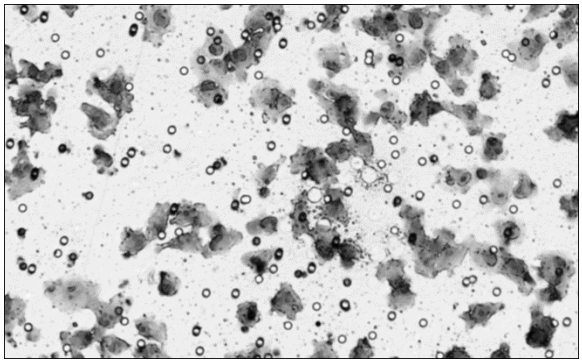


Figure 18 : Filtre moyenneur connexité 48 taille 7x7

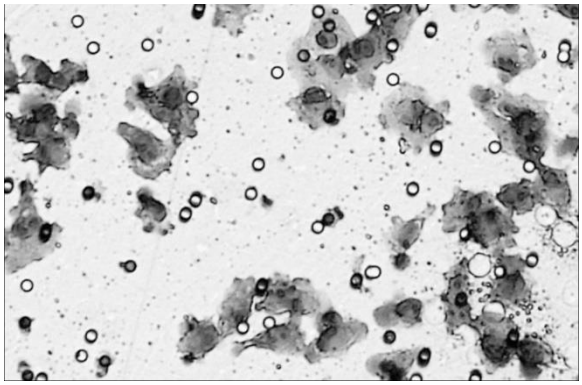


Figure 19 : Filtre moyenneur connexité 4 taille 3x3

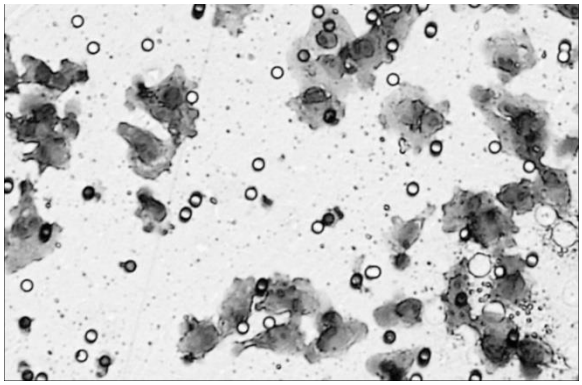


Figure 20 : Filtre moyenneur connexité 8 taille 3x3

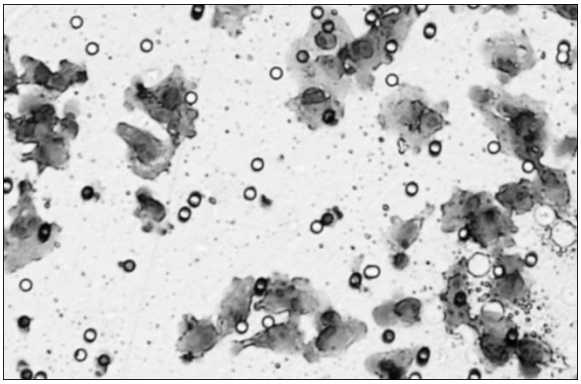


Figure 21 : Filtre moyenneur connexité 24 taille 5x5

Les filtres ont été implémenté de deux façon différentes, une implémentation personnalisée

permettant d'appréhender les concepts et une seconde à l'aide de l'API OpenCV. Un extrait de code tel que celui-ci-dessous permettra d'illustrer la mise en œuvre des différents filtres… Le code des expérimentations a permis d'illustrer le mémoire. Pour rappel, le code est sur un repository public de GitHub.
https://github.com/kwilvers/MasterImageProcessing/tree/master/CancerCellDetection.

Implémentation personnalisée

```
//Chargement de l'image
Bitmap v = (Bitmap)Bitmap.FromFile(@".\echantillon.png");
//Filtre moyenneur connexité 4 taille 3
var resConv = Convolution.Convolve(v, new MeanFilterC4S3());
//Filtre moyenneur Connexité 8 taille 3
var resConv = Convolution.Convolve(v, new MeanFilterC8S3());
//Enregistrement de l'image de sortie
resConv.Output.Save(@".\GrayMeanFilterC48S7Test.png");
```

Implémentation OpenCV

```
//Chargement de l'image
Mat v = Cv2.ImRead(@".\echantillon.png");
Mat output = new Mat();
//Filtre moyenneur 9x9
Cv2.Blur(v, output, new Size(9,9), new Point(-1,-1), BorderTypes.Default );
//Enregistrement de l'image de sortie
Cv2.ImWrite(@".\CvMeanS9Filter.png", output);
```

4.3 Filtre gaussien ou flou gaussien

Le filtre gaussien ou flou gaussien est un filtre passe-bas de lissage, il permet d'atténuer les coupures franches entre une série de pixels clairs et de pixels foncés et de créer un flou. Dans le cas de la détection de contour, le bruit d'une image peut influencer plus ou moins fortement la détection. Par exemple, le filtre de Laplace est très sensible au bruit. En plus de créer un flou, le filtre permet de supprimer une partie du bruit. Le filtre gaussien est moins efficace pour éliminer le bruit, mais il dégrade moins les détails que le filtre moyenneur.

Un flou gaussien est basé sur l'écart type, pour chaque valeur de celui-ci, il existe un kernel. Il existe par conséquent une multitude de kernels de différentes tailles permettant d'obtenir différents niveaux de lissage. Plus le filtre est grand, plus le flou est important.

$$G(x,y) = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} * 1/16 \qquad G(x,y) = \begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 6 & 8 & 6 & 2 \\ 3 & 8 & 10 & 8 & 3 \\ 2 & 6 & 8 & 6 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix} * 1/98$$

$$G(x,y) = \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} * 1/273 \quad G(x,y) = \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * 1/159$$

$$G(x,y) = \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 18 & 30 & 18 & 4 \\ 6 & 30 & 48 & 30 & 6 \\ 4 & 18 & 30 & 18 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} * 1/300 \quad G(x,y) = \begin{bmatrix} 1 & 1 & 2 & 1 & 1 \\ 1 & 2 & 4 & 2 & 1 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \\ 1 & 1 & 2 & 1 & 1 \end{bmatrix} * 1/52$$

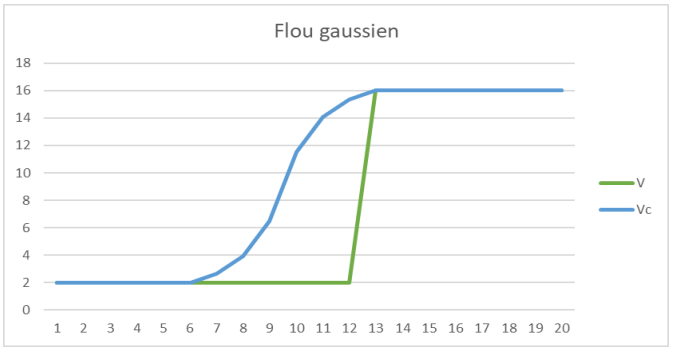
$$G(x,y) = \begin{bmatrix} 1 & 1 & 2 & 2 & 2 & 1 & 1 \\ 1 & 2 & 2 & 4 & 2 & 2 & 1 \\ 2 & 2 & 4 & 8 & 4 & 2 & 2 \\ 2 & 4 & 8 & 16 & 8 & 4 & 2 \\ 2 & 2 & 4 & 8 & 4 & 2 & 2 \\ 1 & 2 & 2 & 4 & 2 & 2 & 1 \\ 1 & 1 & 2 & 2 & 2 & 1 & 1 \end{bmatrix} * 1/140$$

Si la gaussienne centrale G du dernier filtre est appliquée sur le vecteur V, le vecteur Vc est obtenu en sortie. Une représentation graphique permet de constater que le passage de la valeur 2 à la valeur 16 est progressive et plus douce. Ça peut sembler paradoxal de diminuer les contrastes avant de détecter les contours, mais le but est de supprimer une partie du bruit.

G = [2, 4, 8, 16, 8, 4, 2]

V = [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.00, 2.0, 2.0, 2.0, 2.0, 2.0, 16.0, 16.0, 16.0, 16.0, 16.0, 16.0, 16.0, 16.0]

Vc = [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.6, 3.9, 6.5, 11.6, 14.1, 15.4, 16.0, 16.0, 16.0, 16.0, 16.0, 16.0, 16.0, 16.0]



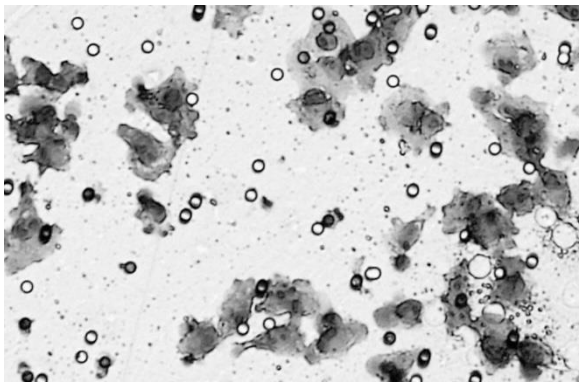


Figure 22 : Originale

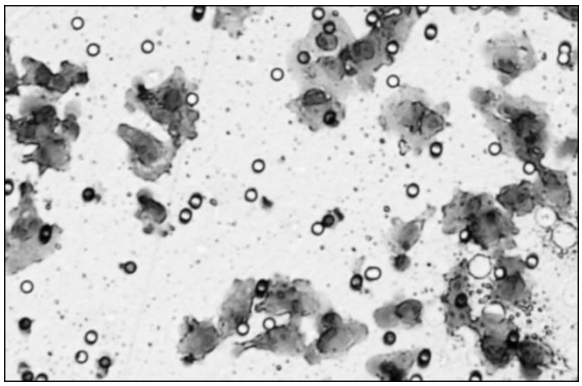


Figure 23 : Filtre gaussien140 taille 7 gris

Implémentation personnalisée
`//Filtre gaussien 9x9 poids 1` `var resConv = Convolution.Convolve(res, new GaussianFilter(9, 1));`
Implémentation OpenCV
`//Filtre gaussien 9x9 sigma 0` `Cv2.GaussianBlur(v, output, new OpenCvSharp.Size(9, 9), 0, 0, BorderTypes.Default);`

4.4 Calcul d'un kernel gaussien

Les kernels utilisés dans la littérature sont rarement identiques. Dans le point précédent, sept des kernels rencontrés ont été énoncés. Cette même littérature ne mentionne que très rarement la possibilité de calculer les kernels de filtre gaussien.

Les méthodes de calcul d'un filtre gaussien peuvent être diverses. Ci-dessous, une seule sera retenue. La formule suivante permet de calculer le poids de chacune des cellules d'une matrice de taille *m*.

$$K(x,y) = \frac{1}{2\,\pi\,\sigma^2}\,e\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

K(x,y) : La valeur de la cellule à l'indice x, y de la matrice du kernel

π : Pi, la valeur constante de la circonférence d'un cercle par rapport à son diamètre

σ : Un facteur spécifique libre défini par l'utilisateur

e : La valeur du nombre d'Euler défini par une constante (2.71828182846) exprimant la fonction exponentielle du dernier terme

x, y : Représente l'indice de la ligne et de la colonne de la matrice, l'indice 0,0 est au centre de cette matrice

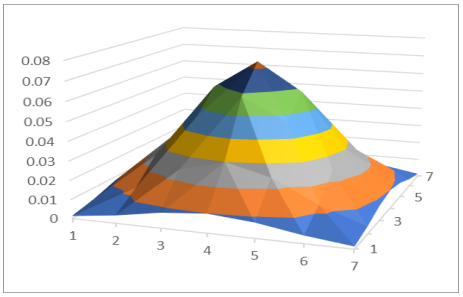
Le facteur de correction du kernel F_k est égal à la somme des cellules du kernel. Cependant, celui-ci peut être ramené à 1 afin de le normaliser en multipliant chaque cellule du kernel par

le quotient de 1 par F_k.

$$F_k = \sum_{x=1}^{n} \sum_{y1}^{n} K(x,y) \qquad K'(x,y) = K(x,y) * \frac{1}{F_k}$$

Pour un kernel de taille 7 dont le facteur σ = 1.5, le kernel suivant est obtenu :

0.00130	0.00394	0.00767	0.00957	0.00767	0.00394	0.00130
0.00394	0.01196	0.02329	0.02908	0.02329	0.01196	0.00394
0.00767	0.02329	0.04535	0.05664	0.04535	0.02329	0.00767
0.00957	0.02908	0.05664	0.07074	0.05664	0.02908	0.00957
0.00767	0.02329	0.04535	0.05664	0.04535	0.02329	0.00767
0.00394	0.01196	0.02329	0.02908	0.02329	0.01196	0.00394
0.00130	0.00394	0.00767	0.00957	0.00767	0.00394	0.00130



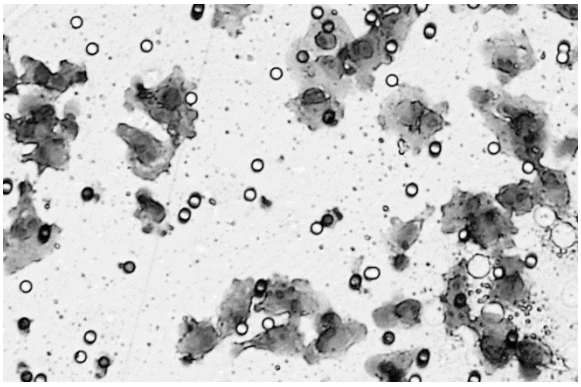


Figure 24 : Originale niveaux de gris

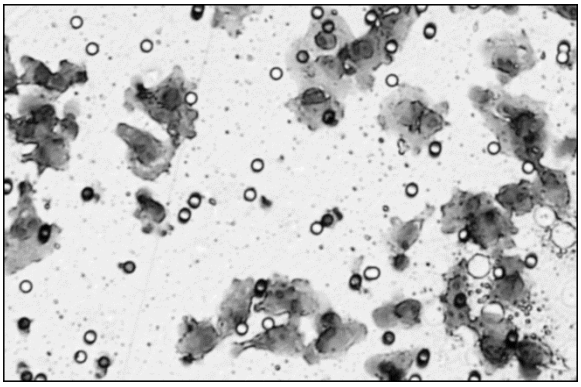


Figure 25 : Filtre gaussien taille = 7, σ = 1.5

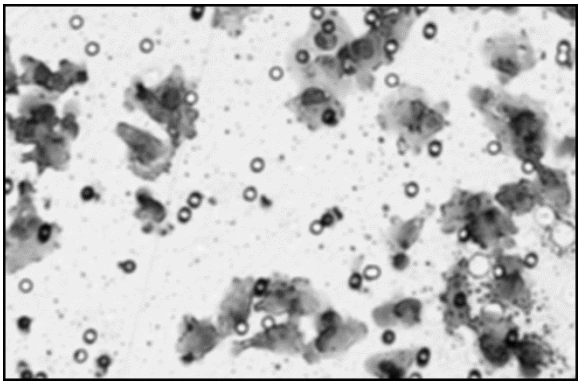


Figure 26 : Filtre gaussien taille 11, σ =5

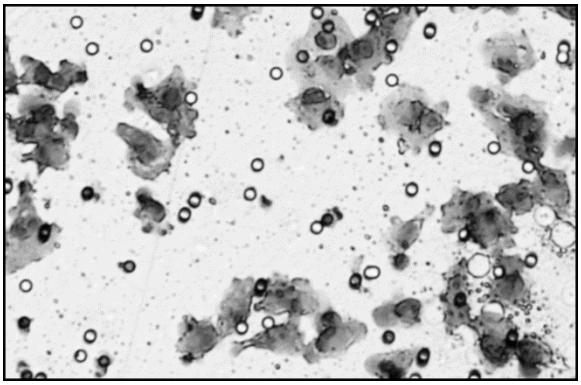
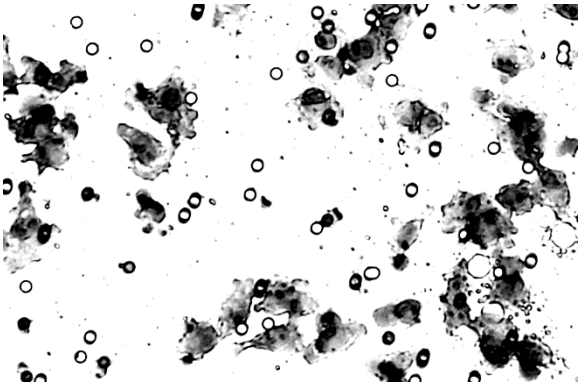


Figure 27 : Filtre gaussien taille 11, σ =1.5



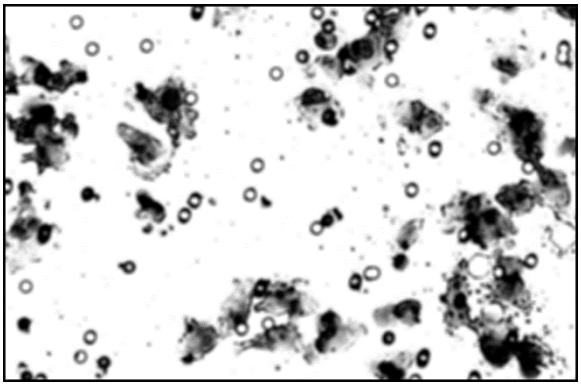


Figure 29 : Augmentation des contrastes de 50 %, filtre

Figure 28 : Augmentation des contrastes de 50 % *gaussien taille 11, σ =5*

Implémentation personnalisée
`//Filtre gaussien calculé 11x11 Sigma 5` `var resConv = Convolution.Convolve(res, new GaussianFilter(11, 5));`
Implémentation OpenCV
`//Filtre gaussien 11x11 Sigma 5` `Cv2.GaussianBlur(v, output, new OpenCvSharp.Size(9, 9), 5, 5, BorderTypes.Default);`

4.5 Filtre Médian

Le filtre médian est un filtre non-linéaire et non un filtre de convolution. Son principe consiste à remplacer la valeur du pixel par la valeur médiane de ses voisins. Si plusieurs voisins sont bruités, ils n'influencent pas le résultat final. Le filtre induit un léger lissage puisque tous les pixels sont traités.

Le filtre médian est meilleur sur du bruit poivre et sel et il conserve mieux les contours que les filtres moyenneur ou gaussien, mais il est aussi beaucoup plus gourmand en termes de ressources. En effet, pour obtenir la valeur médiane, il faut trier le voisinage par ordre croissant, et plus le voisinage est important plus les performances se dégradent. Il faut donc veiller à utiliser un algorithme de tri performant. De plus, le filtre médian n'introduit pas de valeurs autres que celles déjà présentes dans l'image, ce qui n'est pas le cas du filtre moyenneur. Il n'est cependant pas efficace sur du bruit gaussien, ni sur du bruit recouvrant plus de la moitié du voisinage.

Le voisinage peut être de différentes tailles tel un kernel de convolution, par exemple 3x3 ou 5x5.

Pour un pixel et son voisinage de 3x3, il faut trier la valeur des pixels et retenir la valeur médiane.

1	1	2
2	30	3
2	3	4

Pixel trié : [1, 1, 2, **2**, 2, 3, 3, 4, 30] Valeur médiane : 2

Ici, le pixel central de valeur 30 est remplacé par la valeur médiane 2.

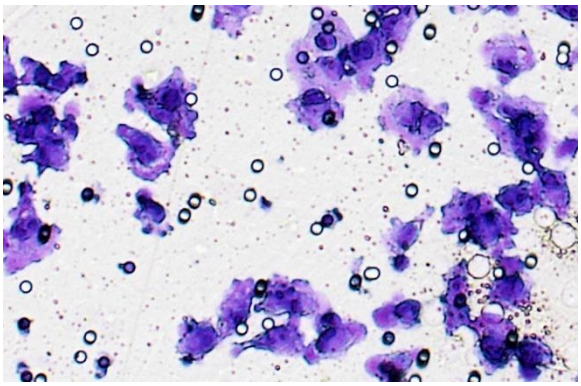


Figure 30 : Originale

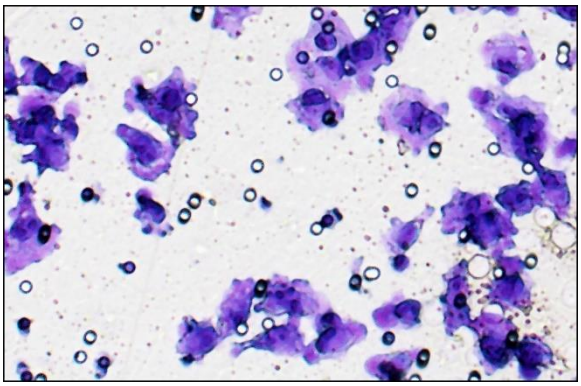


Figure 31 : Filtre médian de taille 7

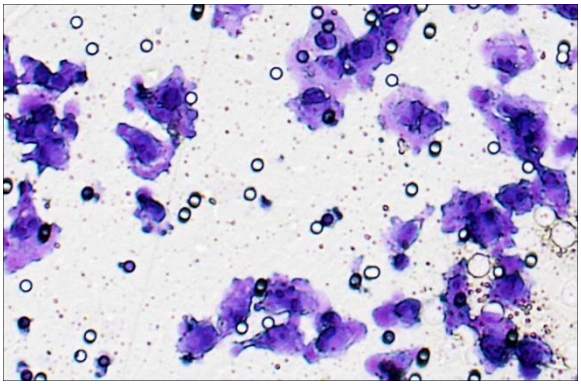


Figure 32 : Filtre médian de taille 3

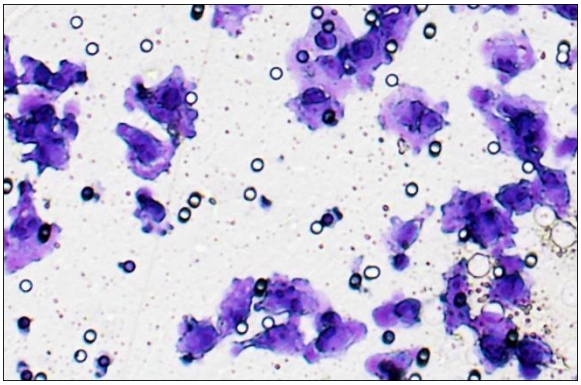


Figure 33 : Filtre médian de taille 5

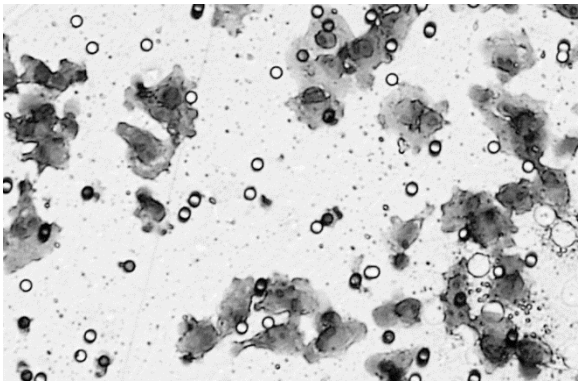


Figure 34 : Originale

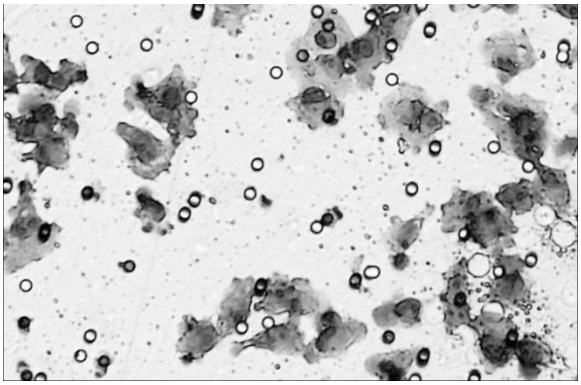


Figure 35 : Filtre médian de taille 3 gris

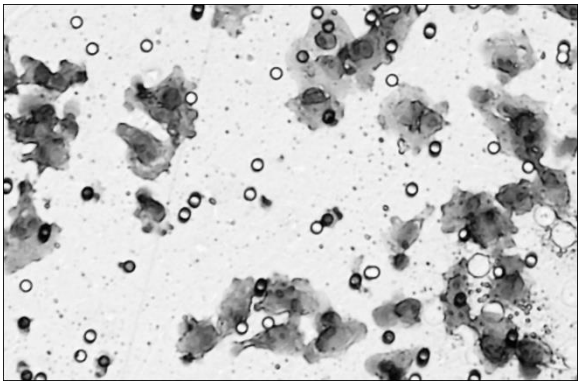


Figure 36 : Filtre médian de taille 5 gris

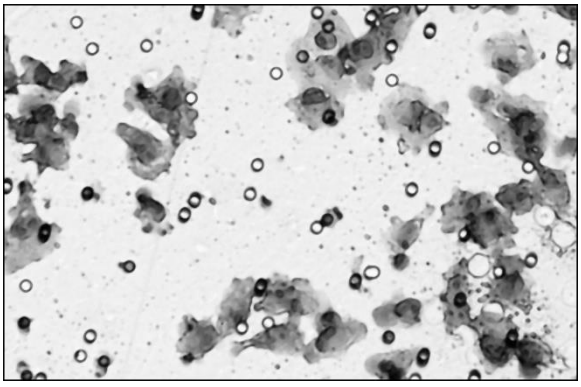


Figure 37 : Filtre médian de taille 7 gris

Implémentation personnalisée
`//Filtre median 7x7`

```
var resConv = ConvolutionMedian.Convolve(v, new MedianFilterS7(), false);
```

Implémentation OpenCV

```
//Filtre median 5x5
Cv2.MedianBlur(v, output, 5);
```

4.6 Filtre de Turkey, la médiane des médianes

Le filtre de Tukey repose sur le même principe que le filtre médian si ce n'est par son approche de médiane des médianes. Il s'agit ici de réaliser la médiane de chacune des lignes du voisinage et d'extraire de ces médianes la médiane finale.

1	1	2
2	30	3
2	3	4

Pour l'exemple précédent, nous obtenons les trois vecteurs de valeurs triées et leurs médianes respectives : [1, **1**, 2] - [2, **3**, 30] - [2, **3**, 4]

Ensuite, un nouveau vecteur trié reprend l'ensemble des médianes : [1, **3**, 3], la nouvelle valeur du pixel est 3.

Ce filtre ne présente pas de véritable intérêt pour des kernels de petites tailles. Les kernels de plus grande taille pourraient être concernés par cette version de filtres médians mais ceux-ci induisent un lissage trop important pour être utilisé dans la détection de contour.

4.7 Filtre médian pondéré

Le filtre médian pondéré fonctionne de la même façon que le filtre médian mais utilise un kernel pour pondérer le nombre d'instances de chacun des pixels. Les filtres médians pondérés permettent de conserver des structures de géométrie complexe.

Un kernel de pondération :

2	1	2
1	3	1
2	1	2

Une portion d'image

1	1	2
2	30	3
2	3	4

Si le kernel de pondération ci-contre est appliqué au pixel et à ses voisins, le vecteur suivant est obtenu : [1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 4, 4, 30, 30, 30] et la valeur médiane : 2.

Dans l'exemple ci-dessous, une comparaison des filtres médian, médian pondéré et moyenneur :

56	37	42
19	7	12
41	83	65

Filtre médian : [7, 12, 19, 37, 41, 42, 56, 65, 83] la médiane : 41

Filtre de Tukey : [37, 42, 56] [7, 12, 19] [41, 65, 83] donne [12, 42, 65] la médiane : 42

2	1	2
1	3	1
2	1	2

Filtre médian pondéré : [7, 7, 7, 12, 19, 37, 41, 41, 42, 42, 56, 56, 65, 65, 83] la médiane : 41

Filtre moyenneur : 362/9 la moyenne 40

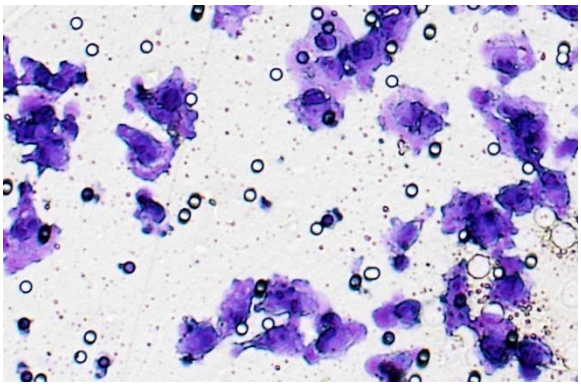


Figure 38 : Originale

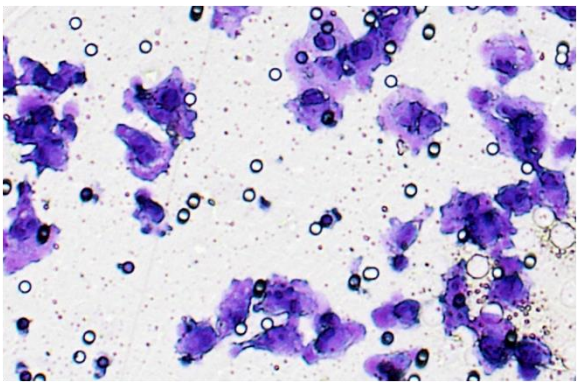


Figure 39 : Filtre médian pondéré

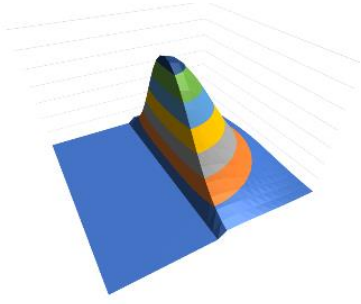
Implémentation personnalisée

```
//Filtre median pondéré 7x7
var resConv = ConvolutionMedian.Convolve(v, new MedianWeightedFilterS5(), true);
```

4.8 Filtre bilatéral

Le filtre bilatéral est méconnu ; il n'est évoqué que très rarement dans la littérature, ce qui explique qu'il est présenté en fin de chapitre, après la rédaction de la conclusion. Ce filtre dépend de la distance euclidienne ($\sqrt{(xA - xB)^2 + (yA - yB)^2}$) au point à lisser et de la similarité des intensités entre les pixels voisins du pixel central. Les pixels dont l'intensité est similaire seront fortement lissés. Les pixels différents ne seront que faiblement lissés.

Les filtres précédents suppriment le bruit et lissent les bords. Ils deviennent moins nets. Avec le filtre bilatéral, ce problème est fortement atténué grâce à un paramètre qui permet d'identifier deux pixels dont les intensités sont similaires.



Ce filtre peut être considéré comme une amélioration du filtre gaussien. Le filtre gaussien va lisser la courbe en incluant une pente lors de changement brusque d'intensité. Le filtre bilatéral quant à lui lisse sans induire cette pente douce et conserve le changement brutal entre deux pixels. Ci-contre, une représentation d'un kernel bilatéral, la moitié du kernel lisse tandis que l'autre conserve les changements brutaux. Une implémentation simpliste est réalisée de la même façon que le filtre gaussien, la moitié du kernel est mis à zéro pour conserver le changement brutal.

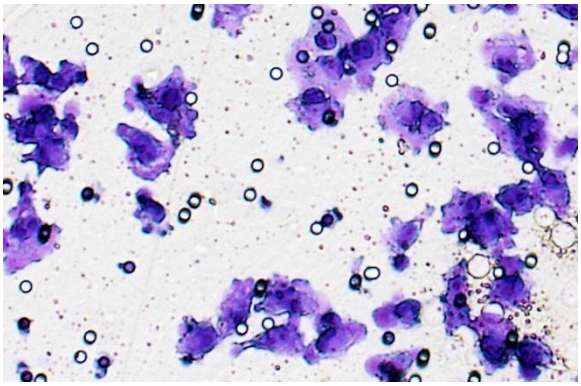


Figure 40 : Originale couleur

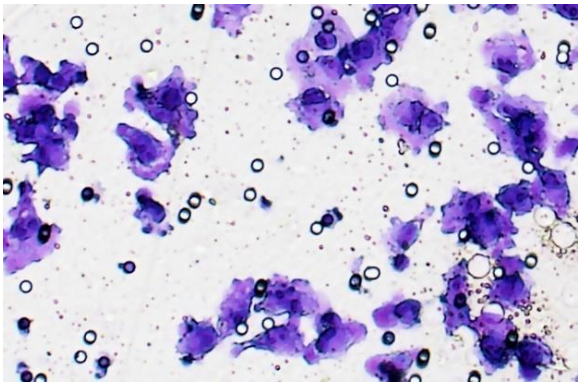


Figure 41 : Filtre bilatéral de taille 15

Implémentation personnalisée

```
//Filtre bilatéral de taille 11
var resConv = Convolution.Convolve(v, new BilateralFilter(11, 5));
```

Implémentation OpenCV

```
//Taille de kernel
int kernel_length = 15;
//Bilateral blur
Cv2.BilateralFilter(v, output, kernel_length, kernel_length * 2, kernel_length / 2);
```

4.9 Conclusion

Le lissage est un domaine assez restreint. Il ne compte que quelques filtres, mais ils présentent les uns et les autres divers avantages.

Le filtre moyenneur a une tendance à dégrader beaucoup plus les images que le filtre médian ou le filtre gaussien.

Le filtre médian a l'avantage de conserver les couleurs. Il est particulièrement efficace sur le bruit poivre et sel. Cependant, plus les filtres médians grandissent, plus ils ont tendance à diminuer le nombre de couleurs du spectre de l'image. Le filtre médian pondéré pourrait permettre d'augmenter les contrastes, mais il est complexe à utiliser, il faudrait l'expérimenter et en définir N afin d'en trouver un qui conviendrait.

Le filtre de Turkey souffre de gros problèmes de performance. Si l'algorithme de tri est performant à une complexité de $n\ log\ n$, la complexité de traitement d'un kernel carré de n x n est $n + 1\ (n\ log\ n)$ et ce pour chaque pixel. Vu les lenteurs théoriques de ce dernier, il n'a pas été implémenté. Il n'est pas non plus présent dans OpenCV.

Le filtre gaussien est le plus polyvalent, et est le plus utilisé. Comme signalé plus haut, la littérature ne mentionne que rarement la possibilité de calculer un kernel gaussien, même si ça semble évident de pouvoir calculer une gaussienne. Il est évident que calculer le kernel est plus sensé que d'établir une liste des kernels les plus fréquemment rencontrés.

Enfin le filtre bilatéral est très bien pour lisser et en même temps conserver les détails, c'est-à-dire les contours. En grande taille, il réduit le nombre de couleur présente en réalisant des

agglomérats de pixels de même couleur. L'implémentation simpliste du filtre bilatéral n'est pas tout à fait correcte, le lissage se fait moyennement et la pente brutale est conservée. Le gros défaut est de ne pas tenir compte de façon systématique de la différence d'intensité. L'implémentation de OpenCV est meilleure et travaille sur un niveau de tolérance de couleur et une largeur d'espace de couleur.

Chapitre 5
Filtre passe-haut de détection

Tirons un peu sur la corde du brûleur et reprenons un peu de hauteur. Vous l'aurez compris, s'il y a des bas, il y a heureusement des hauts et les hauts, au contraire du filtre passe bas, vont nous permettre d'accentuer les détails, accentuer les contrastes. Nous avons jusqu'ici amélioré notre image, converti celle-ci en niveaux de gris et enfin supprimé le bruit et flouté l'image. Maintenant, il nous faut « détecter les cellules » dans l'image.

Le filtre passe-haut accentue les contours et amplifie les détails et de la même façon le bruit résiduel. Une image floue sur laquelle est appliqué un tel filtre, semblera moins floue, les détails seront plus marqués. Vous pouvez penser que c'est paradoxal de flouter une image avant d'en augmenter les contrastes, mais nous avons avant tout supprimé le bruit et aplati les contours avant de les faire ressortir.

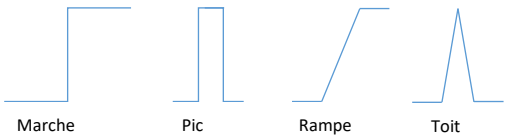
Il existe un très grand nombre de filtres de détection, chacun tentant d'améliorer le précédent. Il en existe des plus spécifiques visant un domaine particulier, et d'autres plus généralistes. Il serait impossible de les aborder tous, les filtres les plus fréquemment rencontrés dans la littérature seront présentés ci-dessous.

5.1 Notion de contour

La notion de contour ne date pas d'hier puisqu'elle apparait fin des années 60. Depuis plus de 50 ans, des milliers de références ont été recensées sur le sujet. Le concours annuel ILSVRC (ImageNet Large Scale Visual Recognition Challenge) vise à présenter des filtres ou algorithmes cherchant à identifier dans une image des objets, leurs nombres et positions. Depuis quelques années, le machine learning remporte haut la main ce type de concours en améliorant significativement les performances de détection, mais il s'agit d'une autre histoire…

Qu'est-ce qu'un contour ? Un contour dans une image est représenté par la variation brutale de la luminosité entre deux pixels ou dans une suite de pixels adjacents, soit par la variation significative de la valeur du niveau de luminosité des pixels. Bien que moins fréquemment utilisés, les contours peuvent être détectés dans une image au format RGB, chaque couche est traitée comme une couche unique en niveaux de gris et les résultats des différentes couches sont ensuite agrégés.

Un contour n'est que très rarement localisé sur deux pixels adjacents. Il est souvent localisé sur une plage de pixels et peut être perturbé par du bruit. Les contours peuvent être classifiés entre quatre types : une marche, un pic, une rampe et un toit. La rampe et le toit sont des contours progressifs caractérisés par une variation de la luminosité répartie sur plusieurs pixels.



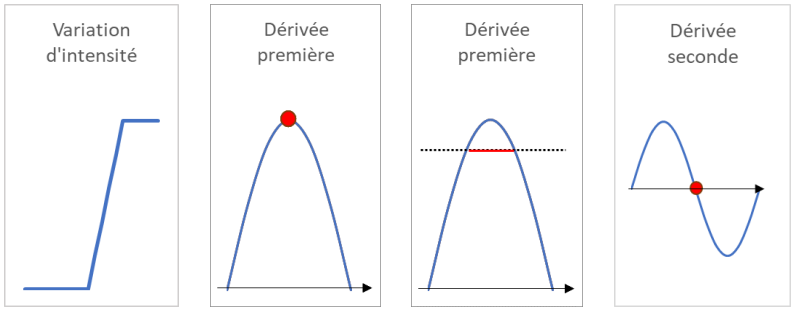
Un contour progressif est une variation du gradient du niveau de luminosité. De nombreux algorithmes utilisent cette propriété au travers du calcul des dérivées premières et secondes.

5.2 Dérivée première et dérivée seconde

Le gradient est caractérisé par la discontinuité de la fonction d'intensité d'une image. Les principes de détection reposent sur l'étude des dérivées de la fonction d'intensité de l'image. La détection des maxima locaux du gradient et les passages par zéro du laplacien. La méthode dérivative est utilisée pour éliminer les variations locales d'intensité des pixels.

Certains détecteurs de contours se base sur la dérivée première, l'étude du gradient tel que Sobel, Prewitt, Roberts. L'emplacement du contour se situe à la variation brutale d'intensité, correspondant au saut de la dérivée première. Il s'agit donc de définir les maximas locaux. Dans la seconde figure ci-dessous, le bord est identifié par le point rouge qui se trouve au sommet de la courbe. Cependant, le bord est généralement dépendant d'un seuil, identifié par la ligne noire de la troisième figure qui donne un bord plus épais.

D'autres détecteurs sont basés sur l'étude de la dérivée seconde et des passages par zéro tel que le masque de Laplace. Pour ces filtres, le bruit est plus impactant, il est donc courant d'utiliser de manière combinée un lissage (par exemple une gaussienne) et un filtre laplacien. Ici, on s'intéresse au passage par zéro qui représente le bord et qui ne mesure qu'un pixel, identifié en rouge dans la quatrième figure.



5.3 L'approche par le gradient

Le gradient d'une image est un changement directionnel de la luminosité des niveaux de gris. C'est un vecteur caractérisé par une amplitude, lié à la variation d'intensité, et une direction orthogonale, liée au contour. Pour calculer la direction du gradient, il faut utiliser deux masques de convolution dont le calcul des dérivées est perpendiculaire l'un à l'autre. Par exemple, les lignes horizontales et les lignes verticales. Les deux vecteurs de gradient obtenus servent ensuite au calcul de la direction.

Par l'approche de dérivée première, le gradient G est calculé pour chaque pixel de l'image.

L'image étant en deux dimensions, deux directions sont nécessaires pour calculer l'amplitude et la direction exactes du gradient. Les lignes et colonnes de l'image donnent un bon point de départ pour calculer les deux gradients, G_x pour les colonnes et G_y pour les lignes. Les dérivées peuvent être obtenues par deux kernels assez simples et très sensibles au bruit, un horizontal et l'autre vertical.

$$G(x,y) = \begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix} \quad G(x,y) = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

L'amplitude M peut être obtenue par l'une des trois formules suivantes, en fonction du nombre de gradients et de la performance désirée :

La racine carrée de la somme des carrés des gradients : $M(x,y) = \sqrt{(G_x(x,y)^2 + G_y(x,y)^2)}$

La somme des valeurs absolues des gradients : $M(x,y) = |G_x(x,y)| + |G_y(x,y)|$

La valeur maximale des gradients : $M(x,y) = Max(G_x(x,y), G_y(x,y))$

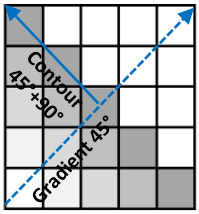
Dans le cas d'un filtre employant plus de deux kernels de convolution, la valeur maximale des gradients est privilégiée, et comporte autant de termes de gradient G_n que de kernels.

La direction du gradient D est obtenue par la tangente inverse du rapport de G_y sur G_x, soit

$$d(x,y) = \begin{cases} Gx = 0 \; alors \; Gy \\ Gy = 0 \; alors \; 90° \\ arctan\left(\dfrac{G_y(x,y)}{G_x(x,y)}\right) \end{cases}$$

La direction est l'angle d'évolution du gradient. Pour un filtre comportant plus de deux kernels, la direction sera équivalente à la direction du kernel dont le gradient est maximum. Les kernels ont une orientation normalisée (0°, 45°, 90°, 135°, 180°). Il est possible d'utiliser celle-ci pour cartographier la direction de l'ensemble des contours. Un contour est perpendiculaire par rapport à son gradient, pour un angle normalisé de 0 à 180 degrés, l'orientation du contour D_c est obtenue en ajoutant ou soustrayant 90° à l'angle du gradient.

$$D_c(x,y) = \begin{cases} d(x,y) + 90 \; si \; d(x,y) < 90 \\ d(x,y) - 90 \; si \; d(x,y) \geq 90 \end{cases}$$



Les filtres gradients ont tous plus ou moins les mêmes défauts. Ils sont sensibles au bruit, le bord est épais et interrompu. Ils peuvent être corrigés par des filtres de lissage.

L'approche traditionnelle de détection liée au gradient se décompose en quatre étapes :

1. Lissage de l'image plus ou moins doux pour supprimer le bruit sans détruire les contours ;
2. Accentuation des contours à l'aide d'un filtre d'accentuation ;
3. Extraction des extrema locaux dans la direction du gradient, soit déterminé que le point p est localement maximal sur la droite passant par p dans la direction de p ;
4. Seuillage par hystérésis des extrema, soit utiliser deux seuils S_b et S_h et sélectionner les pixels dont le gradient est supérieur à S_b où le pixel est connecté à un chemin constitué de pixels dont le gradient est supérieur à S_b et connecté à un pixel dont le gradient est supérieur à S_h.

5.4 Filtre vertical

Le calcul des dérivées a montré qu'il était possible de détecter des contours verticaux et horizontaux mais était fortement sensible au bruit. Le filtre vertical permet de détecter des bords à tendance verticale, soit la frontière entre le noir et le blanc. Avec le filtre horizontal, ce sont les plus simples et intuitifs à comprendre. Le kernel va tenter de trouver des pixels clairs à gauche et sombres à droite tout en effectuant un lissage vertical. En fin de convolution, les pixels négatifs seront forcés à 0. Il est possible de détecter des contours inverses en forçant les pixels négatifs à leur valeurs absolues.

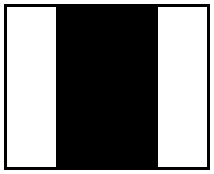
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

*

1	0	-1
1	0	-1
1	0	-1

=

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0



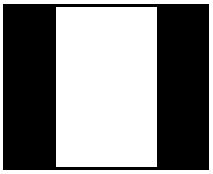
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10

*

1	0	-1
1	0	-1
1	0	-1

=

30	0	0	30
30	0	0	30
30	0	0	30
30	0	0	30



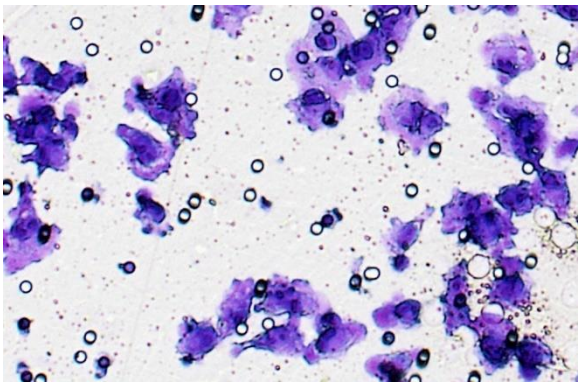


Figure 42 : Originale couleur

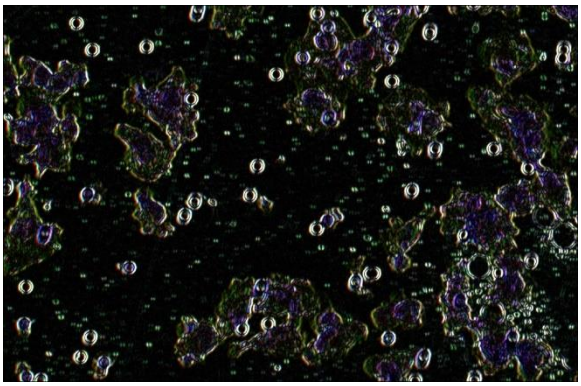


Figure 43 : Filtre vertical couleur

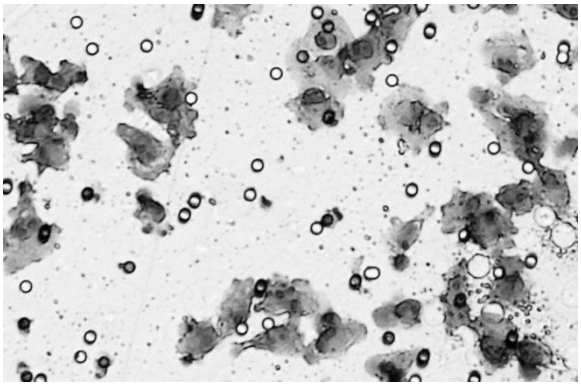


Figure 44 : Originale en niveaux de gris

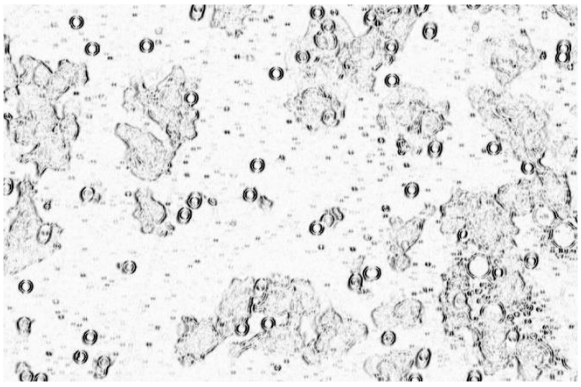


Figure 45 : Filtre vertical gris inverse

Implémentation personnalisée
`//Filtre vertical` `var res = Convolution.Convolve(v, new VerticalFilter());`
Implémentation OpenCV

`//Filtre vertical`
`Mat kernelH = new Mat(3, 1, MatType.CV_32F);`
`kernelH.Set(0, 0,  1.0f);`
`kernelH.Set(1, 0,  0.0f);`
`kernelH.Set(2, 0, -1.0f);`
`Cv2.Filter2D(v, output, v.Depth(), kernelH, new Point(-1, -1), 0, BorderTypes.Default);`

5.5 Filtre Horizontal

Le filtre horizontal diverge par son kernel et le sens de détection des traits qui sont orientés de haut en bas à la place de gauche à droite. En comparant les deux images de sortie, on constate facilement la différence d'orientation des ronds.

$$G(x,y) = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

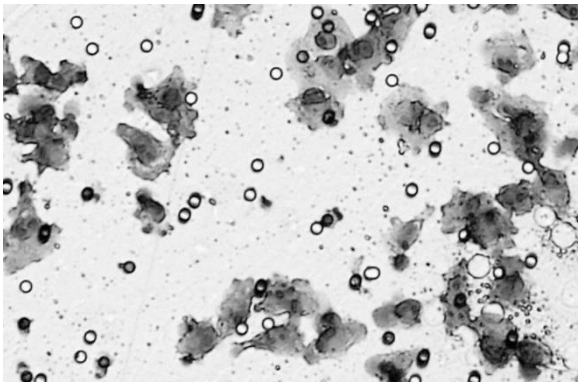


Figure 46 : Originale en niveaux de gris

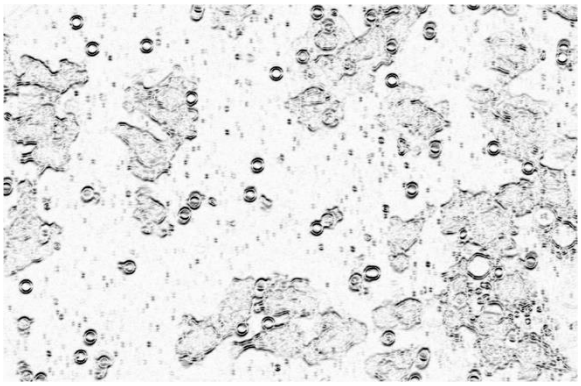


Figure 47 : Filtre horizontal gris inverse

Implémentation personnalisée

```
//Filtre horizontal
var res = Convolution.Convolve(v, new HorizontalFilter());
```

Implémentation OpenCV

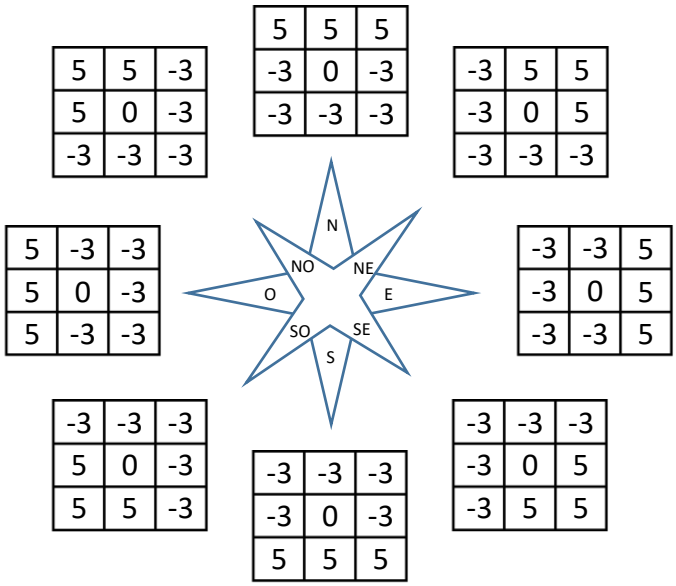
```
//Filtre horizontal
Mat kernelH = new Mat(1, 3, MatType.CV_32F);
kernelH.Set(0,0,1.0f);
kernelH.Set(0, 1,0.0f);
kernelH.Set(0, 2,-1.0f);
Cv2.Filter2D(v, output, v.Depth(), kernelH, new Point(-1, -1), 0, BorderTypes.Default);
```

5.6 Filtre directionnel ou boussole

Les filtres directionnels ou boussole calculent le gradient dans les directions sélectionnées. De la même façon que les filtres comportant deux kernels de convolution, les filtres directionnels comportent jusqu'à huit kernels de convolution, représentant chacun une orientation comme sur le cadran d'une boussole. Pour une telle approche, le calcul du maximum des gradients est utilisé. La valeur maximale des gradients où k représente l'une des orientations de la boussole :

$$M(x,y) = Max_k(|G_k(x,y)|)$$

Ci-dessous l'exemple de kernels directionnels. Les filtres de Sobel, Prewit, Kirsch, Robinson ou encore Scharr peuvent être utilisés comme filtres directionnels. Il suffit de faire pivoter les valeurs du kernel autour de l'axe tel qu'avec l'exemple du filtre de Kirsch.



5.7 Filtre de Roberts (1962)

Le filtre de Roberts recherche les contours en fonction des dérivées des directions des diagonales. L'angle maximum des contours détectés est de 45° Il est composé de deux kernels de convolution dx et dy.

$$\frac{di}{dx} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad \frac{di}{dy} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

La valeur finale du contour est calculée soit par la racine carrée de la somme des carrés des deux convolutions, soit par le maximum des valeurs absolues des deux convolutions :

$$\sqrt{\left(\frac{dI}{dx}\right)^2 + \left(\frac{dI}{dy}\right)^2} \qquad ou \qquad \max\left(\left(\frac{dI}{dx}\right), \left(\frac{dI}{dy}\right)\right)$$

1	4
9	5

*

0	1
-1	0

= 4 + -9 = -5

1	0
0	-1

= 1 + -5 = -4

$= \sqrt{-5^2 + -4^2}$ = $\sqrt{41}$ = 6,4 la valeur du pixel est 6

Ou par le maximum de |-5| et |-4| = 5

Ce filtre possède le défaut d'être fortement sensible au bruit. Il est préférable d'appliquer au préalable un filtre passe-bas afin de lisser les hautes fréquences. Ce filtre détecte trop de contours et les contours diagonaux sont bien détectés à l'instar des verticaux et horizontaux. Il est très rapide et conserve les détails car il utilise des matrices de 2x2. Dans ce cas-ci, la taille du kernel est paire, le point d'ancrage n'est pas central, il est à l'indice [0,0].

La direction du gradient peut être obtenue par la formule suivante :

$$d(x,y) = \frac{\Pi}{4} + arctan\left(\frac{G_y(x,y)}{G_x(x,y)}\right)$$

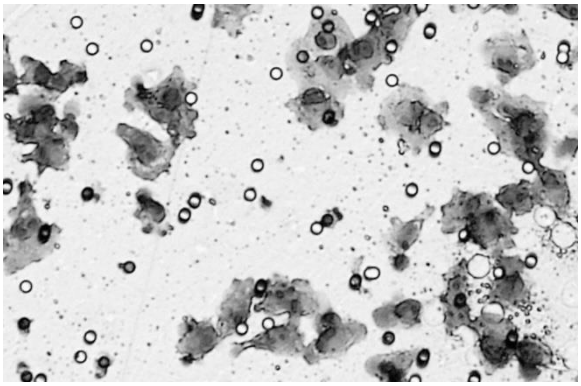


Figure 48 : Originale en niveaux de gris

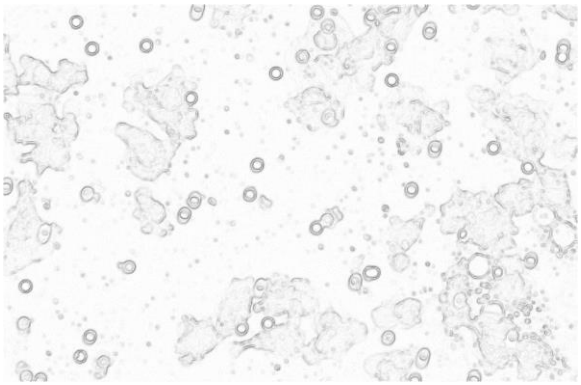


Figure 49 : Filtre de robert inverse

Implémentation personnalisée

```
//Filtre de roberts
var resConv = Convolution.Convolve(res, new RobertsFilter());
```

Implémentation OpenCV

```
//Création des kernels de Roberts
var kernel1 = new Mat(2, 2, MatType.CV_32F);
kernel1.Set(0, 0, 1.0f);  kernel1.Set(0, 1, 0.0f);
kernel1.Set(1, 0, 0.0f);  kernel1.Set(1, 1, -1.0f);
var kernel2 = new Mat(2, 2, MatType.CV_32F);
kernel2.Set(0, 0, 0f);    kernel2.Set(0, 1, 1.0f);
kernel2.Set(1, 0, -1.0f); kernel2.Set(1, 1, 0.0f);
//Convolution par les deux kernels
Cv2.Filter2D(v, output1, v.Depth(), kernel1, new OpenCvSharp.Point(1, 1), 0, BorderTypes.Default);
Cv2.Filter2D(v, output2, v.Depth(), kernel2, new OpenCvSharp.Point(1, 1), 0, BorderTypes.Default);
//Adition des deux matrices
Mat output = output1 + output2;
```

5.8 Filtre de Sobel (1970)

Le Filtre de Sobel est à la fois un filtre de lissage et un détecteur vertical et horizontal. Il est composé de deux kernels de convolution. La valeur du pixel courant est obtenue en calculant la racine carrée de la somme des carrés des gradients X et Y. Ce Filtre est problématique dès que l'image comporte trop de détails car il confond facilement les textures et les objets.

$$\frac{dI}{dx} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * 1/4 \qquad \frac{dI}{dy} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * 1/4$$

10	0	0
10	0	0
10	10	10

*

1	0	-1
2	0	-2
1	0	-1

X 1/4 = 10+20+10-10= 30/4 = 7.5

-1	-2	-1
0	0	0
1	2	1

X 1/4 = -10+10+20+10 = 30/4 = 7.5

$$\sqrt{\left(\frac{dI}{dx}\right)^2 + \left(\frac{dI}{dy}\right)^2} \rightarrow \sqrt{(7.5)^2 + (7.5)^2} = 10.6$$

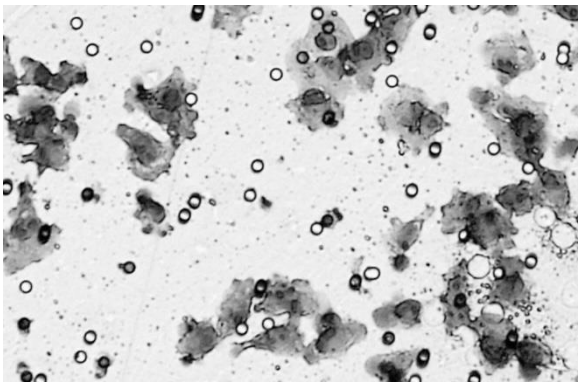


Figure 50 : Originale en niveaux de gris

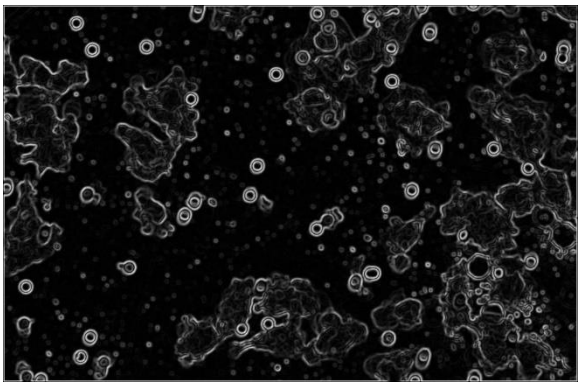


Figure 51 : Filtre de Sobel 4 kernels E, NE, N et NO

Le Filtre de Sobel peut être appliqué aux images RGB pour lesquelles il faut appliquer la convolution pour chaque composante de couleur. Le filtre peut également être appliqué sur la composante de luminance d'une image RGB. Dans ce cas, l'image d'entrée est une image ne comportant qu'une seule composante de luminance Y en niveaux de gris.

Visuellement, le résultat est moyen, les pores de la boîte de Pétri (rond) ressortent fortement et les cellules sont un amas de pixels dont une structure colorée verte ressort.

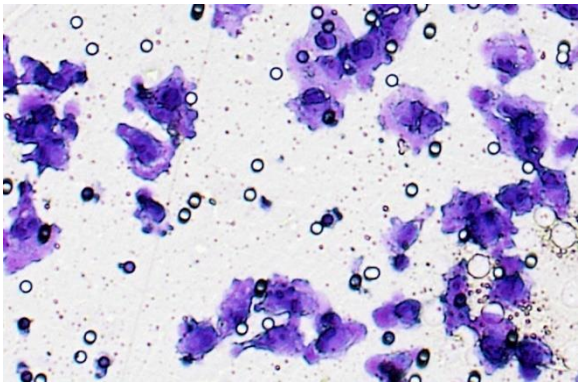


Figure 52 : Originale couleur

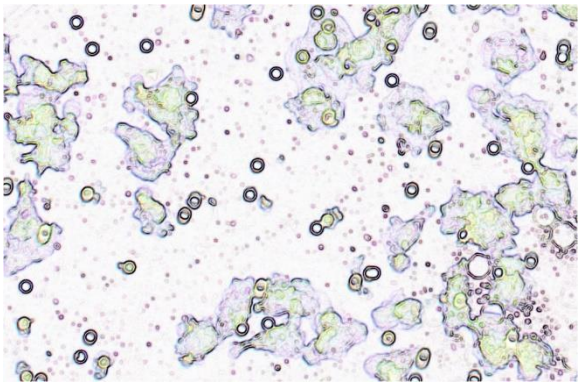


Figure 53 : Filtre de Sobel couleur inverse

Implémentation personnalisée

`//Filtre de Sobel`
`var resConv = Convolution.Convolve(res, new SobelFilter());`
 	
`//Filtre de Sobel à 4 orientations`
`var resConv = Convolution.Convolve(v, new SobelFilter4O());`

Implémentation OpenCV

`//Matrice de gradient X et Y`
`Mat outputX = new Mat(); Mat outputY = new Mat();`
`Mat absX = new Mat();  Mat absY = new Mat();`
`Mat output = new Mat();`
`//Calcul des gradient X et Y`
`Cv2.Sobel(v, outputX, v.Depth(), 1, 0);`
`Cv2.ConvertScaleAbs(outputX, absX);`
`Cv2.Sobel(v, outputY, v.Depth(), 0, 1);`
`Cv2.ConvertScaleAbs(outputY, absY);`
`Cv2.AddWeighted(absX, 0.5, absY, 0.5, 0, output);`

5.9 Filtre de Prewitt (1970)

Le Filtre de Prewitt utilise quatre kernels de convolution. Le premier et troisième kernel sont identiques aux filtres vertical et horizontal, les deux autres kernels sont les diagonales. Le Filtre de Prewitt détecte mieux les contours que le Filtre de Sobel et est moins sensible au bruit, mais plus coûteux dû à ses quatre kernels.

La valeur du pixel est obtenue par la valeur absolue maximale des quatre produits de convolution.

$$\begin{bmatrix}1 & 0 & -1\\1 & 0 & -1\\1 & 0 & -1\end{bmatrix} \quad \begin{bmatrix}1 & 1 & 0\\1 & 0 & -1\\0 & -1 & -1\end{bmatrix} \quad \begin{bmatrix}1 & 1 & 1\\0 & 0 & 0\\-1 & -1 & -1\end{bmatrix} \quad \begin{bmatrix}0 & 1 & 1\\-1 & 0 & 1\\-1 & -1 & 0\end{bmatrix}$$

$$\rightarrow \qquad \searrow \qquad \downarrow \qquad \leftarrow$$

Certaines versions du filtre de Prewit rencontrées dans la littérature ne comportent que les kernels vertical et/ou horizontal. La valeur du gradient peut être obtenue par la racine carré de la somme des carrés des gradients.

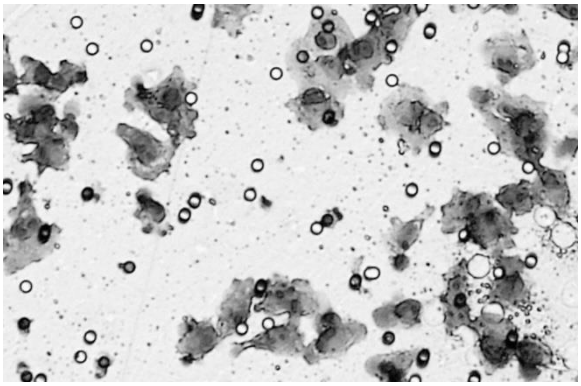


Figure 54 : Originale en niveaux de gris

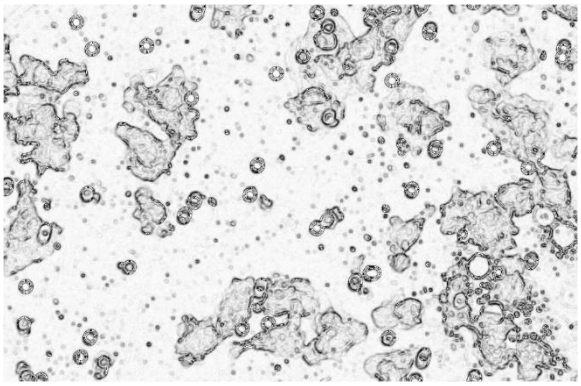


Figure 55 : Filtre de Prewitt gris inverse direction horizontale et verticale

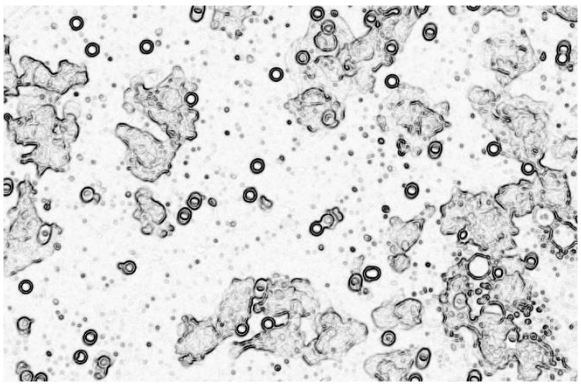


Figure 56 : Filtre de Prewitt 4 directions gris inverse

Implémentation personnalisée

```
//Filtre de Prewitt
var resConv = Convolution.Convolve(res, new PrewittFilter());
//Filtre de Prewitt à 4 orientations
var resConv = Convolution.Convolve(res, new PrewittFilter4O());
```

Implémentation OpenCV

```
//Creation des kernels
var kernelx = new float[,]{
  { 1, 0, -1 },
  { 1, 0, -1 },
  { 1, 0, -1 }
};
var kernely = new float[,]{
  { 1, 1, 1 },
  { 0, 0, 0 },
  { -1, -1, -1 }
};
var kx = new Mat(3, 3, MatType.CV_32F, kernelx);
var ky = new Mat(3, 3, MatType.CV_32F, kernely);
//Convolution par deux kernels
Cv2.Filter2D(v, outputX, -1, kx);
Cv2.Filter2D(v, outputY, -1, ky);
//Conversion en valeurs absolue 8 bits
Cv2.ConvertScaleAbs(outputX, absX);
Cv2.ConvertScaleAbs(outputY, absY);
//Addition de deux matrices dont le poids de chacune des matrices est identique
Cv2.AddWeighted(absX, 0.5, absY, 0.5, 0, output);
```

5.10 Filtre de Scharr

Le Filtre de Scharr est similaire au Filtre de Sobel. La matrice des poids de Sobel est invariante en fonction de la direction du gradient. Elle est isotrope tandis que la matrice des poids de Scharr est dépendante de la direction du gradient. Elle est anisotrope parce que les poids dans les directions diagonales diminuent plus rapidement que dans les directions horizontale et verticale. Différents couples de kernels sont disponibles. Le premier couple est l'original. Le second est simplifié et le dernier est un couple de kernel 5x5. La valeur du gradient est calculée de la même façon que le Filtre de Sobel, la racine carrée des sommes des carrés du produit des convolutions.

$$G(x,y) = \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix} * 1/32 \quad G(x,y) = \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix} * 1/32$$

$$G(x,y) = \begin{bmatrix} -1 & 0 & 1 \\ -3 & 0 & 3 \\ -1 & 0 & 1 \end{bmatrix} * 1/10 \qquad G(x,y) = \begin{bmatrix} -1 & -3 & -1 \\ 0 & 0 & 0 \\ 1 & 3 & 1 \end{bmatrix} * 1/10$$

$$G(x,y) = \begin{bmatrix} -1 & -1 & 0 & 1 & 1 \\ -2 & -2 & 0 & 2 & 2 \\ -3 & -6 & 0 & 6 & 3 \\ -2 & -2 & 0 & 2 & 2 \\ -1 & -1 & 0 & 1 & 1 \end{bmatrix} * 1/60 \quad G(x,y) = \begin{bmatrix} -1 & -2 & -3 & -2 & -1 \\ -1 & -2 & -6 & -2 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 6 & 2 & 1 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix} * 1/60$$

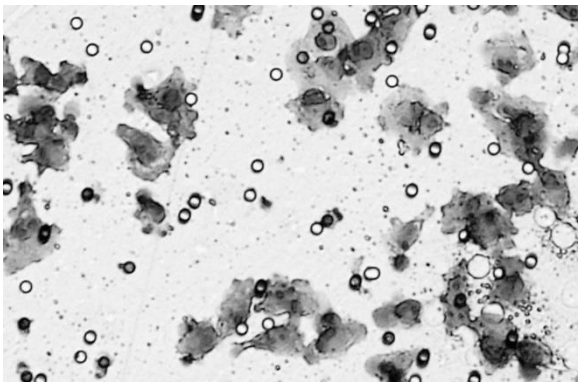


Figure 57 : Originale en niveaux de gris

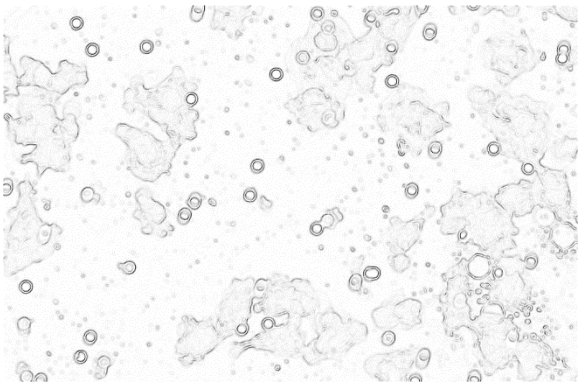


Figure 58 : Filtre de Scharr light gris inverse

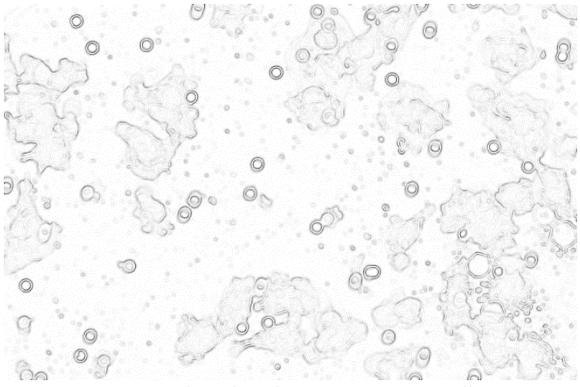


Figure 59 : Filtre de Scharr gris inverse taille 5x5

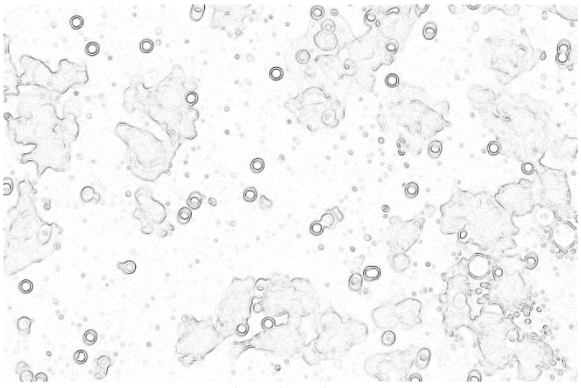


Figure 60 : Filtre de Scharr gris inverse

Implémentation personnalisée
`//Filtre de Scharr` `var resConv = Convolution.Convolve(res, new ScharrFilter());`

Implémentation OpenCV

`//Calcul des gradient X et Y`
`Cv2.Scharr(v, outputX, v.Depth(), 1, 0);`
`Cv2.ConvertScaleAbs(outputX, absX);`
`Cv2.Scharr(v, outputY, v.Depth(), 0, 1);`
`Cv2.ConvertScaleAbs(outputY, absY);`
`Cv2.AddWeighted(absX, 0.5, absY, 0.5, 0, output);`

5.11 Filtre de Kirsch (1971)

Le Filtre de Kirsch ressemble au Filtre de Prewit. Il utilise huit kernels « tournants ». La suite des kernels évolue de 45° dans le sens horaire et donne la direction de détection. Du fait de ses huit kernels, il est plus complexe, l'implémentation est plus coûteuse et nécessite l'utilisation d'un filtre débruiteur tel que le filtre médian.

La valeur du pixel est obtenue par la valeur absolue maximale des huit produits de convolutions dont le facteur est 1/15.

$$\begin{bmatrix}5 & -3 & -3\\ 5 & 0 & -3\\ 5 & -3 & -3\end{bmatrix} \rightarrow \quad \begin{bmatrix}5 & 5 & -3\\ 5 & 0 & -3\\ -3 & -3 & -3\end{bmatrix} \searrow \quad \begin{bmatrix}5 & 5 & 5\\ -3 & 0 & -3\\ -3 & -3 & -3\end{bmatrix} \downarrow \quad \begin{bmatrix}-3 & 5 & 5\\ -3 & 0 & 5\\ -3 & -3 & -3\end{bmatrix} \swarrow$$

$$\begin{bmatrix}-3 & -3 & 5\\ -3 & 0 & 5\\ -3 & -3 & 5\end{bmatrix} \leftarrow \quad \begin{bmatrix}-3 & -3 & -3\\ -3 & 0 & 5\\ -3 & 5 & 5\end{bmatrix} \nwarrow \quad \begin{bmatrix}-3 & -3 & -3\\ -3 & 0 & -3\\ 5 & 5 & 5\end{bmatrix} \uparrow \quad \begin{bmatrix}-3 & -3 & -3\\ 5 & 0 & -3\\ 5 & 5 & -3\end{bmatrix} \nearrow$$

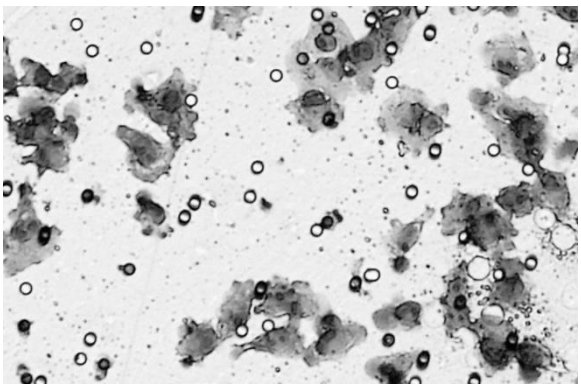


Figure 61 : Originale en niveaux de gris

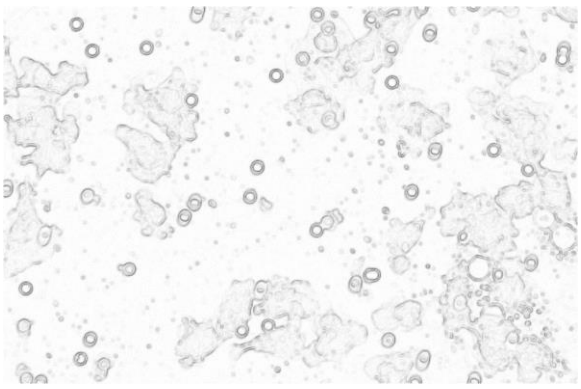


Figure 62 : Filtre de Kirsch gris inverse

Implémentation personnalisée
`//Filtre à 8 directions de Kirsch`

```
var resConv = Convolution.Convolve(res, new KirschFilter());
```

Implémentation OpenCV

```
//Création des kernels
var kernel1 = new float[,]{{ 5, -3, -3 },{ 5,  0, -3 },{ 5, -3, -3 }};
var kernel2 = new float[,]{{  5,  5, -3 },{  5,  0, -3 },{ -3, -3, -3 }};
var kernel3 = new float[,]{{  5,  5,  5 },{ -3,  0, -3 },{ -3, -3, -3 }};
var kernel4 = new float[,]{{ -3,  5,  5 },{ -3,  0,  5 },{ -3, -3, -3 }};
var k1 = new Mat(3, 3, MatType.CV_32F, kernel1);
var k2 = new Mat(3, 3, MatType.CV_32F, kernel2);
var k3 = new Mat(3, 3, MatType.CV_32F, kernel3);
var k4 = new Mat(3, 3, MatType.CV_32F, kernel4);
//Convolution par quatres kernels
Cv2.Filter2D(v, output1, -1, k1);
Cv2.Filter2D(v, output2, -1, k2);
Cv2.Filter2D(v, output3, -1, k3);
Cv2.Filter2D(v, output4, -1, k4);
//Conversion en valeurs absolue 8 bits
Cv2.ConvertScaleAbs(output1, abs1);
Cv2.ConvertScaleAbs(output2, abs2);
Cv2.ConvertScaleAbs(output3, abs3);
Cv2.ConvertScaleAbs(output4, abs4);
Cv2.AddWeighted(abs1, 0.5, abs2, 0.5, 0, output12);
Cv2.AddWeighted(abs3, 0.5, abs4, 0.5, 0, output34);
//Addition de quatre matrices dont le poids de chacune des matrices est identique
Cv2.AddWeighted(output12, 0.5, output34, 0.5, 0, output);
```

5.12 Filtre de Robinson

Le Filtre de Robinson repose sur huit kernels directionnels comme le Filtre de Kirsch et fait l'objet des mêmes problèmes de performance que celui-ci. Tout kernel doit pivoter de 45° pour détecter les contours dans l'ensemble des axes. Il est cependant possible de n'utiliser que quatre des kernels et d'obtenir un résultat pratiquement identique. Le filtre de Robinson permet de mieux percevoir le détail des cellules et fournit des traits plus fins.

La valeur du pixel est obtenue par la valeur absolue maximale des huit convolutions.

Ci-dessous le kernel et ses huit directions dont le facteur est 1/5.

$$\begin{bmatrix}1 & 1 & -1\\ 1 & 2 & -1\\ 1 & 1 & -1\end{bmatrix} \quad \begin{bmatrix}1 & 1 & 1\\ 1 & 2 & -1\\ 1 & -1 & -1\end{bmatrix} \quad \begin{bmatrix}1 & 1 & 1\\ 1 & 2 & 1\\ -1 & -1 & -1\end{bmatrix} \quad \begin{bmatrix}1 & 1 & 1\\ -1 & 2 & 1\\ -1 & -1 & 1\end{bmatrix}$$

$$\rightarrow \qquad \searrow \qquad \downarrow \qquad \swarrow$$

$$\begin{bmatrix}-1 & 1 & 1\\ -1 & 2 & 1\\ -1 & 1 & 1\end{bmatrix} \quad \begin{bmatrix}-1 & -1 & 1\\ -1 & 2 & 1\\ 1 & 1 & 1\end{bmatrix} \quad \begin{bmatrix}-1 & -1 & -1\\ 1 & 2 & 1\\ 1 & 1 & 1\end{bmatrix} \quad \begin{bmatrix}1 & -1 & -1\\ 1 & 2 & -1\\ 1 & 1 & 1\end{bmatrix}$$

$$\leftarrow \qquad \nwarrow \qquad \uparrow \qquad \nearrow$$

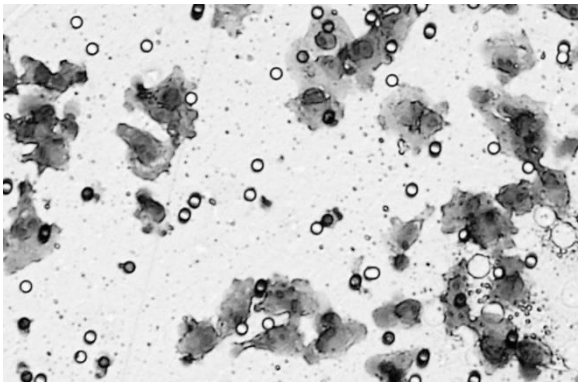


Figure 63 : Originale en niveaux de gris

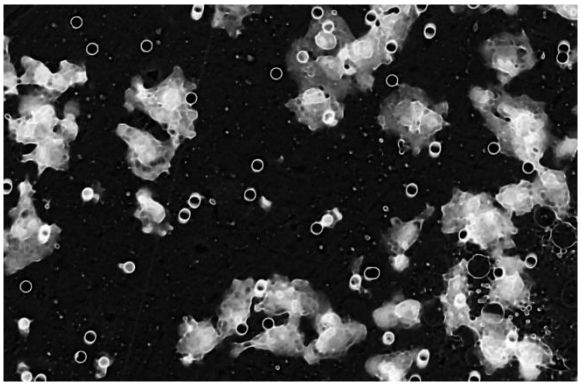


Figure 64: Filtre de Robinsson gris inverse

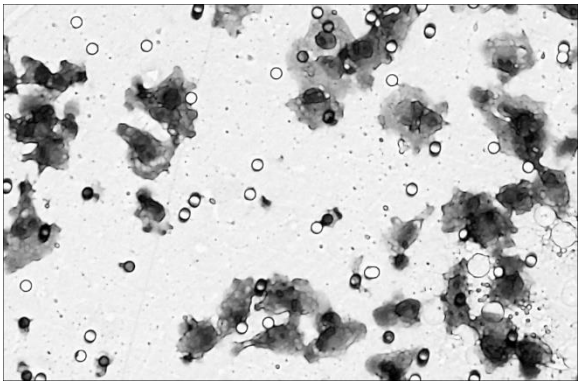


Figure 65 : Filtre de Robinsson gris

Implémentation personnalisée

```
//Filtre de Robinson
var resConv = Convolution.Convolve(res, new RobinsonFilter());
```

Implémentation OpenCV

```
//Creation des kernels
var kernel1 = new double[,] {{1, 1, -1}, {1, 2, -1}, {1, 1, -1}};
var kernel2 = new double[,] {{1, 1, 1}, {1, 2, -1}, {1, -1, -1}};
var kernel3 = new double[,] {{1, 1, 1}, {1, 2, 1}, {-1, -1, -1}};
var kernel4 = new double[,] {{1, 1, 1}, {-1, 2, 1}, {-1, -1, 1}};
var k1 = new Mat(3, 3, MatType.CV_32F, kernel1);
var k2 = new Mat(3, 3, MatType.CV_32F, kernel2);
var k3 = new Mat(3, 3, MatType.CV_32F, kernel3);
var k4 = new Mat(3, 3, MatType.CV_32F, kernel4);
//Convolution par quatres kernels
Cv2.Filter2D(v, output1, -1, k1);
Cv2.Filter2D(v, output2, -1, k2);
Cv2.Filter2D(v, output3, -1, k3);
Cv2.Filter2D(v, output4, -1, k4);
//Conversion en valeurs absolue 8 bits
Cv2.ConvertScaleAbs(output1, abs1);
Cv2.ConvertScaleAbs(output2, abs2);
Cv2.ConvertScaleAbs(output3, abs3);
Cv2.ConvertScaleAbs(output4, abs4);
Cv2.AddWeighted(abs1, 0.5, abs2, 0.5, 0, output12);
Cv2.AddWeighted(abs3, 0.5, abs4, 0.5, 0, output34);
//Addition de quatre matrices dont le poids de chacune des matrices est identique
Cv2.AddWeighted(output12, 0.5, output34, 0.5, 0, output);
```

5.13 Filtre de Pratt et filtre rehausseur

Le filtre de Pratt est un filtre rehausseur, il vise à améliorer la netteté de l'image, à supprimer certains flous ou à ajouter du contraste. Il ne peut pas être considéré comme un détecteur de contour mais peut être utilisé en prémisse d'une détection de contour ou après un filtre de débruitage afin de rétablir les contrastes. Bien que le filtre de Pratt ne présente qu'un seul kernel (le second en partant de la gauche), d'autres kernels de filtre rehausseur sensiblement identiques peuvent lui être associés par mimétisme.

$$G(x,y) = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} * 1/1 \qquad G(x,y) = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix} * 1/1$$

$$G(x,y) = \begin{bmatrix} 1 & -2 & 1 \\ -2 & 5 & -2 \\ 1 & -2 & 1 \end{bmatrix} * 1/1 \qquad G(x,y) = \begin{bmatrix} 1 & -3 & 1 \\ -3 & 9 & -3 \\ 1 & -3 & 1 \end{bmatrix} * 1/1$$

$$G(x,y) = \begin{bmatrix} -1 & -4 & -1 \\ -4 & 27 & -4 \\ -1 & -4 & -1 \end{bmatrix} * 1/1$$

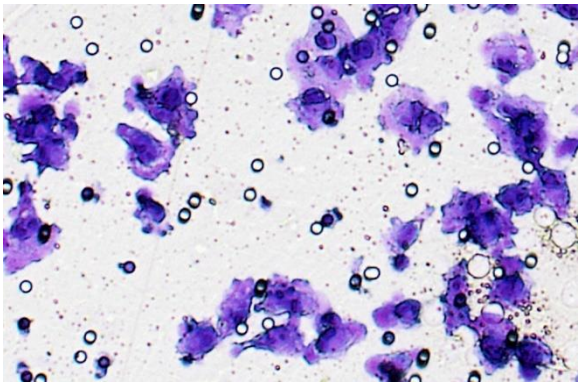


Figure 66 : Originale couleur

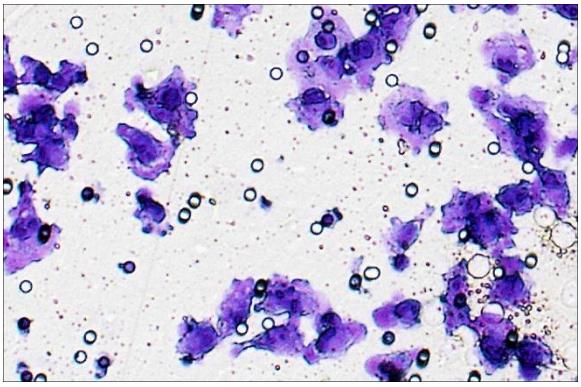


Figure 67 : Filtre rehausseur (-1 5 -1)

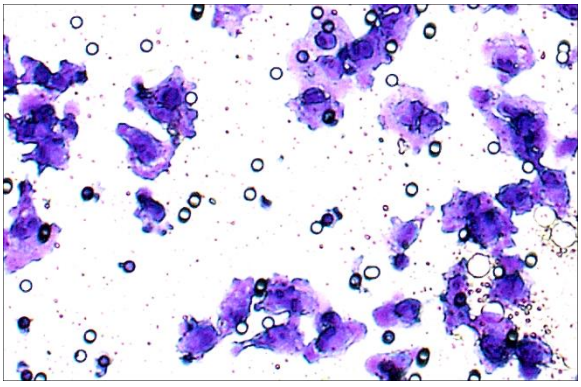


Figure 68 : Filtre rehausseur (-4 27 -4)

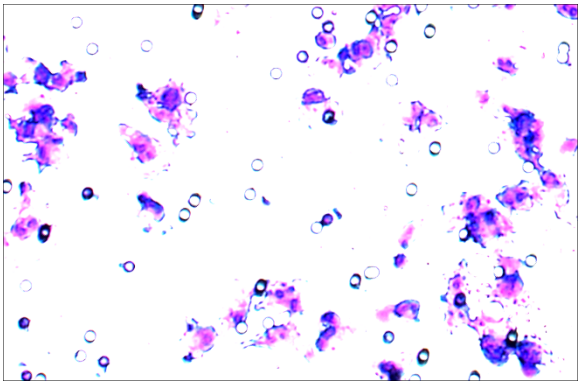


Figure 69 : Filtre de Pratt (-2 5 -2)

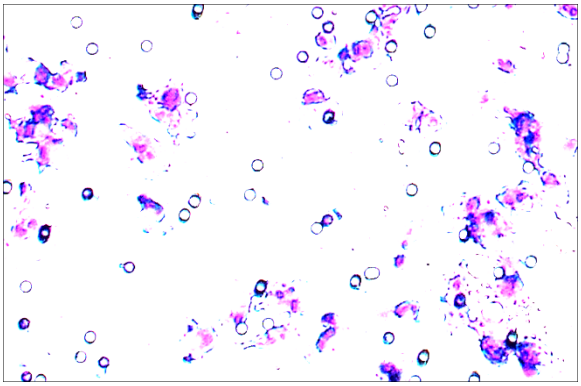


Figure 70 : Filtre de Pratt (-1 9 -1)

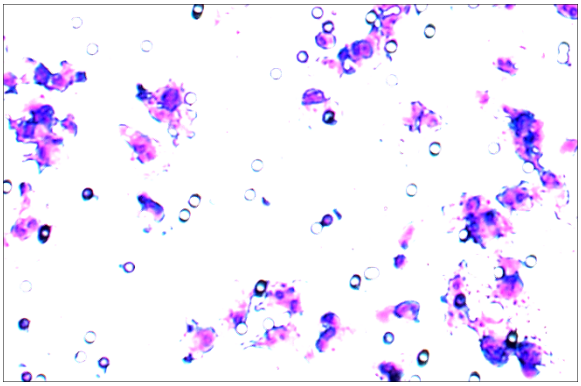


Figure 71 : Filtre de Pratt (-3 9 -3)

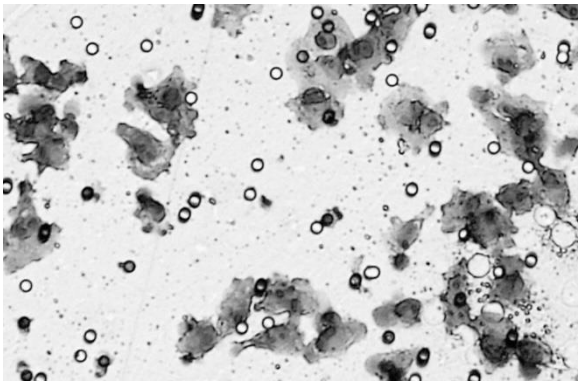


Figure 72 : Originale en niveaux de gris

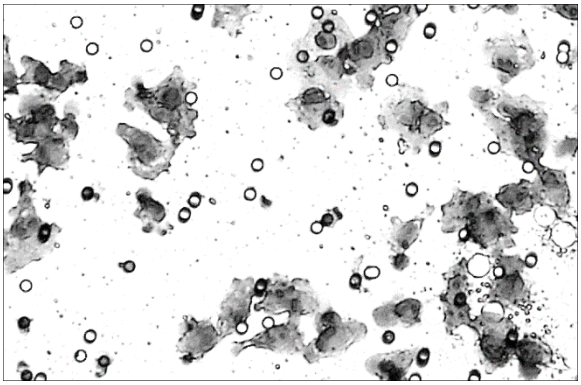


Figure 73 : Filtre de Pratt gris (-4 27 -4)

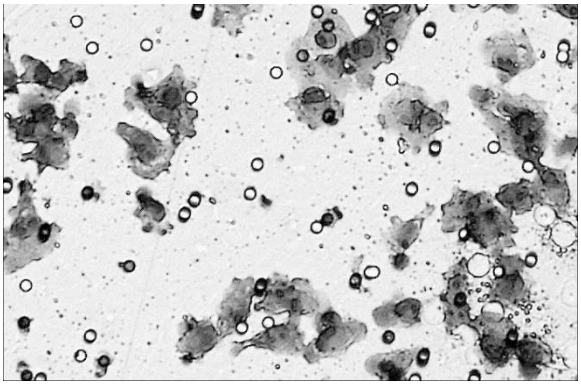


Figure 74 : Filtre de Pratt gris (-1 5 -1)

Implémentation personnalisée

```
//Filtre de Pratt
var resConv = Convolution.Convolve(v, new Pratt51Filter());
```

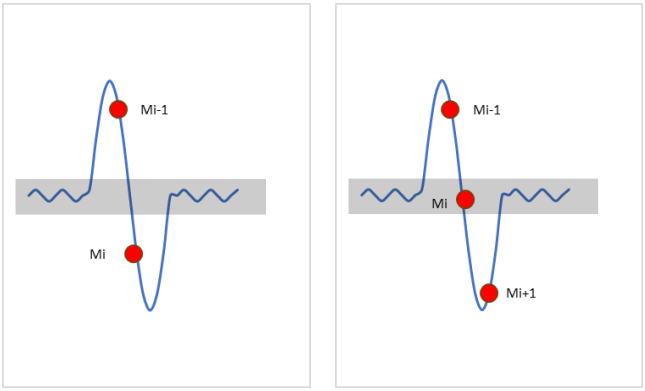
Implémentation OpenCV

```
//Création du kernel
var kernel1 = new float[,] {{0, -1, 0}, {-1, 5, -1}, {0, -1, 0}};
var k1 = new Mat(3, 3, MatType.CV_32F, kernel1);
//Convolution par kernel
Cv2.Filter2D(v, output, -1, k1);
```

5.14 L'Approche du laplacien

L'approche du laplacien cherche à caractériser les contours par des passages par zéro de la dérivée seconde. L'extraction des passages par zéro s'effectue par les étapes suivantes :

1. Calcul du laplacien à l'aide d'un des kernels de convolution
2. Détection des passages par zéro, les pixels pour lesquels le laplacien change de signe sont sélectionnés
3. Création d'une image des passages par zéro
4. Seuillage des passages par zéro de forte amplitude, les faibles gradients sont éliminés. Le seuillage peut être effectué par hystérésis ou par le suivi des frontières
    a. Pour un seuil fixer S
    b. Pour chaque direction des voisins d'un pixel M_i
    c. M_i est un point de contour si le laplacien L est vérifié pour l'une des conditions suivantes :
        i. $L(M_{i-1}) > S \quad et \quad L(M_i) < -S$
        ii. $L(M_{i-1}) > S \quad et \quad |L(M_i)| \leq S \quad et \quad L(M_{i+1}) < -S$



5.15 Filtre de Laplace

Le Filtre de Laplace utilise un seul kernel de convolution, ce qui le rend performant. Il repose sur le principe d'approximation de la somme des dérivées secondes. Il permet de détecter des lignes horizontales, verticales et diagonales. Le Laplacien est très sensible au bruit. Il est donc conseillé d'utiliser au préalable un filtre de lissage, par exemple un filtre gaussien de taille 7 au minimum ou un filtre moyenneur. Il existe deux kernels principaux, l'un de connexité 4 et l'autre de connexité 8, cependant d'autres kernels peu répandus et offrant d'autres approximations peuvent être utilisés.

$$L(x,y) = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} * 1/1 \qquad L(x,y) = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} * 1/1$$

$$L(x,y) = \begin{bmatrix} -1 & 0 & 0 & -1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & -1 \end{bmatrix} * 1/4$$

Le troisième kernel est de taille paire, le point d'ancrage représenté par un fond gris n'est pas central, il est donc essentiellement approximé par la partie droite inférieure du kernel.

En combinaison avec d'autres kernels, le masque de Laplace peut accentuer les contours pour les rendre plus apparent, en d'autres termes augmenter les contrastes.

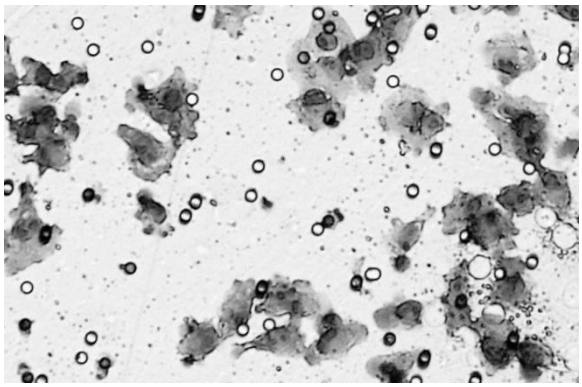


Figure 75 : Originale en niveaux de gris

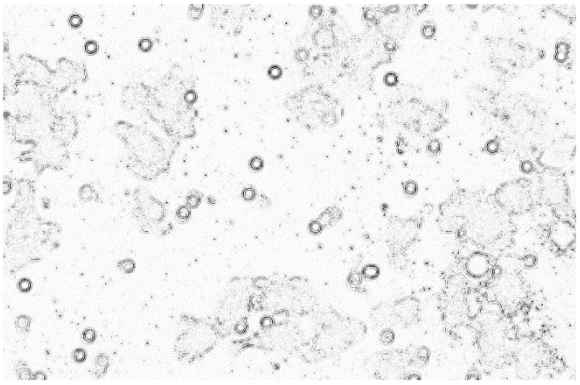


Figure 76 : Filtre de Laplace taille 4 connexité 4

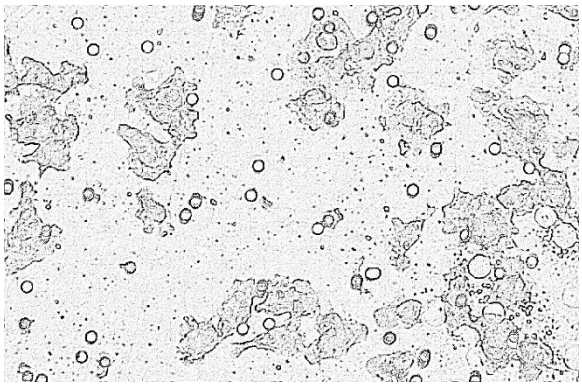


Figure 77 : Filtre de Laplace taille 3 connexité 4

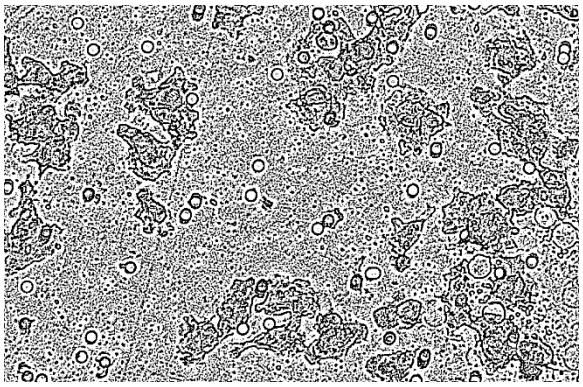


Figure 78 : Filtre de Laplace taille 5

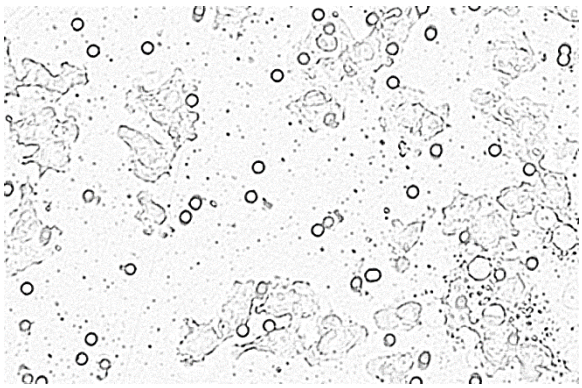


Figure 79 : Laplacien d'une gaussienne

Implémentation personnalisée
`//Filtre laplacien`

```
var resConv = Convolution.Convolve(res, new LaplacianS3C4Filter());
```

Implémentation OpenCV

```
//Filtre Laplacien
Cv2.Laplacian(v, output, v.Depth(), 3, 5);

//Filtre Laplacien d'une gaussienne
//Gaussian blur
Cv2.GaussianBlur(v, blur, new OpenCvSharp.Size(7, 7), 5, 5, BorderTypes.Default);
//Convolution par kernel
Cv2.Laplacian(blur, output, v.Depth(), 3, 5);
```

5.16 La méthode de Canny (1986)

La méthode de Canny est née du constat que les précédents filtres ne permettaient pas une approche fiable et optimale de la détection de contour. Canny a donc établi une liste de trois qualités attendues d'un bon détecteur de contour.

Une bonne détection : Maximiser les bonnes réponses et minimiser les fausses réponses. Tous les contours doivent être détectés sans en omettre, et uniquement les contours, le bruit doit être ignoré. Plus le lissage est important meilleure est la détection.

Une bonne localisation : Le contour détecté doit être aussi proche que possible du contour réel, la distance entre le contour et contour réel doit être minimal. Moins le lissage est important meilleure est la localisation. Il faut un filtre de lissage qui offre un juste milieu entre détection et localisation.

Une réponse unique : un contour ne doit fournir qu'une seule réponse, il faut faire la différence entre deux contours et un contour bruité.

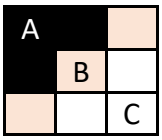
L'algorithme de Canny est décomposé en cinq étapes :

1. Filtrer le bruit de l'image à l'aide d'un filtre gaussien. Plus le filtre est large plus la détection est efficace et moins la localisation est précise.
2. Calculer le gradient de chaque pixel à l'aide du filtre de Sobel. La norme est calculée par la racine carrée de la somme des carrés des convolutions verticale et horizontale. D'autres kernels peuvent être utilisés pour réduire le bruit ou avoir une meilleure symétrie (Sobel 5x5, Prewit, Scharr ou Roberts).
3. En même temps, calculer la direction du gradient par la fonction arc tangente. Une carte des gradients et de leur direction est obtenue. Les directions sont corrélées en fonction de leurs voisins afin de leur donner une direction cohérente. La direction est approximée à $\frac{\pi}{4}$ près. Le kernel de voisinage à considérer pour l'évaluation de la direction est un kernel de 3x3 ou 5x5 dont le pixel central est évalué. La direction des pixels est d'abord normalisée pour obtenir l'une des 4 directions suivantes :
    a. 0 degré si le voisinage indique un contour horizontal
    b. 45 degrés si le voisinage indique un contour où $x = y$
    c. 90 degrés si le voisinage indique un contour vertical
    d. 135 degrés si le voisinage indique un contour où $x = -y$

Les angles sont normalisés selon les conditions suivantes :

$$d(x,y) = \begin{cases} 0° & si\ d(x,y) > 0° \ et\ \leq 22.5° \\ 45° & si\ d(x,y) > 22.5° \ et\ \leq 67.5° \\ 90° & si\ d(x,y) > 67.5° \ et\ \leq 112.5° \\ 135° & si\ d(x,y) > 112.5° \ et\ \leq 157.5° \\ 0° & si\ d(x,y) > 157.5° \ et\ \leq 202.5° \\ 45° & si\ d(x,y) > 202.5° \ et\ \leq 247.5° \\ 90° & si\ d(x,y) > 247.5° \ et\ \leq 297.5° \\ 135° & si\ d(x,y) > 297.5° \ et\ \leq 337.5° \\ 0° & si\ d(x,y) > 337.5° \ et\ \leq 360° \end{cases}$$

4. Supprimer les non-maximas, c'est-à-dire les pixels qui ne font pas partie d'un contour. Le but est d'obtenir le bord le plus fin possible dont les pixels sont dans la direction du contour et dont la valeur est maximale. Les pixels voisins sont ceux qui suivent l'angle du gradient auquel on ajoute 90°, dans une matrice carrée, il s'agit des deux pixels voisins dont la direction est perpendiculaire au gradient.



Par exemple, trois pixels (A, B, C) sont dans la direction du gradient de A vers C soit 135°, le contour, la ligne rouge, est orienté à 90° de plus ou de moins soit 225° ou 45°. Les pixels sont vérifiés pour obtenir le maxima, les deux autres sont supprimés. Si B est le maxima, il fait partie du contour et A et C sont mis à zéro.

5. Seuiller les contours par hystérésis en définissant un seuil haut S_h et un seuil bas S_b. S_h est fixé à une valeur et S_b à la moitié de S_h soit $S_b = \frac{1}{2} * S_h$
   a. Si le gradient est nul, ne rien faire
   b. Si le gradient du pixel est plus grand que le seuil haut, il est considéré comme faisant partie du contour : $si\ G > S_h\ alors\ G = 255$
   c. Si le gradient du pixel est inférieur au seuil bas, il est rejeté et ne fait pas partie du contour : $si\ G < S_b\ alors\ G = 0$
   d. Si le gradient du pixel est entre les deux seuils, il est accepté si le gradient de l'un des pixels du voisinage (3x3) est supérieur au seuil haut :

$$G = \begin{cases} 255\ si\ voisin(G) == true \\ 0\ si\ voisin(G) == false \end{cases}$$

   e. S_h peut être adaptable manuellement et permet d'obtenir plus ou moins de détails.
      i. Si S_h est trop bas, il y a un risque d'obtenir trop de contours, et des contours parasites ;
      ii. Si S_h est trop haut, il y a un risque d'obtenir peu de contours ;
      iii. Il est possible de déterminer S_h de manière plus ou moins automatique en fonction de la distribution des valeurs sur l'histogramme et notamment à l'aide de la méthode d'Otsu.

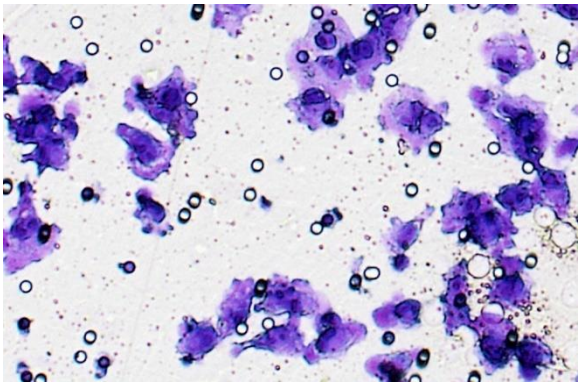


Figure 80 : Originale couleur

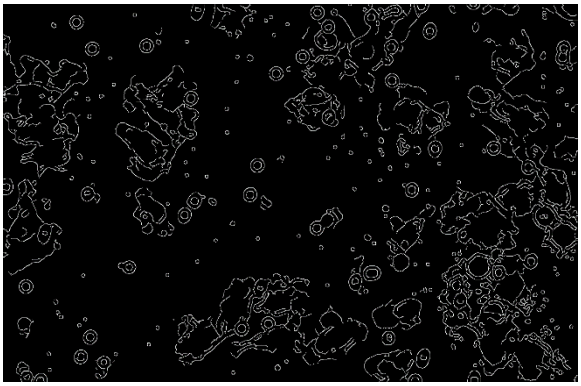


Figure 81 : Contour selon la méthode de Canny

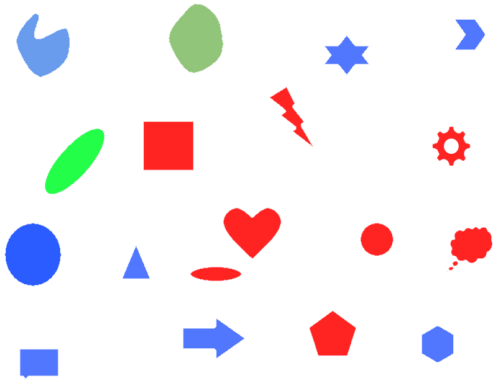


Figure 82 : Image originale

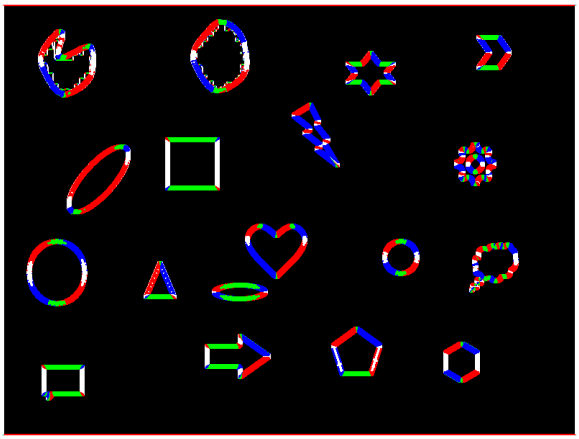


Figure 83 : Représentation visuelle de la normalisation des directions

Implémentation personnalisée

```
//Conversion en niveaux de gris selon la norme Bt709
var res = GrayScaleConverter.ToGray(v, GrayScaleConverter.GrayConvertionMethod.Bt709);
//Application d'un filtre gaussien de taille 5
var resGaus = Convolution.Convolve(res, new GaussianFilter159S5());
//Filtre de détection de Sobel de quatre orientations
var resSobel = Convolution.Convolve(resGaus.Output, new SobelFilter4O(), true);
//Suppression des non-maximas sur trois pixels
var max = NonMaximumSuppression.Apply(resSobel.Output, resSobel.Directions);
//Calcul du seuil maximum par la méthode de Otsu
int th = (int)OtsuThresholding.Compute(max);
double min = (double)th / 2;
//Seuillage par hystérésis
var resThr = HysteresisThresholdingFilter.Apply(max, (int) min, th);
```

Implémentation OpenCV

```
//Gaussian blur
Cv2.GaussianBlur(v, blur, new OpenCvSharp.Size(7, 7), 5, 5, BorderTypes.Default);
//Convolution par kernel
Cv2.Canny(blur, output, lowThreshold, lowThreshold*2, 3);
```

5.17 Filtre de Deriche (1987)

Le filtre de Deriche est une optimisation du filtre de Canny dont la performance est nettement supérieure. La première étape de filtrage est différente par le type de filtre utilisé. En effet, Deriche utilise un filtre de réponse impulsionnelle infinie. La performance du filtre de Canny

est directement liée à la taille du filtre gaussien qui est utilisé pour le lissage de l'image. Plus le filtre est grand plus le traitement est long. La performance du filtre de Deriche est quant à lui indépendant de la qualité du lissage.

Le second avantage est la possibilité de modifier la réponse du filtre. Il est configurable en fonction du bruit. Soit le bruit est important et requiert un lissage important, soit le bruit est faible et permet un lissage plus doux et donc une localisation plus précise.

Le filtre de Deriche est réputé comme récursif. Le traitement se fait en deux phases : la première traite les colonnes de gauche à droite et de droite à gauche, la seconde traite les lignes de haut en bas et de bas en haut.

L'implémentation est beaucoup plus complexe et ne sera pas abordée. De plus, cette méthode n'est pas présente dans l'API OpenCV, ce qui pose la question du bien fondé de celle-ci.

5.18 Conclusion

Nous venons d'explorer une douzaine de filtres de détection et nous sommes loin d'avoir fait le tour de la question. En effet, seuls les filtres les plus populaires ont eu droit à une attention particulière. Chaque année, de nombreux chercheurs s'affrontent dans une compétition afin de tenter d'établir de nouveaux algorithmes ou filtres en vue de détecter une série d'objets et d'améliorer la précision, mais visiblement, ce sont les vieux de la vieille qui font toujours recette.

Les filtres les plus utilisés sont les filtres vertical, horizontal et Sobel. La méthode de Canny utilise le filtre de Sobel et est généralement utilisée dans les méthodes de détection des librairies de traitement d'images. Ceci est en fait, probablement, le filtre le plus utilisé au même titre que le filtre gaussien. Notre implémentation de la méthode de Canny nous a donné du fil à retordre. En effet, elle manque cruellement de performance et de précision. La méthode de suppression des non-maximas est la plus complexe et résulte des étapes précédentes qui doivent pour cela être parfaitement réalisées.

OpenCV ne comporte pas de filtre directionnel mais il est facilement implémentable mais plus gourmand en mémoire. Dans notre implémentation, les différents kernels sont appliqués simultanément et la valeur du gradient est calculée au même moment. La librairie OpenCV impose d'appliquer les différents kernels et ensuite de calculer le gradient. Ce petit défaut est largement compensé par la vitesse d'exécution des méthodes de OpenCV écrite en C++.

Nous venons de voir un ensemble de technique permettant de faire ressortir les contours. Nous serions en droit de nous demander pourquoi il n'existe pas de détecteur d'amas de pixels ? Nous y viendrons dans le chapitre 7.3 Détection de blobs.

Chapitre 6
Filtres et traitements divers

Notre voyage continue avec un ensemble de filtres inclassables mais qui toutefois sont utiles à l'amélioration de l'image d'origine ou à l'amplification des objets de l'image de sortie.

6.1 Isolation de couleur

Isoler une couleur primaire signifie supprimer une ou deux composantes d'un pixel. Isoler la couleur rouge d'une couleur C(r :100, g :150, b :200) signifie supprimer la composante verte et la composante bleue de *C*:

$C(r:100, g:150, b:200)$ ➔ $C'(100, 0, 0)$.

Pourquoi faire ça ? Dans le cas d'une image avec une couleur principale, du mauve par exemple, il peut être intéressant de supprimer la composante verte pour accentuer les contrastes, et donc faciliter la détection de cette couleur.

L'isolation de la couleur peut servir à supprimer une composante avant de transformer en niveaux de gris.

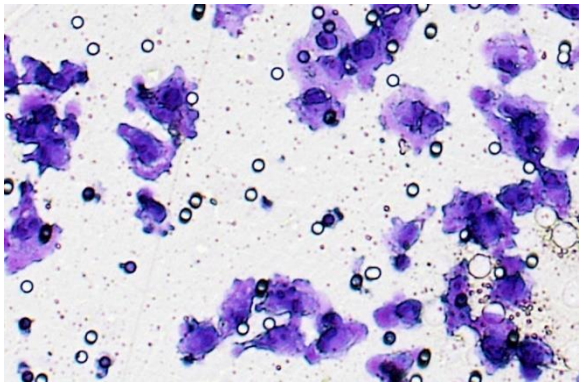


Figure 84 : Originale couleur

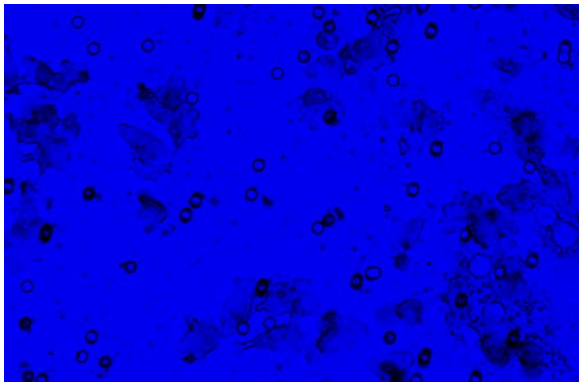


Figure 85 : Isolation du bleu

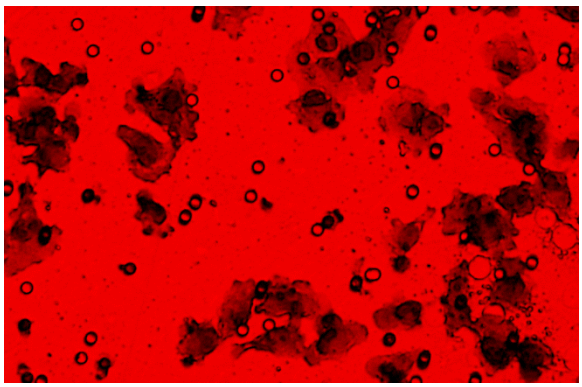


Figure 86 : Isolation du rouge

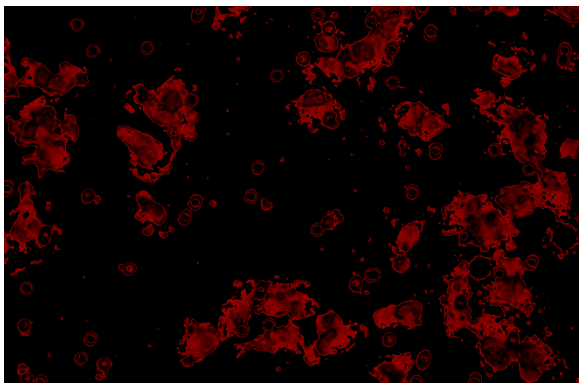


Figure 87 : Isolation du rouge, seuillage par 0 à 160

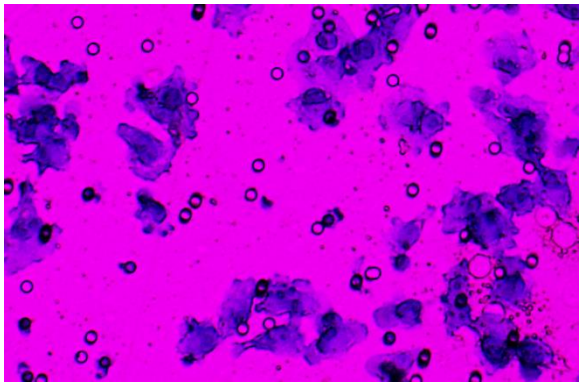


Figure 88 : Isolation du rouge et bleu

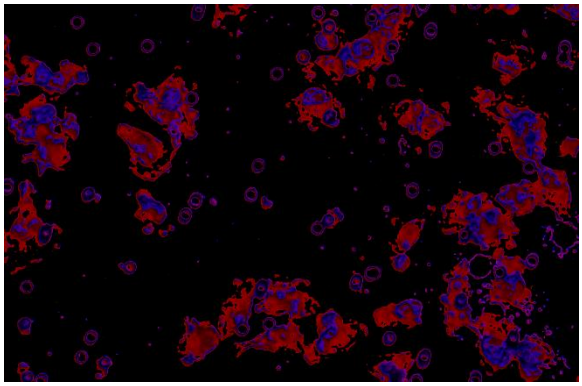


Figure 89 : Isolation du rouge et bleu, seuillage par de 0 à 160

6.1 Conversion en niveaux de gris

La conversion en niveaux de gris peut être atteinte de différentes façons, soit en faisant la moyenne des trois composantes de couleurs, soit en appliquant différents poids aux trois couleurs, soit en ne conservant qu'une partie des composantes de couleurs. Les différentes fonctions permettent d'obtenir des résultats différents, faisant ressortir des zones, appuyant les couleurs les plus communes ou simplement flattant le rendu d'un portrait en noir et blanc.

Les algorithmes de détection de contour, de détection de forme ou de détection de zone de pixels sont effectués sur base d'images en niveaux de gris. Elles subissent des morphismes, accentuations et filtres de toutes sortes. C'est pourquoi il est intéressant d'avoir à sa disposition un ensemble de fonctions de conversion pour pourvoir optimiser le traitement de l'image.

6.1.1 La moyenne de composantes

Deux ou trois composantes de couleurs sont additionnées pour obtenir une moyenne. Cette moyenne est ensuite attribuée aux trois composantes.

$$C'(Y,Y,Y) = C(r,g,b) : Y = (R+G+B)/3$$

$$C'(Y,Y,Y) = C(r,g,b) : C1 \epsilon \{\mathrm{RGB}\}, C2 \epsilon \{RGB\} \setminus C1, Y = (C1+C2)/2$$

6.1.2 Le codage BT709

Il s'agit d'un standard utilisé pour le format HDTV.

$$C'(Y,Y,Y) = C(r,g,b) : Y = (0.2126 * R + 0.7152 * G + 0.0722 * B)$$

6.1.3 L'isolation d'un des composants

L'une des composantes de couleurs peut être utilisée pour forcer les deux autres composantes et donc créer un niveau de gris lié à une seule composante.

$$C'(Y,Y,Y) = C(r,g,b) : Y\ in\ \{rgb\}$$

6.1.4 L'isolation de la composante de luminance ou code rec601

Le modèle YUV comporte une composante de luminance Y et deux composantes de chrominance U et V. Un ratio différent est appliqué à chacune des couleurs pour obtenir une somme pondérée. Cette méthode est utilisée dans les systèmes vidéo télévisés tels que PAL, SECAM et NTSC.

$$C'(Y) = C(r,g,b) : Y = (0.299\ R) + (0.587\ G) + (0.144\ B)$$

La chrominance peut être obtenue par :

$$U = 0.493(B - Y)$$

$$V = 0.877(R - Y)$$

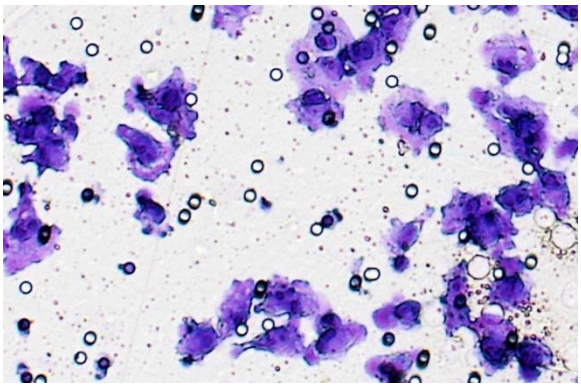


Figure 90 : Originale couleur

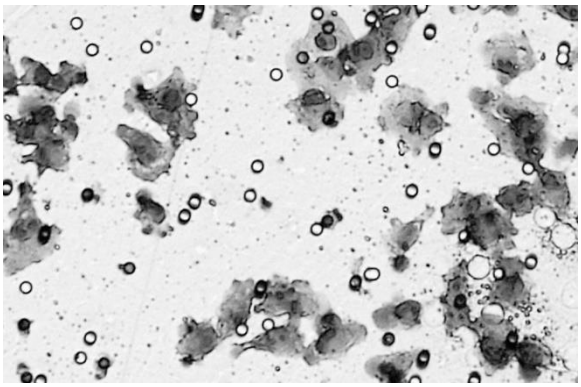


Figure 91 : Moyenne des composants

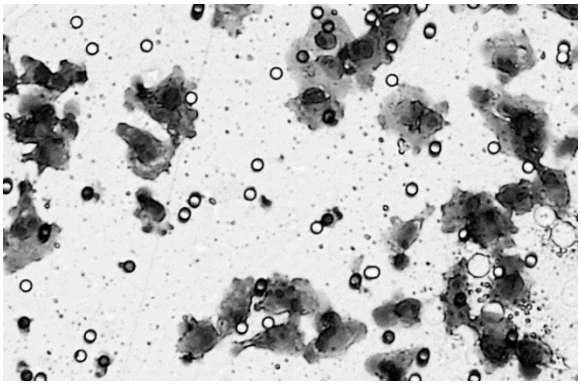


Figure 92 : BT709

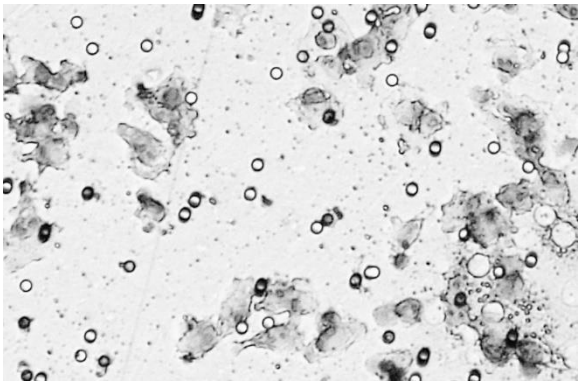
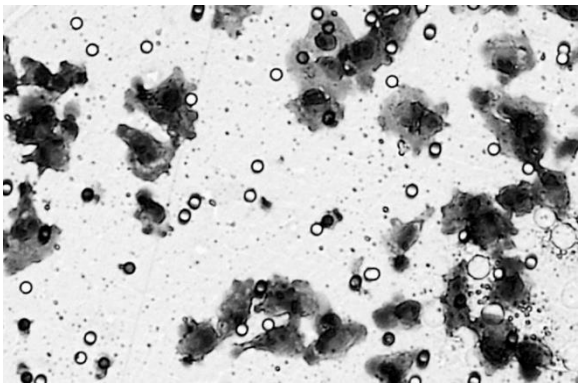


Figure 93 : Composante bleu



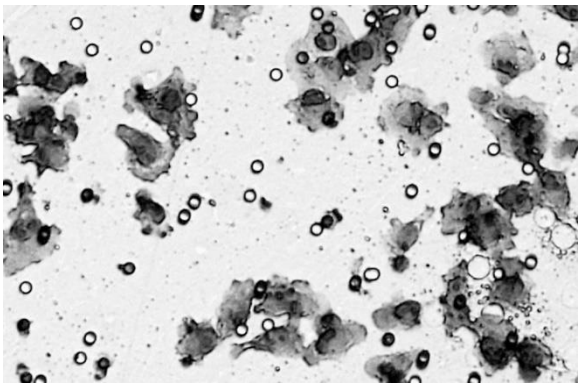


Figure 94 : Composante verte

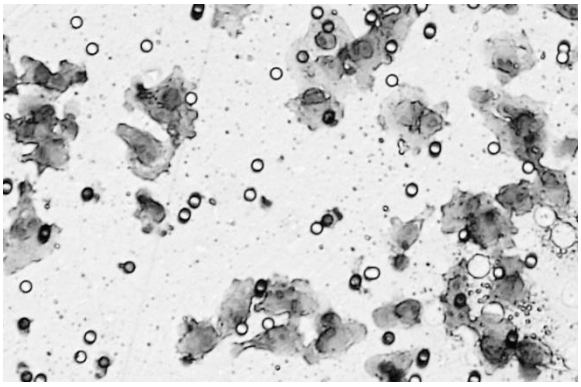


Figure 95 : Composante rouge

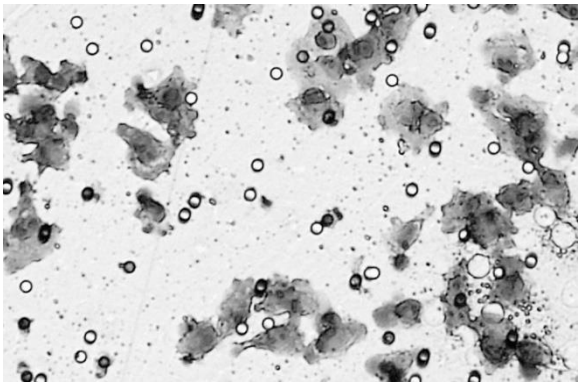


Figure 96 : Composantes rouge et bleu

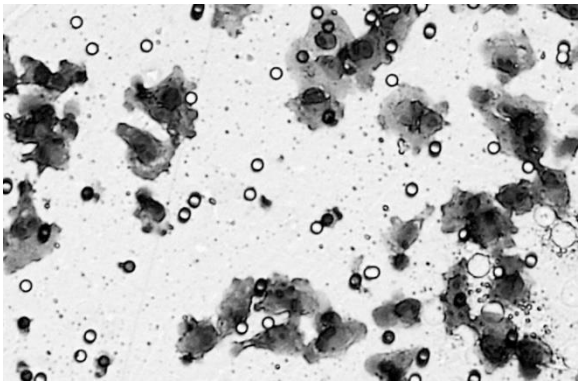


Figure 97 : Composantes bleu et verte

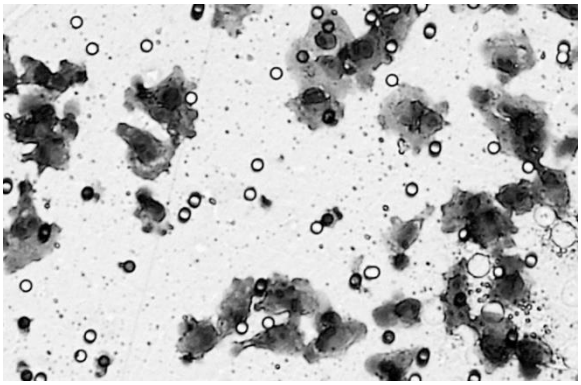


Figure 98 : Composantes rouge et verte

Figure 99 : Luminance

Implémentation personnalisée

```
//Conversion en niveaux de gris selon la moyenne
var res = GrayScaleConverter.ToGray(v, GrayScaleConverter.GrayConvertionMethod.Average);
//Conversion en niveaux de gris selon la norme BT709
var res = GrayScaleConverter.ToGray(v, GrayScaleConverter.GrayConvertionMethod.Bt709);
//Conversion en niveaux de gris selon une composante
var res = GrayScaleConverter.ToGray(v, GrayScaleConverter.GrayConvertionMethod.FromRed);
//Conversion en niveaux de gris selon la luminance
var res = GrayScaleConverter.ToGray(v, GrayScaleConverter.GrayConvertionMethod.FromBrightness);
```

Implémentation OpenCV

```
//Conversion en niveaux de gris Rec601
Cv2.CvtColor(v, output, ColorConversionCodes.RGB2GRAY);
```

6.2 Filtre inverse

Le filtre inverse, également appelé négatif, permet de changer la teinte d'un pixel par son inverse, un pixel foncé deviendra clair et un pixel clair deviendra foncé, un pixel noir deviendra blanc et vice versa. La formule de conversion est simple, il s'agit de retirer de la valeur maximum d'une composante la valeur dudit pixel $C_{initial}$ pour en obtenir la valeur inverse C_{final}.

$$C_{final} = 255 - C_{initial}$$

Le but de ce filtre est dans notre cas essentiellement destiné à l'affichage des différents

résultats des filtres, noir sur blanc est plus lisible que blanc sur noir surtout lors de l'impression.

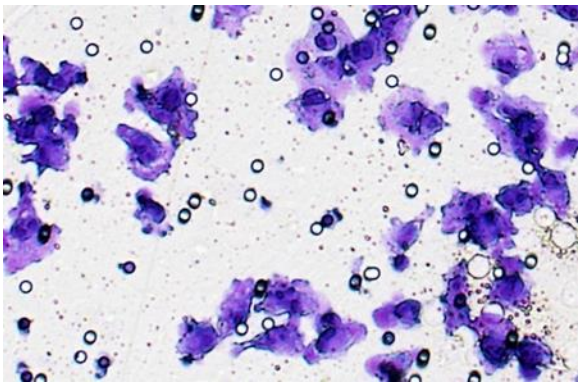


Figure 100 : Originale

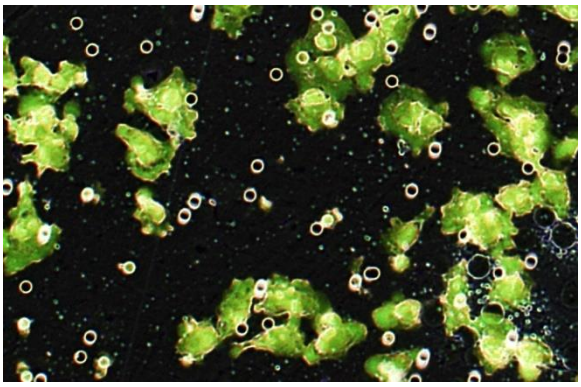


Figure 101 : Inversion image RGB

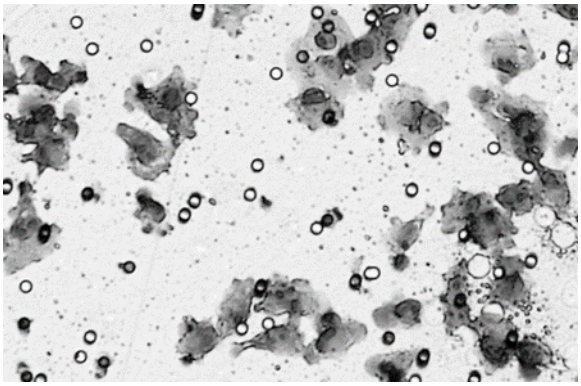


Figure 102 : Original niveaux de gris

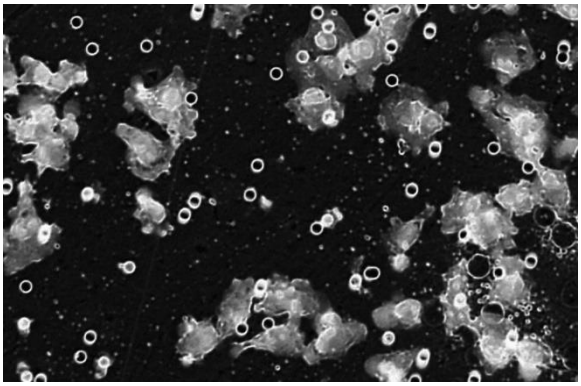


Figure 103 : Inversion niveaux de gris

Implémentation personnalisée
`//Filtre inverse` `var resInv = InverterFilter.Invert(res);`
Implémentation OpenCV
`//Filtre inverse` `Cv2.BitwiseNot(v, output);`

6.3 Seuillage

L'opération de seuillage permet de supprimer les intensités ne faisant pas partie du seuil défini et de les remplacer par des valeurs minimales ou maximales. Il est fréquent d'utiliser le seuillage à la suite d'un filtre de détection de contour. Plusieurs types de seuillage sont possibles.

6.3.1 Seuillage binaire

Le seuillage binaire consiste à convertir l'image en deux classes d'intensité, une classe contenant les blancs et l'autre contenant les noirs. Un seuil est défini pour effectuer la binarisation des deux classes. Les valeurs en-dessous du seuil sont converties en noir et les

valeurs au-dessus du seuil en blanc.

$$G(x,y)_{final} = \begin{cases} 255 \; si \; G(x,y)_{initial} \geq S \\ 0 \quad si \; G(x,y)_{initial} < S \end{cases}$$

6.3.2 Seuillage tronqué

Le seuillage tronqué consiste à appliquer le seuil aux pixels dont la valeur est supérieure ou égale à celui-ci sinon le pixel conserve son intensité.

$$G(x,y)_{final} = \begin{cases} S & si \; G(x,y)_{initial} \geq S \\ G(x,y)_{initial} & si \; G(x,y)_{initial} < S \end{cases}$$

6.3.3 Seuillage par zéro

Le seuillage par zéro applique aux valeurs inférieures au seuil la valeur zéro, sinon le pixel demeure tel quel.

$$G(x,y)_{final} = \begin{cases} G(x,y)_{initial} & si \; G(x,y)_{initial} \geq S \\ 0 & si \; G(x,y)_{initial} < S \end{cases}$$

Une seconde version du seuillage par zéro existe, il s'agit d'appliquer la valeur 0 au pixel supérieur ou égal au seuil et le pixel demeure tel quel dans le cas inverse.

$$G(x,y)_{final} = \begin{cases} 0 & si \; G(x,y)_{initial} \geq S \\ G(x,y)_{initial} & si \; G(x,y)_{initial} < S \end{cases}$$

6.3.4 Seuillage par hystérésis

Le seuillage par hystérésis repose sur l'établissement de deux seuils, un seuil haut S_h et un seuil bas S_b (souvent moitié moins grand que S_h). Tout pixel dont le gradient est inférieur au seuil bas est mis à zéro, et tout pixel dont le gradient est supérieur au seuil haut est considéré comme un contour et mis à la valeur maximum. Les pixels situés entre le seuil bas et le seuil haut sont acceptés comme contours et mis au maximum uniquement s'ils possèdent un contour voisin dans le sens du gradient, sinon ils sont rejetés et mis à 0.

$$G(x,y)_{final} = \begin{cases} 0 \quad si \; G(x,y)_{initial} \leq S_b \\ 255 \quad si \; G(x,y)_{initial} \geq S_h \\ 0 \quad si \; S_b < G(x,y)_{initial} < S_h \; et \; ne \; possède \; pas \; un \; voisin \\ 255 \quad si \; S_b < G(x,y)_{initial} < S_h \; et \; possède \; un \; voisin \end{cases}$$

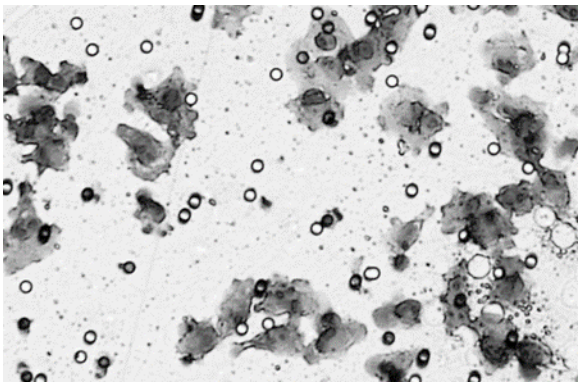


Figure 104 : Original niveaux de gris

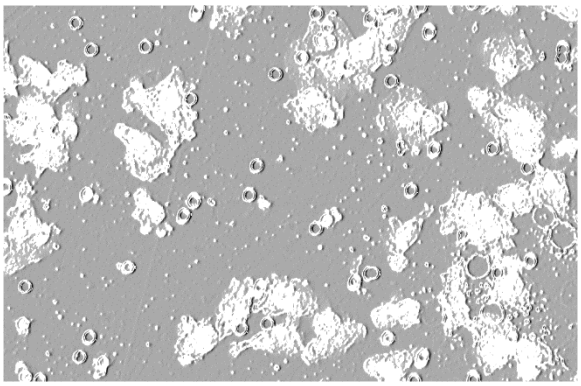


Figure 105 : Seuillage par zéro, seuil à 60

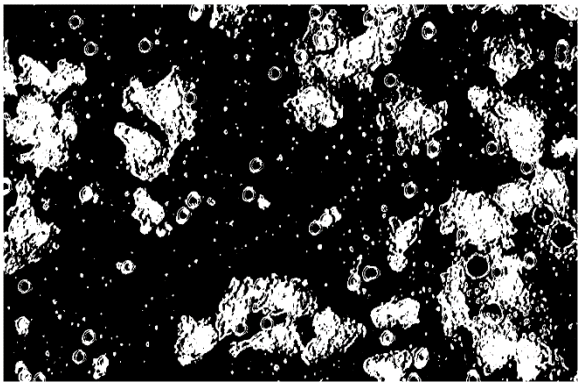


Figure 106 : Seuillage binaire, seuil à 60

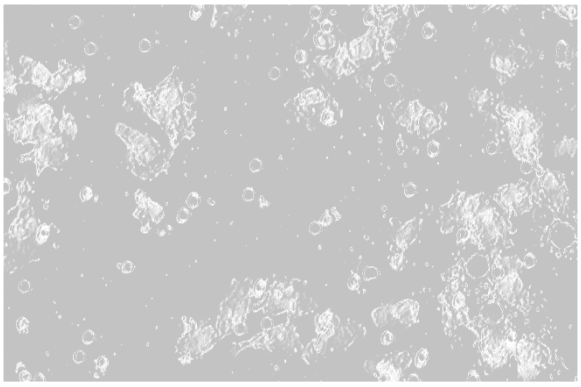


Figure 107 : Seuillage tronqué, seuil à 60

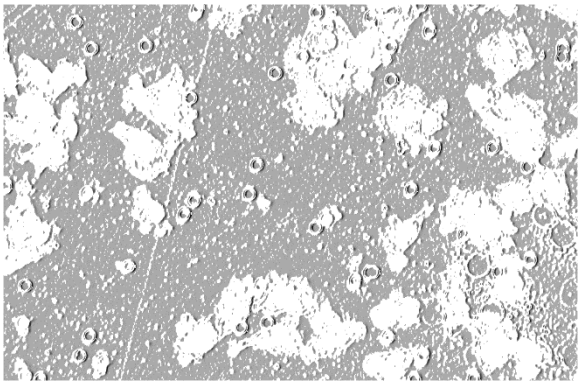


Figure 108 : Seuillage par hystérésis, seuil min 40 max 80

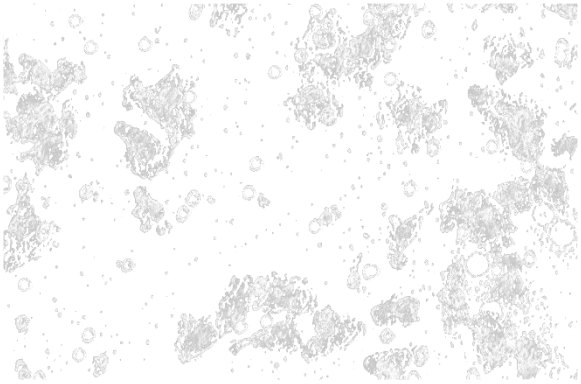


Figure 109 : Seuillage par zéro maximum, seuil à 60

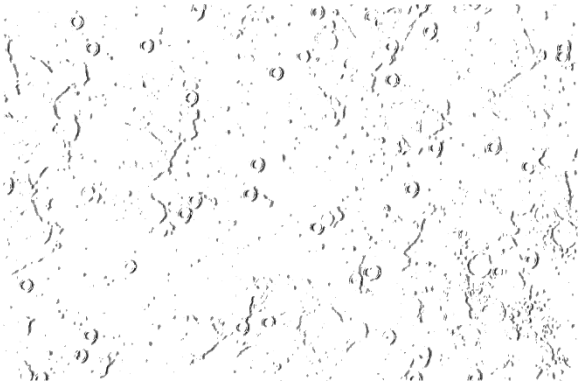


Figure 110 : Seuillage par hystérésis, seuil min 50 max 100

Implémentation personnalisée

```
//Seuillage par Hystérésis
var resThr = HysteresisThresholdingFilter.Apply(resConv.Output, 50, 100);
//Seuillage binaire
var resThr = BinaryThresholdingFilter.Apply(resConv.Output, 60);
//Seuillage tronqué
var resThr = TruncatedThresholdingFilter.Apply(resConv.Output, 60);
//Seuillage par zéro
var resThr = ZeroThresholdingFilter.Apply(resConv.Output, 60, false);
//Seuillage par zéro inverse
var resThr = ZeroThresholdingFilter.Apply(resConv.Output, 60, true);
```

Implémentation OpenCV

```
//Seuillage binaire
Cv2.Threshold(v, output, 60, 255, ThresholdTypes.Binary);
//Seuillage binaire inverse
Cv2.Threshold(v, output, 60, 255, ThresholdTypes.BinaryInv);
//Seuillage par zéro
Cv2.Threshold(v, output, 60, 255, ThresholdTypes.Tozero);
//Seuillage par zéro inverse
Cv2.Threshold(v, output, 60, 255, ThresholdTypes.TozeroInv);
//Seuillage tronqué
Cv2.Threshold(v, output, 60, 255, ThresholdTypes.Trunc);
```

6.4 Seuillage par la méthode d'Otsu

La méthode d'Otsu permet de réaliser un calcul du seuillage de façon automatique. Le seuillage par hystérésis est une opération qui impose de fixer le seuil haut afin de déterminer le seuil bas et de permettre le filtrage. Cette opération manuelle peut être lente à mettre en œuvre et demande un certain nombre d'essais/erreurs pour isoler un seuil pertinent.

La méthode d'Otsu se base sur l'histogramme et divise les pixels de l'image en deux classes, l'une représentant le fond et l'autre représentant les objets. L'algorithme calcule le seuil optimal qui sépare les deux classes afin de réduire la variance au sein d'une classe.

Les étapes de l'algorithme sont les suivantes :

1. Calculer l'histogramme et les probabilités de chaque niveau d'intensité
2. Définir les poids $\omega_i(0)$ et moyenne $\mu_i(0)$ initiaux des deux classes
3. Parcourir l'ensemble des seuils de 1 à 256
    a. Mettre à jour les ω_i et μ_i
    b. Calculer les variances des deux classes $\sigma_b^2(t)$
4. Le seuil désiré correspond au $\sigma_b^2(t)$ maximum

L'algorithme peut être coûteux pour les images de grande taille mais permet d'obtenir un seuil de façon automatique. Dans le cas présent, le facteur temps n'est pas un problème puisqu'il ne s'agit nullement de faire du temps réel ou de l'analyse d'image en continu.

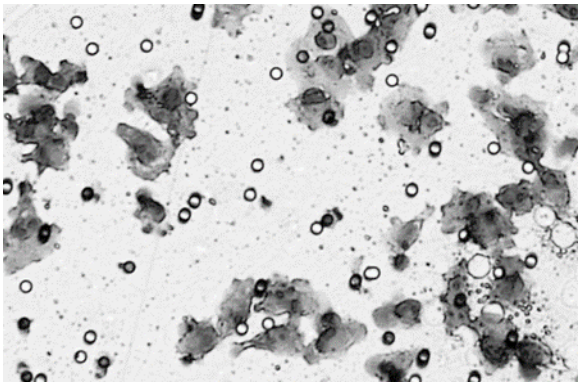


Figure 111 : Original niveaux de gris

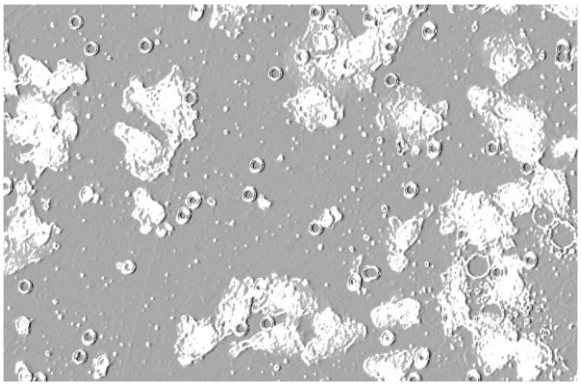


Figure 112 : Seuillage par hystérésis gris, le seuil est 58, il est calculé par la méthode de Otsu

Implémentation personnalisée

`//Calcul du seuil par la méthode de Otsu`
`int th = (int) OtsuThresholding.Compute(resConv.Output);`
`//Seuillage par hystérésis calculé par la méthode de Otsu`
`var resThr = HysteresisThresholdingFilter.Apply(resConv.Output, th/2, th);`

Implémentation OpenCV
`//Seuillage par hystérésis et la méthode de Otsu` `Cv2.Threshold(v, output, 60, 255, ThresholdTypes.Otsu);`

6.5 Seuillage par bande de couleur

Le seuillage par bande est une implémentation du filtre passe-bande. Le filtre passe-bande ne laisse passer que les fréquences comprises entre deux bornes afin de ne conserver qu'une gamme de teinte de couleur. La teinte T est soumise à une tolérance donnant une gamme caractérisée par un seuil bas S_b et un seuil haut S_h. Les pixels en dehors de la teinte sont forcés à la couleur noire.

$$T(x,y)_{final}\begin{cases} T & si\ S_b < T(x,y)_{initial} < S_h \\ 0 & sinon \end{cases}$$

Pour une binarisation dans le but d'obtenir un masque binaire :

$$T(x,y)_{final}\begin{cases} 255 & si\ S_b < T(x,y)_{initial} < S_h \\ 0 & sinon \end{cases}$$

Pour le seuillage de teintes primaires, une série de tests simples peuvent être utilisés. Ils permettent de comprendre le principe de base du seuillage de teinte.

$$S_b < T_{bleu} < S_h\ et\ T_{bleu} > T_{vert} + 40\ et\ T_{bleu} > T_{rouge} + 40$$

$$S_b < T_{rouge} < S_h\ et\ T_{rouge} > T_{vert} + 40\ et\ T_{rouge} > T_{bleu} + 40$$

$$S_b < T_{vert} < S_h\ et\ T_{vert} > T_{rouge} + 40\ et\ T_{vert} > T_{bleu} + 40$$

Dans le cas qui nous occupe, la quantification et localisation de surfaces colorées dans un

intervalle de couleurs (le mauve) pourraient aider à nettoyer les régions en dehors de la bande de couleurs et ne conserver que les régions ayant un intérêt. La quantification pourrait permettre, sur base d'une analyse statistique, d'établir une relation entre la quantité de pixels et le nombre de cellules cancéreuses. Il faudrait pour cela analyser un échantillon représentatif d'image pour lesquelles le nombre des cellules est connu. Après une régression linéaire, il serait alors possible d'obtenir une estimation du nombre de cellules d'une nouvelle image à un facteur d'incertitude près.

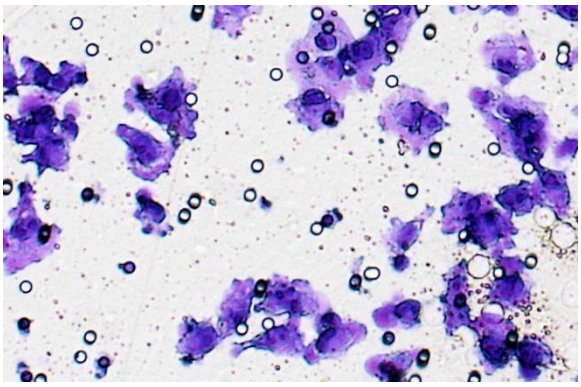


Figure 113 : Originale couleur

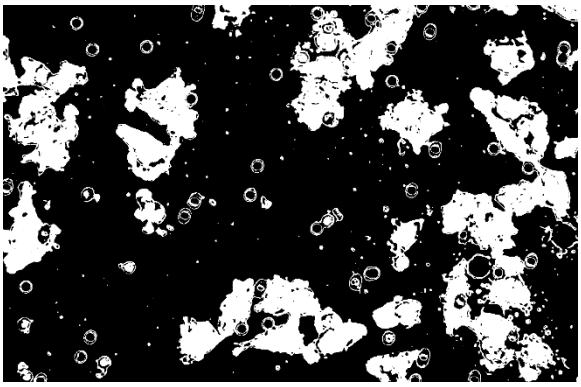


Figure 114 : Seuillage par bande RGB(200,137,233), tolérance 60

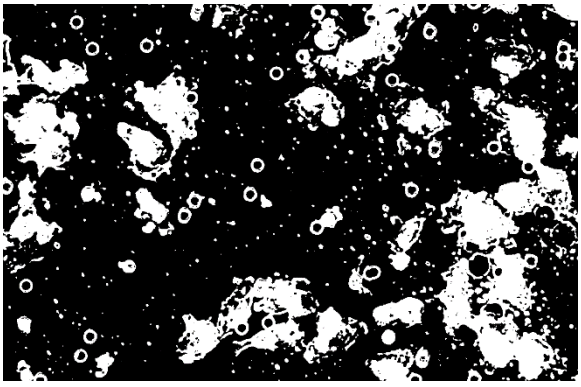


Figure 115 : Seuillage par bande de couleur HSV 60° à 180°

Implémentation personnalisée

```
//Déclaration de la couleur de seuil
Color c = Color.FromArgb(200, 147, 233);
//Seuillage de bande de couleur
var resThr = BandThresholdingFilter.Apply(res, c, 100, 60);
```

Implémentation OpenCV

```
//Déclaration des seuils
var lowher_color = new Scalar(lowH, lowS, lowV);
var higher_color = new Scalar(highH, highS, highV);
//Seuillage par couleur et création du masque
Cv2.InRange(hsv, lowher_color, higher_color, mask);
//Opération de masquage pour ne conserver que les pixels de seuillés
Cv2.BitwiseAnd(ori, ori, output, mask);
```

Le seuillage par bande de couleur peut être appliqué à une image en niveaux de gris. En effet, si les seuils bas et haut sont des couleurs dont les composantes sont identiques, le seuil est un niveau de gris. Il s'agirait alors d'un filtre coupe bande.

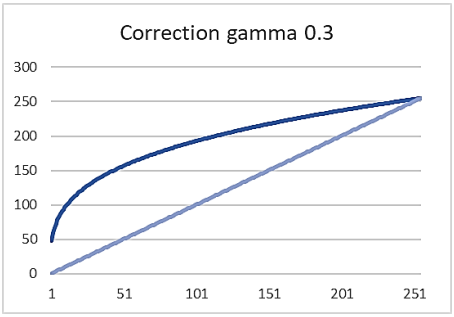
6.6 Correction gamma

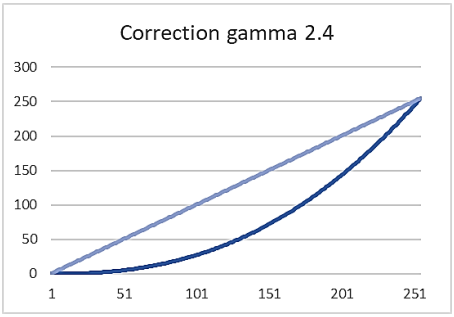
La correction gamma (γ) est une fonction qui permet de corriger l'exposition d'une image, de l'éclaircir ou l'assombrir en fonction du défaut de celle-ci. La correction gamma était utilisée sur les moniteurs CRT ou lors de processus d'impression. Elle peut servir dans différentes méthodes d'acquisition et de constitution de l'imagerie numérique.

La correction gamma n'est pas linéaire et permet de préserver les détails dans les teintes moyennes (les ombres). La formule suivante permet d'appliquer une correction gamma.

$$Pixel_{out} = 255\left(\frac{Pixel_{in}}{255}\right)^{\gamma}$$

Si le paramètre γ est < 1 les teintes claires sont assombries. Si γ est > 1, les teintes sombres sont éclaircies. Si γ est 1 la correction est nulle. Les deux représentations suivantes illustrent deux corrections gamma, l'une de 0.3 où l'intervalle des teintes sombres est élargi, l'intervalle d'intensité se situant entre 0 et 51 est augmenté et élargi dans l'intervalle d'intensité de 50 à 150. L'autre de 2.4 où l'intervalle des teintes sombre situé entre 0 à 50 est rétréci et devient 0 à 5. Une correction gamma sur une image RGB est appliquée de la même façon sur chacune des composantes de couleurs.





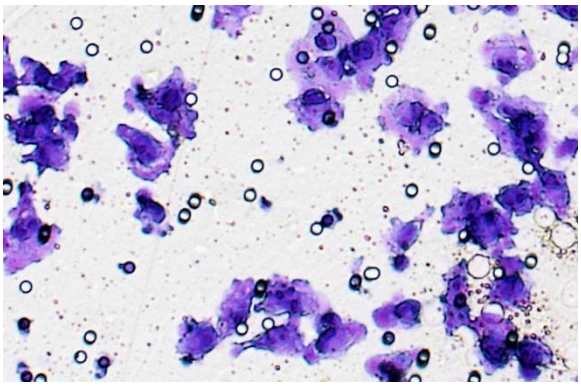


Figure 116 : Originale

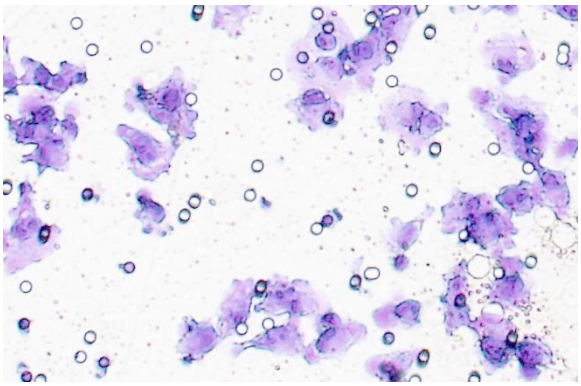


Figure 117 : Correction gamma 0.3

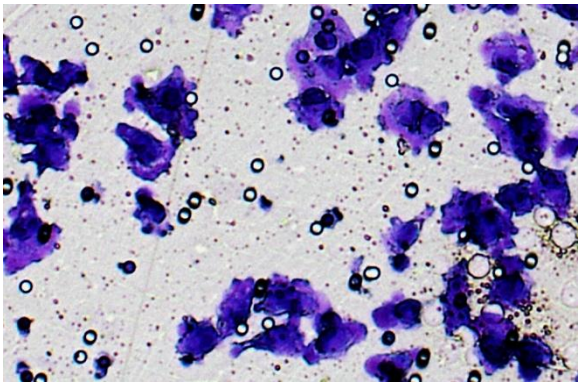


Figure 118 : Correction gamma 2.4

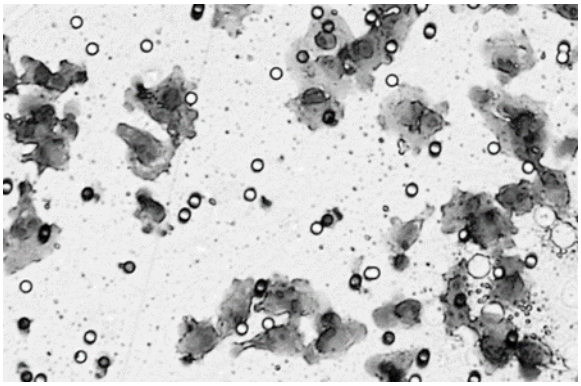


Figure 119 : Original niveaux de gris

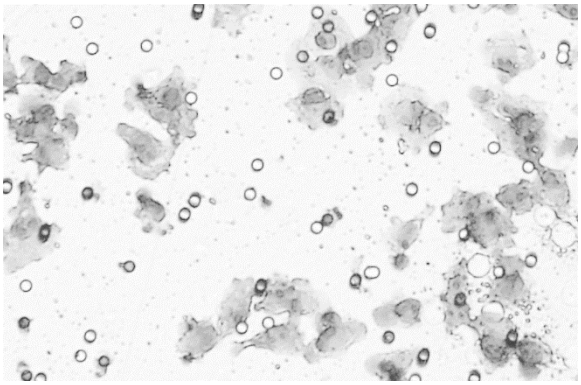


Figure 120 : Correction gamma 0.3 (niveaux de gris)

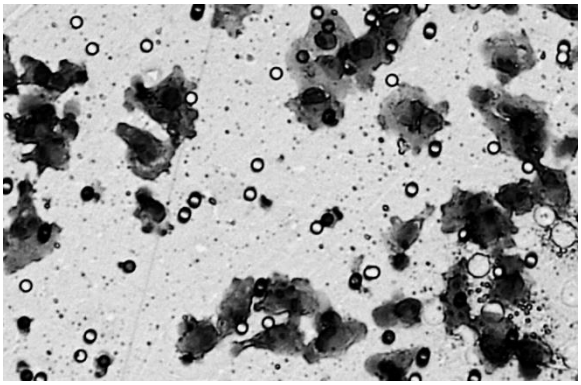


Figure 121 : Correction gamma 2.4 (niveaux de gris)

Implémentation personnalisée

```
//Application de la correction gamma
var resInv = GammaCorrection.Correct(res, 0.3);
```

Implémentation OpenCV

```
//Création de la table lut en fonction du facteur de correction gamma
byte[] lookUpTable = new byte[256];
double gamma = 0.5;
for (int i = 0; i < 256; ++i)
    lookUpTable[i] = (byte)Math.Round(Math.Pow(i / 255.0, gamma) * 255.0);
//Application de la correction gamma
Cv2.LUT(v, lookUpTable, output);
```

6.7 Correction de contraste

Le contraste d'une image est l'opposition entre les couleurs sombres et les couleurs claires, soit la répartition de la luminance d'une image. Une image « bien contrastée », devrait être étalée sur l'histogramme complet. Le contraste permet d'augmenter ou diminuer la perception d'objet dans une image, de les rendre plus ou moins distincts.

L'augmentation du contraste permet de faire ressortir les objets d'une image, la diminution atténue la différence entre les objets. Le contraste peut être exprimé par un seuil compris dans un intervalle T de ±100 où les valeurs positives augmentent le contraste et les valeurs négatives diminuent le contraste. La formule suivante permet d'obtenir le facteur de correction de contraste C.

$$C = \left(\frac{100 + T}{100}\right)^2$$

Chacune des composantes de couleur $C_{originale}$ est ensuite adaptée à l'aide du facteur de contraste C pour obtenir les composantes de sortie C_{output}. Les valeurs de sortie doivent être bornées de 0 à 255.

$$C_{output} = \left(\left(\left(\frac{C_{originale}}{255} - 0.5\right) * C\right) + 0.5\right) * 255$$

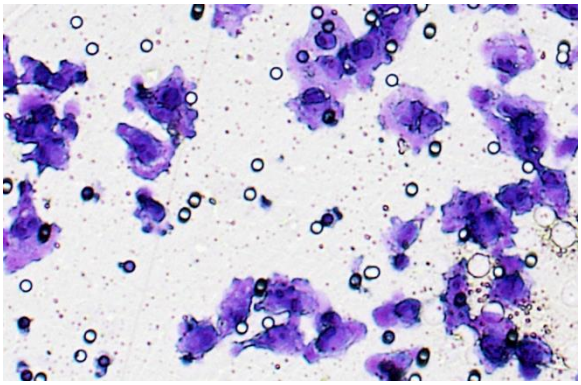


Figure 122 : Originale couleur

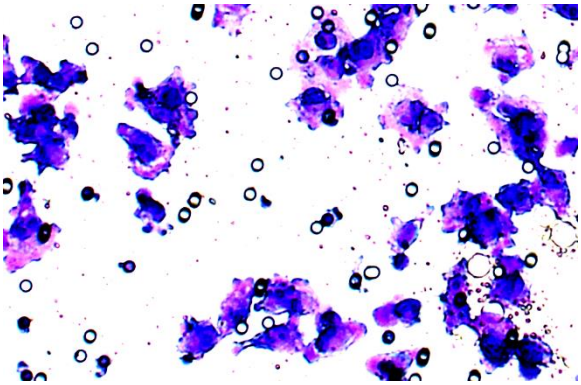


Figure 123 : Correction de contraste +50

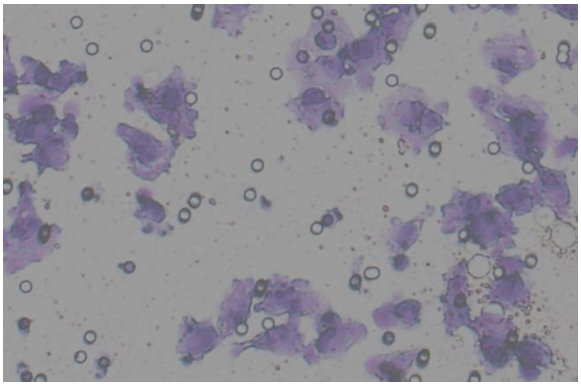


Figure 124 : Correction de contraste -50

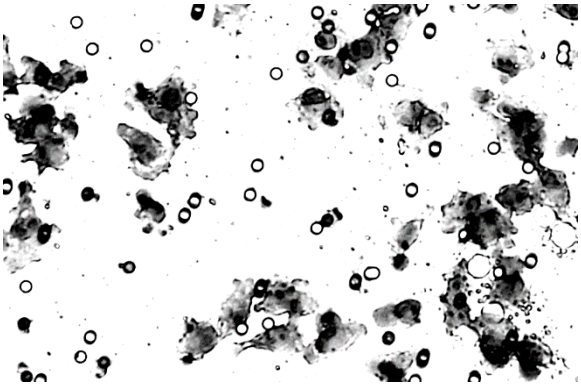


Figure 125 : Correction de contraste +50 gris

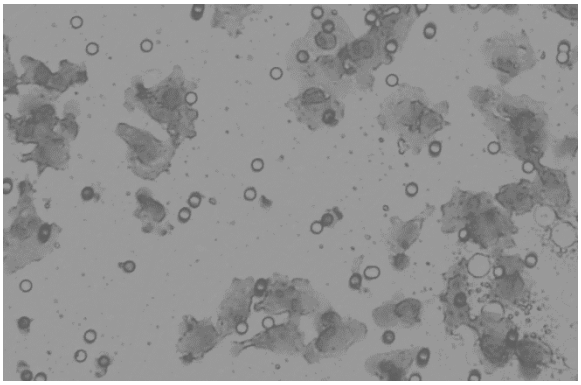


Figure 126 : Correction de contraste -50 gris

Implémentation personnalisée

```
//Correction de contraste de +50%
var res = ContrastCorrection.Correct(v, 50);
//Correction de contraste de -50%
var res = ContrastCorrection.Correct(v, -50);
```

Implémentation OpenCV

```
//Augmente le contraste de 10%
v.ConvertTo(output, v.Depth(), 1.1, 0);
//Diminue le contraste de 50%
v.ConvertTo(output, v.Depth(), 0.5, 0);
```

6.8 Morphologie mathématique

La morphologie mathématique permet d'effectuer, soit du filtrage de lissage ou d'accentuation, soit de la segmentation afin d'isoler les objets du fond. Elle fait appel à la théorie des ensembles et notamment aux opérateurs définis par Minkowski.

Un élément structurant a la même forme qu'un kernel. Il permet de définir le voisinage pris en compte lors d'une opération de dilatation ou d'érosion. Les éléments structurants peuvent avoir différentes formes (disque, carré, triangle, …) et tailles. Ci-dessous, trois exemples d'éléments structurants, l'un sous forme de croix en connexité quatre A_4, un autre carré en connexité huit A_8 et le dernier « rond » en connexité vingt-neuf A_{29}.

$$A_4 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad A_8 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad A_{29} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

6.9 Dilatation

Une dilatation est une opération morpho-mathématique, l'addition de deux ensembles définis par Minkowski, d'une part un objet au sein d'une image et d'autre part un élément structurant.

La dilatation repose sur le même principe que la convolution, chaque pixel est parcouru et traité à l'aide du voisinage de l'élément structurant A. Si le pixel V_n est un pixel de « fond » et qu'il possède un voisin dans l'élément structurant appartenant à un objet, le pixel prend la valeur du contour. Dans une image en noir et blanc, les pixels de fond sont noirs et les pixels des objets sont blancs.

$$C = A \oplus V_n$$

$$C = \begin{cases} 255 \; si \; V_n \in fond \; et \; A \in contour \\ 0 \; sinon \end{cases}$$

Pour une image en niveaux de gris, les pixels de fonds sont noirs ou de teinte inférieure à un seuil bas S_b, par exemple le seuil bas correspondant à la moitié du seuil obtenu grâce à la méthode de Otsu. Le reste des pixels appartiennent aux objets. La nouvelle valeur d'un pixel pour une réponse positive correspond à la valeur maximum du voisinage.

$$C = \begin{cases} 0 \; si \; V_n < S_b \\ \max(voisinage) \end{cases}$$

Le résultat, comme son nom l'indique, est une dilatation des objets, ils grossissent de la taille de l'élément structurant, des trous peuvent être comblés durant le processus et des objets proches peuvent fusionner. La dilatation d'une image en niveaux de gris aura tendance à éclaircir l'image.

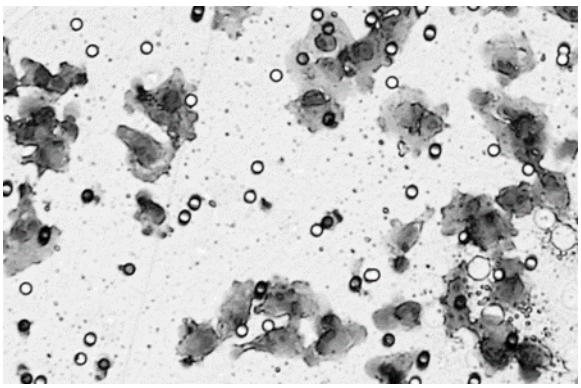


Figure 127 : Original niveaux de gris

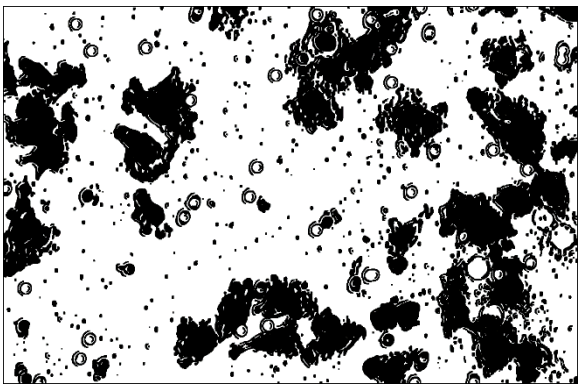


Figure 128 : Dilatation en carré

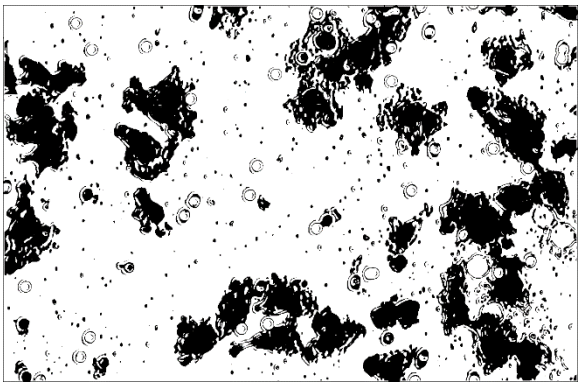


Figure 129 : Filtre gaussien, Sobel et seuillage binaire (seuil 48)

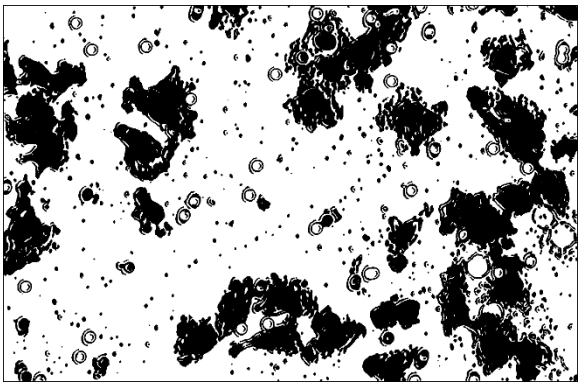


Figure 130 : Dilatation kernel en croix

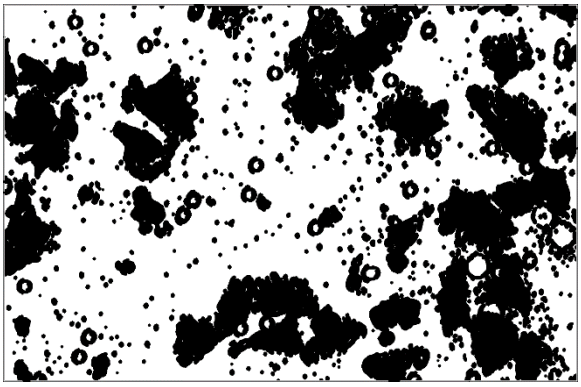


Figure 131 : Dilatation kernel en rond

Implémentation personnalisée

```
//Applique un filtre gaussien
var gaus = Convolution.Convolve(res, new GaussianFilter159S5());
//Calcul des gradients
var sobl = Convolution.Convolve(gaus.Output, new SobelFilter4O(), true);
byte otsuTh = (byte) OtsuThresholding.Compute(sobl.Output);
//Seuillage par hystérésis
var th = BinaryThresholdingFilter.Apply(sobl.Output, otsuTh);
th.Save(@".\otsu.png");
//Dilatation
var dilate = Dilate.Apply(th, new RoundStructuredElement(), otsuTh);
```

Implémentation OpenCV

```
//Filtre gaussien 9x9
Cv2.GaussianBlur(v, gaus, new OpenCvSharp.Size(9, 9), 0, 0, BorderTypes.Default);
//Calcul des gradients X et Y
Cv2.Sobel(v, outputX, v.Depth(), 1, 0);
Cv2.ConvertScaleAbs(outputX, absX);
Cv2.Sobel(v, outputY, v.Depth(), 0, 1);
Cv2.ConvertScaleAbs(outputY, absY);
Cv2.AddWeighted(absX, 0.5, absY, 0.5, 0, sobel);
//Dilatation dont l'élément structurant est une ellipse de 5x5
Mat output = sobel.Dilate(Cv2.GetStructuringElement(MorphShapes.Ellipse, new Size(5, 5)));
```

6.10 Erosion

L'érosion est l'opération inverse de la dilatation, elle correspond à la soustraction de Minkowski. L'opération utilise de la même façon les éléments structurants définis au point précédent, chaque pixel est parcouru, si le pixel fait partie d'un objet et qu'il possède un voisin dans l'élément structurant appartenant au fond, le pixel prend la valeur du fond.

$$C = A \ominus V_n$$

$$C = \begin{cases} 255 \; si \; V_n \in au \; fond \; et \; A \in contour \\ 0 \; si \; V_n \in au \; fond \; et \; A \notin contour \\ V_n \; sinon \end{cases}$$

Pour une image en niveaux de gris, tel que la dilatation, il s'agit de considérer un seuil bas sous lequel le fond est représenté. La nouvelle valeur d'un fond sera la valeur minimale du voisinage.

L'érosion permet d'affiner un objet de la taille de l'élément structurant. Les objets plus petits que l'élément structurant vont disparaître, les trous seront accentués et un objet peut être scindé en plusieurs objets. L'érosion d'une image en niveaux de gris aura tendance à assombrir l'image.

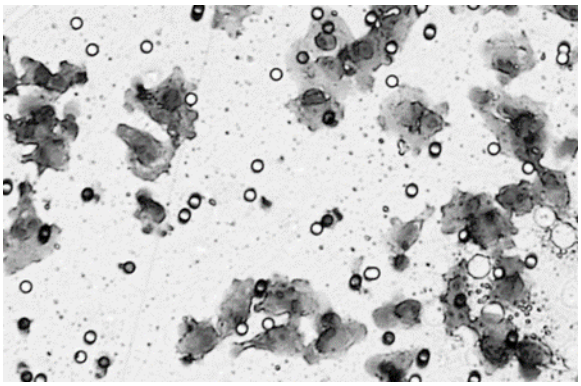


Figure 132 : Original niveaux de gris

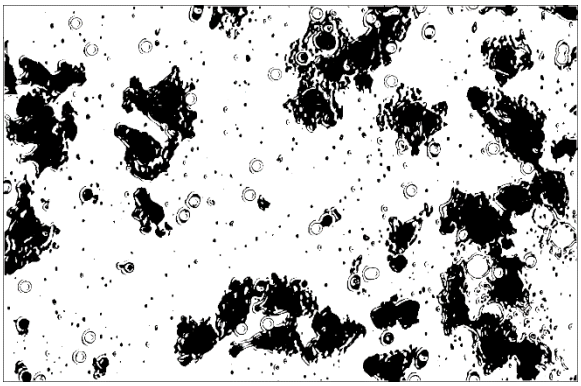


Figure 133 : Filtre gaussien Sobel seuillage binaire

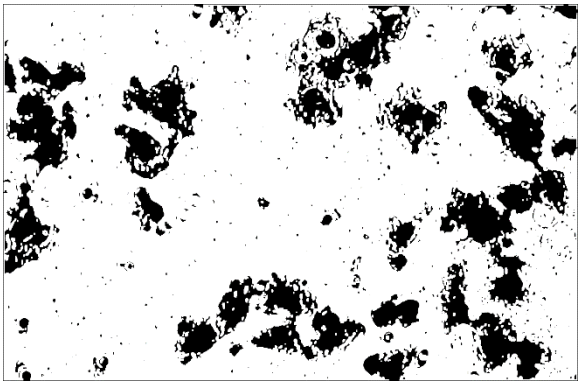


Figure 134 : Erosion en carré

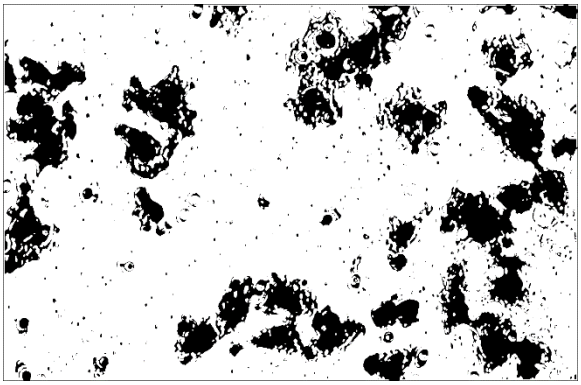


Figure 135 : Erosion kernel en croix

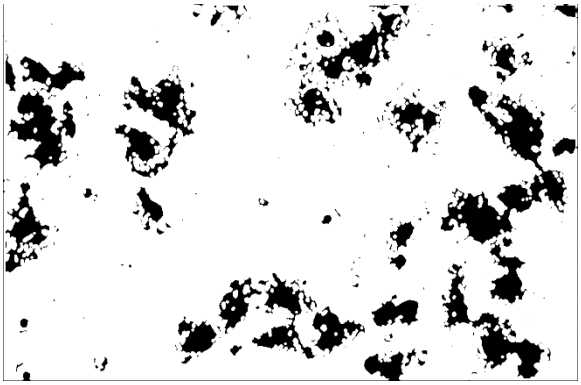


Figure 136 : Erosion kernel en rond

Implémentation personnalisée

```
//Applique un filtre gaussien
var gaus = Convolution.Convolve(res, new GaussianFilter159S5());
//Calcul des gradients
var sobl = Convolution.Convolve(gaus.Output, new SobelFilter4O(), true);
byte otsuTh = (byte) OtsuThresholding.Compute(sobl.Output);
//Seuillage par hystérésis
var th = BinaryThresholdingFilter.Apply(sobl.Output, otsuTh);
th.Save(@".\otsu.png");
//Erosion
var ero = Erode.Apply(th, new SquareStructuredElement(), otsuTh);
```

Implémentation OpenCV

```
//Filtre gaussien 9x9
Cv2.GaussianBlur(v, gaus, new OpenCvSharp.Size(9, 9), 0, 0, BorderTypes.Default);
//Calcul des gradients X et Y
Cv2.Sobel(v, outputX, v.Depth(), 1, 0);
Cv2.ConvertScaleAbs(outputX, absX);
```

```
Cv2.Sobel(v, outputY, v.Depth(), 0, 1);
Cv2.ConvertScaleAbs(outputY, absY);
Cv2.AddWeighted(absX, 0.5, absY, 0.5, 0, sobel);
//Erosion dont l'élément structurant est une ellipse de 5x5
Mat output = sobel.Erode(Cv2.GetStructuringElement(MorphShapes.Ellipse, new Size(5, 5)));
```

6.11 Ouverture et fermeture - Erosion et dilatation

Une ouverture est un enchaînement d'une érosion suivie d'une dilatation. Dans la manœuvre, les détails disparaissent lors de l'érosion et ne sont pas recréés lors de la dilatation. Les petits morceaux de contour et bruit sont supprimés.

$$C = (A \ominus V_n) \oplus V_n$$

Une fermeture est l'enchainement inverse, une dilatation suivie d'une érosion. Les petits trous sont bouchés et les structures fines sont préservées.

$$C = (A \oplus V_n) \ominus V_n$$

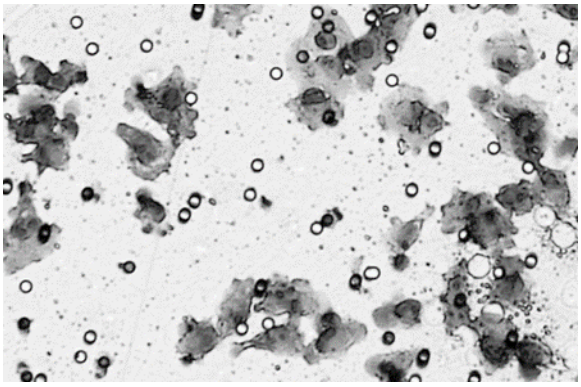


Figure 137 : Original niveaux de gris

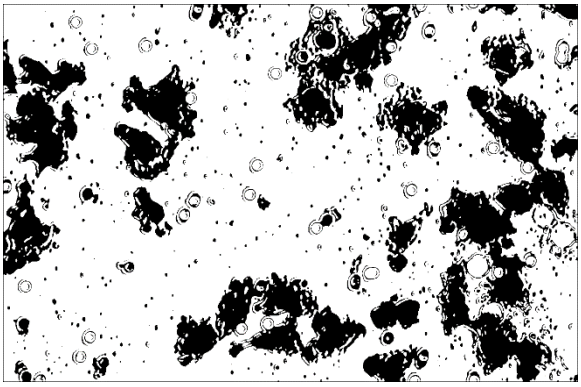


Figure 138 : Fermeture (dilatation et érosion)

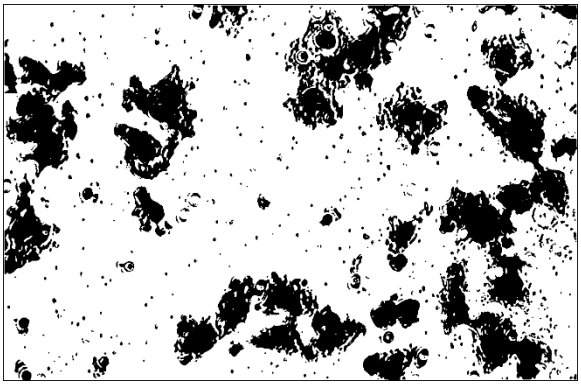


Figure 139 : Ouverture (érosion et dilatation)

Implémentation personnalisée

```
//Ouverture
//Erosion
var ero = Erode.Apply(th, new CrossStructuredElement(), otsuTh);
//Dilatation
var dil = Dilate.Apply(ero, new CrossStructuredElement(), otsuTh);
//Fermeture
//Dilatation
```

```
var dil = Dilate.Apply(th, new CrossStructuredElement(), otsuTh);
//Erosion
var ero = Erode.Apply(dil, new CrossStructuredElement(), otsuTh);
```

Implémentation OpenCV

```
//Ouverture
Mat element = Cv2.GetStructuringElement(MorphShapes.Ellipse, new Size(5,5));
Cv2.MorphologyEx(output, ero, MorphTypes.Open, element);
//Fermeture
Mat element = Cv2.GetStructuringElement(MorphShapes.Ellipse, new Size(5, 5));
Cv2.MorphologyEx(output, ero, MorphTypes.Close, element);
```

6.12 Contour intérieur et extérieur

Le contour intérieur d'un objet est obtenu par la différence entre l'image d'origine et de son érodé, seul le contour intérieur est conservé.

$$C = A/(A \ominus V_n)$$

Le contour extérieur d'un objet est obtenu par la différence entre le dilaté de l'image d'origine et l'image elle-même, seul le contour extérieur des objets est conservé.

$$C = (A \oplus V_n)/A$$

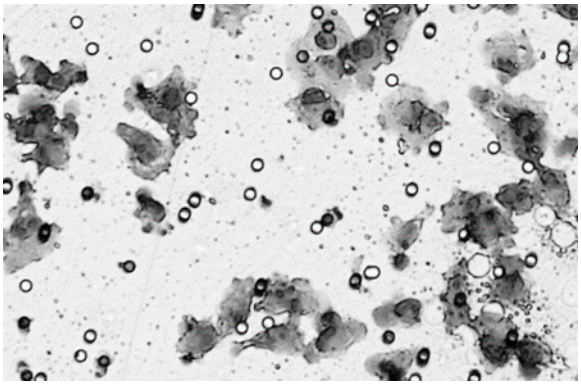


Figure 140 : Original niveaux de gris

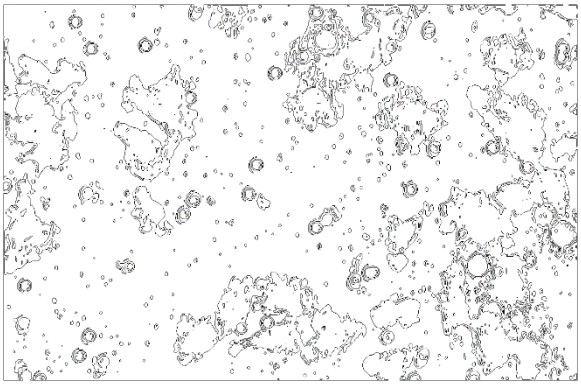


Figure 141 : Extérieur flou gaussien Sobel gris inversé

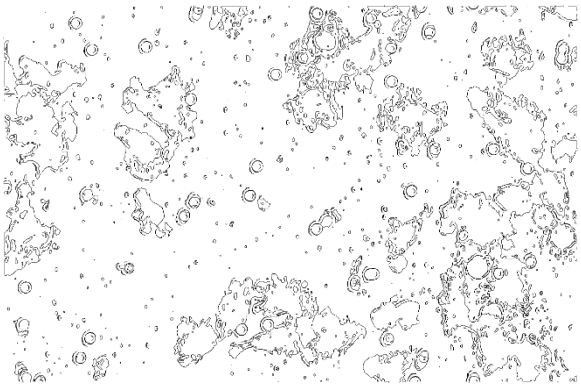


Figure 142 : Intérieur flou gaussien Sobel gris inversé

Implémentation personnalisée

```
//Filtre de Sobel
var sobl = Convolution.Convolve(gaus.Output, new SobelFilter4O(), true);
byte otsuTh = (byte)OtsuThresholding.Compute(sobl.Output);
//Seuillage par binarisation
var th = BinaryThresholdingFilter.Apply(sobl.Output, otsuTh);
//Erosion
var ero = Erode.Apply(th, new CrossStructuredElement(), otsuTh);
//Contour intérieur
var interior = Morpho.Sub(th, ero);
//Dilatation
var dil = Dilate.Apply(th, new CrossStructuredElement(), otsuTh);
//Contour extérieur
var ext = Morpho.Sub(dil, th);
```

Implémentation OpenCV

```
//Seuillage par binarisation
Cv2.Threshold(output, th, 0, 255, ThresholdTypes.Otsu|ThresholdTypes.Binary);
//Erosion
Mat element = Cv2.GetStructuringElement(MorphShapes.Ellipse, new Size(3,3));
Cv2.MorphologyEx(th, ero, MorphTypes.Erode, element);
//Soustraction de l'érodé, contour intérieur
var interior = th - ero;
//Dilatation
Cv2.MorphologyEx(th, dilate, MorphTypes.Dilate, element);
//Soustraction de la dilatation, contour extérieur
var exterior = dilate-th;
```

6.13 Gradient morphologique

Le gradient morphologique ou contour moyen est l'addition du contour intérieur et du contour extérieur.

$$C = (A \oplus V_n)/(A \ominus V_n)$$

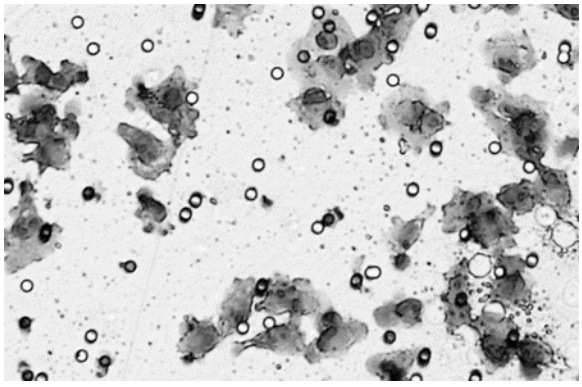


Figure 143 : Original niveaux de gris

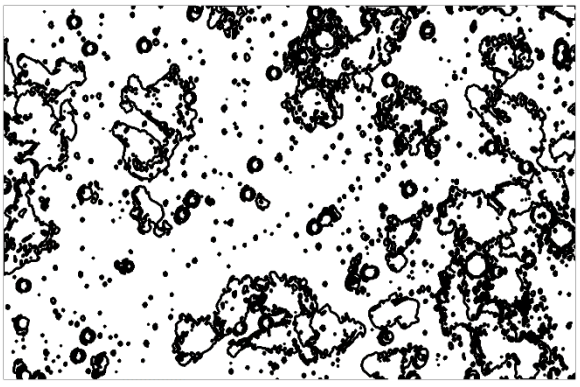


Figure 144 : Gradient morphologique rond flou gaussien Sobel gris inversé

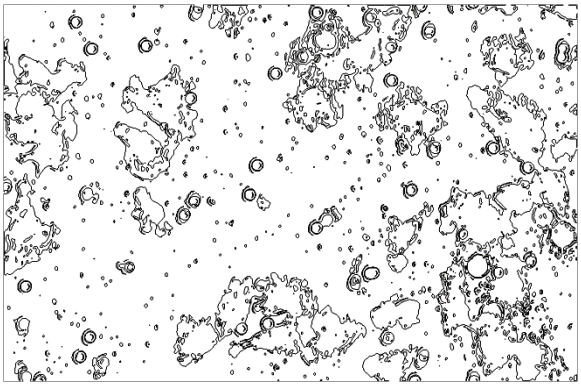


Figure 145 : Gradient morphologique en croix flou gaussien Sobel gris inversé

Implémentation personnalisée

```
//Sobel 4 orientations
var sobl = Convolution.Convolve(gaus.Output, new SobelFilter4O(), true);
byte otsuTh = (byte)OtsuThresholding.Compute(sobl.Output);
//Seuillage par binarisation
var th = BinaryThresholdingFilter.Apply(sobl.Output, otsuTh);
//Erosion
var ero = Erode.Apply(th, new RoundStructuredElement(), otsuTh);
//Dilatation
var dil = Dilate.Apply(th, new RoundStructuredElement(), otsuTh);
//Contour intérieur
var inter = Morpho.Sub(th, ero);
//contour extérieur
var ext = Morpho.Sub(dil, th);
//Gradient morphologique
var grad = Morpho.Add(inter, ext);
```

Implémentation OpenCV

```
//Gradient morphologique en croix
//Open cv définit le gradient morphologique par la différence entre la dilatation et l'érosion
Mat element = Cv2.GetStructuringElement(MorphShapes.Cross, new Size(5, 5));
Cv2.MorphologyEx(output, ero, MorphTypes.Gradient, element);
```

6.14 Intersection de masque

L'intersection de masque ne fait pas partie des opérateurs morpho mathématiques. Il s'agit de l'opération ensembliste permettant d'obtenir, sur base d'une image *A* et d'un masque *B*, une troisième image *C* ne contenant que l'intersection des deux images précédentes.

$$C = A \cap B$$

Le résultat de l'intersection de l'image et du masque ne contient que les éléments appartenant aux deux ensembles. L'intersection comporte les pixels blancs du masque et les pixels à la même position de l'image originale. Le masque peut être obtenu soit par seuillage binaire, soit par un seuillage par bande de couleur et binarisation du résultat, soit par l'empilage de plusieurs filtres suivi d'un seuillage par hystérésis ou seuillage par zéro. Les pixels de fond du masque *B* de couleur noire ne font pas partie de l'intersection avec l'image *A*. Pour tous pixel de l'image, si le pixel du masque aux mêmes coordonnées n'est pas un pixel de fond (noir), le pixel de sortie est égal à l'intensité du pixel de l'image d'origine.

$$P(x,y) = \begin{cases} 0 & si\ masque(x,y) = 0 \\ original(x,y) & sinon \end{cases}$$

Ci-dessous, un masque créé avec un seuillage par bande, trois érosions pour se débarrasser d'un maximum de bruit suivi de trois dilatations pour recouvrer la taille « d'origine ». L'image de sortie est le résultat de l'intersection entre l'image originale et le masque précédemment obtenu. Le bruit et les ports ont disparu pour ne laisser place qu'aux régions contenant des cellules. Le bruit résultant est négligeable.

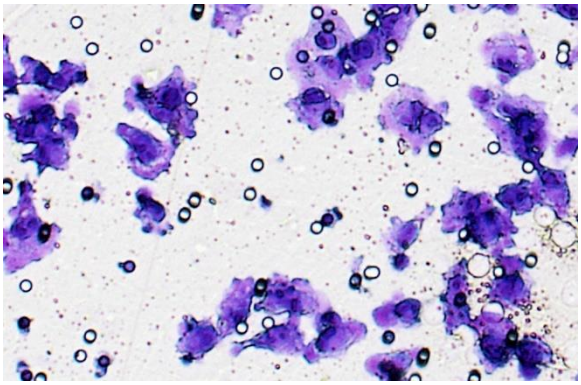


Figure 146 : Originale

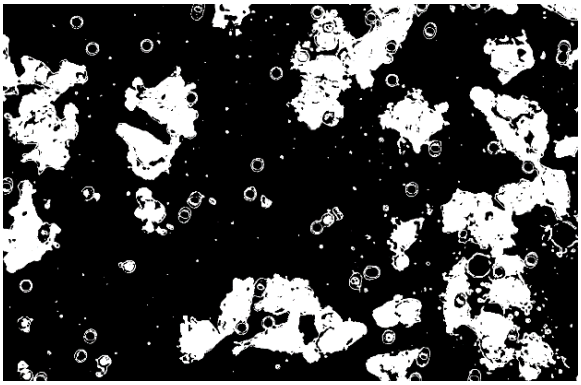


Figure 147 : Masque par seuillage de couleur

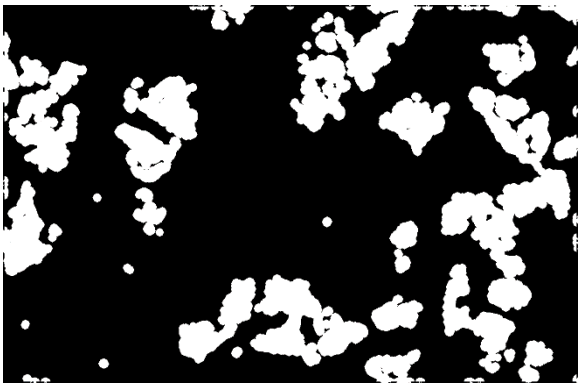


Figure 148 : Masque par seuillage de couleur, érosions et dilatations

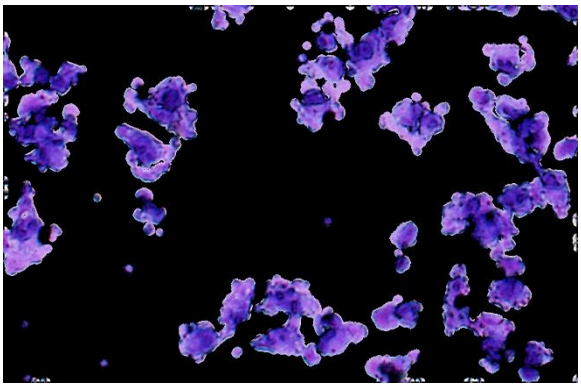


Figure 149 : Intersection du masque et de l'image originale

Implémentation personnalisée

```
var res = GammaCorrection.Correct(v, 0.3);
//Seuillage par bande de couleur RGB
Color c = Color.FromArgb(200, 147, 233);
var resThr = BandThresholdingFilter.Apply(res, c, 100, 60);
//Erosions du masque
var ero = Erode.Apply(resThr, new RoundStructuredElement(), 128);
ero = Erode.Apply(ero, new RoundStructuredElement(), 128);
//Dilatations du masque
var dil = Dilate.Apply(ero, new RoundStructuredElement(), 128);
dil = Dilate.Apply(dil, new RoundStructuredElement(), 128);
dil = Dilate.Apply(dil, new CrossStructuredElement(), 128);
dil.Save(@".\52BandThresholdingOpenTest.png");
//Intersection du masque et de l'image originale
var sub = Morpho.Intersec(v, dil);
```

Implémentation OpenCV

```
//Convertion RGB vers HSB
Mat hsv = new Mat();
Cv2.CvtColor(v, hsv, ColorConversionCodes.BGR2HSV);
//Déclaration des seuil de couleurs HSB
```

```
var lowher_color = new Scalar(120, 0, 0);
var higher_color = new Scalar(280, 255, 220);
//Seuillage par bande de couleur
Mat mask = new Mat();
Cv2.InRange(hsv, lowher_color, higher_color, mask);
//Erosion
var erode = mask.Erode(Cv2.GetStructuringElement(MorphShapes.Ellipse, new Size(11, 11)));
//Dilatation
var morpho = erode.Dilate(Cv2.GetStructuringElement(MorphShapes.Ellipse, new Size(15, 15)));
//Intersection du masque et de l'image originale
Mat output = new Mat();
Cv2.BitwiseAnd(v, v, output, morpho );
```

6.15 Conclusion

Ce chapitre est l'un des plus intéressants et des plus riches. Comparativement aux deux précédents qui se basent essentiellement sur la convolution, celui-ci regorge d'algorithmes et de transformations.

La fin du chapitre est un réel plaisir. Grâce au seuillage par couleur et à l'opération d'intersection de masque, il est possible d'obtenir une image ne contenant presque que des cellules. Il n'est pas encore question d'en localiser les noyaux mais nous n'en sommes pas loin. De plus, les tests accomplis ne mettent pas en œuvre les filtres de lissage tel que le filtre bilinéaire qui devrait pouvoir aplanir les couleurs et assurer une meilleure localisation.

Notre implémentation est parfois plus facile à mettre en œuvre et à comprendre par rapport à la librairie OpenCV. La documentation de la librairie OpenCV existe, mais elle est parfois trop concise, voire floue, et ne permet pas de la mettre en œuvre sans recourir à d'autres ressources.

Chapitre 7
Segmentation

Notre voyage initiatique touche doucement à sa fin. Cette dernière étape va nous mener vers l'un des domaines les plus importants et les plus difficiles du traitement d'image numérique : la segmentation.

La segmentation permet d'isoler des parties de l'image, des groupes de pixels ayant une relation les uns par rapport aux autres. Elle permet de partitionner l'image, d'identifier des objets de la vie courante ou des éléments de ceux-ci. La segmentation est complexe à cause de la grande diversité des objets à identifier et propre à chaque domaine d'application. Dans notre cas, il s'agit bien évidemment de pouvoir identifier et énumérer les cellules, mais aussi les noyaux de celles-ci. Il est peu probable que le système que nous tentons de construire puisse reconnaitre des cellules dans d'autres conditions sans y apporter des modifications.

Nous aborderons les deux méthodes de segmentation : la première, la segmentation par régions qui consiste à identifier les groupements de pixels dont les caractéristiques sont similaires. La seconde, la détection par contour qui consiste à identifier les frontières des régions.

7.1 Détection de régions par seuillage

Le seuillage a déjà été évoqué dans le « Chapitre 6 ». Le seuillage est régulièrement utilisé pour isoler les régions. Il ne s'agit pas ici d'un algorithme unique mais d'un empilage de filtres tels que les filtres de lissage, filtres passe haut et seuillage. Le résultat de cette méthode est une image en noir et blanc représentant respectivement le fond et les régions.

7.2 Détection de région par classification

Une deuxième technique de segmentation par seuillage consiste à effectuer une classification des intensités. Une première méthode consiste à repérer sur l'histogramme les pics correspondant à une forte présence d'intensité dans l'image et à le considérer comme une classe. Lorsque les pics sont identifiés, un seuillage par bande de couleur dont les seuils sont des niveaux de gris permet de récupérer l'ensemble de la classe. L'amplitude des seuils est soit une valeur arbitraire fixe soit une valeur calculée à l'aide de la variance des intensités (la moyenne des carrés des écarts à la moyenne). L'image finale est obtenue en combinant les différentes classes.

Ce partitionnement peut être réalisé à l'aide de l'algorithme k-mean, fréquemment utilisé en data science pour partitionner une population et identifier des profils au sein de celle-ci. K-mean peut aider à réaliser le partitionnement, surtout si le nombre de pics est plus important que le nombre de classes ou si les pics sont difficilement identifiables.

L'algorithme se divise en trois étapes :

- Initialisation : Le nombre de classes est choisi arbitrairement en fonction des besoins. Un nombre d'intensités correspondant au nombre de classes est tiré aléatoirement et sera considéré comme les centroïdes.
- Etape 1 : La distance euclidienne entre chaque intensité et les centroïdes est calculée. La distance euclidienne se calcule par $\sqrt{(xA - xB)^2 + (yA - yB)^2}$.
- Etape 2 : Les intensités sont assignées au centroïde le plus proche.
- Etape 3 : Les moyennes de chaque classe sont calculées et deviennent les nouveaux centroïdes
- Les étapes 1 à 3 sont répétées tant que des intensités sont déplacées de groupes. Le seuil bas d'une classe est le minimum de la classe et le seuil haut est le maximum.

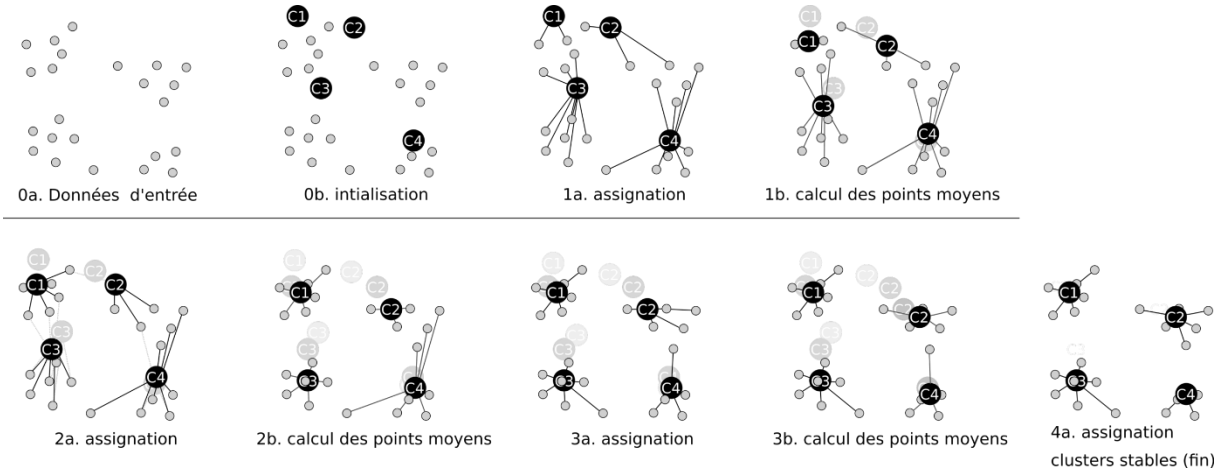


Figure 150 : Illustration du déroulement de l'algorithme K-means[3]

Une question se pose : serait-il possible de partitionner une image afin d'obtenir une classe pour le fond, une classe pour les cellules et une classe pour les noyaux ?

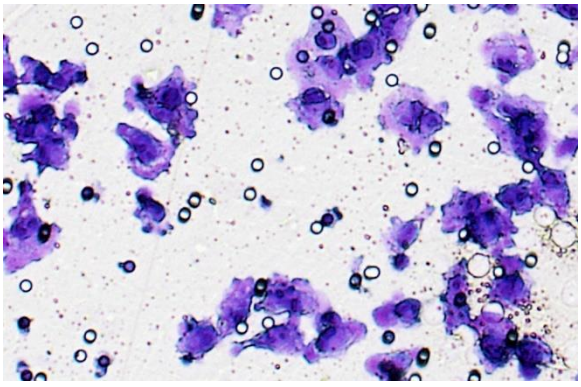


Figure 151 : Originale couleur

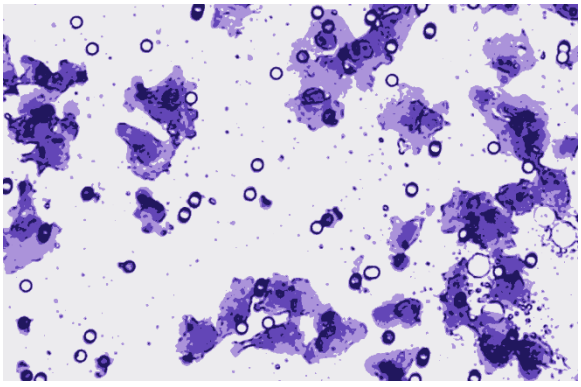


Figure 152 : Filtre Kmean

[3] https://fr.wikipedia.org/wiki/K-moyennes#/media/File:K-means.png

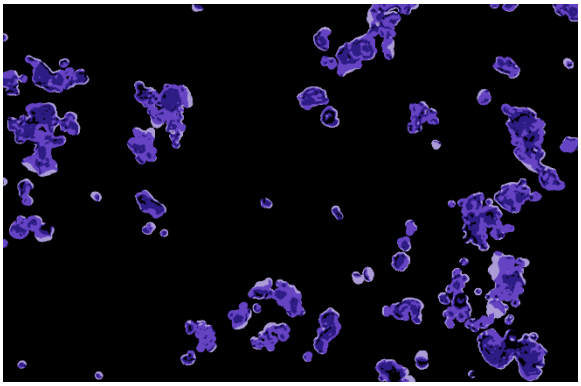


Figure 153 : Effacement des pores avec inpainting (chapitre 7.7), seuillage par bande de couleur et filtre kmean

Implémentation OpenCV

```
int width = input.Cols;
int height = input.Rows;

points.Create(width * height, 1, MatType.CV_32FC3);
centers.Create(k, 1, points.Type());
result.Create(height, width, input.Type());

int i = 0;
for (int y = 0; y < height; y++)
{
    for (int x = 0; x < width; x++, i++)
    {
        Vec3f vec3f = new Vec3f
        {
            Item0 = input.At<Vec3b>(y, x).Item0,
            Item1 = input.At<Vec3b>(y, x).Item1,
            Item2 = input.At<Vec3b>(y, x).Item2
        };

        points.Set<Vec3f>(i, vec3f);
    }
}

//Calcul du kmean
Cv2.Kmeans(points, k, labels, new TermCriteria(CriteriaType.Eps | CriteriaType.MaxIter,
10, 1.0), 3, KMeansFlags.PpCenters, centers);

i = 0;
//Application des couleurs des centroïdes
for (int y = 0; y < height; y++)
{
    for (int x = 0; x < width; x++, i++)
    {
        int idx = labels.Get<int>(i);

        Vec3b vec3b = new Vec3b();

        int tmp = Convert.ToInt32(Math.Round(centers.At<Vec3f>(idx).Item0));
        tmp = tmp > 255 ? 255 : tmp < 0 ? 0 : tmp;
        vec3b.Item0 = Convert.ToByte(tmp);

        tmp = Convert.ToInt32(Math.Round(centers.At<Vec3f>(idx).Item1));
        tmp = tmp > 255 ? 255 : tmp < 0 ? 0 : tmp;
        vec3b.Item1 = Convert.ToByte(tmp);

        tmp = Convert.ToInt32(Math.Round(centers.At<Vec3f>(idx).Item2));
```

```
            tmp = tmp > 255 ? 255 : tmp < 0 ? 0 : tmp;
            vec3b.Item2 = Convert.ToByte(tmp);

            result.Set<Vec3b>(y, x, vec3b);

        }
    }
```

7.3 Détection de blobs

Un algorithme d'agglomération de pixels ou de blob, consiste à attacher des groupes de pixels voisins dont les caractéristiques sont similaires tel que le niveau d'intensité. La détection de blob permet d'identifier les régions contenant ces blobs.

La librairie OpenCV implémente une méthode de détection de blob, cet algorithme de détection de blob fonctionne en quatre étapes :

- Le seuillage par échelon, permettant de générer *n* images binaires correspondant chacune à un échelon,
- Les pixels connectés dans chaque image binaire sont groupés pour former des blobs,
- Les blobs des différentes images binaires sont fusionnés s'ils sont assez proches,
- Les centres et rayon des blobs sont calculés pour former une liste de blobs.

Cette méthode de détection de blobs possède un ensemble de paramètres permettant de filtrer les blobs en fonction de leurs couleurs, tailles ou formes. Les blobs sont détectés et peuvent être dénombré. OpenCV retourne une liste de blobs avec leur centre et leur rayon.

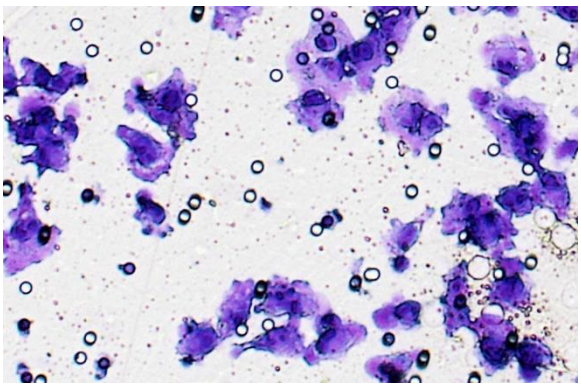


Figure 154 : Originale couleur

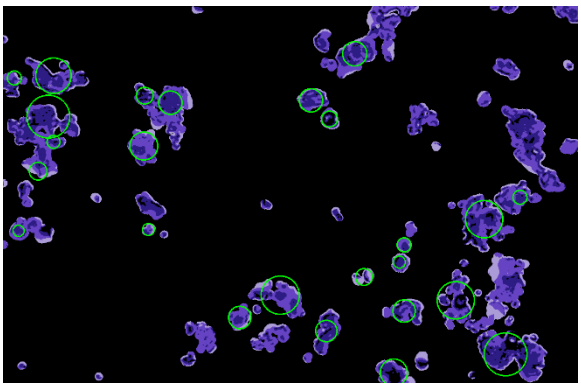


Figure 155 : Seuillage par bande de couleur, filtre kmean et détection de blobs

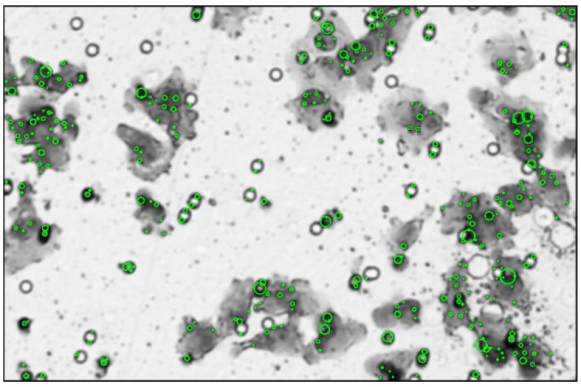


Figure 156 : Niveaux de gris, Filtre gaussien et détection de

blobs

Implémentation OpenCV

```
//Créer les paramètres de détection de blob
var detector = SimpleBlobDetector.Create(BlobParam.Param);
//Détection des blobs
var keypoints = detector.Detect(gray);
Mat im_with_keypoints = new Mat();
//Dessine les cercles représentant les blobs
Cv2.DrawKeypoints(image, keypoints, im_with_keypoints, new Scalar(0, 255, 0),
                  DrawMatchesFlags.DrawRichKeypoints);
```

7.4 Autres détecteurs de régions

D'autres algorithmes sont présents dans la littérature, mais rarement implémentés.

Un premier algorithme de division et de fusion, utilise un arbre quadratique pour classifier les pixels d'une région carrée de l'image. L'image est examinée. Si le bloc de l'image n'est pas homogène, elle est divisée en quatre blocs. Ils sont à leur tour examinés et le cas échéant à leurs tours divisés en quatre autres blocs, et ainsi de suite. Le résultat final permet de naviguer dans l'arbre à la recherche des régions, de leur intensité et de leur position. Une région à cheval sur deux blocs a systématiquement identifié comme deux régions.

Un second consiste à retrouver des régions caractérisées par un motif tel qu'un logo. Ce genre d'algorithme est fortement lié à son application. La recherche dépend de la position, de la mise à l'échelle, de l'orientation, de l'exposition, … Les patterns recherchés doivent être polymorphes selon ces différents critères.

7.5 Transformation de Hough

La transformé de Hough, du nom de son inventeur, est une technique de détection de formes créée en 1952. D'abord utilisée pour la détection de ligne, cette méthode a été généralisée pour repérer des éléments structurels plus complexe comme des cercles ou des ellipses.

Pour effectuer une transformé de Hough, il faut au préalable réaliser une détection de contour, par exemple à l'aide de la méthode de Canny, et ensuite appliquer la transformé.

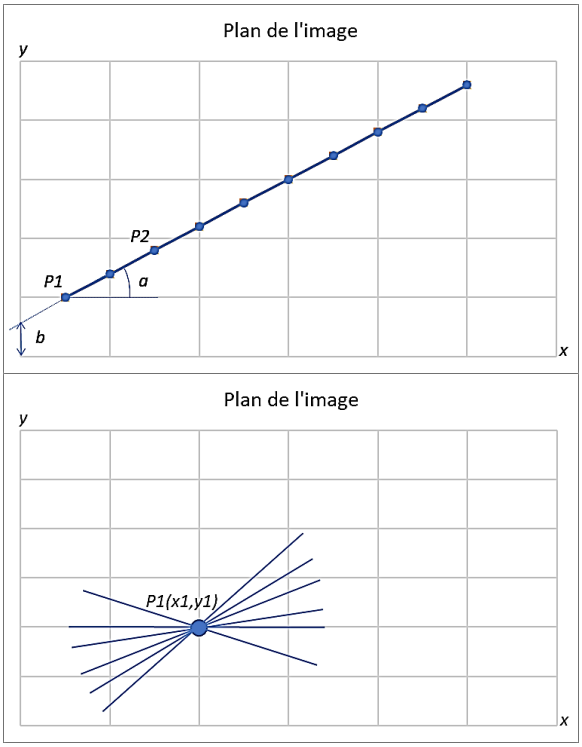
7.5.1 Le système cartésien

La base de la transformé de Hough est l'équation cartésienne d'une droite $(y = ax + b)$ déterminée par deux paramètres. Les paramètres de la droite sont *a, la pente de la droite et b* son ordonnée à l'origine, c'est-à-dire la valeur de *y* pour *x valant 0,* autrement dit, l'intersection de cette droite sur l'axe *y*. L'ensemble des valeurs des paramètres *a* et *b* représente le plan des paramètres.

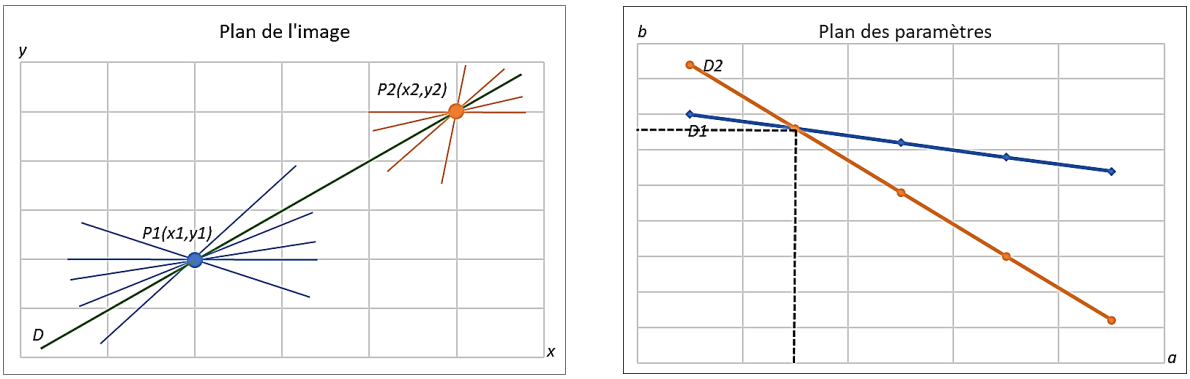
Pour deux pixels $P1(x1, y1)$ et $P2(x2, y2)$ dans le plan de l'image :

La pente est obtenue à l'aide de l'équation : $a = \frac{y2-y1}{x2-x1}$, et l'ordonnée à l'origine grâce à l'équation : $b = y1 - ax1$. Cette équation dans le plan des paramètres donne une droite unique *D1*.

Cette même équation appliquée au pixel *P2* donne une seconde droite *D2* dans l'espace des paramètres. *D1* représente l'ensemble des droites passant par les pixels *P1*. Dans le plan des paramètres, le point d'intersection de ces deux droites donne les paramètres permettant de tracer la droite passant par les deux points *P1* et *P2* dans le plan de l'image. Chaque droite dans l'espace de l'image est donc transformée en un seul point dans l'espace des paramètres.



L'ensemble des droites passant par un point P1 est $y1 = ax1 + b$, ou *a* et *b* ont différentes valeurs. Ci-dessous, l'image comporte une droite dont deux des pixels sont représentés. En traçant l'ensemble des droites dans l'espace de l'image, deux droites sont créées dans l'espace des paramètres. L'intersection de ces deux droites donne le paramètre a et b correspondant à la droite *D*.



L'implémentation de la transformé de Houg se réalise en quelques étapes simples. Elle requiert que l'espace des paramètres soit discrétisé et borné afin de réduire l'ensemble des

droites à un espace fini.

Pour trouver les droites dans les contours d'une image, l'algorithme de la transformée de Hough peut être défini de la sorte :

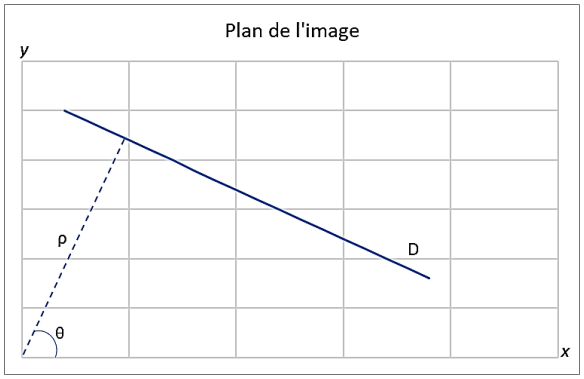
- Définir l'espace des paramètres au travers des bornes et paliers pour chacuns des paramètres,
- Appliquer un filtre de Canny à l'image d'origine pour en extraire les contours,
- Initialiser une matrice d'accumulation dont les dimensions correspondent aux espaces de *a* et de *b*,
- Pour chaque pixel appartenant aux contours
    - Pour chaque valeur comprise dans l'espace de *a*
        - Générer la droite passant par le point de contour sélectionné en calculant la valeur de *b*
        - Ajouter un vote dans la matrice d'accumulation à l'indice de *a* et *b*
- Sélectionner les valeurs maximales de l'accumulateur pour lesquels ces valeurs dépassent un seuil défini. Les valeurs sélectionnées correspondent aux droites reliant un grand nombre de pixels. Les paramètres (*a,b*) sélectionnés correspondant aux lignes comprisses dans l'image.

7.5.2 Le système polaire

L'équation cartésienne présente un problème pour les droites purement verticales. En effet, les valeurs des paramètres *a* et *b* sont infinies. Pour cette raison, il est préférable d'utiliser la seconde approche qui consiste à utiliser l'équation polaire.

Cette deuxième approche comporte également deux paramètres Thêta (θ), la distance entre l'origine et la droite et Rho (ρ) l'angle entre l'axe x et le vecteur thêta. Ces deux paramètres sont bornés, thêta est compris entre 0 et la diagonale de l'image et rho est compris entre 0 et 2 pi. L'équation polaire permet de calculer Rho : $\rho = \sin(\theta)\, y + \cos(\theta)\, x$.

Le principe de cette approche est identique, pour chaque pixel p(x,y), pour chaque valeur de thêta, une valeur de Rho est calculée. Contrairement à la première approche, ces valeurs ne construisent pas une droite dans l'espace des paramètres mais une courbe sinusoïdale. Les intersections des sinusoïdes de l'espace des paramètres indiquent les paramètres des droites dans l'espace del'image.



L'algorithme fonctionne de la même façon que pour le système cartésien, une matrice

d'accumulation est créée afin de récolter le nombre de votes par jeu de paramètre. Les maximas donnent les droites présentes sur le plan de l'image.

7.5.3 Les variantes

La transformé de Hough existe dans diverses formes, essentiellement dans une optique de performance.

La transformation probabiliste se contente de calculer une partie des pixels appartement au contour. La méthode choisit une partie des pixels aléatoirement, 20% des pixels devraient suffire pour obtenir un résultat acceptable.

La transformation par tirage aléatoire sélectionne des couples de points, calcule la transformé et incrémente seulement l'accumulateur passant par les deux points.

La transformation hiérarchique apporte une solution au problème de dimensionnement des foulées des paramètres. Cette méthode se déroule en deux passes, une première passe avec une foulée de paramètres assez large permettant d'obtenir de bonnes performances et une seconde passe va subdiviser les intervalles ayant obtenu le plus grand nombre de votes permettant ainsi d'obtenir une bonne précision.

7.5.4 Détection de courbes

La détection de courbes à l'aide de la transformation de Hough est similaire à la détection de droites, seule l'équation et l'espace des paramètres sont différents.

Les paramètres sont (*ab*) de la détection d'une droite donnant le centre du cercle et (*R*) donnant le rayon du cercle.

L'équation est la suivante : $(x-a)^2 + (y-b)^2 - R^2 = 0$

L'accumulateur est un peu plus complexe, il s'agit de créer une matrice multidimensionnelle pour chaque triplet de paramètres (a, b, R).

L'idée est similaire à la détection de droites. Les pixels de contour sont parcourus et pour chacun d'eux, les cercles possibles de l'image sont calculés. Si le pixel appartient au cercle, la matrice d'accumulation est incrémentée pour ce jeu de paramètres. Si lors de la détection, le rayon du cercle à trouver est connu, la performance de détection s'en trouve accrue puisqu'il suffit de ne traiter que des rayons connus et non des rayons allant de 1 à la dimension maximum de l'image. Les valeurs maximums de la matrice d'accumulation représentent les centres et rayons des cercles présents dans le plan de l'image.

7.5.5 Généralisation

La transformation de Hough, comme pour la détection de cercle, peut être généralisée afin de détecter d'autres formes, telles que des ellipses ou des formes complexes pouvant être définies par une équation paramétrique. Le principe est toujours identique, sur base des contours, il faut appliquer aux pixels l'équation et incrémenter la matrice d'accumulation

correspondante si le pixel fait partie de la forme complexe.

7.5.6 En image

Ci-dessous, une détection de cercles à l'aide de la méthode de Hough, la taille du rayon des cercles est de 13 à 15 pixels. Les pores sont presque tous détectés et d'autres cercles sont détectés dans la structure des cellules. Un deuxième essai avec un rayon inférieur permet de détecter plus de pores mais donne beaucoup plus de faux positifs.

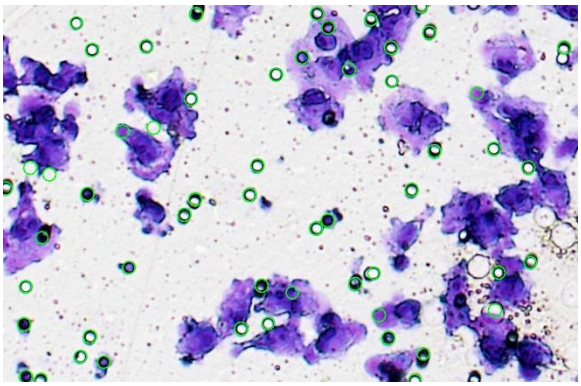


Figure 157 : Transformation de Hough détection de cercles

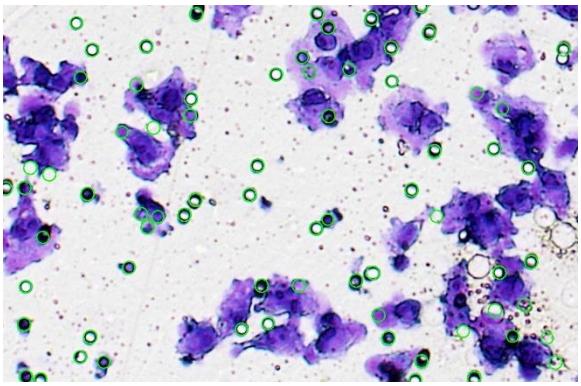


Figure 158 : Transformation de Hough paramètre de rayon de 12

Implémentation OpenCV

```
//Transformé de Hought
//14.5 distance minimum entre les cercles
//200 seuil haut pour la méthode de Canny
//100 Seuil pour la détection de centre
//13 et 15 tailles minimum et maximum du rayon des cercles à détecter
var circles = Cv2.HoughCircles(gray, HoughMethods.Gradient, 1, 14.5, 200, 10, 13, 15);

//Trace les cercles sur l'image d'origine
foreach (var circle in circles)
    Cv2.Circle(v, (int)circle.Center.X, (int)circle.Center.Y, (int)circle.Radius, new
Scalar(0,0,255), 2);

Cv2.ImWrite(@".\HoughtCircleTest.png", v);
```

7.6 Contour d'une image binaire

Suzuki et Abe ont défini un algorithme de chainage de contour dans l'article *Topological Structural Analysis of Digitized Binary* en 1983. L'algorithme inspecte les pixels et leur connexité du résultat d'une détection de contour, par exemple Canny, et d'un seuillage binaire d'une image afin de détecter les polygones présents dans l'image. La valeur des pixels de l'image binaire initiale est zéro ou un. À la suite de la détection de contour, les pixels se sont vu assigner des valeurs suivant une politique de labélisation unique permettant d'identifier un contour contigu par la simple énumération des pixels dont la valeur est équivalente à l'identifiant donné. Le résultat de cette détection est stocké dans un arbre binaire, les parties de « droites » peuvent ainsi être reliées pour former des polygones complexes.

Cet algorithme est souvent associé au code de Freeman, les contours sont encodés comme un

chainage de déplacement permettant de le tracer. Dans cette représentation, tous les points de contour ne sont pas retenus, seules les directions normalisées, par exemple de 1 à 8, sont retenues.

La librairie OpenCV implémente cet algorithme, ci-dessous, en vert les contours détectés par cette méthode. Cette méthode détecte trop d'éléments, les pores ont été gommés mais de nombreux petits éléments apparaissent.

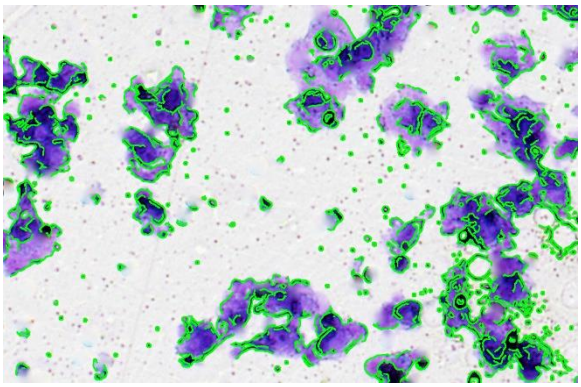


Figure 159 : Filtre gaussien, Canny, seuille de Otsu et détection de contours

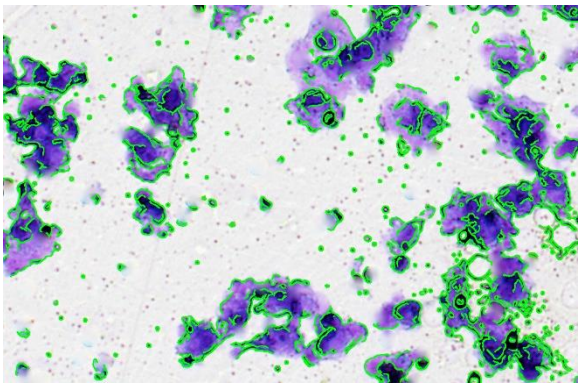


Figure 160 : Filtre gaussien, Canny, seuille de Otsu, inpainting et détection de contours

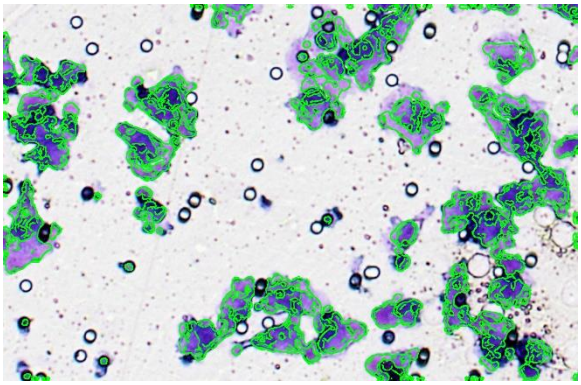


Figure 161 : Filtre gaussien, Canny, seuille de Otsu, seuillage par bande de couleur et détection de contours

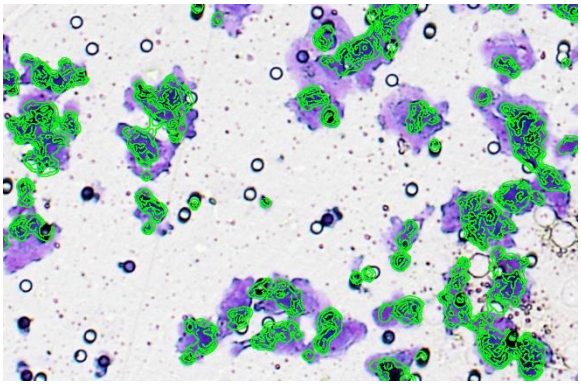


Figure 162 : Filtre gaussien, Canny, seuille de Otsu, inpainting, seuillage par bande de couleur et détection de contours

Implémentation OpenCV

```
//Filtre gaussien
Cv2.GaussianBlur(v, blur, new OpenCvSharp.Size(7, 7), 5, 5, BorderTypes.Default);
//Méthode de Otsu
int thOtsu = (int)OtsuThresholding.Compute(v2);
//Détecteur de contours de Canny
Cv2.Canny(blur, output, thOtsu/3, thOtsu);
//Enumération des contours
Cv2.FindContours(output, out contours, out hierarchy, RetrievalModes.List,
ContourApproximationModes.ApproxTC89KCOS);

//Dessine les contours en vert
for(var i =0; i<contours.Length; i++)
    Cv2.DrawContours(v, contours, i, new Scalar(0,255,0), 2);
```

7.7 Retouche, interpolation d'image (Inpainting)

L'un des plus gros chalenges de l'imagerie numérique est la possibilité de reconstruire ou d'effacer une partie de l'image en laissant une sensation, pour qui l'observe, que l'objet n'a jamais existé. Cette technique, consiste à combler de manière itérative une zone ou à incorporer des données manquantes. Elle est empruntée au monde artistique et à la restauration d'œuvre d'art.

Dans le monde numérique, la méthode Inpainting permet de reconstruire des parties manquantes ou endommagées d'une image et permet également de supprimer des logos, textes ou objets indésirables.

Il existe trois grandes familles d'algorithme de retouche : la retouche structurelle, la retouche texturale et la retouche mixte qui est une combinaison des deux précédentes.

La retouche structurelle vise à reconstruire des structures géométriques, pour rétablir une cohérence des contours, tels que le prolongement d'une ligne ou la fermeture d'un cercle.

La retouche texturale quant à elle travaille à reconstruire des textures, des modèles répétitifs. La reconstruction de motifs tels que la peau, du gazon ou des nuages.

La retouche mixte combine la retouche structurelle et la retouche texturale. En effet, les images sont un mix d'éléments structurel et texturaux. Un mix de basse fréquence et de haute fréquence, un mix de contours et de fonds. Il est en finalité possible de reconstruire un objet en reconstruisant les contours et textures voisines.

Les techniques de retouche sont variées, copie de texture avoisinante, détection et fermeture de contour, équation différentielle, variation du voisinage, détection de motif similaire, … Seules les méthodes de Telea et de Navier Stokes seront présentées. Ces deux techniques sont parfaites pour faire disparaitre les pores, de plus elles sont implémentées dans la librairie OpenCV.

7.7.1 La méthode de Telea (2004)

Cette méthode a été définie par Alexandru Telea dans un article de 2004 « An Image Inpainting Technique Based on the Fast Marching Method ».

Le principe est assez simple : pour une région devant être retouchée, l'algorithme commence à partir de la limite de la région et se déplace petit à petit à l'intérieur de cette région. Un pixel peut être remplacé, si le nombre de voisins est suffisant, par une somme pondérée normalisée des pixels du voisinage. La pondération est définie en fonction de la proximité du pixel à traiter, plus le pixel est proche plus la pondération est importante, plus il est éloigné moins la pondération est élevée. Le centre de la région est retouché à l'aide des pixels de voisinage précédemment traités.

7.7.2 La méthode Navier-Stokes (2001)

Cette méthode est basée sur le livre « Navier-Stokes, Fluid Dynamics, and Image and Video

Inpainting » par Bertalmio, Marcelo, Andrea L. Bertozzi et Guillermo Sapiro en 2001. L'algorithme se déplace d'abord le long des frontières de régions connues vers des régions inconnues. Il va poursuivre des contours d'image de luminosité égale en respectant les vecteurs de gradient de la région à peindre en utilisant certaines méthodes de la dynamique des fluides. Une fois les contours obtenus, les trous sont remplis.

7.7.3 Inpainting sous OpenCV

OpenCV implémente les deux méthodes décrites ci-dessus. La retouche se réalise en deux phases, la détection des cercles pour la création d'un masque et la retouche par InPainting à l'aide du masque.

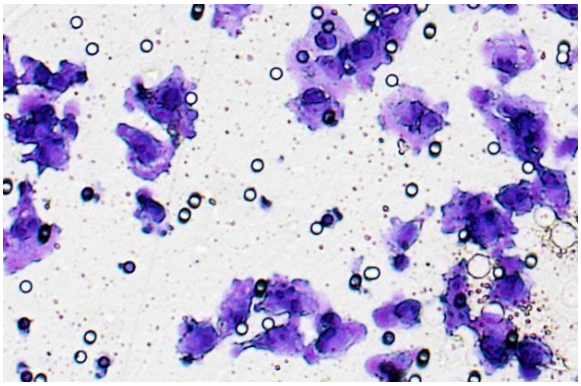


Figure 163 : Originale

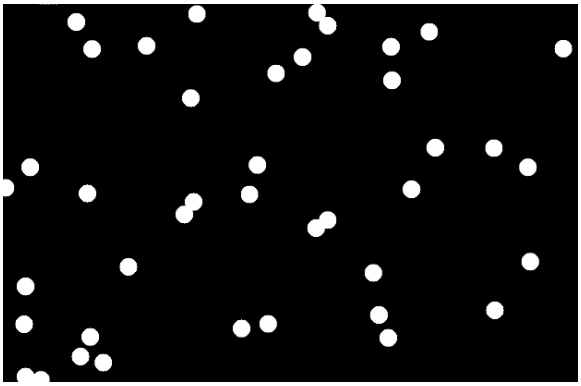


Figure 164 : Masque contenant les cercles repérés

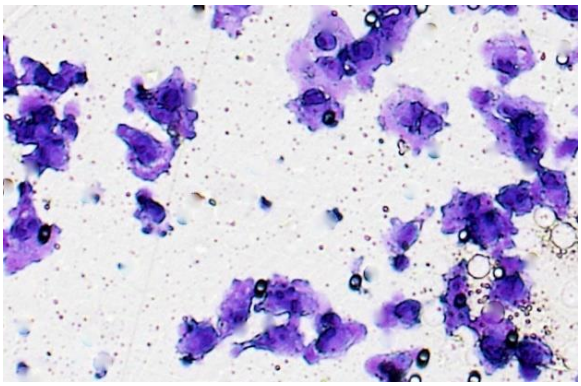


Figure 165 : Retouche par lissage du voisinage méthode de Telea

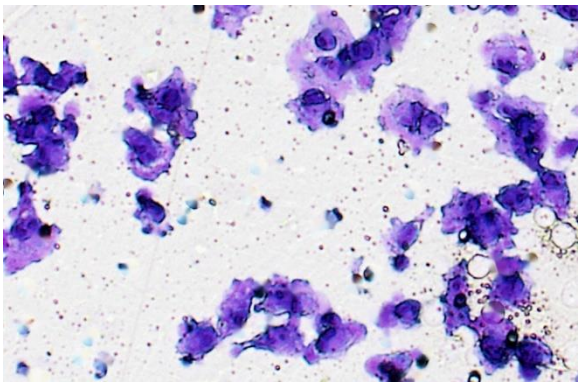


Figure 166 : Retouche par lissage du voisinage méthode de Navier Stokes

Implémentation OpenCV

```
Mat output = new Mat();
Mat gray = new Mat();
//Conversion en niveaux de gris
Cv2.CvtColor(v, gray, ColorConversionCodes.BGR2GRAY);
//Récupération des cercles à partir de l'image en niveaux de gris
var circles = Cv2.HoughCircles(gray, HoughMethods.Gradient, 1, 26, 200, 10, 14, 15);
//Création de la matrice 8 bits du masque
Mat mask = new Mat(v.Size(), MatType.CV_8U);
//Dessine les cercle sur le masque
var w = new Scalar(255) ;
foreach (var circle in circles)
    Cv2.Circle(mask, (int) circle.Center.X, (int)circle.Center.Y, (int) circle.Radius+5, w, -1);
//Effacement des objets du masque par floutage du voisinage avec la méthode de Telea
Cv2.Inpaint(v, mask, output, 30, InpaintMethod.Telea);
//Effacement des objets du masque par floutage du voisinage avec la méthode de Navier Stokes
```

```
Cv2.Inpaint(v, mask, output, 25, InpaintMethod.NS);
```

7.8 Conclusion

Une dernière conclusion pour clôturer cette première partie que nous qualifierons d'état de l'art. Tout en émettant une réserve sur l'expression « état de l'art » puisqu'il serait presque impossible d'en réaliser un de manière exhaustive. En effet, le domaine est vaste et nous nous sommes limités aux méthodes les plus populaires de traitements d'image et à la plus populaire des librairies de traitement.

Le chapitre précédent était l'un des plus riches, celui-ci est l'un des plus complexes. Nous avons implémenté les chapitres précédents mais celui-ci demanderait beaucoup trop de temps pour probablement n'obtenir que des résultats partiels avec des performances médiocres.

Après autant de temps de recherches et de tests, il est satisfaisant de enfin pouvoir localiser des objets sur une image. Malheureusement, il devient évident que la portion d'image que nous avons choisie pour nos illustrations n'est pas la plus claire et les cellules qui s'y trouvent sont difficilement identifiables à l'œil nu. De ce fait, il est difficile de trouver les bons réglages permettant d'identifier les cellules. De plus, nous utilisons des tests unitaires et non une interface graphique pour nos recherches, le jeu d'essais-erreurs est certaines fois fastidieux et long.

Nous pouvons néanmoins conclure que la détection de cercles fonctionne bien sur les pores même si quelques faux positifs sont détectés. La détection de blob de OpenCV n'est pas très efficace telle quelle, il est même difficile de savoir ce qui est vraiment détecté. Le seuillage par couleurs couplé à un filtre K-means semble être la bonne voie pour isoler les masses de couleurs. Par la suite, il devrait être possible d'identifier des amas de pixels ayant la couleur des noyaux. Cependant, pour en être certain, il nous faudra une interface permettant de facilement adapter les paramètres des différents traitements et ainsi trouver la bonne configuration permettant une détection optimale.

Chapitre 8
Implémentation

Après avoir regroupé et synthétisé l'ensemble des informations concernant les filtres et traitement divers, il était temps de passer à la phase d'implémentation et de test. Chaque filtre et traitement divers a été implémenté afin d'une part, illustrer les chapitres de 4 à 8, et d'autre part, maitriser le sujet, la terminologie et les algorithmes principaux.

Pour des raisons purement pratiques, l'implémentation personnelle s'est faite à l'aide du framework .Net et Visual Studio. Il n'y a rien de spécifique au C# ; le code pourrait être porté dans n'importe quel autre langage sans grande difficulté.

Nous verrons plus loin que, l'implémentation personnalisée des filtres, bien que correcte, manque cruellement de performance, de portabilité et de maintenabilité. Les tests de détection de cellules seront, par conséquent, implémentés à l'aide de la libraire OpenCv qui est très réputée dans le milieu.

8.1 L'implémentation des filtres

Le framework .Net possède une classe Bitmap permettant d'accéder aux informations d'une image enregistrée sur le disque. Les propriétés width et height permettent d'accéder au nombre de pixels en largeur et en hauteur ainsi qu'à la valeur de chaque pixel.

Parcours linéaire des pixels

```
// Retrieve the image.
var image1 = new Bitmap(@".\image.bmp", true);

// Loop through the images pixels to set color.
for(x=0; x<image1.Width; x++)
{
    for(y=0; y<image1.Height; y++)
    {
        Color pixelColor = image1.GetPixel(x, y);
        Color newColor = Color.FromArgb(pixelColor.R, 0, 0);
        image1.SetPixel(x, y, newColor);
    }
}
```

Les premiers tests ont montré que cette méthode n'est pas performante. Une seconde méthode utilisant un pointeur mémoire d'un tableau à une dimension permet d'obtenir des performances bien meilleures. Cependant, cette méthode est moins pratique que la précédente puisqu'il faut calculer les offsets de chacun des pixels en fonction des lignes ainsi que des trois composantes de couleur. La position des couleurs est inversée par rapport à la lecture RGB, l'ordre des bytes est BGR.

Parcours linéaire du pointeur

```
Bitmap source = (Bitmap)Bitmap.FromFile(@".\echantillon.png");
```

```
BitmapData data = source.LockBits(new Rectangle(0, 0, source.Width, source.Height),
ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);

IntPtr ptr = data.Scan0;

int tenPercent = source.Height / 10;

int bytes = Math.Abs(data.Stride) * tenPercent;
byte[] rgb = new byte[bytes];

// Copie les valeurs RGBdans le tableau
Marshal.Copy(ptr, rgb, 0, bytes);

int gray = 0;

//si 10% des composants des pixels de l'image ont la même valeur nous supposont qu'elle est en
niveaux de gris
for (int i = 0; i < rgb.Length; i += 3)
{
    if (rgb[i] == rgb[i + 1] && rgb[i + 1] == rgb[i + 2])
        gray += 3;
}

source.UnlockBits(data);
```

L'implémentation de la convolution personnalisée se fait par le biais du pointeur. L'adressage est complexe à cause de l'alignement sur 4 bytes des données, et du fait de la linéarité de la représentation mémoire. L'application d'un filtre se fait grâce à la méthode statique paramétrique Convolve de la classe Convolution.

Application de N kernel sur une image

```
//Foreach rows
for (int rowIndex = padding; rowIndex < height - padding; rowIndex++)
{
    //foreach lines
    for (int lineIndex = padding; lineIndex < width - padding; lineIndex++)
    {
        //Compute the current pixel byte offset
        byteOffset = rowIndex * stride + lineIndex * 3;

        //foreach kernel
        for (int i = 0; i < kernelCount; i++)
        {
            var kernel = kernels[i];
            var k = kernel.Kernel;
            var factor = kernel.Factor;
            gray[i] = 0;

            //foreach row in kernel
            for (int filterRowIndex = -padding; filterRowIndex <= padding; filterRowIndex++)
            {
                //foreach line in kernel
                for (int filterLineIndex = -padding; filterLineIndex <= padding; filterLineIndex++)
                {
                    calcOffset = byteOffset +
                                 (filterLineIndex * 3) +
                                 (filterRowIndex * stride);
                    gray[i] += (double)(pixelBuffer[calcOffset]) *
                             k[filterRowIndex + padding, filterLineIndex + padding];
                }
            }
            gray[i] = filter.ForceAbsoluteValue
                ? Math.Abs(factor * gray[i])
                : factor * gray[i];
            gray[i] = gray[i] > 255 ? 255 : gray[i] < 0 ? 0 : gray[i];
        }

        if (twoKernel)
        {
```

```
            if (computeDirection)
                res.Directions[byteOffset] = ToDirection(gray[0], gray[1]);
            var result = (byte)Math.Sqrt(Math.Pow(gray[0], 2) + Math.Pow(gray[1], 2));
            resultBuffer[byteOffset + 1] = resultBuffer[byteOffset + 2] = resultBuffer[byteOffset] =
                                           result;
        }
        else
        {
            if (computeDirection)
                res.Directions[byteOffset] = ToDirection(filter.Kernels, gray);
            resultBuffer[byteOffset + 1] = resultBuffer[byteOffset + 2] = resultBuffer[byteOffset] =
                                           (byte)Max(gray);
        }
    }
}
```

Chaque filtre de convolution dérive de la classe ConvolutionFilterBase et nécessite de surcharger la méthode InitKernels qui ajoute au filtre les différents kernels, leur orientation et leur coefficient.

Application de N kernel sur une image

```
/** @overview Filtre de détection de Roberts
  * @specfields name:String //"Roberts Filter" */
public class RobertsFilter : ConvolutionFilterBase
{
    public override string Name => "Roberts Filter";

    /**
    * @see base.InitKernels();
    */
    protected override void InitKernels()
    {
        var k1 = new double[,]{
            { 0, 0,  0 },
            { 0, 1,  0 },
            { 0, 0, -1 }
        };
        var k2 = new double[,]{
            { 0,  0, 0 },
            { 0,  0, 1 },
            { 0, -1, 0 }
        };

        this.AddKernel(k1, 1, KernelOrientation.EasternNorth);
        this.AddKernel(k2, 1, KernelOrientation.WesternNorth);
    }
}
```

Bien que les filtres principaux soient implémentés dans OpenCv, ils ne le sont pas tous. La méthode de convolution de OpenCV ne tient pas compte du coefficient, c'est pourquoi il est nécessaire de l'appliquer sur le kernel avant de convoluer. Le résultat de la fonction de convolution doit encore être traité, d'une part pour s'assurer que les valeurs soient positives et d'autre part pour assembler les résultats de sortie des différents kernels.

Application de deux filtres de convolution avec OpenCV

```
//Creation des kernels
var kernelx = new float[,]{
                { 1, 0, -1 },
                { 1, 0, -1 },
                { 1, 0, -1 }
            };
var kernely = new float[,]{
                { 1, 1, 1 },
                { 0, 0, 0 },
                { -1, -1, -1 }
```

```
            };
var kx = new Mat(3, 3, MatType.CV_32F, kernelx);
var ky = new Mat(3, 3, MatType.CV_32F, kernely);
//Convolution par deux kernels
Cv2.Filter2D(v, outputX, -1, kx);
Cv2.Filter2D(v, outputY, -1, ky);
//Conversion en valeurs absolue 8 bits
Cv2.ConvertScaleAbs(outputX, absX);
Cv2.ConvertScaleAbs(outputY, absY);
//Addition de deux matrices dont le poids de chacune des matrices est identique
Cv2.AddWeighted(absX, 0.5, absY, 0.5, 0, output);
```

Pour d'autres types de filtres, le parcours de la matrice avec OpenCV est plus simple et plus intuitif que la version C#. En effet, les propriétés Cols et Rows de la classe Mat permettent un accès aisé aux pixels.

Parcours linéaire de la matrice avec OpenCV

```
//Déclaration de variable pour diminuer les temps d'accès
int width = input.Cols;
int height = input.Rows;

//Parcourt en lignes et colonnes de la matrice
for (int y = 0; y < height; y++)
{
    for (int x = 0; x < width; x++)
    {
        //Lecture de la valeur du pixel courant
        var v = dest.At<Vec3b>(y, x);
        //Modification des composantes de couleur
        v.Item0 = removeBlue ? (byte)0 : v.Item0;
        v.Item1 = removeGreen ? (byte)0 : v.Item1;
        v.Item2 = removeRed ? (byte)0 : v.Item2;

        //Ecriture de la nouvelle valeur du pixel
        dest.Set(y, x, v);
    }
}
```

8.2 Analyse des performances

L'implémentation de traitement d'image dans un langage tel que C# ou Java montre assez rapidement ses limitations en termes de performances. En effet, derrière l'ensemble des appels, réside un mécanisme de sécurité permettant de ne pas atteindre une zone mémoire non allouée ou de dépasser la capacité d'un tableau. Un second mécanisme, le « *garbage collector* », s'assure qu'un objet ne soit pas libéré s'il est toujours référencé et à l'inverse, le libère s'il n'est plus référencé. Bref, ce genre de mécanisme a un coût sur les performances d'exécution.

De plus, la philosophie de ces langages vise à encapsuler les propriétés d'une classe afin qu'elle ne soit pas accessible de l'extérieur. Dans le cas de la classe Bitmap de C#, la propriété largeur (Width) masque l'appel à une fonction. En Java, la fonction se nommerait GetWidth. L'impact de ce mécanisme force le compilateur à passer par la pile d'appel afin d'obtenir la valeur de la propriété, sans parler des éventuels traitements annexes effectués avant le retour de la valeur. Ce style d'implémentation a également un impact sur la performance et l'utilisation de la mémoire.

Les tests unitaires des différents filtres implémentés ont montré que le temps d'exécution peut atteindre des sommets vertigineux. En effet, pour de petites images, le temps de réponse peut être acceptable si le processus ne doit pas être en temps réel. En revanche, pour des images obtenues à la Faculté de médecine, dont les hauteur et largeur sont de l'ordre de 14.000 pixels, le temps d'exécution dépasse très largement la minute.

À la suite de cette constatation, un simple test de performance permet de montrer une certaine linéarité entre le temps de fonctionnement et la taille de l'image. Le test proposé mesure le temps d'exécution du traitement d'image ainsi que son impact en mémoire. Le traitement est des plus basiques : l'image est chargée, convertie en niveaux de gris, un filtre gaussien et filtre de Sobel sont appliqués et enfin l'image est enregistrée sur disque. Les tests de performance ont été réalisé au sein de tests unitaires afin qu'ils ne soient pas impactés par l'exécution d'une interface utilisateur. Trois images de tailles différentes ont été utilisées :

- La petite image servant à l'illustration des filtres (1235x809),
- Une moyenne dont la taille est inférieure au quart d'une image réelle (6022x6022),
- Et une grande image dont la taille réelle est (14048x14003).

Afin d'être complet, les tests ont également été réalisés avec la librairie OpenCv dans le même contexte. Les résultats sont mis en parallèle afin de pouvoir les comparer. Inutile de préciser que OpenCv sort du lot et affiche des performances bien supérieures. Rappelons que OpenCV est compilé en code natif à l'aide des options d'optimisation de performance.

La méthode de mesure du temps d'exécution de la charge mémoire

```
public void ComputeCpuMemoryUsage(Action action, long length, int iteration = 1,
                                   [CallerMemberName] string caller = null)
{
    //Calcul du temps d'exécution pour n itération
    Stopwatch stopwatch = Stopwatch.StartNew();
    for(int i=0; i<iteration; i++)
        action();
    stopwatch.Stop();
    //Affichage des résultats dans la sortie standard
    Process proc = Process.GetCurrentProcess();
    CultureInfo ci = new CultureInfo("en-US");
    CultureInfo.CurrentCulture = ci;
    Console.WriteLine("Method : {0}", caller);
    Console.WriteLine("Time elapsed={0}", stopwatch.Elapsed);
    //Affichage de la charge mémoire maximum
    Console.WriteLine("Memory usage (bytes)={0}", proc.PeakWorkingSet64);
    Console.WriteLine("Memory usage (MByte)={0}", proc.PeakWorkingSet64/1048576);
    Console.WriteLine("Lenght : {0}", length);
    //Sauvegarde des résultats dans un fichier CSV
    string str = $"{caller};{stopwatch.Elapsed.TotalSeconds};{proc.PeakWorkingSet64};
                  {proc.PeakWorkingSet64 / 1048576};{length};{iteration}";
    var writer = File.AppendText(@".\performance.csv");
    writer.AutoFlush = true;
    writer.WriteLine(str);
    writer.Close();
}
```

Implémentation personnalisée

```
[TestMethod]
public void SmallTestPerso()
{
    ComputeCpuMemoryUsage(() =>
    {
        //Chargement de l'image
        Bitmap v = (Bitmap)Bitmap.FromFile(@".\echantillon.png");
        //Convertion en niveaux de gris
        var res = GrayScaleConverter.ToGray(v, GrayScaleConverter.GrayConvertionMethod.Bt709);
```

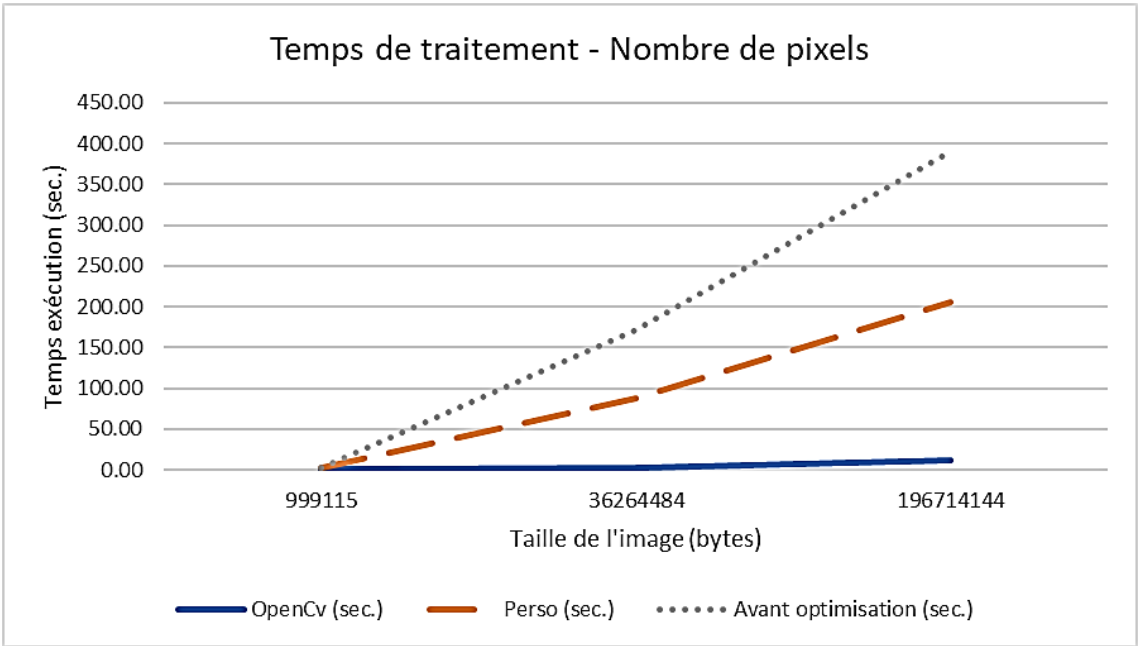
```
        //Application d'un filtre gaussien de 9x9
        var gaus = Convolution.Convolve(res, new GaussianFilter(9, 5));
        //Application d'un filtre de Sobel à 4 orientations
        var resConv = Convolution.Convolve(gaus.Output, new SobelFilter4O());
        //Sauvegarde du résultat
        resConv.Output.Save(@".\SmallTestPerso.png");
    }, 1235*809);
}
```

Implémentation OpenCV

```
[TestMethod]
public void SmallTestOpenCv()
{
    ComputeCpuMemoryUsage(() =>
    {
        //Chargement de l'image en niveaux de gris
        Mat v = Cv2.ImRead(@".\echantillon.png", ImreadModes.AnyDepth);
        Mat gaus = new Mat();
        //Application d'un filtre gaussien de 9x9
        Cv2.GaussianBlur(v, gaus, new OpenCvSharp.Size(9, 9), 0, 0, BorderTypes.Default);
        //Déclaration des matrice
        Mat output1 = new MatOfDouble();
        Mat output2 = new MatOfDouble();
        Mat output3 = new MatOfDouble();
        Mat output4 = new MatOfDouble();
        Mat output12 = new Mat();
        Mat output34 = new Mat();
        Mat output = new Mat();
        //Creation des kernels
        var kernel1 = new float[] {-1, 0, 1, -2, 0, 2, -1, 0, 1};
        var kernel2 = new float[] {-2, -1, 0, -1, 0, 1, 0, 1, 2};
        var kernel3 = new float[] {-1, -2, -1, 0, 0, 0, 1, 2, 1};
        var kernel4 = new float[] {0, -1, -2, 1, 0, -1, 2, 1, 0};
        var k1 = new Mat(3, 3, MatType.CV_32F, kernel1);
        var k2 = new Mat(3, 3, MatType.CV_32F, kernel2);
        var k3 = new Mat(3, 3, MatType.CV_32F, kernel3);
        var k4 = new Mat(3, 3, MatType.CV_32F, kernel4);
        //Application d'un filtre de Sobel à 4 orientations
        Cv2.Filter2D(v, output1, -1, k1);
        Cv2.Filter2D(v, output2, -1, k2);
        Cv2.Filter2D(v, output3, -1, k3);
        Cv2.Filter2D(v, output4, -1, k4);
        //Maximisation des résultats locaux
        Cv2.Max(output1, output2, output12);
        Cv2.Max(output3, output4, output34);
        Cv2.Max(output12, output34, output);
        //Sauvegarde du résultat
        Cv2.ImWrite(@".\SmallTestOpenCv.png", output);
    }, 1235 * 809);
}
```

Les résultats obtenus :

	Elapsed (secondes)			Memory (Mb)			
	OpenCv	Perso	Avant Optim.	OpenCv	Perso	Pixel count	Itéra.
1 SmallTest	0.29	2.01	3.10	101	84	999115	1
5 MediumTest	2.56	87.63	171.55	1339	1131	36264484	1
6 LargeTest	11.38	205.38	389.15	6164	5789	196714144	1



Les courbes, nous montrent un temps d'exécution linéaire en fonction du nombre de pixels. De haut en bas, le graphique comporte trois courbes :

- Une courbe grise en pointillés qui représente l'implémentation personnelle, sans optimisation,
- Une courbe orange discontinue qui représente l'implémentation personnelle ayant fait l'objet d'optimisation basique,
- Une courbe bleue qui représente l'utilisation de la librairie OpenCv.

Le résultat est sans appel, la librairie OpenCv est plus performante que la librairie personnelle. Le facteur de performance n'est pas linéaire, il varie entre 10 et 30.

Tel qu'évoqué ci-dessus, une phase d'optimisation a été réalisée sur la librairie personnelle. Le test sur une image large ayant pris plus de sept minutes, il était nécessaire de se pencher sur les raisons d'un tel temps d'exécution. Par chance, Visual Studio fournit une analyse détaillée (Figure 167 : Fenêtre D'analyse Des Performances De Visual Studio) de la pile d'appels et du temps processeur occupé pour chaque ligne de code et il met en évidence les méthodes les plus gourmandes. A l'aide de cette analyse, il est constatable que l'appel aux propriétés (Width, Height, stride, Kernel.Count…) est chronophage. Une seconde constatation a permis d'identifier la fonction Linq.Max() comme source de lenteur. Trois mesures ont été adoptées pour résoudre ces problèmes et diminuer le temps d'exécution :

- Des variables locales ont été introduites pour remplacer l'appel aux propriétés,
- Une implémentation de la fonction Max() pour un vecteur a été réalisée,
- L'option d'optimisation de code a été activée donnant un gain de +-30%.

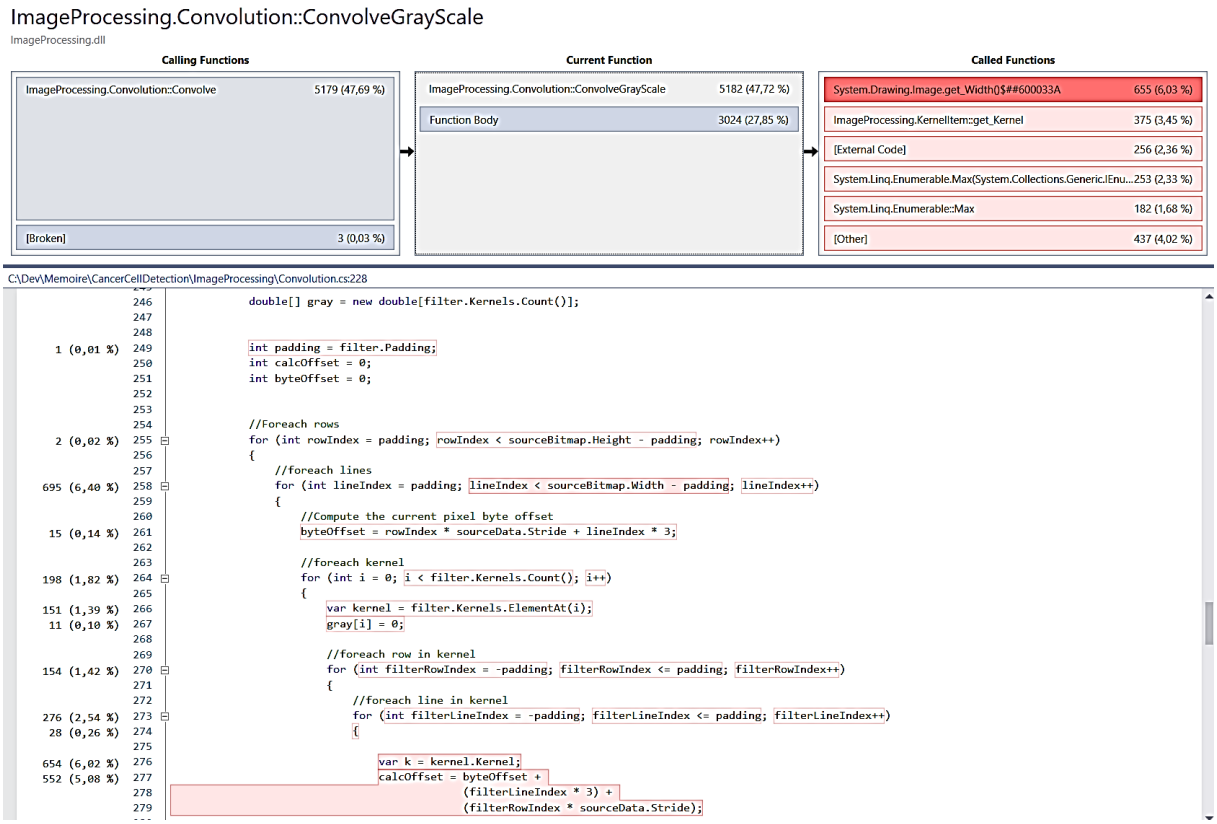
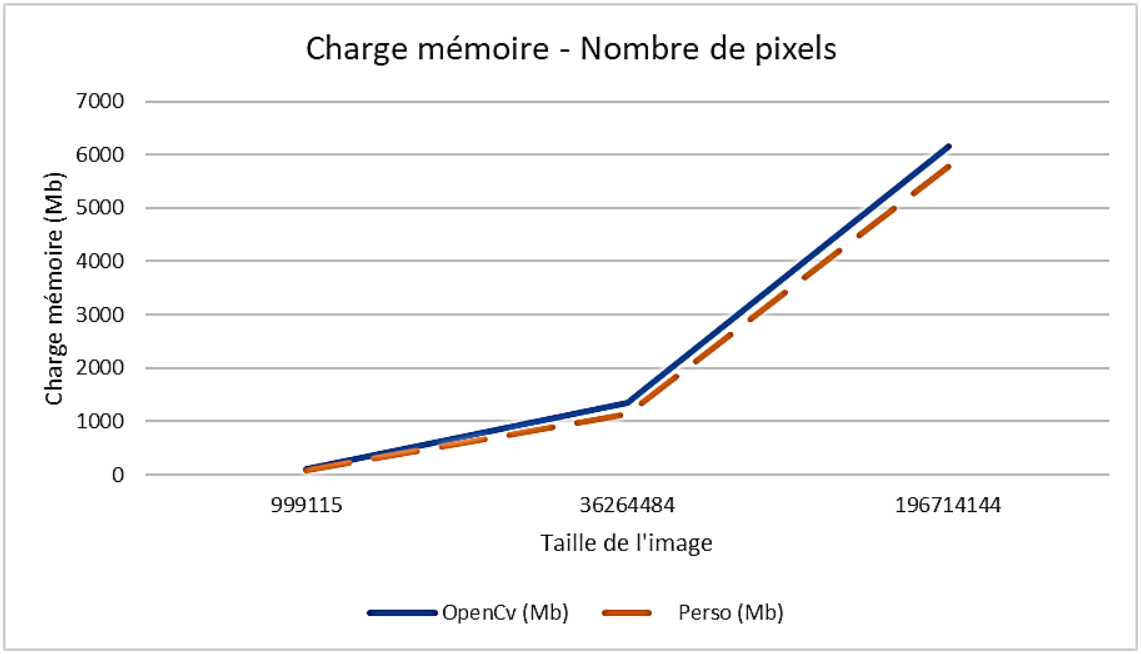


Figure 167 : Fenêtre d'analyse des performances de Visual Studio

En parallèle, la charge de la mémoire a été évaluée pour les trois tests. De la même façon, l'implémentation personnelle est comparée à l'utilisation de la librairie OpenCv. Il y a une différence significative de l'ordre de 6 à 17% en défaveur de OpenCv. La librairie doit être compilée en 64 bits pour pouvoir utiliser plus de 2GO en mémoire.



8.3 La librairie OpenCV

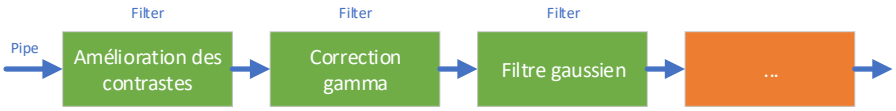
OpenCv est une librairie de traitement graphique écrite en C/C++ optimisée et spécialisée dans le traitement temps réel tirant profit du traitement multicœur. Un « wrapper » C# englobé dans un système de package NuGet permettra d'utiliser les fonctions OpenCV. Le NuGet OpenCvSharp3 sera préféré pour ses prototypes de fonctions identiques aux fonctions C++ de l'API originale.

Installation du package
Install-Package OpenCvSharp3-AnyCPU

8.4 Tests concerts, Pipe and filter

Comme déjà évoqué, l'idée première est de mettre en place un système de flux, comme le modèle de conception architecturale « Pipe and Filter ». Ce modèle linéaire prend en entrée une donnée, notre image, et fait transiter celle-ci au travers d'un ensemble de filtres, nos traitements d'images, afin d'en obtenir un résultat, le dénombrement de cellules. Chaque filtre est indépendant et effectue une tâche unique sur la donnée en entrée avant de la transmettre à un éventuel filtre en sortie.

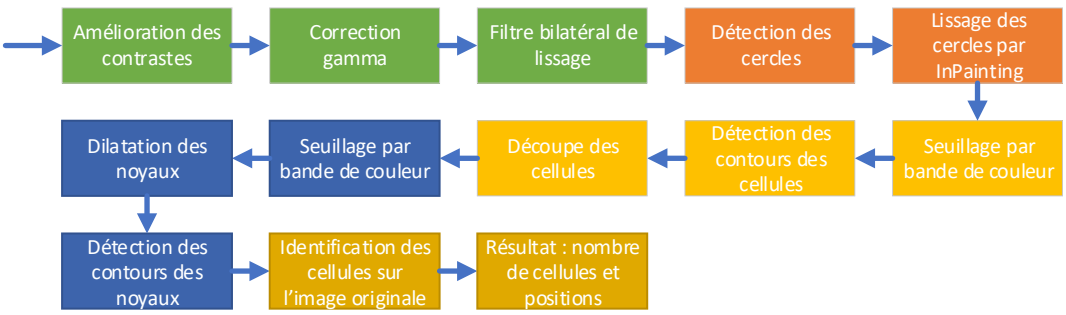
L'image d'origine est transmise à l'entrée d'un filtre, ce filtre effectue un traitement numérique sur l'image avant de transmettre le résultat au filtre suivant. Le résultat final est le nombre de cellules et leur positionnement.



Bien que nous ayons déjà réalisé quelques tests d'enchainement, il est maintenant temps d'en effectuer de nouveaux plus complets.

Les tests suivants peuvent paraître succincts, mais ils résultent de longues et laborieuses adaptations de paramètres. Ne disposant pas d'une interface utilisateur, l'adaptation s'est faite au sein de tests unitaires. L'adaptation des paramètres est manuelle et a été réalisée avec une image dont la boîte de Pétri a été supprimée. La taille originale est diminuée de 35%.

1er test : Un workflow complet



Le premier test est basé sur ce workflow contient cinq phases. La première est une phase

d'amélioration de l'image, la seconde phase tente de supprimer les pores par un filtre de InPainting, la troisième phase identifie les cellules avec leur cytoplasme et leur noyau, la quatrième identifie les noyaux au sein du premier seuillage et enfin la dernière phase identifie les cellules sur l'image d'origine. La troisième phase d'identification découpe les cellules et les places dans un répertoire particulier. Seuls les contours dont la distance euclidienne entre le coin supérieur gauche et le coin inférieur droit est supérieur à 25 sont conservés. Les cellules ainsi découpées alimenteront le réseau de neurones à convolution. Pour bien faire, les images de cellules devrons être classifiées.

L'image d'origine, boîte de Pétri incluse, compte 2750 cellules. Le décompte de cellules après le premier seuillage est de 1465. Ceci s'explique d'une part par une perte de certaines cellules lors de la phase de détection de cercle et de InPainting et d'autre part par l'agglomération de plusieurs cellules très proches ou en cours de division. La seconde phase de seuillage identifie les cellules, l'identification des noyaux est plus périlleuse, les contours des noyaux sont souvent discontinus, un noyau peut être identifié par plusieurs contours. Ceci explique que 5523 contours sont dénombrés. Les contours sont colorés, la couleur est choisie parmi cinq couleurs rendant possible la visualisation des différents contours identifiés.

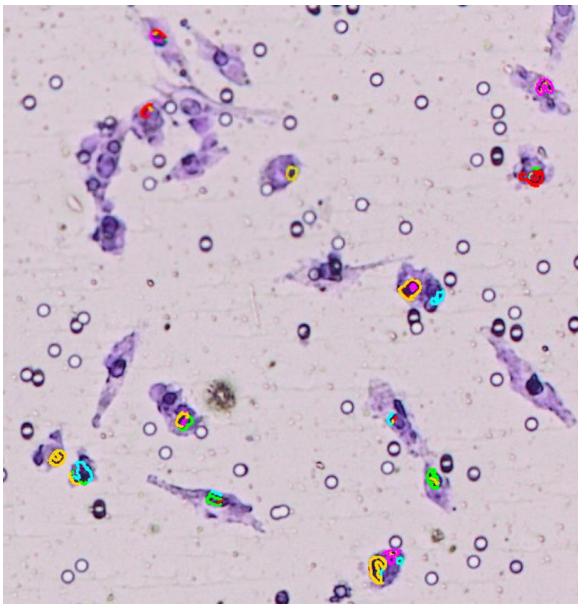


Figure 168 : Identification des noyaux

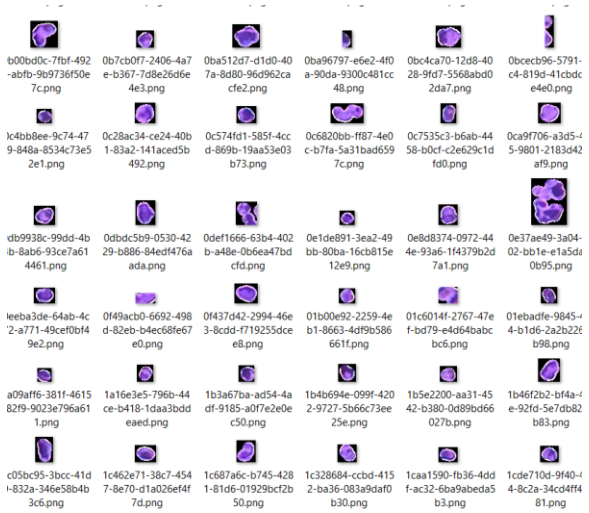
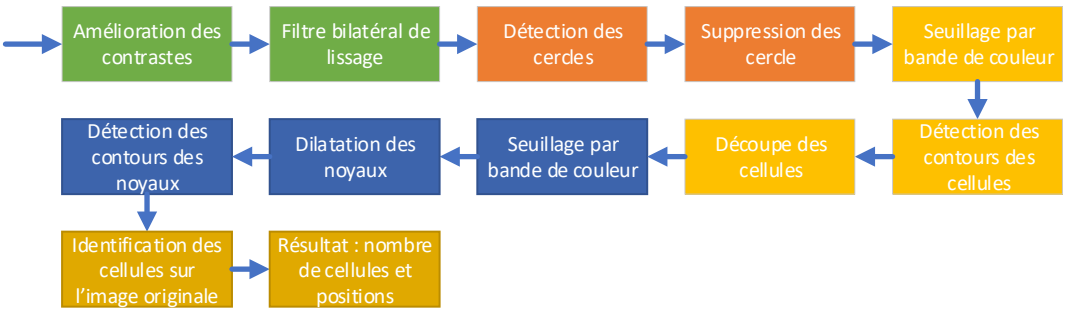


Figure 169 : Echantillon de découpe de cellules

2ème test : Sans l'InPainting



Le temps d'exécution de la méthode de InPainting est pénalisant, le flux s'exécute en près de 4 minutes. Le second test diffère du premier par l'adaptation de quelques paramètres. La correction de contraste est augmentée de 30%, la correction gamma est supprimée, la méthode InPainting est remplacée par l'impression d'un cercle blanc sur les pores. La couleur blanche étant en dehors des seuils de couleurs, les pores seront exclus lors du seuillage.

Outre l'avantage de rapidité, l'application d'une pastille blanche à la place de la méthode de InPainting ne produit pas un flou coloré qui pourrait être confondu avec un noyau. A contrario, l'application d'une pastille proche d'un noyau pourrait partiellement ou totalement la recouvrir.

Les résultats de ce second test sont légèrement meilleurs. Plusieurs contours identifient le même noyau, la découpe des cellules semble être un peu moins bonne, mais c'est très difficile à dire.

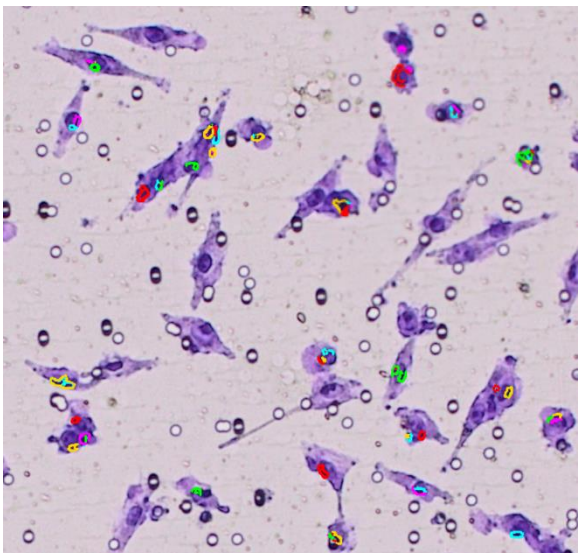


Figure 170 : Identification des noyaux

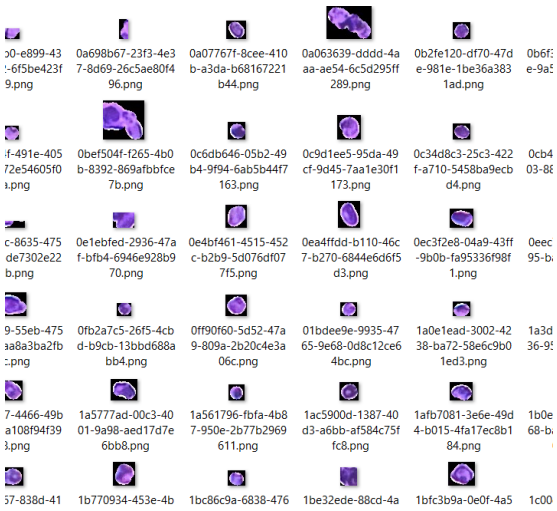
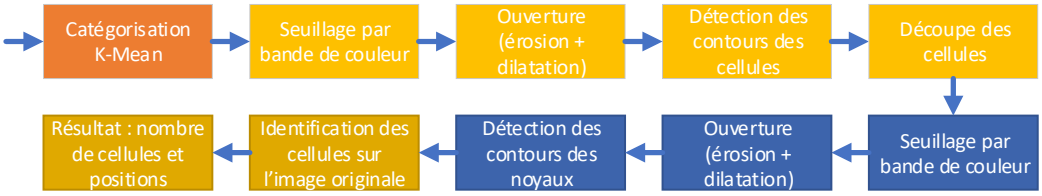


Figure 171 : Echantillon de découpe de cellules

3ème test : K-mean



Durant ce dernier test, il a été remarqué que la méthode de détection et de lissage des cercles était trop destructrice de noyaux. En effet, lors des tests préliminaires sur un échantillon, le résultat était concluant mais sur l'image finale, les noyaux ont presque la même taille que les pores. Pour ce test, les pores ne seront pas supprimés.

Le filtre k-mean est pénalisant, la durée du workflow est de plus de 10 minutes, ce qui est un peu long pour un processus d'essais erreur.

Bien que visuellement ça semble une bonne idée, au final le résultat est plutôt mitigé. La découpe des cellules englobe énormément de pores, de défaut et de partie de cellules. Plus de 4133 cellules ont été isolée alors qu'il en existe seulement la moitié. De nombreux pores sont détectés comme étant des noyaux.

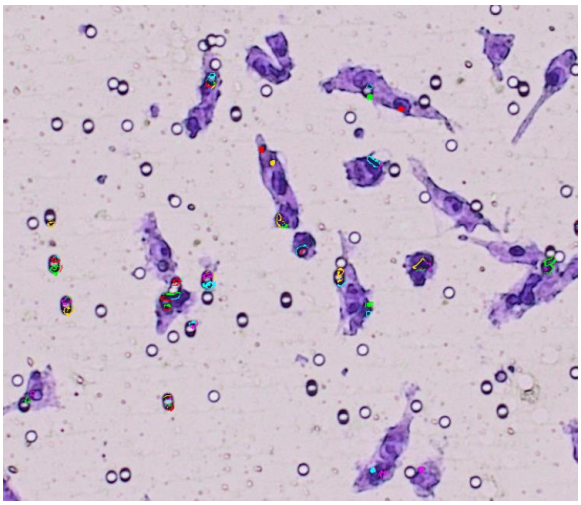


Figure 172 : Identification des noyaux

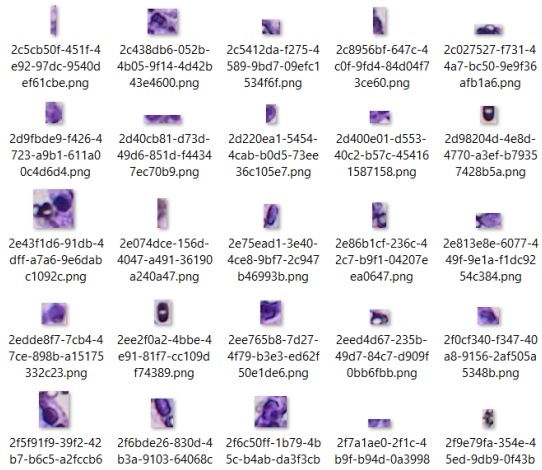


Figure 173 : Echantillon de découpe de cellules

8.5 Conclusion

L'implémentation de fonctions de traitement d'image doit se faire avec beaucoup de précaution, il ne s'agit pas simplement de réaliser une tâche donnée mais de la réaliser dans un temps donné. Nous avons pu le constater dans les tests de performances mais aussi dans les tests grandeurs nature. Traiter une image en plus de 10 minutes peut être acceptable si personne n'attend le résultat mais dans une phase de mise au point, c'est un peu pénible. Certaines fonctions peuvent cependant être conservées, voire réimplémentées dans un autre langage. En revanche, la majeure partie n'aura servi qu'à appréhender et maitriser les concepts sous-jacents.

En ce qui concerne les tests concrets, le manque d'une interface de paramétrage, de réalisation d'un workflow et de visualisation partielle de résultat a fait cruellement défaut. Nous ne sommes pas sûr que le résultat aurait été meilleur mais il aurait été obtenu plus rapidement.

Certaines améliorations peuvent être apportées au traitement des noyaux. La première consisterait à agglomérer les contours très proches, la seconde consisterait à exclure les contours les plus petits, la suivante exclurait les contours de noyaux non inclus dans un contour de cellules et la dernière exclurait les cellules n'ayant pas de noyaux. Une autre piste d'amélioration consisterait à trouver dans les images découpées des cellules les régions pouvant être des noyaux, mais ça c'est une autre histoire.

Dans l'état actuel des choses et compte tenu de la mauvaise qualité des images, il est difficile d'imaginer détecter facilement les noyaux de manière significative. Peut-être les algorithmes de deep learning sont-ils capables de faire mieux.

Chapitre 9
Introduction aux data sciences

Difficile de parler de traitement d'image numérique et de détection d'objets sans évoquer les sciences des données dont deux des domaines qui sont le machine learning et le deep learning. Comme indiqué dans le titre, il ne s'agira ici que d'une introduction.

Nous allons cibler un sujet au sein de ces domaines. Il traitera de la régression linéaire simple et la régression linéaire multiple. Nous tenterons d'établir une corrélation entre la couleur entourant les cellules, la couleur des noyaux et le nombre de cellules présentes sur l'image. Nous utiliserons des filtres de lissage, de seuillage, k-mean et conversion en niveaux de gris pour dénombrer les pixels de la couleur qui nous intéresse.

Nous utiliserons différents outils lors de nos expérimentations. Tout d'abord Excel et Gretl (Gnu Regression, Econometrics and Time-series Library) nous servirons pour valider l'hypothèse qu'il existe une linéarité entre le nombre de pixels et le nombre de cellules.

Nous n'aborderons pas ici les réseaux de neurones à convolution, ce sujet bien que passionnant pourrait à lui seule remplir un mémoire. Nous ne désespérons pas de pouvoir un jour le mettre en œuvre mais pour le moment la place et le temps nous manque.

9.1 Régression linéaire simple et régression linéaire multiple

Régression : Détermination de la grandeur approximative d'un phénomène (d'une variable), correspondant à la grandeur certaine d'un autre phénomène (d'une ou plusieurs variables).[4]

9.1.1 Régression linéaire

Concernant notre sujet, une question se pose : existe-t-il une relation entre le nombre de pixels d'une gamme de couleur et le nombre de cellules comprises dans une image ? Si la relation est linéaire, alors la régression linéaire va permettre de répondre à cette question.

La régression linéaire établit une relation de linéarité entre une variable *X* et une fonction appliquée à une variable *Y*.

La variable *X* est la variable mesurée, la variable explicative, soit le nombre de pixels de couleur. Elle sera communément appelée la variable indépendante. La variable X doit être correcte, elle ne doit pas contenir d'erreur ou si elle en contient, l'erreur doit être négligeable.

La variable *Y* représente la variable pour laquelle il est tenté d'établir un lien de linéarité, la

[4] Définition du Larousse dont les termes variables ont été ajoutés

variable à expliquer, soit le nombre de cellules dans l'image. La variable *Y* est la variable qui doit être prédite. Cette variable est appelée variable dépendante. Cette variable comporte une erreur, c'est-à-dire que *Y* est une vraie valeur plus une erreur. Cette erreur ne doit porter que sur *Y* et ne doit pas avoir d'influence sur *X*.

La relation linéaire entre *X* et *Y* est moyenne, c'est-à-dire que la variance sur l'erreur ε de *Y* est constante et permet de définir la relation de linéarité. La dispersion se fait autour de la droite de régression et non sur la droite, ce qui explique que pour un point connu, il est peu probable que la prédiction soit identique à la valeur réelle.

La relation fonctionnelle peut être écrite par la fonction affine suivante : $f(x) = \alpha X + \beta$ ou α est la pente de la courbe, le coefficient directeur et β est l'ordonnée à l'origine. Il est également possible de noter $Y = \alpha X + \beta + \varepsilon$.

9.1.1 Régression linéaire multiple

La régression linéaire multiple établit une relation entre un ensemble de variables X appelé régresseurs et la variable dépendante *Y*. Grâce à la régression linéaire multiple, il est toujours question de réaliser une régression et de prédire le nombre de cellules dans une image en se basant sur des composantes de couleurs. La régression précédente ne tient compte que d'une seule gamme de couleur reprenant l'ensemble d'une cellule.

La régression linéaire multiple peut compter plusieurs régresseurs, par exemple, un premier comptant les pixels mauve clair entourant les cellules et un second régresseur comptant les pixels mauve foncé des noyaux. Il est imaginable d'ajouter à notre modèle d'autres régresseurs qui pourraient définir le type de cellules, le produit de contraste utilisé ou tout autre donnée permettant d'affiner le modèle.

Pour faciliter la compréhension de la fonction de la régression linéaire multiple, la fonction de régression linéaire simple peut être transformée de la sorte : $Y = \beta_0 + \beta_1 X_1$.

La fonction linéaire multiple est similaire si ce n'est qu'elle possède plusieurs variables indépendantes ayant chacune un coefficient directeur : $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n$.

Il existe plusieurs méthodes pour réaliser un modèle de régression linéaire multiple. La méthode utilisée dans ce chapitre est la méthode Backward Elimination. Il s'agit de la méthode la plus facile à mettre en œuvre et elle donne de très bons résultats. Cette méthode se base sur le calcul et l'interprétation de la variable p-value, qui indique le niveau de corrélation de la variable indépendante sur la variable dépendante. Les variables indépendantes sont toutes ajoutées au modèle et ensuite supprimées en fonction de leur p-value.

En détail, la mise en œuvre se déroule en 5 étapes :

Etape 1 : Etablir un seuil de significativité de la p-value qui restera constant dans l'ensemble du modèle. Généralement, elle est définie à 5% soit SS = 0.05

Etape 2 : Ajouter au modèle toutes les variables indépendantes disponibles dans le jeu de données

Etape 3 : Après avoir calculé le modèle, les p-values de chaque régresseur ont été calculées. Sélectionner le régresseur pour lequel la p-value est supérieur au seuil de significativité, soit : $p\_value > SS$. S'il n'en existe pas, le modèle est prêt, sinon se rendre à l'étape 4

Etape 4 : Supprimer la variable indépendante sélectionnée du modèle

Etape 5 : Relancer le modèle et recommencer l'étape 3

9.1.2 Dispersion graphique

Avant toute chose, il est indispensable de vérifier la dispersion de la population afin de déterminer s'il existe une relation de linéarité entre les variables de *X* et de *Y*. Les données sont affichées dans un graphique nuage de point. Grâce à ça, il est facile de voir le nombre de populations et la variance de *Y*, c'est à dire si *Y* évolue de façon constante avec *X*.

Ci-dessous, un exemple de dispersion du nombre de pixels par rapport au nombre des cellules. Les pixels retenus sont différents du noir. Le premier filtre isole les couleurs bleu et rouge, le second réalise un filtre par bande de couleur et le dernier est un filtre par seuillage par zéro sur une image en niveaux de gris. Ce graphe est obtenu grâce à Excel. Les données sont obtenues à partir du tableau en annexe 1.

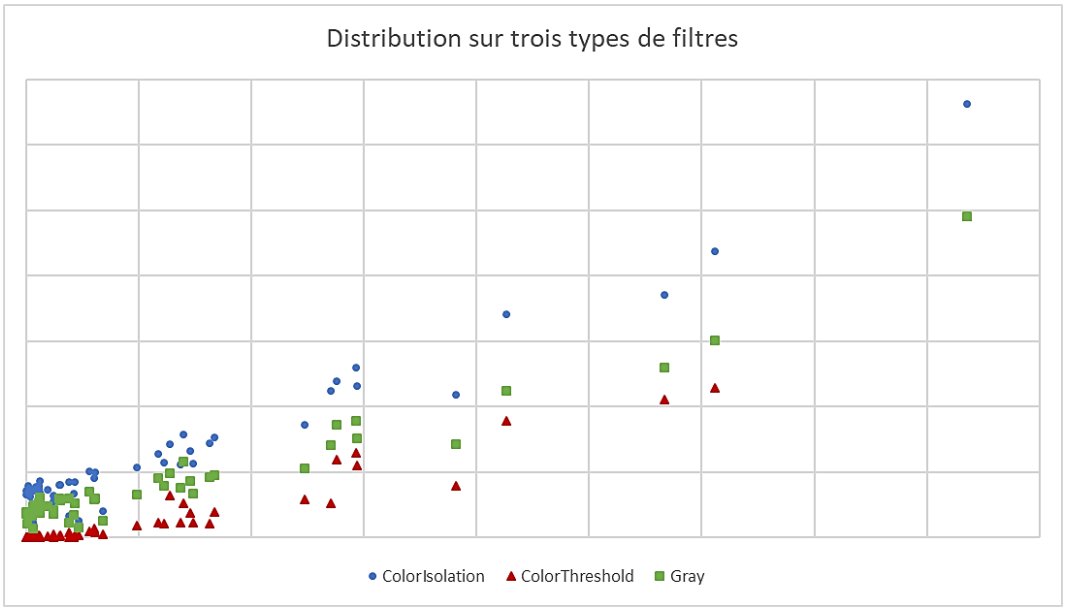


Figure 174 : Distribution du nombre de pixels / de cellules (isolation de couleur, bande de couleur et niveaux de gris)

9.1.1 La méthode des moindres carrés ordinaires

La droite de régression linéaire, qui est la droite qui se rapproche le plus possible des points, peut être obtenue par la méthode des moindres carrés. La méthode des moindres carrés cherche à minimiser la somme des différences entre les points observés *Y* et leurs prédictions *Y^* au carré.

$$sum(y - y\hat{})^2 \rightarrow min$$

Le minimum est obtenu en évaluant toutes les droites de prédiction et en calculant pour chacune d'elles la somme des différences au carré. La droite de régression est celle pour laquelle cette somme est minimale.

Ci-dessous une représentation des droites de régression linéaire simple calculée par Excel.

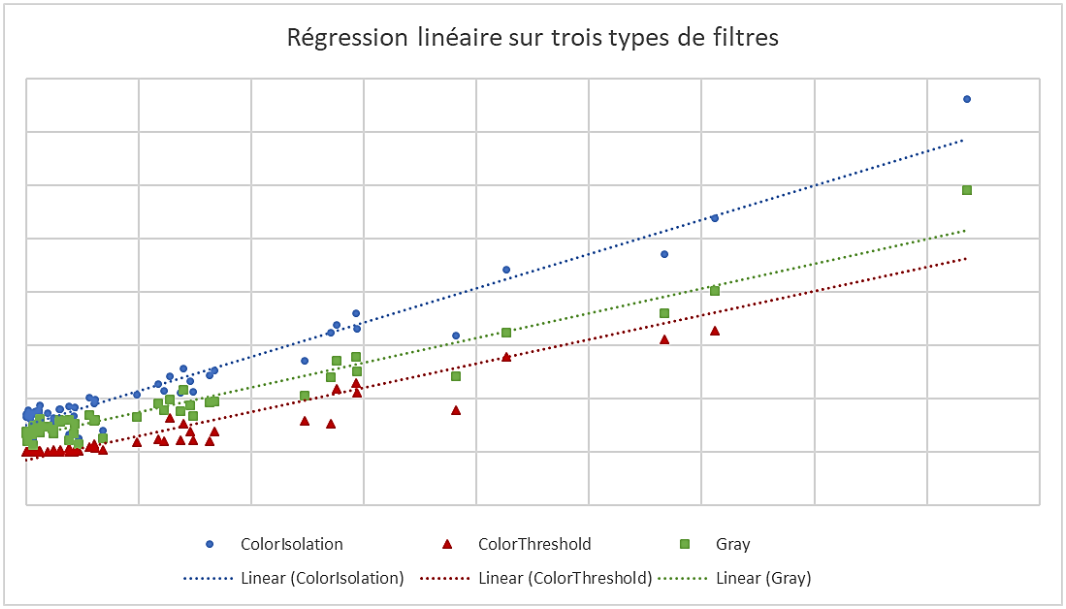
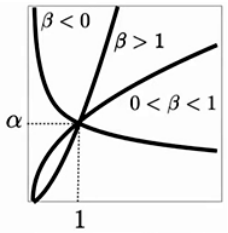


Figure 175 : Linéarisation des trois distributions

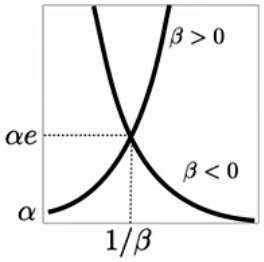
9.1.1 Linéarisation des données

Dans le cas où la variance de *Y* ne progresse pas de manière constante, cela signifie que le nuage de point n'est pas linéarisable. Il est cependant possible d'appliquer à X ou à Y une transformation permettant d'établir une linéarité.

Voici quelques transformations[5] à appliquer si le nuage de points a l'une des tendances suivantes :

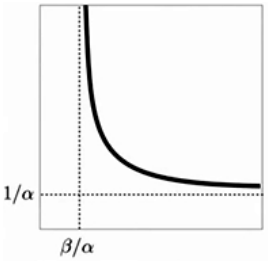


La tendance du nuage : $y = \alpha x^{\beta}$
La transformation des variables :
$y' = \log(y)$
$x' = \log(x)$



La tendance du nuage : $y = \alpha e^{\beta x}$
La transformation des variables :
$y' = \log(y)$

[5] Cours en ligne de statistique, Thierry Verdel, Université de Lorraine Mines Nancy

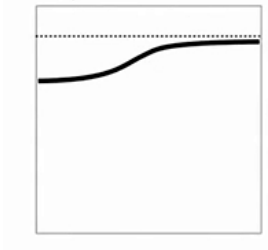


La tendance du nuage : $y = \frac{x}{\alpha x - \beta}$

La transformation des variables :

$y' = 1/y$

$x' = 1/x$



La tendance du nuage : $y = \frac{e^{\alpha x+\beta}}{1+e^{\alpha x+\beta}}$

La transformation des variables :

$y' = \log(\frac{y}{1-y})$

9.1.1 Filtrage des données

La partie de préparation des données est un point important dans la réalisation d'une régression linéaire. En effet, il faut pouvoir exclure des résultats qui sembleraient trop éloignés de la dispersion moyenne. Si plusieurs populations semblent apparaître, il serait peut-être bon de les séparer, ou de trouver une propriété permettant de les catégoriser et donc de créer un nouveau régresseur.

9.2 Dénombrement par régression linéaire simple ou multiple

Le dénombrement par régression va s'effectuer en plusieurs étapes :

- La première consiste à obtenir le nombre des cellules à partir des métadonnées des images ;
- Ensuite, obtenir le nombre des pixels de couleur ;
- Jointer les différentes données de quantification ;
- Diviser l'ensemble des données en deux, un ensemble d'entrainement et un ensemble de test ;
- Entrainer le modèle à l'aide des données d'entrainement ;
- Tester le modèle en réalisant des prédictions avec l'ensemble des données de test ;
- Recommencer au second point si le résultat n'est pas satisfaisant.

Après quelques tests, il a été évident que l'image d'origine contenant la boîte de Pétri est trop lourde en mémoire et comporte trop de pixels de couleur pouvant être confondus avec les cellules. Le choix a été fait de supprimer la boîte de Pétri et de ne conserver que la zone. Les données ont été classées dans un tableau repris à l'annexe 14.2. Ces données ont été obtenue par les méthodes de l'annexe 14.3.

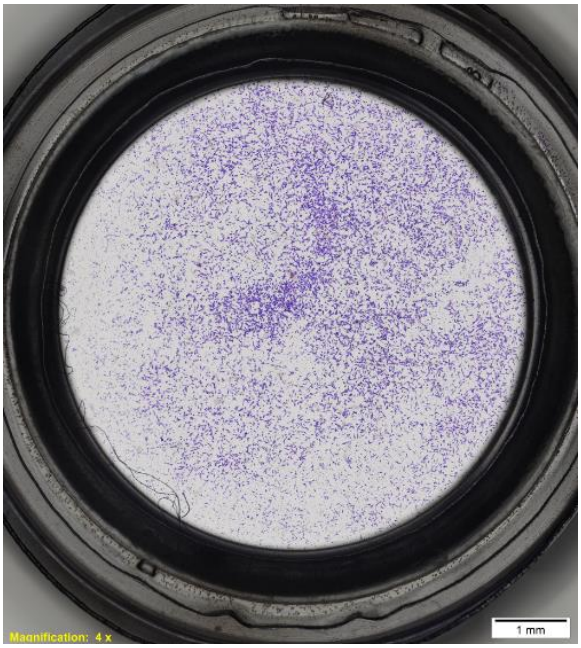


Figure 176 : Image originale

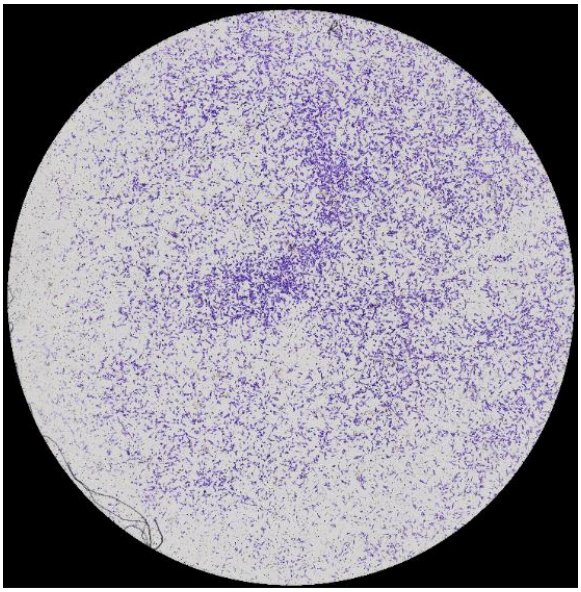


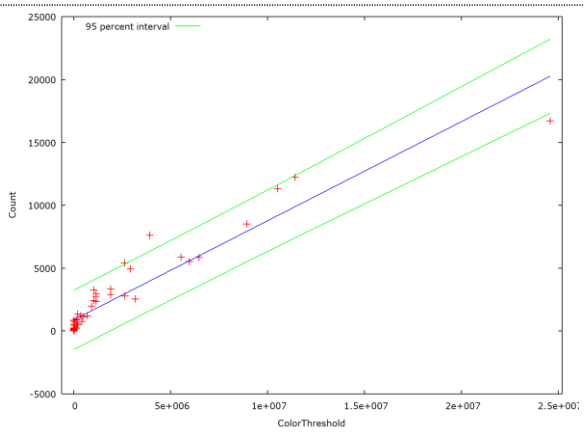
Figure 177 : Image nettoyée

9.2.1 Régression simple par bande de couleur

Le premier essai de régression linéaire consiste à compter le nombre de pixels contenus dans une image filtrée par gamme de couleur, le mauve. Les filtres suivants seront appliqués à l'image d'origine :

- Un filtre gaussien pour lisser l'image d'origine et la rendre homogène ;
- Un seuillage de couleur permettant d'obtenir une image ne contenant que les pixels compris dans la gamme de couleur, les autres pixels étant forcés à 0 (le noir) ;
- Le nombre des pixels différents de noir est calculé.

L'outil Gretl fournit trois sorties intéressantes. Tout d'abord, un graphe de dispersion contenant la droite de régression en bleu, deux droites en vert modélisant l'intervalle de confiance à 95 % de la prédiction et enfin un tableau de valeurs mettant en relation les valeurs mesurées et les valeurs prédites. Ensuite, un tableau récapitulatif des coefficients et mesures sorties du modèle. Et enfin, un tableau des valeurs brutes mesurées et prédites




```
Model 2: OLS, using observations 1-50
Dependent variable: Count

                  coefficient    std. error    t-ratio   p-value
  ---------------------------------------------------------------
  const           885.151        182.843         4.841   1.38e-05  ***
  ColorThreshold    0.000788936    3.90695e-05  20.19    4.19e-025 ***

Mean dependent var   2501.600   S.D. dependent var   3545.104
Sum squared resid    64856932   S.E. of regression   1162.405
R-squared            0.894682   Adjusted R-squared   0.892488
F(1, 48)             407.7628   P-value(F)           4.19e-25
Log-likelihood      -422.8387   Akaike criterion     849.6774
Schwarz criterion    853.5015   Hannan-Quinn         851.1336
```

	Count	prediction	std. error	95% interval
1	2750.00	1791.44	1174.498	-570.04 - 4152.93
2	2910.00	2384.85	1173.986	24.39 - 4745.30
3	75.00	889.96	1176.682	-1475.91 - 3255.84
4	11331.00	9180.89	1219.680	6728.56 - 11633.22
5	8524.00	7935.26	1204.415	5513.62 - 10356.90
6	1128.00	1269.94	1175.555	-1093.67 - 3633.56
7	596.00	929.89	1176.549	-1435.72 - 3295.50
8	2442.00	1699.37	1174.644	-662.41 - 4061.15
9	3262.00	1707.64	1174.630	-654.11 - 4069.40
10	1973.00	1615.02	1174.793	-747.06 - 3977.09
11	137.00	899.68	1176.649	-1466.13 - 3265.49
12	1225.00	1175.66	1175.807	-1188.46 - 3539.78
13	5853.00	5979.21	1186.536	3593.52 - 8364.90
14	389.00	931.61	1176.544	-1433.99 - 3297.21
15	167.00	888.37	1176.687	-1477.51 - 3254.26
16	249.00	968.62	1176.424	-1396.74 - 3333.98
17	1209.00	1448.58	1175.129	-914.18 - 3811.33

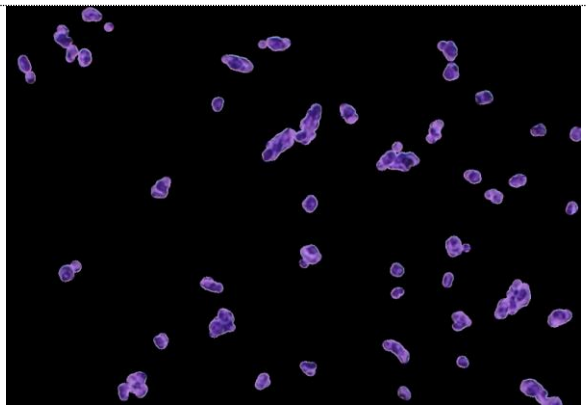
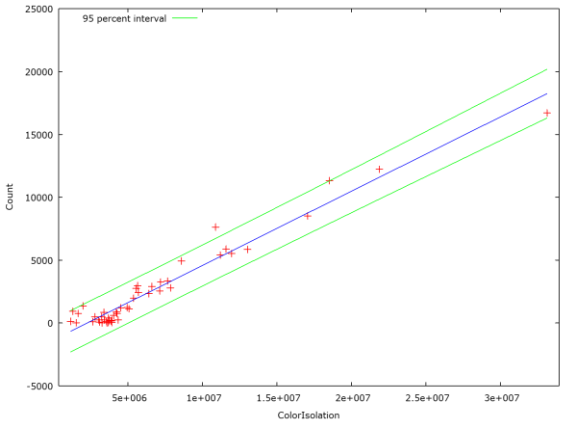


Figure 178 : Sortie de Gretl et échantillon de régression par bande de couleur

Sur le graphique, nous pouvons observer que quelques-unes des valeurs sortent de l'intervalle à 95% et il existe une forte concentration de valeurs dans le bas de la droite. La p-value est très faible, 4.19^{e}-25, ce qui indique une forte corrélation. Nous savons également que le coefficient ColorThreshold indique que pour chaque pixel de couleur, le nombre de cellules augmente de 0.00079. Une petite règle de trois nous permet de dire qu'une cellule occupe en moyenne 1265 pixels. Dans les faits, ce n'est pas tout à fait vrai puisque certains pores, les taches et autres défauts, n'ont pas été supprimés.

9.2.2 Régression simple par isolation de couleur

Ce second essai de régression linéaire se base sur l'isolation des composantes principales du mauve, le rouge et le bleu. Même principe que précédemment, lissage gaussien, la couleur de fond est noire après le filtre d'isolation du rouge et du bleu et les pixels restants sont considérés comme appartenant à des cellules mais les pores n'ont pas été supprimés.



Model 1: OLS, using observations 1-50
Dependent variable: Count

	coefficient	std. error	t-ratio	p-value
const	-1335.78	170.384	-7.840	3.85e-010 ***
ColorIsolation	0.000591145	1.95936e-05	30.17	7.31e-033 ***

Mean dependent var	2501.600	S.D. dependent var	3545.104
Sum squared resid	30847456	S.E. of regression	801.6579
R-squared	0.949908	Adjusted R-squared	0.948865
F(1, 48)	910.2437	P-value(F)	7.31e-33
Log-likelihood	-404.2605	Akaike criterion	812.5209
Schwarz criterion	816.3450	Hannan-Quinn	813.9772

30	3344.00	3193.71	809.960	1565.18 -	4822.25
31	852.00	667.23	811.915	-965.24 -	2299.69
32	2796.00	3303.84	810.071	1675.08 -	4932.60
33	127.00	766.93	811.674	-865.05 -	2398.91
34	4940.00	3741.59	810.677	2111.61 -	5371.56
35	863.00	1162.48	810.850	-467.84 -	2792.81
36	35.00	838.32	811.510	-793.33 -	2469.97
37	3.00	594.70	812.098	-1038.14 -	2227.53
38	40.00	980.58	811.203	-650.45 -	2611.62
39	232.00	978.37	811.207	-652.67 -	2609.41
40	9.00	785.65	811.630	-846.24 -	2417.54
41	128.00	-652.30	816.356	-2293.69 -	989.09
42	929.00	-570.60	816.013	-2211.30 -	1070.11
43	25.00	-437.26	815.473	-2076.87 -	1202.36
44	758.00	-350.36	815.134	-1989.30 -	1288.58
45	1363.00	-160.66	814.429	-1798.18 -	1476.86
46	2347.00	2448.88	809.637	820.99 -	4076.76
47	5524.00	5735.71	816.700	4093.62 -	7377.79
48	12232.00	11596.87	863.938	9859.80 -	13333.93
49	5405.00	5281.17	814.860	3642.78 -	6919.55
50	16708.00	18248.55	963.289	16311.73 -	20185.38

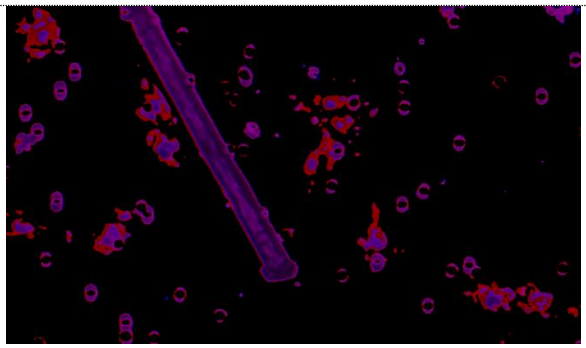


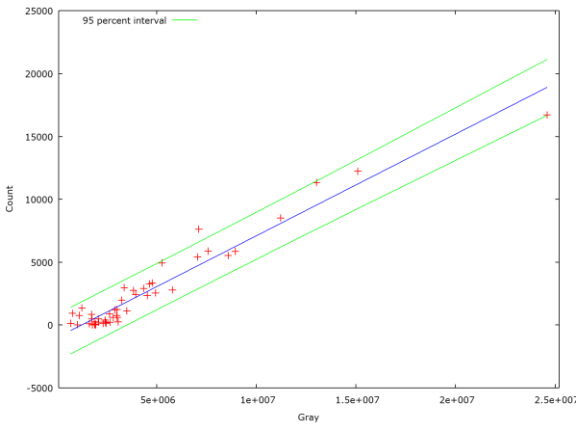
Figure 179 : Sortie de Gretl et échantillon de régression par isolation de couleur

Graphiquement, la population est mieux dispersée, un seul point est en dehors de l'intervalle des 95%. La p-value à 7.31^{e}-33 est inférieure au test précédent, ce qui indique une forte corrélation. Le coefficient de ColorIsolation indique que la superficie d'une cellule est plus étendue que dans le cas précédent. Cela s'explique par le fait que 100% des pores sont visibles sur l'image de sortie et donc comptabilisés comme faisant partie des pixels des cellules. D'un point de vue prédiction, l'écart type est inférieur au précédent mais il indique des prédictions négatives pour les échantillons de 41 à 45. Dans ce type de filtrage, le cytoplasme en rouge se distingue du noyaux mauve foncé.

Une tentative a été effectuée pour détecter et supprimer les pores à l'aide des méthodes de Hough et de inpainting, mais les temps de calculs était considérables. Après plus de 25 minutes la première image n'était toujours pas traitée. Ce test a été avorté.

9.2.3 Régression simple en niveaux de gris et seuillage par zéro

Pour ce dernier essai qui est sensiblement identique au précédent, les filtres appliqués sont différents mais l'idée sous-jacente est la même, ne retenir que des pixels significatifs. Les campagnes de tests sont espacées dans le temps et ne portent pas sur des composants identiques. Ce fait porte à penser que la gamme de couleur peut différer d'une campagne à l'autre. En ramenant le tout à 256 niveaux de gris, les intensités seront plus proches et plus susceptibles d'être identiques. Le seuillage par zéro va convertir les blancs faisant partie de l'image d'origine en noir de fond. Les seuils ont été fixés arbitrairement après quelques essais manuels. Tels qu'au point précédent, les pores demeurent sur l'image de sortie.



Model 3: OLS, using observations 1-50
Dependent variable: Count

	coefficient	std. error	t-ratio	p-value	
const	-997.755	184.106	-5.419	1.91e-06	***
Gray	0.000809563	3.05687e-05	26.48	2.69e-030	***

Mean dependent var	2501.600	S.D. dependent var	3545.104
Sum squared resid	39445616	S.E. of regression	906.5228
R-squared	0.935946	Adjusted R-squared	0.934612
F(1, 48)	701.3705	P-value(F)	2.69e-30
Log-likelihood	-410.4072	Akaike criterion	824.8144
Schwarz criterion	828.6384	Hannan-Quinn	826.2706

For 95% confidence intervals, t(48, 0.025) = 2.011

	Count	prediction	std. error	95% interval
1	2750.00	2106.22	915.665	265.15 - 3947.29
2	2910.00	2517.83	915.543	677.01 - 4358.66
3	75.00	583.38	918.404	-1263.20 - 2429.95
4	11331.00	9538.89	953.325	7622.10 - 11455.68
5	8524.00	8086.76	939.519	6197.73 - 9975.78
6	1128.00	1842.60	915.881	1.10 - 3684.10
7	596.00	1446.23	916.410	-396.34 - 3288.80
8	2442.00	2198.06	915.615	357.10 - 4039.03
9	3262.00	2767.01	915.598	926.08 - 4607.95
10	1973.00	1629.54	916.135	-212.47 - 3471.56
11	137.00	987.24	917.327	-857.17 - 2831.65
12	1225.00	1445.51	916.411	-397.06 - 3288.08
13	5853.00	6246.87	926.401	4384.21 - 8109.52
14	389.00	949.92	917.416	-894.67 - 2794.50
15	167.00	998.15	917.302	-846.21 - 2842.51
16	249.00	1487.01	916.344	-355.42 - 3329.44
17	1209.00	1346.68	916.581	-496.23 - 3189.59

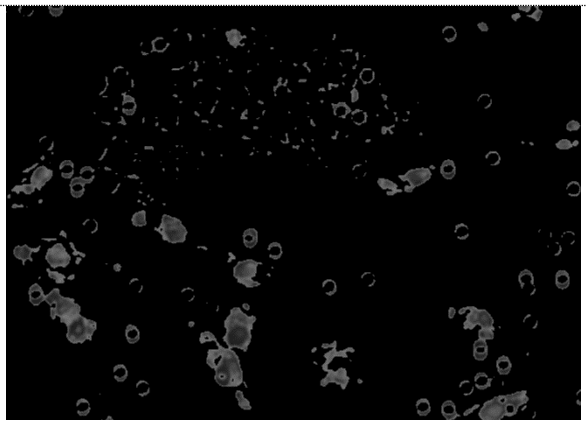


Figure 180 : Sortie de Gretl et échantillon de régression par seuillage par zéro en niveaux de gris

Belle dispersion, nous constatons ici aussi un point en dehors de l'intervalle à 95%. Peut-être faudrait-il le repérer et l'exclure du jeu de test ? Très bonne p-value, elle se situe entre les deux p-value précédentes. Les prédictions sont parfois très bonnes comme très mauvaises, ici aussi le résultat peut être négatif.

9.2.4 Modèle et prédiction

Les modèles sont identiques, il s'agit de mettre en relation un nombre de cellules et un nombre de pixels, qu'il soit coloré ou gris n'impacte en rien le modèle. Il existe une seule restriction à l'obtention d'un résultat cohérent, les images pour lesquelles il est réalisé une prédiction doivent être prétraitées de la même façon que les images ayant servi à nourrir le modèle.

Comme nous avons pu le constater, les prédictions peuvent être bonnes et mauvaises, quel que soit le format de l'image. Peut-être faudrait-il pousser plus loin les tests en associant le masque obtenu par le filtre passe bande aux deux autres afin d'en isoler les pores ? Nous avons un échantillon de 50 images, quel serait le résultat pour un échantillon dix fois plus grand ? Et si maintenant nous essayons d'établir une corrélation entre les différents dénombrements de pixels et le nombre de cellules ? Belle façon d'amener la régression linéaire multiple…

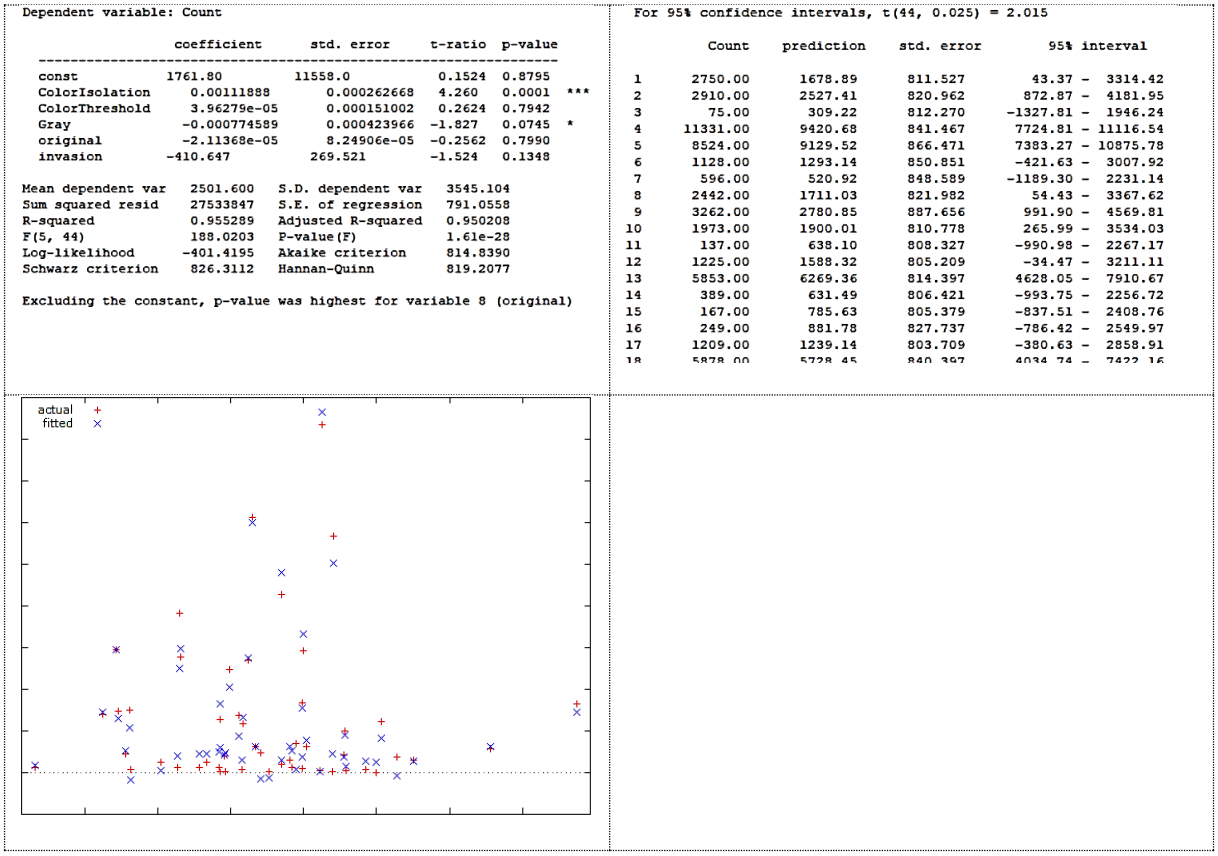
9.2.5 Régression linéaire multiple

Le modèle est réalisé de manière itérative à l'aide de la méthode Backward Elimination. Comme pour la régression linéaire simple, Gretl est utilisé pour créer le modèle. Les 5 étapes de la mise en œuvre de la méthode sont donc manuelles. Le seuil de significativité est fixé par convention à 0.05. Deux nouvelles variables indépendantes ont été ajoutées au jeu de données : la taille d'origine des images et le type de test effectué (invasion ou migration).

1ere itération :

Tous les régresseurs sont ajoutés au modèle : le nombre de pixels dans l'image filtrée par bande de couleur (ColorThreshold), le nombre de pixels obtenus par le filtre d'isolation de couleur (ColorIsolation), le nombre de pixels gris restants après seuillage par zéro (Gray), le nombre de pixels de l'image d'origine : hauteur X largeur (original) et le type de test effectué

dont les valeurs sont 0 ou 1 (invasion). La variable migration n'est pas ajoutée puisqu'il s'agit de l'inverse de la variable invasion, ce serait équivalent à ajouter deux fois le même régresseur, ce qui n'aurait pas de sens.



Dependent variable: Count

	coefficient	std. error	t-ratio	p-value	
const	1761.80	11558.0	0.1524	0.8795	
ColorIsolation	0.00111888	0.000262668	4.260	0.0001	***
ColorThreshold	3.96279e-05	0.000151002	0.2624	0.7942	
Gray	-0.000774589	0.000423966	-1.827	0.0745	*
original	-2.11368e-05	8.24906e-05	-0.2562	0.7990	
invasion	-410.647	269.521	-1.524	0.1348	

Mean dependent var	2501.600	S.D. dependent var	3545.104
Sum squared resid	27533847	S.E. of regression	791.0558
R-squared	0.955289	Adjusted R-squared	0.950208
F(5, 44)	188.0203	P-value(F)	1.61e-28
Log-likelihood	-401.4195	Akaike criterion	814.8390
Schwarz criterion	826.3112	Hannan-Quinn	819.2077

Excluding the constant, p-value was highest for variable 8 (original)

For 95% confidence intervals, t(44, 0.025) = 2.015

	Count	prediction	std. error	95% interval
1	2750.00	1678.89	811.527	43.37 - 3314.42
2	2910.00	2527.41	820.962	872.87 - 4181.95
3	75.00	309.22	812.270	-1327.81 - 1946.24
4	11331.00	9420.68	841.467	7724.81 - 11116.54
5	8524.00	9129.52	866.471	7383.27 - 10875.78
6	1128.00	1293.14	850.851	-421.63 - 3007.92
7	596.00	520.92	848.589	-1189.30 - 2231.14
8	2442.00	1711.03	821.982	54.43 - 3367.62
9	3262.00	2780.85	887.656	991.90 - 4569.81
10	1973.00	1900.01	810.778	265.99 - 3534.03
11	137.00	638.10	808.327	-990.98 - 2267.17
12	1225.00	1588.32	805.209	-34.47 - 3211.11
13	5853.00	6269.36	814.397	4628.05 - 7910.67
14	389.00	631.49	806.421	-993.75 - 2256.72
15	167.00	785.63	805.379	-837.51 - 2408.76
16	249.00	881.78	827.737	-786.42 - 2549.97
17	1209.00	1239.14	803.709	-380.63 - 2858.91
18	5878.00	5728.45	840.397	4034.74 - 7422.16

Figure 181 : Sortie de Gretl et graphe de dispersion de la variable Original

Cette première itération nous montre que la p-value de la variable « Original » est maximale et dépasse le seuil de significativité de 5%. Elle sera donc supprimée de la seconde itération. Les étoiles indiquent les deux variables les plus significatives, ColorIsolation et Invasion. Les prédictions ne sont pas très bonnes. Sur le graphe de dispersion du nombre de cellules par rapport à la taille originale, on peut effectivement constater qu'il n'y a pas de relation, ils ne suivent ni une droite ni une courbe.

2ème itération :

Le régresseur « Original » est supprimé de la liste des variables de régression.

Dependent variable: Count

	coefficient	std. error	t-ratio	p-value	
const	−1198.26	361.601	−3.314	0.0018	***
ColorIsolation	0.00112311	0.000259412	4.329	8.24e-05	***
ColorThreshold	4.16215e-05	0.000149228	0.2789	0.7816	
Gray	−0.000782678	0.000418377	−1.871	0.0679	*
invasion	−418.397	265.024	−1.579	0.1214	

Mean dependent var	2501.600	S.D. dependent var	3545.104
Sum squared resid	27574933	S.E. of regression	782.8003
R-squared	0.955222	Adjusted R-squared	0.951242
F(4, 45)	239.9919	P-value(F)	1.00e-29
Log-likelihood	−401.4568	Akaike criterion	812.9136
Schwarz criterion	822.4737	Hannan-Quinn	816.5541

Excluding the constant, p-value was highest for variable 4 (ColorThreshold)

For 95% confidence intervals, t(44, 0.025) = 2.015

	Count	prediction	std. error	95% interval
1	2750.00	1714.50	787.925	126.54 - 3302.46
2	2910.00	2547.82	780.724	974.38 - 4121.27
3	75.00	279.96	785.460	−1303.03 - 1862.95
4	11331.00	10044.75	896.798	8237.37 - 11852.13
5	8524.00	9571.86	883.261	7791.76 - 11351.95
6	1128.00	1235.88	797.529	−371.44 - 2843.19
7	596.00	564.28	812.561	−1073.33 - 2201.88
8	2442.00	1625.36	794.161	24.84 - 3225.89
9	3262.00	2895.80	797.848	1287.84 - 4503.76
10	1973.00	1788.51	787.454	201.50 - 3375.52
11	137.00	593.07	784.965	−988.93 - 2175.06
12	1225.00	1561.34	781.779	−14.23 - 3136.91
13	5853.00	6631.82	820.058	4979.11 - 8284.54
14	389.00	596.85	783.914	−983.03 - 2176.72
15	167.00	749.71	782.417	−827.15 - 2326.57
16	249.00	852.76	799.143	−757.80 - 2463.33
17	1209.00	1223.99	780.850	−349.71 - 2797.68
18	5878.00	5847.88	806.224	4223.04 - 7472.72

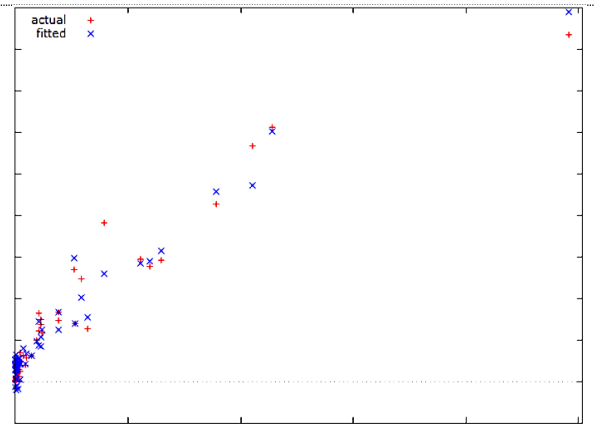


Figure 182 : Sortie de Gretl et graph de dispersion de la variable ColorThreshold

La p-value du régresseur ColorTreshold est maintenant maximale et est supérieure au seuil de significativité. Les prédictions sont « plus précises » en ce sens où les écarts types sont plus étroits. Le graphe de dispersion de la variable gray à une tendance linéaire, pourquoi est-elle rejetée du modèle ? Sa p-value indique une faible significativité.

3ème itération :

Cette troisième itération sans la variable ColorThreshold semble être la bonne.

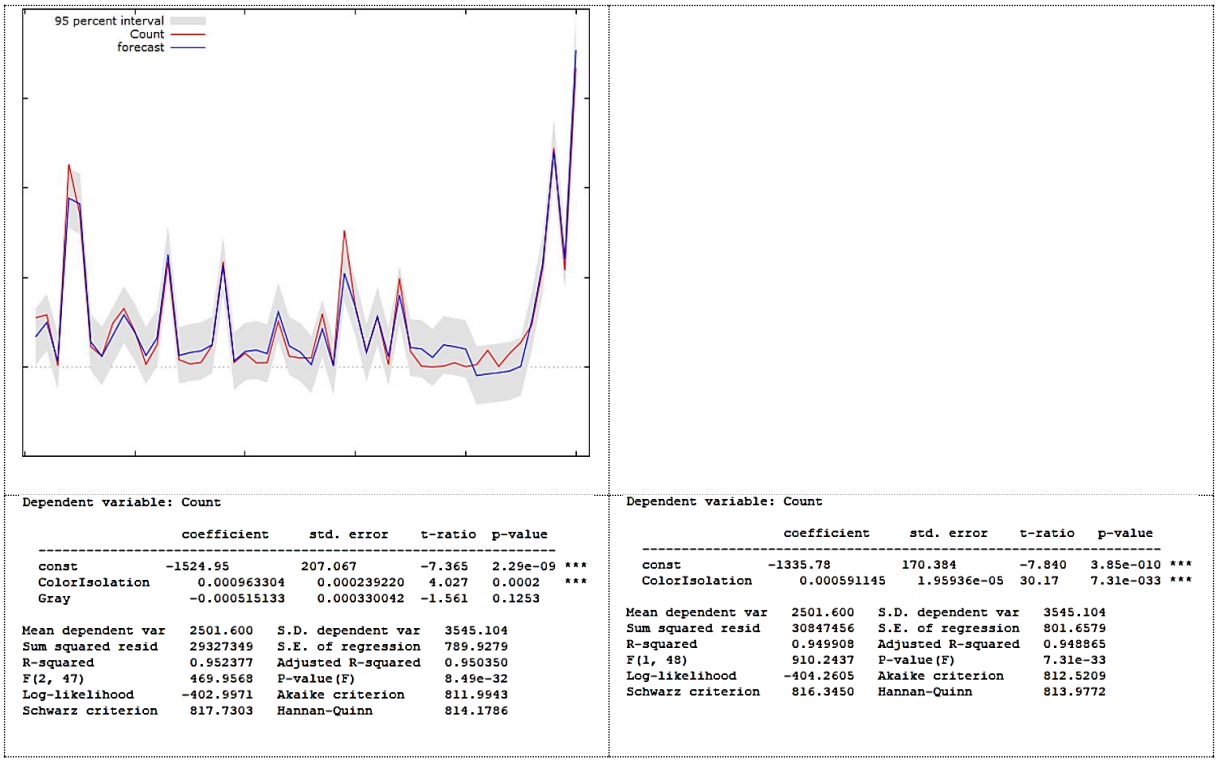
Dependent variable: Count

	coefficient	std. error	t-ratio	p-value	
const	−1269.47	253.481	−5.008	8.55e-06	***
ColorIsolation	0.00110596	0.000249480	4.433	5.72e-05	***
Gray	−0.000718330	0.000345500	−2.079	0.0432	**
invasion	−433.150	257.076	−1.685	0.0988	*

Mean dependent var	2501.600	S.D. dependent var	3545.104
Sum squared resid	27622602	S.E. of regression	774.9138
R-squared	0.955145	Adjusted R-squared	0.952220
F(3, 46)	326.5092	P-value(F)	5.28e-31
Log-likelihood	−401.5000	Akaike criterion	810.9999
Schwarz criterion	818.6480	Hannan-Quinn	813.9124

For 95% confidence intervals, t(46, 0.025) = 2.013

	Count	prediction	std. error	95% interval
1	2750.00	1685.13	791.268	92.39 - 3277.87
2	2910.00	2483.97	786.000	901.83 - 4066.11
3	75.00	317.76	789.687	−1271.80 - 1907.32
4	11331.00	9432.27	822.096	7777.48 - 11087.06
5	8524.00	9106.49	842.515	7410.60 - 10802.38
6	1128.00	1386.24	792.308	−208.60 - 2981.07
7	596.00	594.66	808.701	−1033.17 - 2222.49
8	2442.00	1767.55	791.513	174.32 - 3360.78
9	3262.00	2913.92	786.985	1329.80 - 4498.03
10	1973.00	1914.83	788.982	326.69 - 3502.96
11	137.00	636.07	790.556	−955.23 - 2227.38
12	1225.00	1606.80	787.245	22.16 - 3191.45
13	5853.00	6275.13	797.147	4670.55 - 7879.70
14	389.00	637.10	789.828	−952.74 - 2226.94
15	167.00	800.47	787.947	−785.58 - 2386.53
16	249.00	890.41	797.170	−714.21 - 2495.03
17	1209.00	1227.69	786.732	−355.92 - 2811.30



Dependent variable: Count

	coefficient	std. error	t-ratio	p-value	
const	−1524.95	207.067	−7.365	2.29e-09	***
ColorIsolation	0.000963304	0.000239220	4.027	0.0002	***
Gray	−0.000515133	0.000330042	−1.561	0.1253	

Mean dependent var	2501.600	S.D. dependent var	3545.104
Sum squared resid	29327349	S.E. of regression	789.9279
R-squared	0.952377	Adjusted R-squared	0.950350
F(2, 47)	469.9568	P-value(F)	8.49e-32
Log-likelihood	−402.9971	Akaike criterion	811.9943
Schwarz criterion	817.7303	Hannan-Quinn	814.1786

Dependent variable: Count

	coefficient	std. error	t-ratio	p-value	
const	−1335.78	170.384	−7.840	3.85e-010	***
ColorIsolation	0.000591145	1.95936e-05	30.17	7.31e-033	***

Mean dependent var	2501.600	S.D. dependent var	3545.104
Sum squared resid	30847456	S.E. of regression	801.6579
R-squared	0.949908	Adjusted R-squared	0.948865
F(1, 48)	910.2437	P-value(F)	7.31e-33
Log-likelihood	−404.2605	Akaike criterion	812.5209
Schwarz criterion	816.3450	Hannan-Quinn	813.9772

Figure 183 : Sortie de Gretl et courbe Count vs Prediction

Les p-value sont entre 0.05 et 0.1, c'est pourquoi les prédicteurs ont tous une étoile. Bien que toutes les p-values soient plus grandes que le seuil de significativité, Gretl ne nous propose pas de supprimer un nouveau régresseur. Si nous appliquons la méthode jusqu'au bout, il faudrait supprimer le régresseur Invasion qui est à 9.9% et ensuite le régresseur Gray qui obtiendrait une p-value de 12% dans un quatrième modèle. Le cinquième et dernier modèle nous donnerait un modèle avec une seule variable de régression : ColorIsolation.

Mais alors pourquoi Gretl ne nous propose-t-il pas de continuer ? Pour cela il faut s'intéresser au paramètre r-square et adjusted r-square. Le paramètre r-square est l'indicateur permettant de savoir à quel point la droite de régression linéaire est proche des points d'observation. Le paramètre adjusted r-square est similaire mais il possède en plus un facteur de pénalisation le rendant insensible au nombre de régresseurs du modèle. En effet, r-square augmente à l'ajout de chaque régresseur rendant sont interprétation moins évidente.

R-square et Adjusted r-square modèle de 1 à 5				
0.955289	0.955222	0.955145	0.952377	0.949908
0.950208	0.951242	0.952220	0.950350	0.948865

Le premier paramètre r-square diminue petit à petit en fonction du nombre de régresseurs. Le paramètres adjusted r-square quant à lui, augmente du modèle 1 à 3 et ensuite diminue. Le modèle 3 possède donc la droite de régression linéaire la plus proche de nos points d'observation, il est plus robuste.

9.2.6 Modèle et prédiction

Le modèle final comporte trois régresseurs, ColorIsolation, Gray et Invasion. Les profils des droites d'observations et de prédictions se suivent mais la droite de prédiction sort à plusieurs reprises de l'intervalle de 95% (la droite rouge et non bleue du schéma ci-dessus).

Le coefficient de la variable ColorIsolation est positif, ce qui signifie que le nombre de cellules augmente quand le nombre de pixels augmente. En revanche, le coefficient invasion est négatif, ce qui signifie qu'il évolue dans le sens inverse du nombre de cellules. Nous pouvons en tirer une conclusion, quand la variable invasion augmente à la valeur 1, le nombre de cellules diminue. Ça a du sens puisque lors d'une invasion, les cellules doivent d'abord traverser une membrane avant de passer de l'autre coté en empruntant les pores.

9.2.7 Conclusion

Nous parlerons ici d'un essai non transformé. Ce type d'estimation demanderait encore quelques ajustements pour donner une courbe plus ou moins correcte. Dans les défauts, nous pouvons pointer :

- Le manque d'observations. En effet, 50 images de migration et d'invasion semblent insuffisantes pour dresser un profil.
- Les images ont été prétraitée afin de supprimer les bords de la boîte de Pétri. Cette opération manuelle a supprimé une partie des cellules ayant très probablement été comptabilisées. L'image d'origine perd plus de 30 % de sa superficie initiale. De plus la partie rognée est faite à main levée et est différente d'une image à l'autre.
- De nombreuses images ont des défauts. Des taches ou crasses apparaissent au centre de l'image. Une image ayant de nombreuses cellules ne sera que faiblement impactée mais une image ne comportant que quelques cellules pourraient compter plus de pixels parasites que de pixels de cellules.
- Le masque du filtre par bande de couleur était-il adéquat ? La bande de couleur n'était-elle pas trop faible ? En effet, le cytoplasme a été tronqué par endroit.

Il nous semble que c'est une piste qui mériterait d'être approfondie.

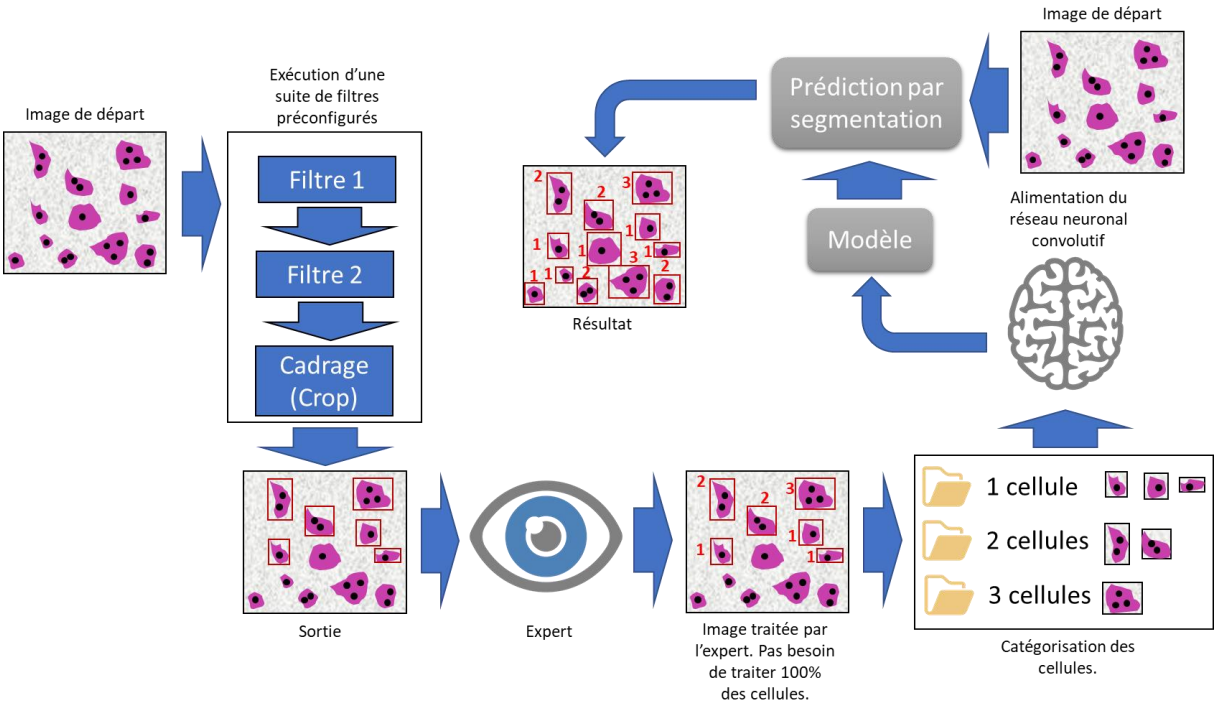
Chapitre 10
Vers un système d'aide à l'analyse d'images

Ce chapitre n'a pas pour prétention de proposer une analyse complète de l'application idéale, mais bien de dépeindre notre vision du produit. N'ayant eu que peu d'interaction avec de potentiels utilisateurs, le sujet est abordé telle une startup imaginant son futur produit phare.

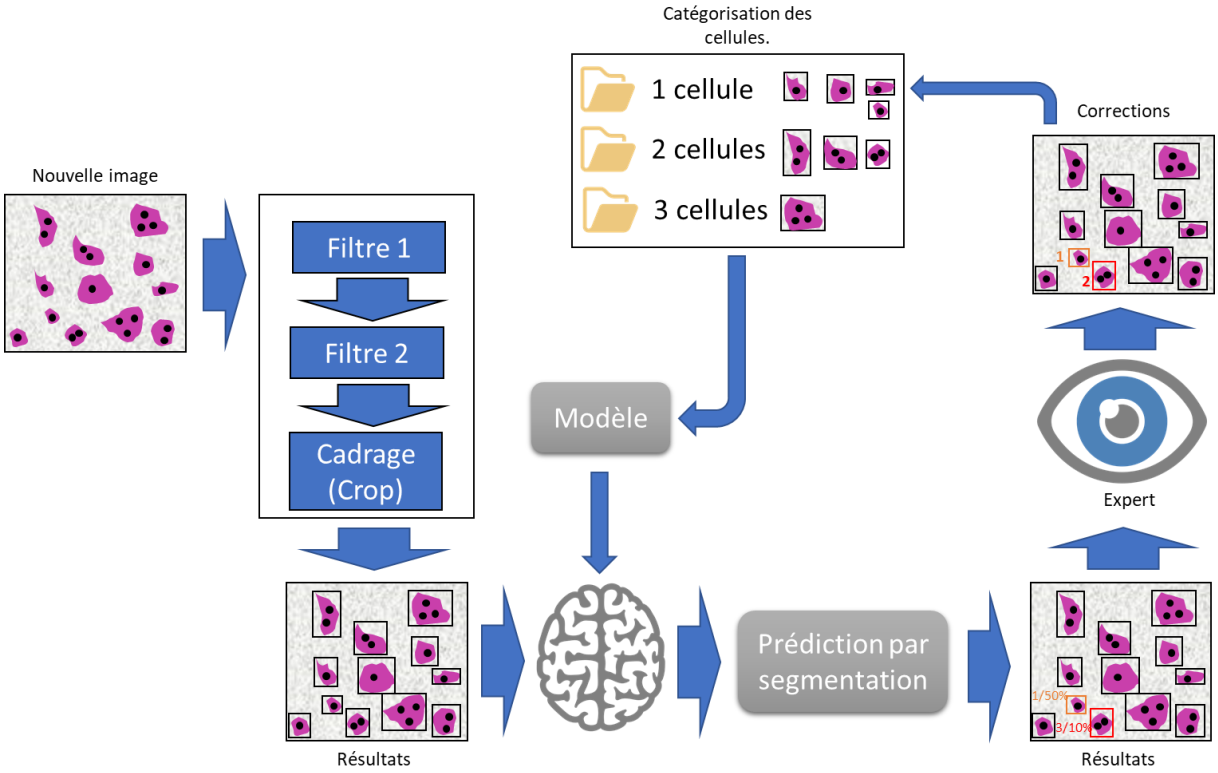
Cette analyse arrive assez tard dans le processus de rédaction de ce travail. Il était en effet indispensable de maîtriser le sujet vaste et complexe qu'est le traitement d'image numérique avant de se lancer dans l'élaboration d'une solution.

A la fin de l'analyse nous aurons une ébauche des user stories de notre application idéale. Nous utiliserons des « fiches » tirées des méthodes agiles pour présenter les différentes user stories. Ces fiches comprendront un identifiant, un nom et une description (formatée de telle sorte que l'on y retrouve l'intervenant, l'action et l'accomplissement).

10.1 Idée générale

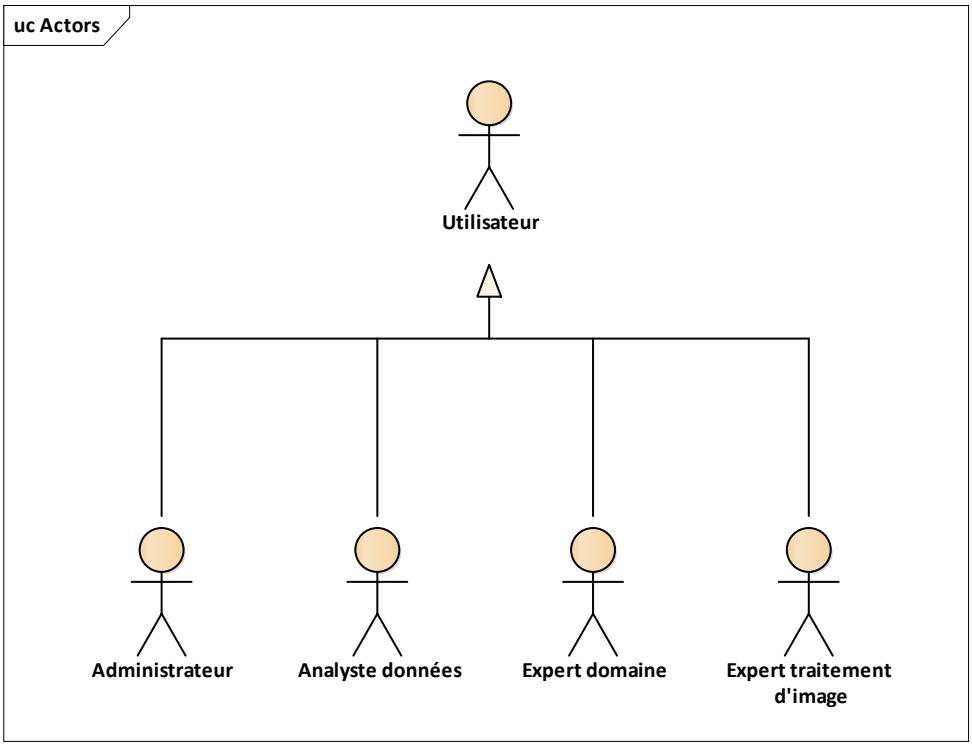


Les idées gravitant autour de cette application idéale se base sur notre première impression (voir §1.3), mais aussi sur les connaissances acquises depuis. Comme dit précédemment, le travail à effectuer sur une image peut se modéliser sous forme de workflow. La première partie sera un enchainement de filtres qui permettront d'isoler les occurrences de ce que l'on cherche (des cellules cancéreuses dans notre cas, mais ça pourrait très bien être des rhinocéros blancs sur une photo satellite). Une fois les coordonnées des occurrences isolées, l'application pourra les soumettre à l'œil d'un expert, qui pourra les dénombrer ou corriger l'information déjà présente. L'image sera alors découpée en petites images et catégorisées (une cellule, deux cellules, un rhinocéros mâle…) celles-ci serviront d'apprentissage au réseau neuronal à convolution (RNC). Le RNC générera alors une prédiction par segmentation qui pourra de nouveau être soumis à l'œil de l'expert.



De manière générale, dès que le réseau neuronal convolutif est suffisamment alimenté, l'expert peut traiter d'autres images du même type. Un des avantages du RNC, c'est qu'en plus du résultat, il offre un degré de certitude qui peut être exploité. L'application pourra donc proposer à l'expert les groupes de cellules dont la réponse du RNC est la moins sûr. L'expert les validera ou il les corrigera. Ces occurrences pourront alors compléter les catégories et le modèle pourra être reconstruit et utilisé pour les images suivantes.

10.2 Intervenants



10.2.1 Expert traitement d'image

C'est l'expert qui va s'occuper de configurer le workflow, le processus permettant de cadrer les occurrences de ce que l'on cherche en appliquant toute une série de filtres. Connaître l'emplacement de ces occurrences permet au système de les itérer suivant un ordre bien définit et ainsi proposer à l'utilisateur une information pertinente. De plus, le système pourra extraire ces occurrences sous forme de petites images qui servira à alimenter le réseau neuronal convolutif.

10.2.2 Expert domaine

C'est l'utilisateur principal de l'application. Il est expert dans son domaine d'application et ses observations sont considérées comme 100% fiable. C'est à lui que le système proposera les occurrences afin qu'il puisse les catégoriser ou les corriger.

10.2.3 Réseau neuronal convolutif

C'est lui qui va mâcher le travail de l'expert domaine. Une fois son modèle alimenté, il pourra catégoriser chaque occurrence qui lui sera proposée et lui assigner un degré de certitude. L'expert domaine pourra alors vérifier les résultats en se basant sur le degré de certitude. Suivant les corrections qu'il apportera, le modèle pourra alors être enrichi et gagner en efficacité lors du traitement d'images similaires.

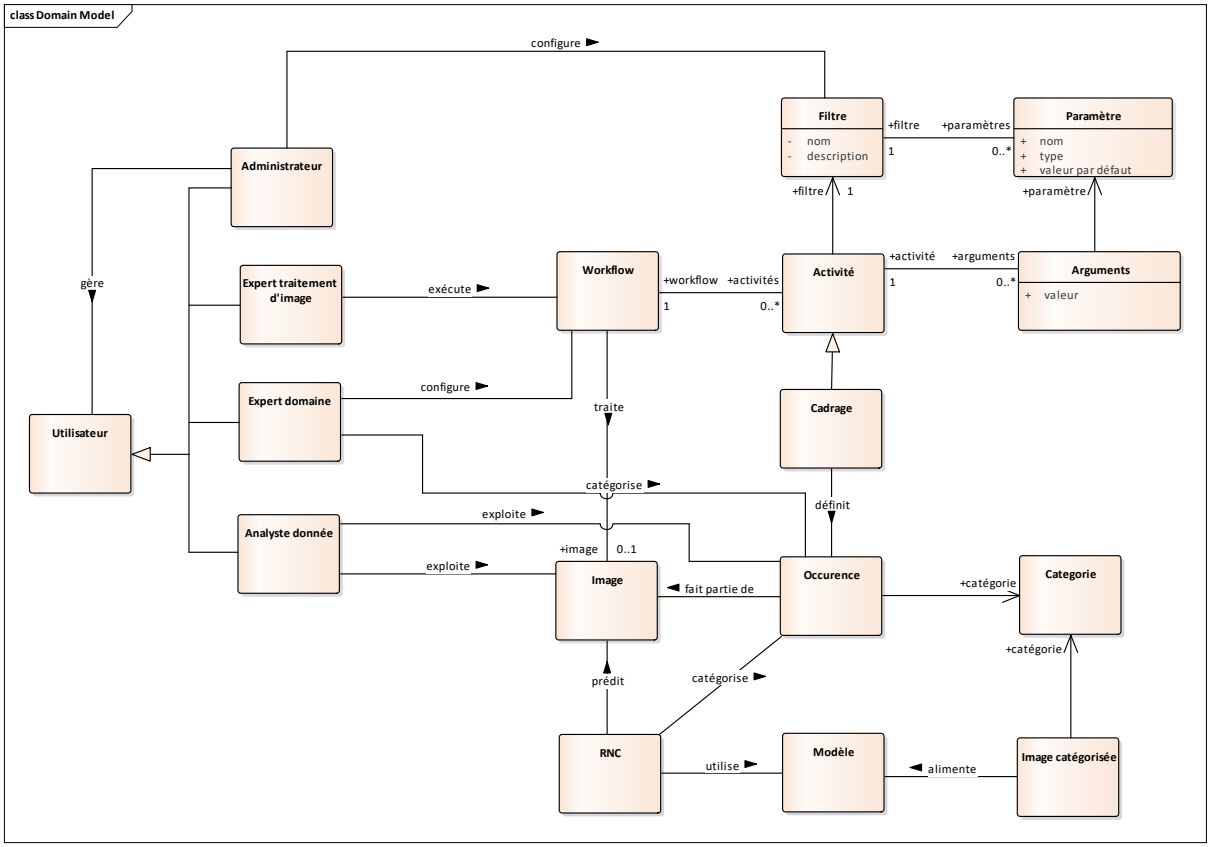
10.2.4 Analyste donnée

C'est la personne qui s'occupera d'extraire les résultats afin de les analyser et en tirer des conclusions. Il est probablement également expert dans son domaine d'application.

10.2.5 Administrateur

C'est la personne en charge de la configuration du système, probablement un IT. Il gère les paramètres du système (emplacement des fichiers, connexions à une éventuelle base de données…), les utilisateurs et leur attribue des rôles, rôles qui leur donneront accès à certaines parties de l'application.

10.3 Domain model du système



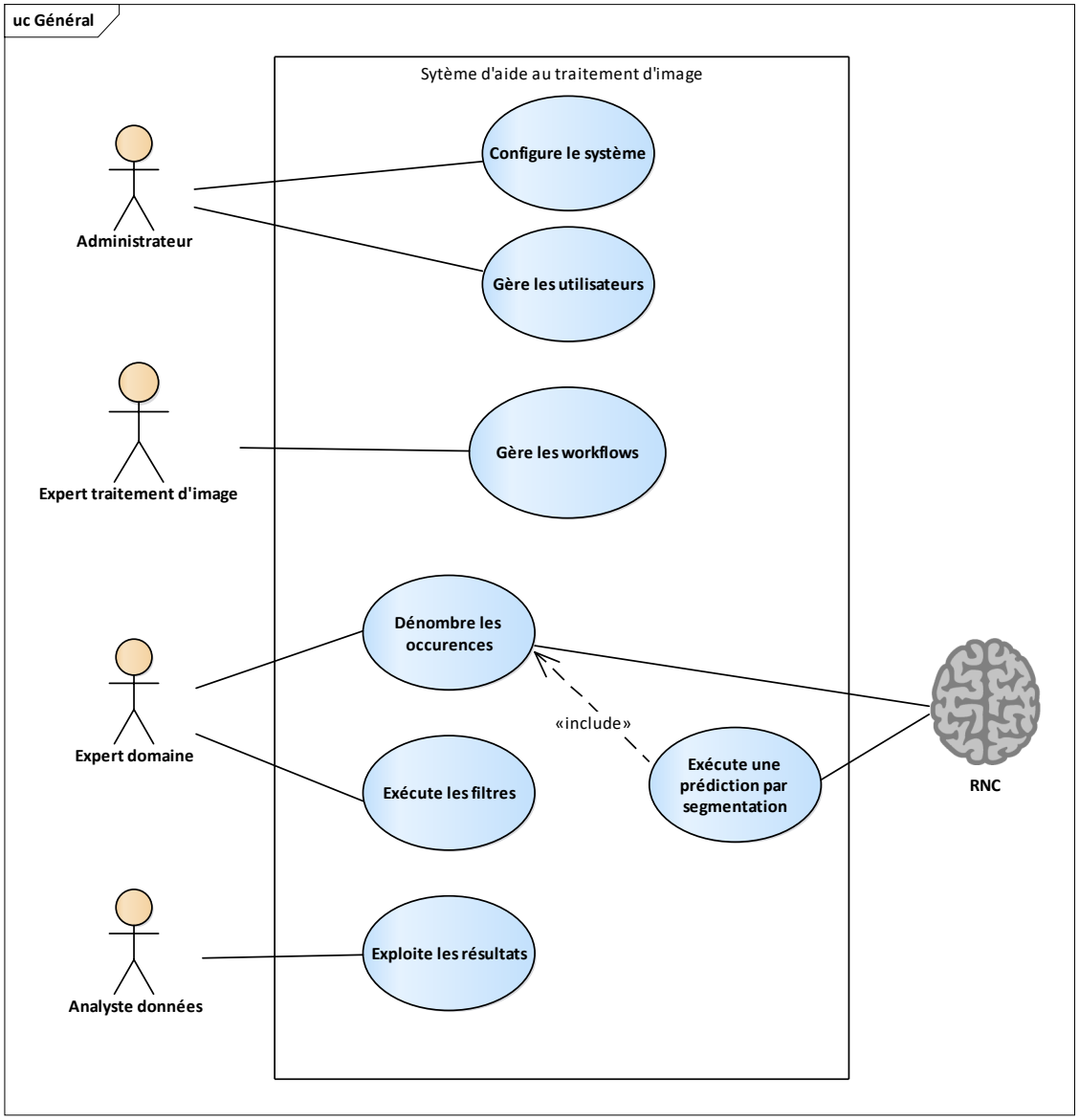
10.4 Lexique

- Filtre : En mathématique, un filtre linéaire est une opération qui transforme un ensemble de données d'entrée en un ensemble de données de sortie. Cette opération s'appelle une convolution. En traitement d'image, un filtre va transformer une image d'entrée en une image de sortie (voir §3.1).
- Paramètre : Les paramètres d'un filtre permettent de configurer ce filtre. Par exemple, pour la plupart des filtres linéaires, l'un des paramètres sera la taille du kernel.
- Workflow : C'est le processus qui mènera au résultat escompté. Chaque filtre prend

en entrée une image et génère une image en sortie. Le workflow est donc un enchainement de filtres paramétrés.

- Activité : Une activité est un filtre auquel on a associé des valeurs de paramètres, c'est à dire des arguments.
- Argument : Un argument est un paramètre auquel a été associé une valeur.
- Cadrage : Le cadrage est le résultat escompté du workflow, il permet de déterminer la position et la taille des occurrences.
- Image : C'est une image source sur laquelle le workflow va tenter de détecter des occurrences.
- Occurrence : Chaque détection d'un élément cherché sur une image, par exemple un rhinocéros blanc sur une photo satellite, est une occurrence.
- Catégorie : Les éléments détectés sur une image sont catégorisés. Par exemple, si l'on cherche les mammifères sur une photo, les catégories seraient : chien, chat, gorille, …
- Image catégorisée : Pour alimenter le réseau neuronal convolutif, il est nécessaire de lui fournir des images pour chaque catégorie recherchée, ainsi une image catégorisée est une occurrence extraite de son image d'origine et enregistrée en tant que petite image et mise à disposition du RNC.

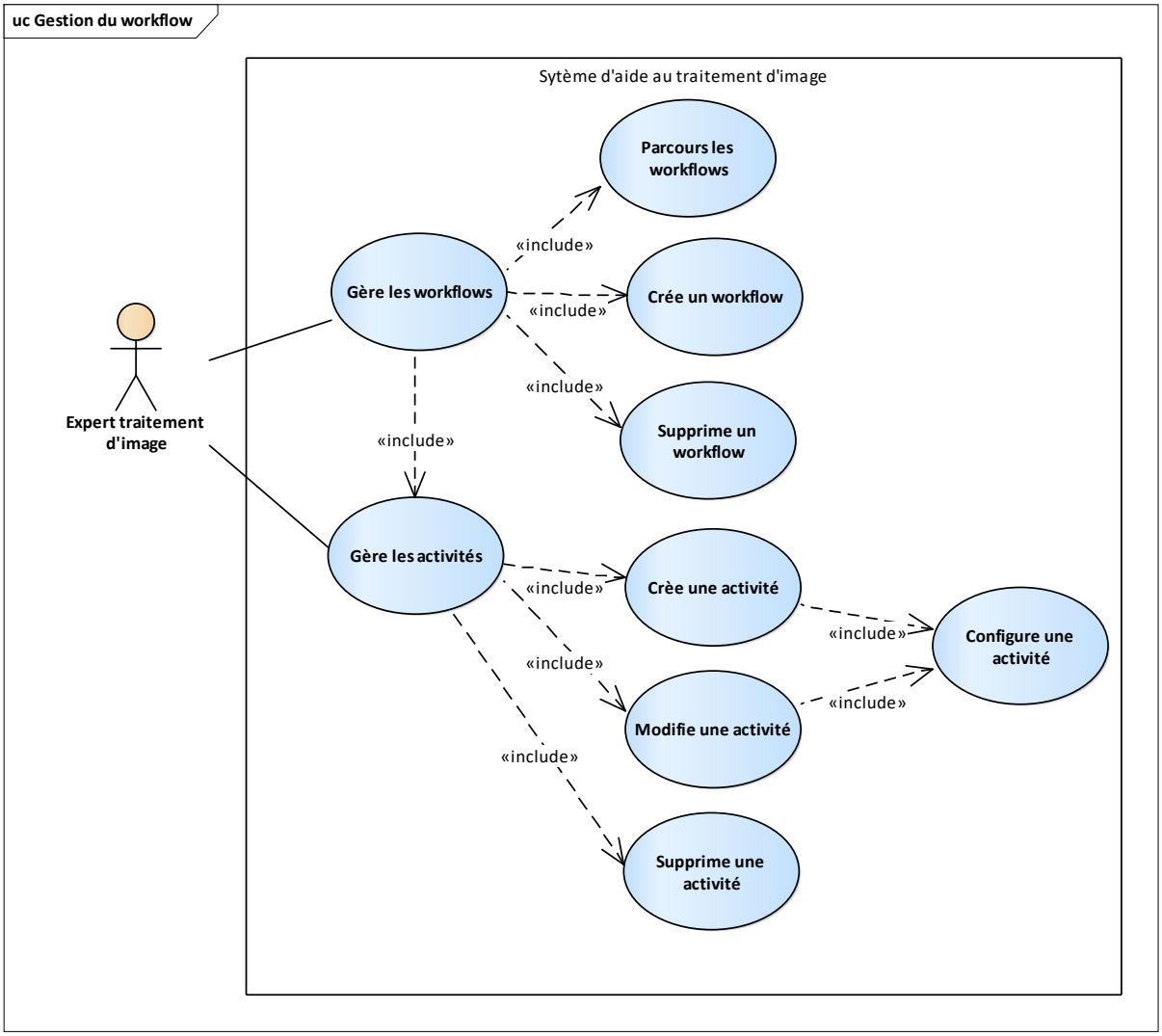
10.5 Vue générale



Cette vue générale reprend les principales activités de chaque type d'intervenant. L'administrateur gère les utilisateurs et configure le système. L'expert en traitement d'image crée et configure le workflow. L'expert domaine exécute les filtres sur une image donnée afin de dénombrer les éléments qui la composent. Et enfin l'analyste donnée exploite les résultats.

Le réseau neuronal à convolution fait également partie des intervenants, ses prédictions pouvant également dénombrer les éléments d'une image une fois son modèle correctement alimenté.

10.6 Gestion du processus de traitement d'image



L'expert en traitement d'image doit pouvoir créer, modifier ou supprimer un processus de traitement d'image (workflow). Ce qui implique la possibilité d'ajouter, modifier ou supprimer des activités (filtres) sur ce workflow.

10.6.1 US 1.1 - Créer un workflow

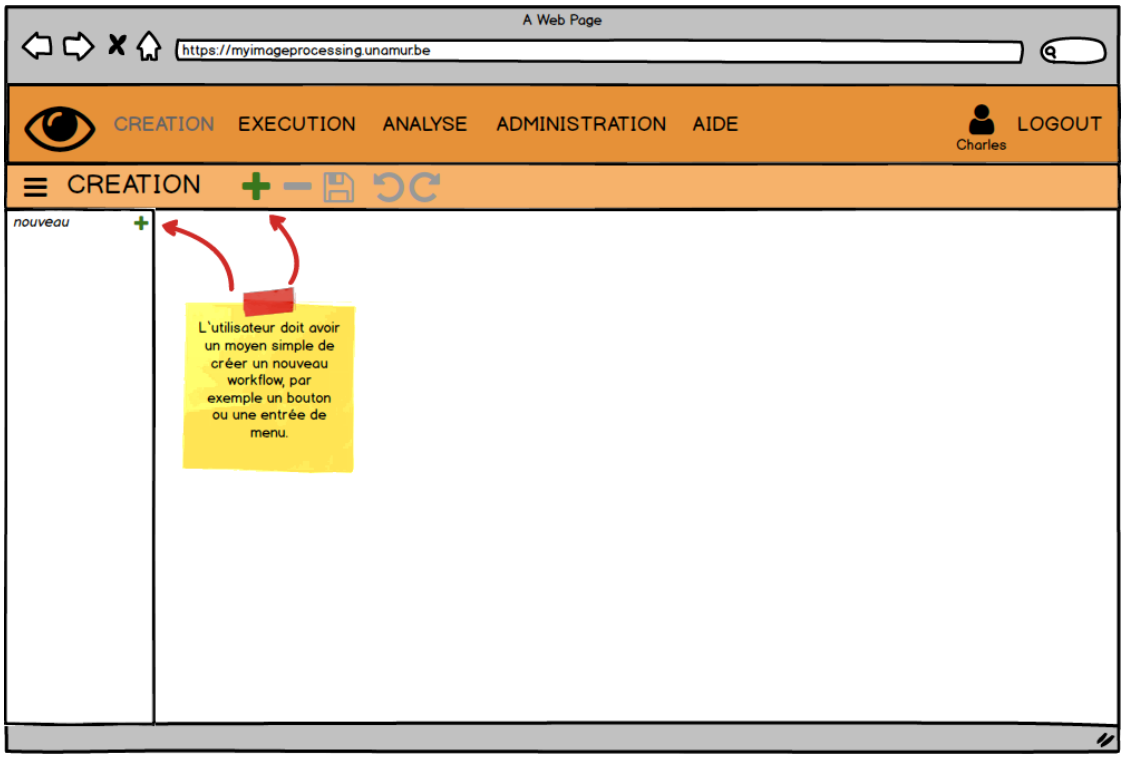
Créer un workflow	US 1.1

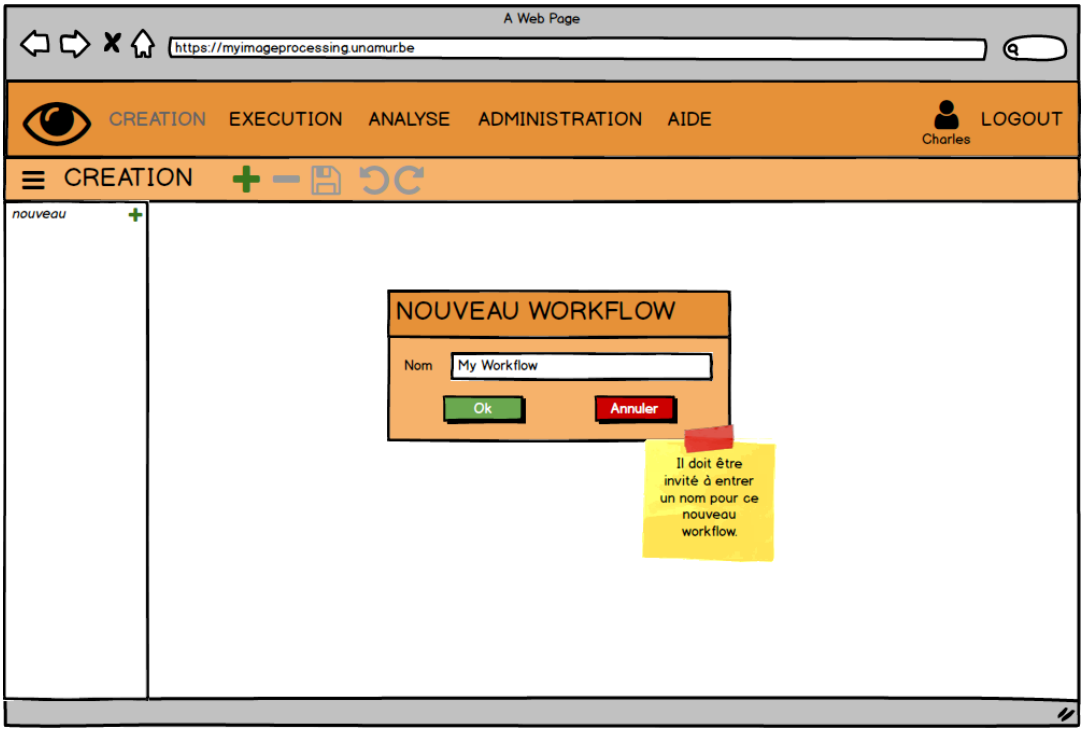
Description

En tant que : expert en traitement d'image
Je veux : être capable de créer un nouveau processus de traitement d'image
Afin de : avoir un point de départ à la création d'une séquence de filtres qui permettront à terme de détecter les contours des éléments à dénombrer.

Critères d'acceptance

L'utilisateur doit avoir un moyen simple de créer un nouveau workflow, par exemple un bouton ou une entrée de menu. Il doit être invité à entrer un nom pour ce nouveau workflow.





10.6.2 US 1.2 - Parcourir les workflows

Parcourir les workflows	US 1.2

Description

En tant que : expert en traitement d'image
Je veux : pouvoir visualiser les workflows existants
Afin de : avoir la possibilité de les modifier, les exécuter ou les supprimer.

Critères d'acceptance

L'utilisateur doit avoir un aperçu des workflows existants et avoir la possibilité d'en sélectionner un. Le workflow sélectionné doit être affiché en détail.

10.6.3 US 1.3 - Supprimer un workflow

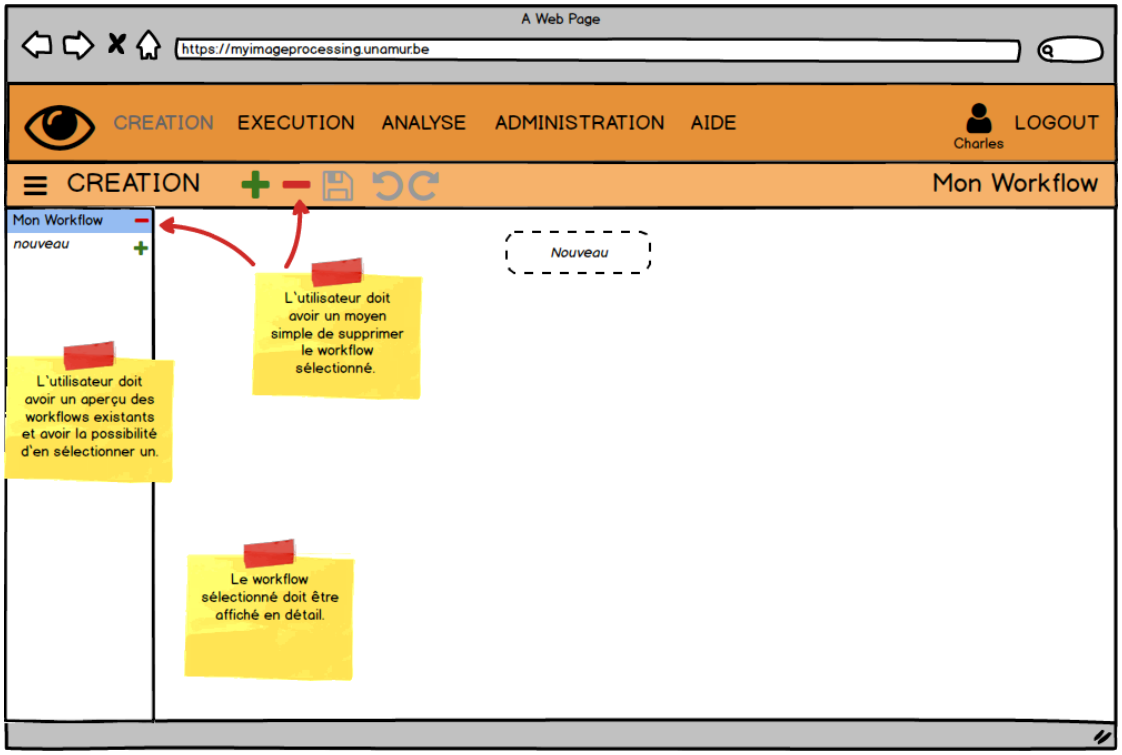
Supprimer un workflow	US 1.3

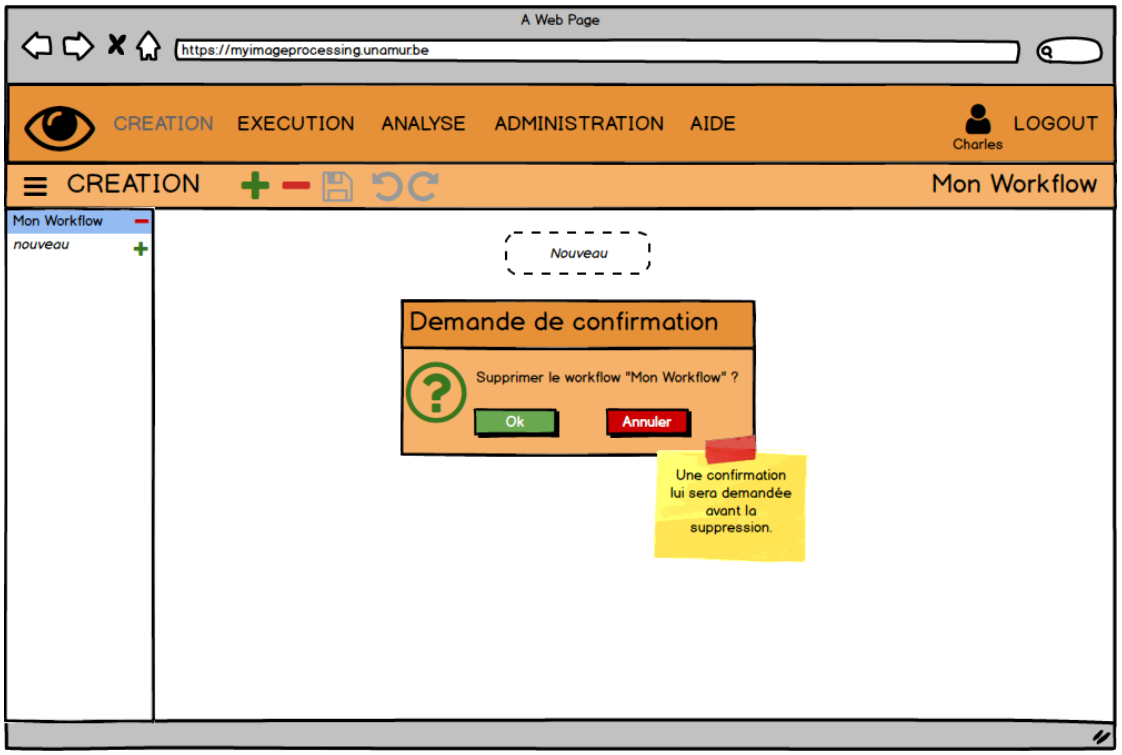
Description

En tant que : expert en traitement d'image
Je veux : supprimer un workflow existant
Afin de : ne plus voir apparaître un workflow inutile dans la liste des workflows.

Critères d'acceptance

L'utilisateur doit avoir un moyen simple de supprimer le workflow sélectionné. Une confirmation lui sera demandée avant la suppression.


A Web Page
https://myimageprocessing.unamur.be
CREATION
EXECUTION
ANALYSE
ADMINISTRATION
AIDE
LOGOUT
Charles
CREATION
Mon Workflow
Mon Workflow
nouveau
Nouveau
L'utilisateur doit avoir un moyen simple de supprimer le workflow sélectionné.
L'utilisateur doit avoir un aperçu des workflows existants et avoir la possibilité d'en sélectionner un.
Le workflow sélectionné doit être affiché en détail.


A Web Page
https://myimageprocessing.unamur.be
CREATION
EXECUTION
ANALYSE
ADMINISTRATION
AIDE
LOGOUT
Charles
CREATION
Mon Workflow
Mon Workflow
nouveau
Nouveau
Demande de confirmation
Supprimer le workflow "Mon Workflow" ?
Ok
Annuler
Une confirmation lui sera demandée avant la suppression.

10.6.4 US 1.4 - Créer une activité

Créer une activité	US 1.4

Description

En tant que : expert en traitement d'image
Je veux : être capable de créer une activité
Afin de : pouvoir configurer les filtres nécessaires à l'exécution du workflow.

Critères d'acceptance

L'utilisateur doit pouvoir ajouter une activité et la positionner dans l'ordre dans lequel il veut qu'elle s'exécute. Il doit en outre pouvoir sélectionner le filtre qu'il souhaite. Chaque filtre doit être rapidement identifiable.

10.6.5 US1.5 - Modifier une activité

Modifier une activité	US 1.5

Description

En tant que : expert en traitement d'image
Je veux : pouvoir modifier une activité existante en changeant l'ordre d'exécution ou le type de filtre
Afin de : pouvoir affiner le workflow pour atteindre l'objectif désiré.

Critères d'acceptance

L'utilisateur doit pouvoir simplement réordonnancer les activités ou changer le type de filtre.

10.6.6 US 1.6 - Configurer une activité

Configurer une activité	US 1.6

Description

En tant que : expert en traitement d'image
Je veux : pouvoir configurer les paramètres du filtre sélectionné
Afin de : d'exploité au mieux chaque type de filtre.

Critères d'acceptance

L'utilisateur doit avoir accès à toute une série de paramètres propres à chaque filtre.

10.6.7 US 1.7 - Supprimer une activité

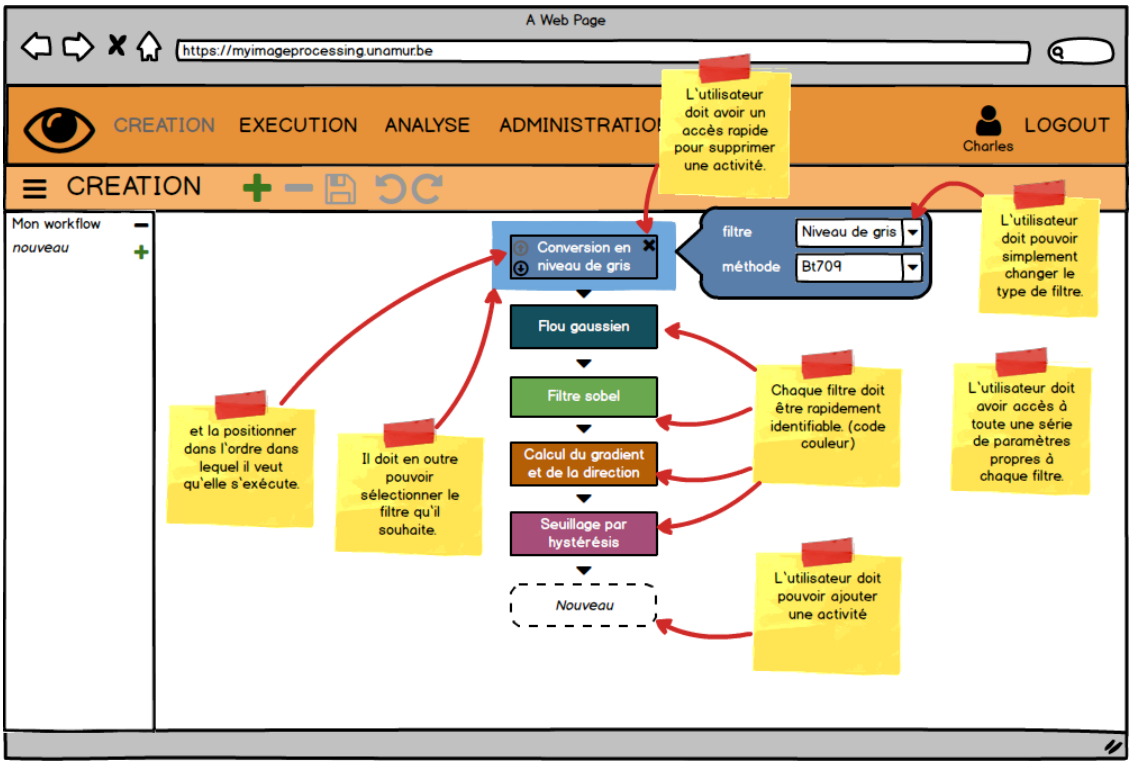
Supprimer une activité	US 1.7

Description

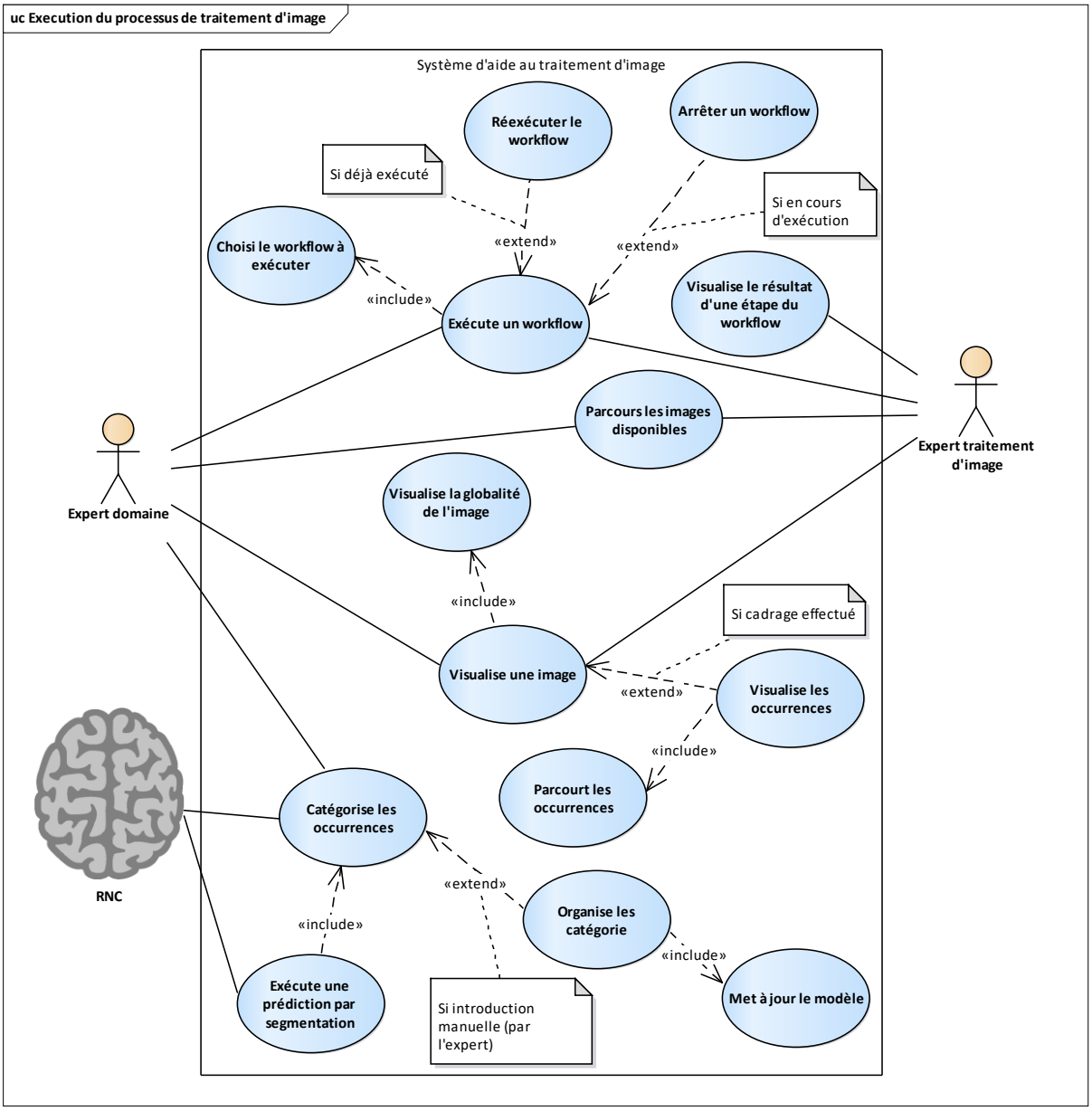
En tant que : expert en traitement d'image
Je veux : pouvoir supprimer l'activité sélectionnée
Afin de : de ne pas garder une activité qui ne serait plus utile.

Critères d'acceptance

L'utilisateur doit avoir un accès rapide pour supprimer une activité. Une confirmation sera demandée.



10.7 Exécution du processus de traitement d'image



L'expert en traitement d'image doit pouvoir tester un workflow en l'exécutant, puis en visualisant le résultat. Dès que ce workflow est correctement configuré, l'expert domaine peut l'utiliser.

L'expert domaine doit pouvoir utiliser le workflow, c'est-à-dire l'exécuter pour obtenir le cadrage des occurrences, puis visualiser celles-ci et éventuellement les corriger. Il doit pouvoir se déplacer d'occurrence en occurrence en suivant un ordre prédéfini, par exemple, quand le RNC a effectué sa prédiction, il serait intéressant de se déplacer des dénombrements les plus incertains vers ceux qui ont la plus grande certitude. Le RNC devrait avoir le même comportement, quel que soit l'exhaustivité de son modèle, c'est-à-dire qu'il doit s'exécuter après l'exécution du workflow. Si son modèle est incomplet, ou si son modèle n'a pas encore été généré, les occurrences auront une incertitude de 100% et donc l'expert domaine sera invité à corriger l'information.

10.7.1 US 2.1 - Parcourir les images disponibles

Parcourir les images disponibles	US 2.1

En tant que : expert en traitement d'image ou expert domaine
Je veux : avoir une vue globale des images disponibles dans le système
Afin de : de pouvoir les traiter.

Critères d'acceptance

L'utilisateur doit avoir une visibilité sur les images à traiter, les images en cours d'exécution, les image à vérifier et les images terminées.

10.7.2 US 2.2 - Exécuter un processus de traitement d'image

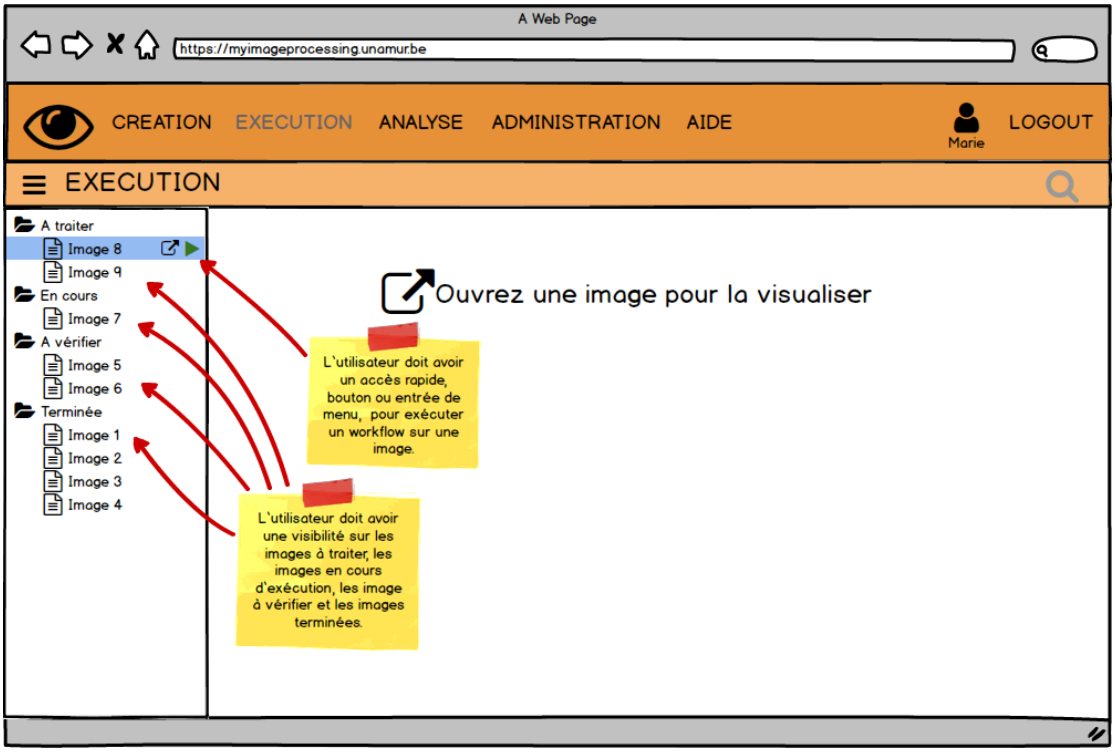
Exécuter un processus de traitement d'image	US 2.2

Description

En tant que : expert en traitement d'image ou expert domaine
Je veux : pouvoir exécuter un processus de traitement d'image
Afin de : de visualiser les occurrences.

Critères d'acceptance

L'utilisateur doit avoir un accès rapide, bouton ou entrée de menu, pour exécuter un workflow sur une image.



10.7.3 US 2.3 - Ouvrir une image

Ouvrir une image	US 2.3

Description

En tant que : expert en traitement d'image ou expert domaine
Je veux : avoir un moyen simple d'ouvrir une image dans la vue principale
Afin de : de visualiser son contenu.

Critères d'acceptance

L'utilisateur doit avoir un accès rapide, bouton ou entrée de menu, pour visualiser une image dans l'interface.

10.7.4 US 2.4 - Arrêter un processus de traitement d'image en cours

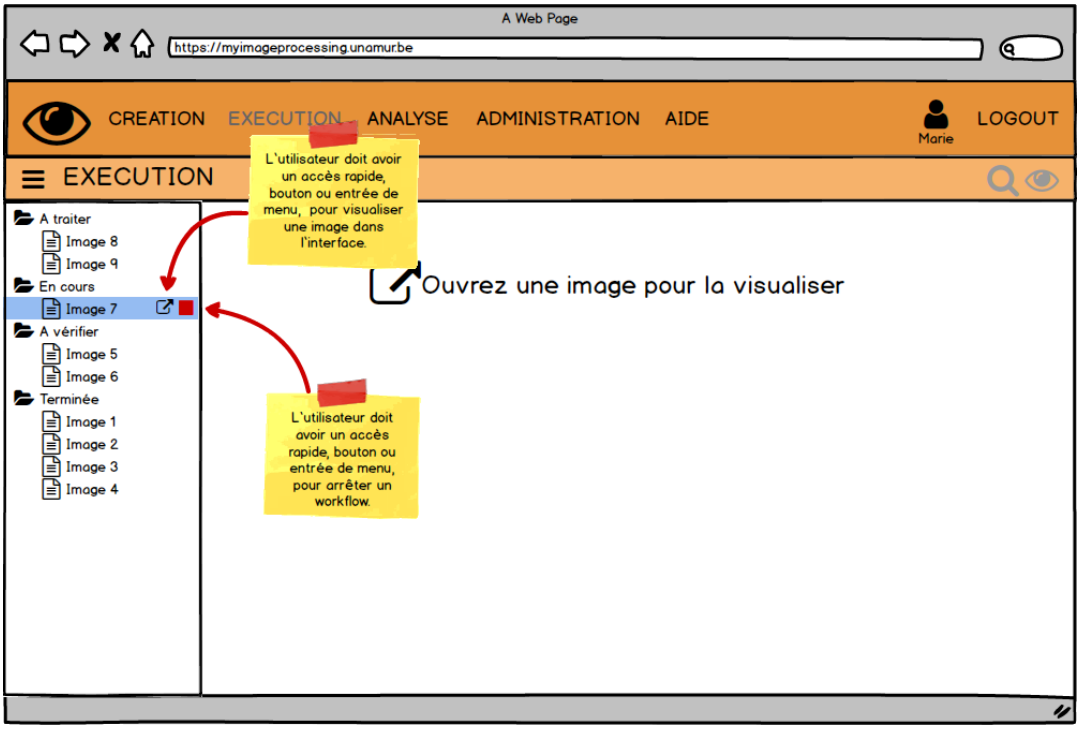
Arrêter un processus de traitement d'image en cours	US 2.4

Description

En tant que : expert en traitement d'image ou expert domaine
Je veux : pouvoir arrêter un processus de traitement d'image en cours
Afin de : éviter de perdre du temps avec un traitement qui serait erroné.

Critères d'acceptance

L'utilisateur doit avoir un accès rapide, bouton ou entrée de menu, pour arrêter un workflow.



10.7.5 US 2.5 - Visualiser une image

Visualiser une image	US 2.5

Description

En tant que : expert en traitement d'image ou expert domaine
Je veux : pouvoir visualiser une image
Afin de : d'analyser son contenu.

Critères d'acceptance

Lorsque l'utilisateur ouvre une image, elle doit apparaître dans la zone principale de l'application.

10.7.6 US 2.6 - Visualiser la globalité de l'image

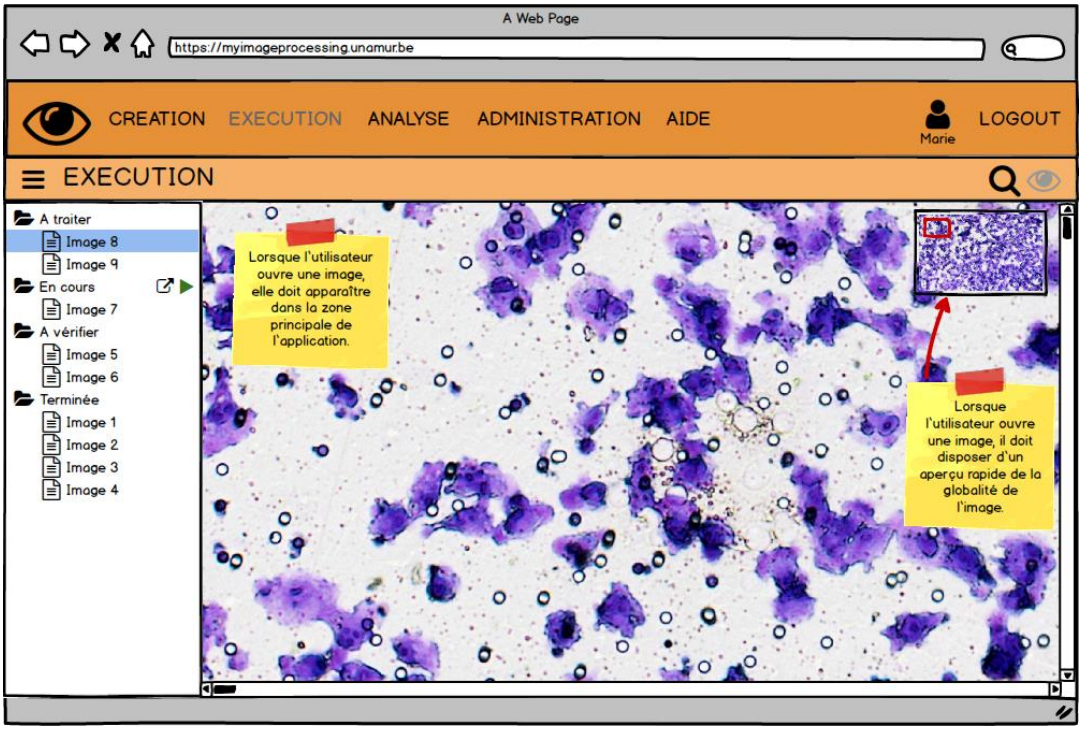
Visualiser la globalité de l'image	US 2.6

Description

En tant que : expert en traitement d'image ou expert domaine
Je veux : avoir un aperçu global de l'image
Afin de : de connaitre la position de la portion visible par rapport à la globalité de l'image et de se déplacer aisément vers un autre emplacement.

Critères d'acceptance

Lorsque l'utilisateur ouvre une image, il doit disposer d'un aperçu rapide de la globalité de l'image.



10.7.7 US 2.7 - Choisir le processus de traitement d'image à exécuter

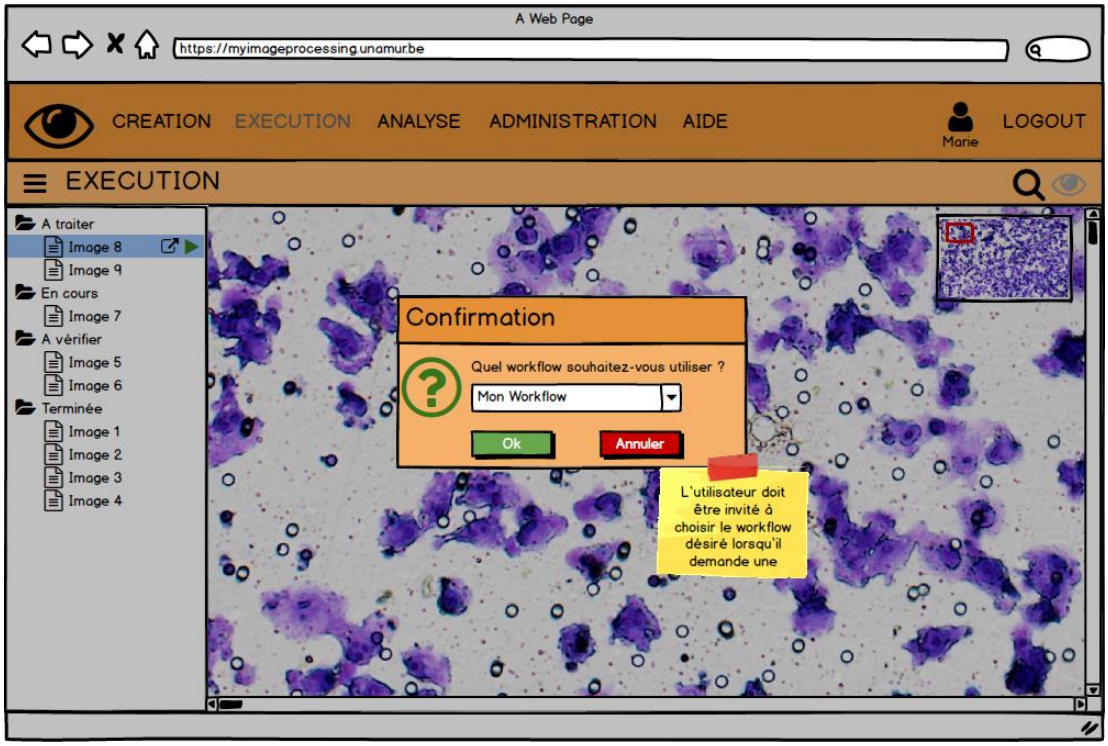
Choisir le processus de traitement d'image à exécuter	US 2.7

Description

En tant que : expert en traitement d'image ou expert domaine
Je veux : pouvoir choisir le processus de traitement d'image à exécuter
Afin de : d'appliquer à l'image, l'enchaînement de filtres souhaité.

Critères d'acceptance

L'utilisateur doit être invité à choisir le workflow désiré lorsqu'il demande une exécution.



10.7.8 US 2.8 - Réexécuter un processus de traitement d'image

Réexécuter un processus de traitement d'image	US 2.8

Description

En tant que : expert en traitement d'image ou expert domaine
Je veux : pouvoir relancer un processus de traitement d'image en cours
Afin de : d'appliquer, à l'image, les corrections effectuées sur le processus de traitement d'image.

Critères d'acceptance

L'utilisateur doit avoir un accès rapide, bouton ou entrée de menu, pour relancer un workflow.

10.7.9 US 2.9 – Visualiser le résultat d'une étape d'un processus de traitement d'image

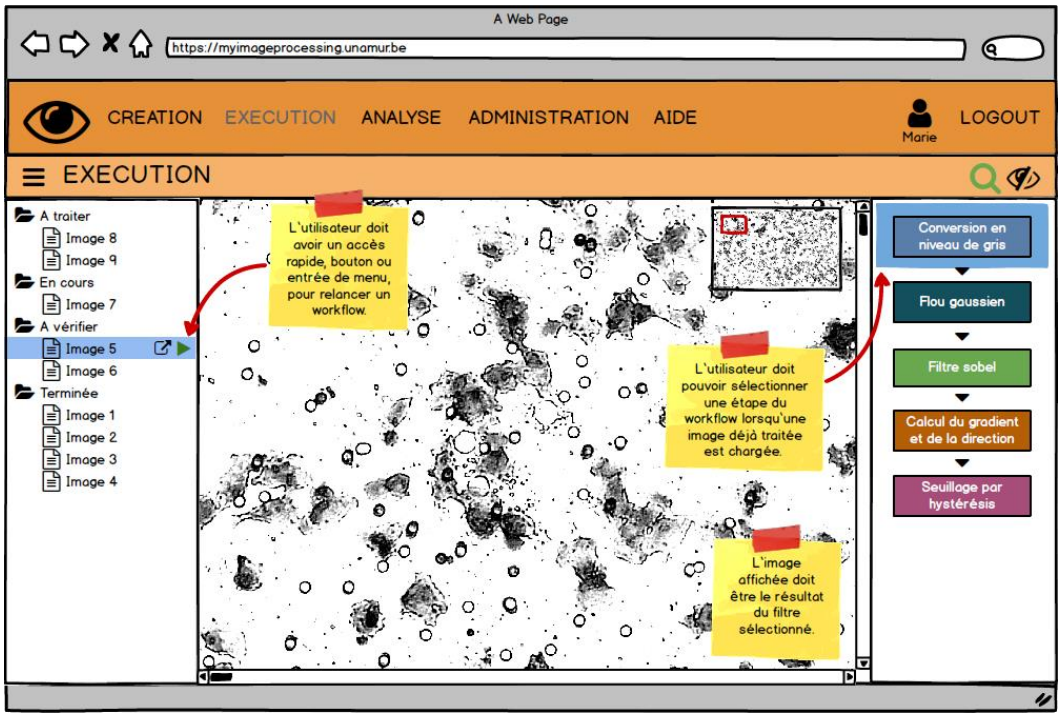
Visualiser le résultat d'une étape d'un processus de traitement d'image	US 2.9

Description

En tant que : expert en traitement d'image
Je veux : pouvoir visualiser le résultat d'une étape d'un processus de traitement d'image
Afin de : de vérifier que l'étape s'exécute correctement.

Critères d'acceptance

L'utilisateur doit pouvoir sélectionner une étape du workflow lorsqu'une image déjà traitée est chargée. L'image affichée doit être le résultat du filtre sélectionné.



10.7.10 US 2.10 - Visualiser les occurrences

Visualiser les occurrences	US 2.10

Description

En tant que : expert domaine
Je veux : pouvoir visualiser les occurrences détectées après l'exécution du processus de traitement d'image
Afin de : de vérifier valider ou corriger l'information.

Critères d'acceptance

L'utilisateur doit pouvoir visualiser et sélectionner une occurrence.

10.7.11 US 2.11 - Parcourir les occurrences

Parcourir les occurrences	US 2.11

Description

En tant que : expert domaine
Je veux : pouvoir parcourir les occurrences par ordre inverse de certitude
Afin de : de vérifier, valider ou corriger les informations les plus douteuses.

Critères d'acceptance

L'utilisateur doit pouvoir itérer les occurrences de la plus incertaine à celle qui offre la meilleure certitude.

10.7.12 US 2.12 - Catégoriser une occurrence

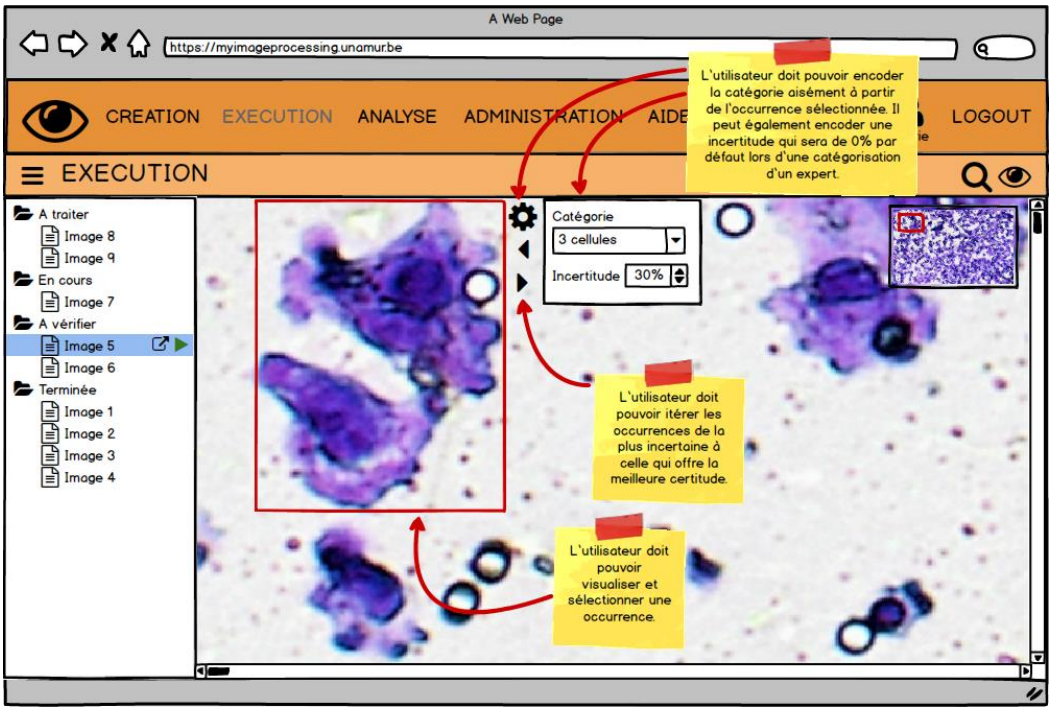
Catégoriser une occurrence	US 2.12

Description

En tant que : expert domaine
Je veux : avoir la possibilité d'encoder la catégorie (le dénombrement) d'une occurrence
Afin de : de corriger une information erronée et ainsi enrichir le système de mon expertise.

Critères d'acceptance

L'utilisateur doit pouvoir encoder la catégorie aisément à partir de l'occurrence sélectionnée. Il peut également encoder une incertitude qui sera de 0% par défaut lors d'une catégorisation d'un expert.



10.7.13 US 2.13 - Organiser les catégories

Organiser les catégories	US 2.13

Description

En tant que : expert domaine
Je veux : que le système organise les catégories lorsque je corrige une information à ce propos
Afin de : que le modèle de prédiction puisse s'affiner et ainsi m'octroyer des prédictions plus justes.

Critères d'acceptance

Lors d'une modification de la catégorie d'une occurrence par l'expert domaine, le système doit rendre disponible cette information pour la mise à jour du modèle de prédiction.

10.7.14 US 2.14 - Mettre à jour le modèle de prédiction

Mettre à jour le modèle de prédiction	US 2.14

Description

En tant que : expert domaine
Je veux : que le système mette à jour le modèle de prédiction lorsque des modifications sont apportées aux catégories
Afin de : que le réseau neuronal convolutif puisse affiner ses prédictions.

Critères d'acceptance

Lors d'une modification de la catégorie d'une occurrence par l'expert domaine, le système doit mettre à jour le modèle de prédiction.

10.7.15 US 2.15 - Exécuter une prédiction par segmentation

Exécuter une prédiction par segmentation	US 2.15

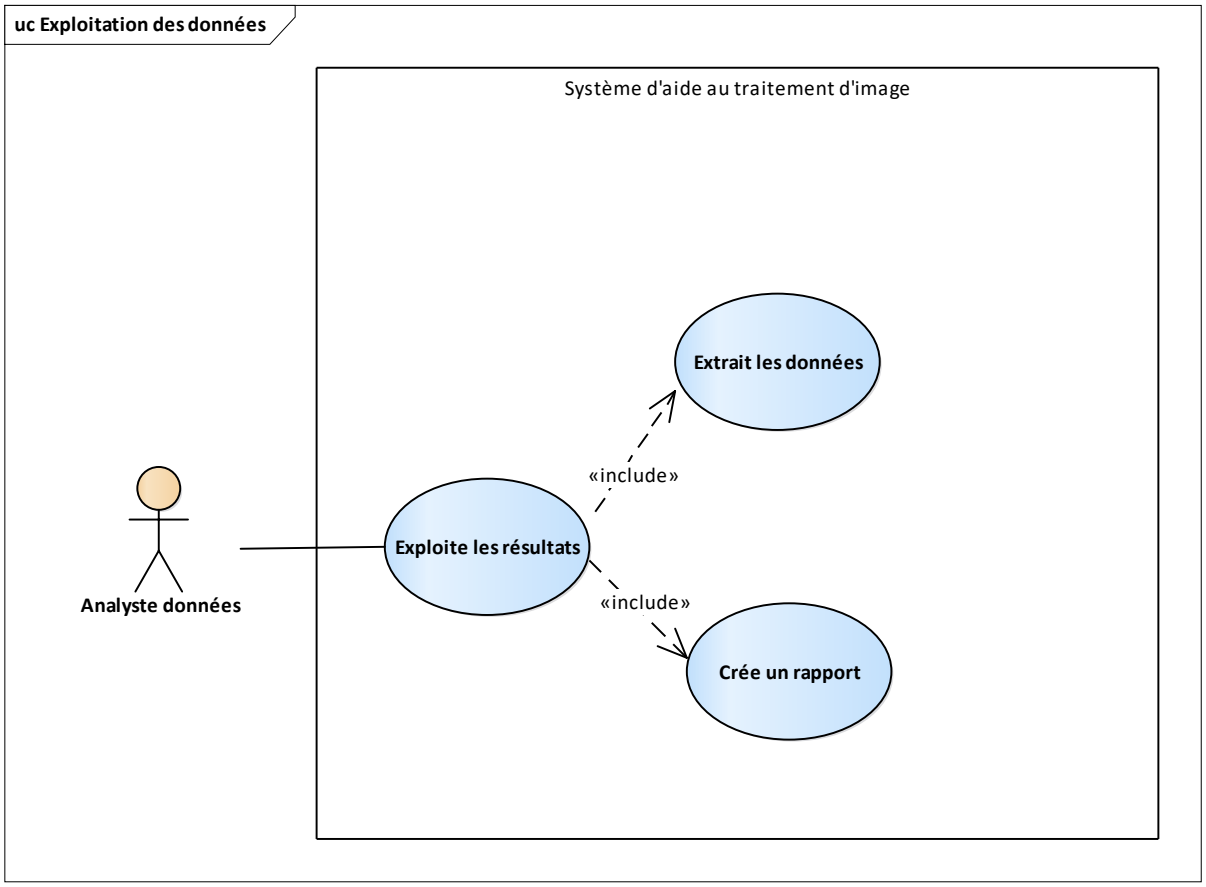
Description

En tant que : expert domaine
Je veux : que le réseau neuronal convolutif fasse une prédiction à la fin de l'exécution d'un processus
Afin de : de prédire les catégories des occurrences.

Critères d'acceptance

Lorsque la position des occurrences est connue, le RNC peut effectuer une prédiction sur base du modèle.

10.8 Exploitation des données



L'analyste doit essentiellement pouvoir consulter les résultats des analyses, soit par un menu directement dans l'application, soit par extraction de ces données vers Excel. Si les données sont stockées en base de données, il pourrait également effectuer des rapports au travers d'outils externes.

10.8.1 US 3.1 - Créer un rapport

Créer un rapport	US 3.1

Description

En tant que : analyste données
Je veux : avoir la possibilité de visualiser les données en relation avec les images analysées
Afin de : de pouvoir les exploiter.

Critères d'acceptance

Grille de données avec filtres sur les catégories et sur la date d'échantillonnage.

10.8.2 US 3.2 - Extraire les données

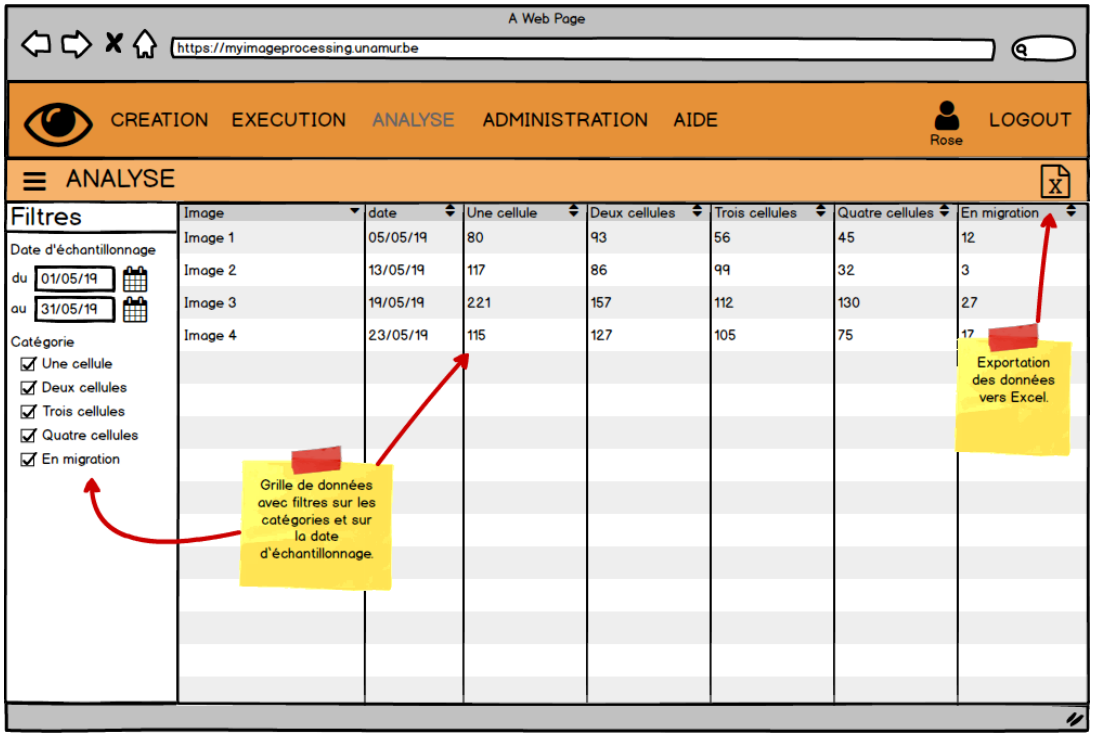
Extraire les données	US 3.2

Description

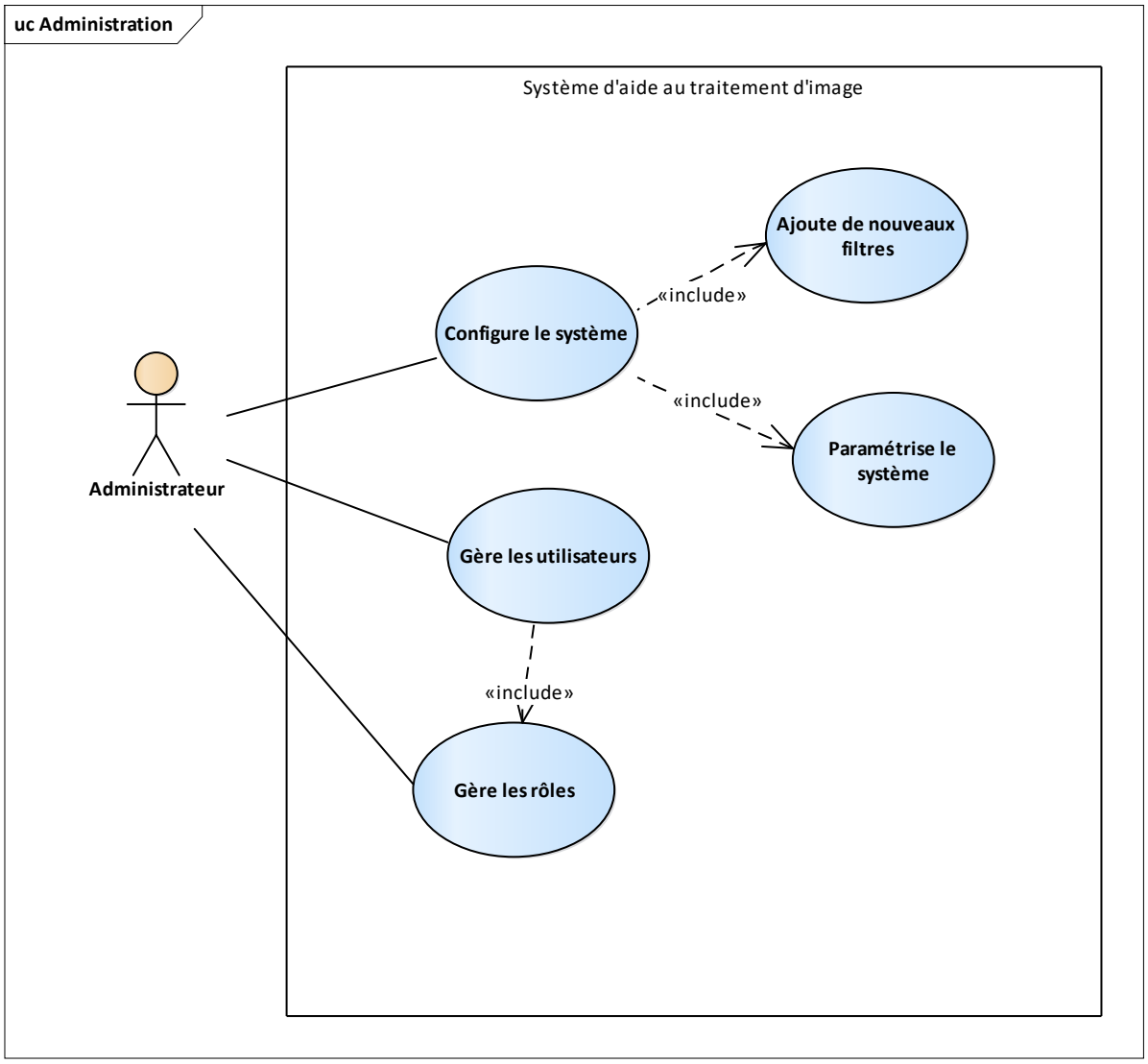
En tant que : analyste données
Je veux : avoir la possibilité d'extraire les données
Afin de : de pouvoir les exploiter.

Critères d'acceptance

Exportation des données vers Excel.



10.9 Administration



La partie administration sort un peu du champ d'application de ce travail, cependant, l'ajout de nouveaux filtres n'est pas sans intérêt. L'idée est de pouvoir ajouter de nouveaux filtres sans avoir à recompiler l'application. Comment ? En utilisant la flexibilité d'un langage interprété tel que Python. Pour ajouter un filtre, nous avons besoin d'un script (Python ou autre) dans lequel l'on introduit des paramètres, une liste des arguments éditables par l'utilisateur, un nom et un code couleur. Bien sûr un tel script ouvre la porte à des problèmes évidents de sécurité. C'est pourquoi seul un administrateur doit pouvoir le faire.

Les autres user stories concernent la gestion des utilisateurs et leur configuration (attribution de rôle), ainsi que la configuration du système (définir une chaîne de connexion vers une base de données par exemple).

10.9.1 US 4.1 - Ajouter de nouveaux filtres

Ajouter de nouveaux filtres	US 4.1

Description

En tant que : administrateur
Je veux : avoir la possibilité d'ajouter de nouveaux filtres
Afin de : de pouvoir offrir de nouvelles options à l'expert en traitement d'image.

Critères d'acceptance

L'administrateur doit pouvoir ajouter ces filtres sans qu'il soit nécessaire de recompiler la solution. Il doit en outre pouvoir leur donner un nom, un code couleur, des paramètres et un script Python.

10.9.2 US 4.2 - Paramétriser le système

Paramétriser le système	US 4.2

Description

En tant que : administrateur
Je veux : avoir la possibilité d'éditer les paramètres de l'application
Afin de : de pouvoir configurer le système et veiller à son bon fonctionnement.

Critères d'acceptance

L'administrateur doit pouvoir configurer l'application sans qu'il soit nécessaire de recompiler la solution.

10.9.3 US 4.3 - Gérer les utilisateurs et leurs rôles

Gérer les utilisateurs et leurs rôles	US 4.3

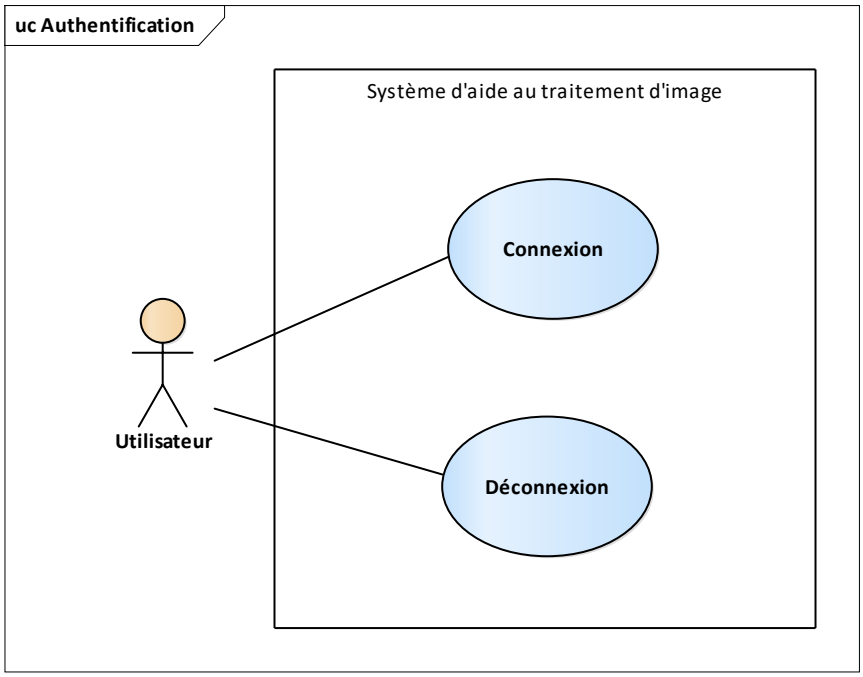
Description

En tant que : administrateur
Je veux : avoir la possibilité d'ajouter, modifier supprimer des utilisateurs et pouvoir leur attribuer des rôles
Afin de : de leur attribuer les accès nécessaires à leur travail.

Critères d'acceptance

L'administrateur doit pouvoir donner des accès aux utilisateurs selon les modules de l'application dont ils ont besoin.

10.10 Authentification



Pour des raisons de sécurité, permettre aux utilisateurs de s'identifier et les cloisonner dans leurs rôles respectifs semble une évidence.

10.10.1 US 5.1 - Connexion

Connexion	US 5.1

Description

En tant que : utilisateur
Je veux : avoir la possibilité de me connecter
Afin de : que le système puisse me donner accès aux modules nécessaires à mon travail.

Critères d'acceptance

L'utilisateur doit pouvoir s'authentifier. Dès qu'il est reconnu par le système, les menus dont il a un droit d'accès se déverrouillent.

10.10.2 US 5.2 - Déconnexion

Déconnexion	US 5.2

Description

En tant que : utilisateur
Je veux : avoir la possibilité de me déconnecter
Afin de : qu'une personne peu scrupuleuse ne puisse pas utiliser mes accès à des fins peu honorables.

Critères d'acceptance

L'utilisateur doit pouvoir se déconnecter. Il sera alors redirigé vers la page d'accueil. Seuls les menus accessibles sans authentification seront déverrouillés.

10.11 Conclusion

Cette analyse donne un aperçu de ce que pourrait être l'application idéale. Bien entendu, le manque d'interaction avec d'éventuels utilisateurs finaux nous a obligé à faire preuve d'imagination et de créativité. Cette analyse n'est ni parfaite ni exhaustive, certaines parties comme l'analyse de données ou l'exécution automatique des workfows ont été négligées voire ignorées. Cependant elle laisse un avant-goût de ce que pourrait être l'application et, pourquoi pas, pourrait servir de base à une analyse plus poussée.

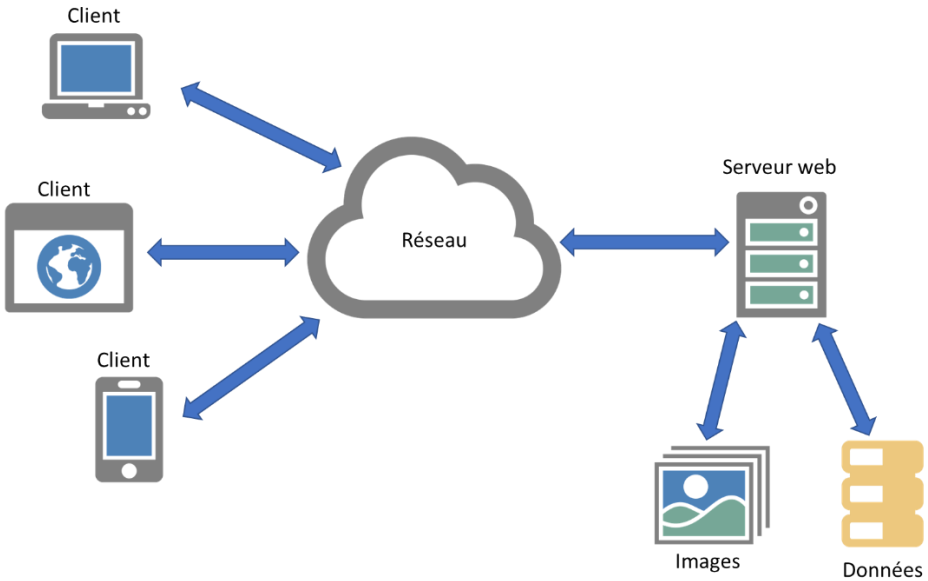
Chapitre 11
Architecture d'une future application

Toute bonne application commence par le choix et l'élaboration de son architecture. Nous aborderons dans ce chapitre l'architecture de l'application idéale (§1.3), qui servira de ligne directrice à la réalisation d'une application démontrant les concepts abordés dans cet ouvrage.

Ce chapitre vise toute personne désireuse de poursuivre ou de s'inspirer de ce travail, en lui exposant les lignes directrices qui ont permis d'opter pour tel ou tel style architectural. Nous prendrons soin de justifier chacun des choix en exposant notre point de vue.

11.1 Architecture REST

Nous avons fait mention à plusieurs reprises d'une application basée sur une architecture client-serveur voir un style architectural REST. REST semble en effet un bon candidat. En effet, les contraintes architecturales d'un service REST vont impliquer certaines qualités non-fonctionnelles intéressantes voire indispensables.



11.1.1 Maintenabilité

Nous sommes dans un cadre académique. Ce travail pourrait donc être repris, en partie ou complètement par d'autres. La maintenabilité est, de ce fait, hautement importante. Une

architecture avec des qualités de séparation des préoccupations doit être envisagée.

L'une des contrainte du style architectural REST est que les responsabilités doivent être séparées entre le client et le serveur. Un découplage de l'interface et des données améliore grandement la maintenabilité.

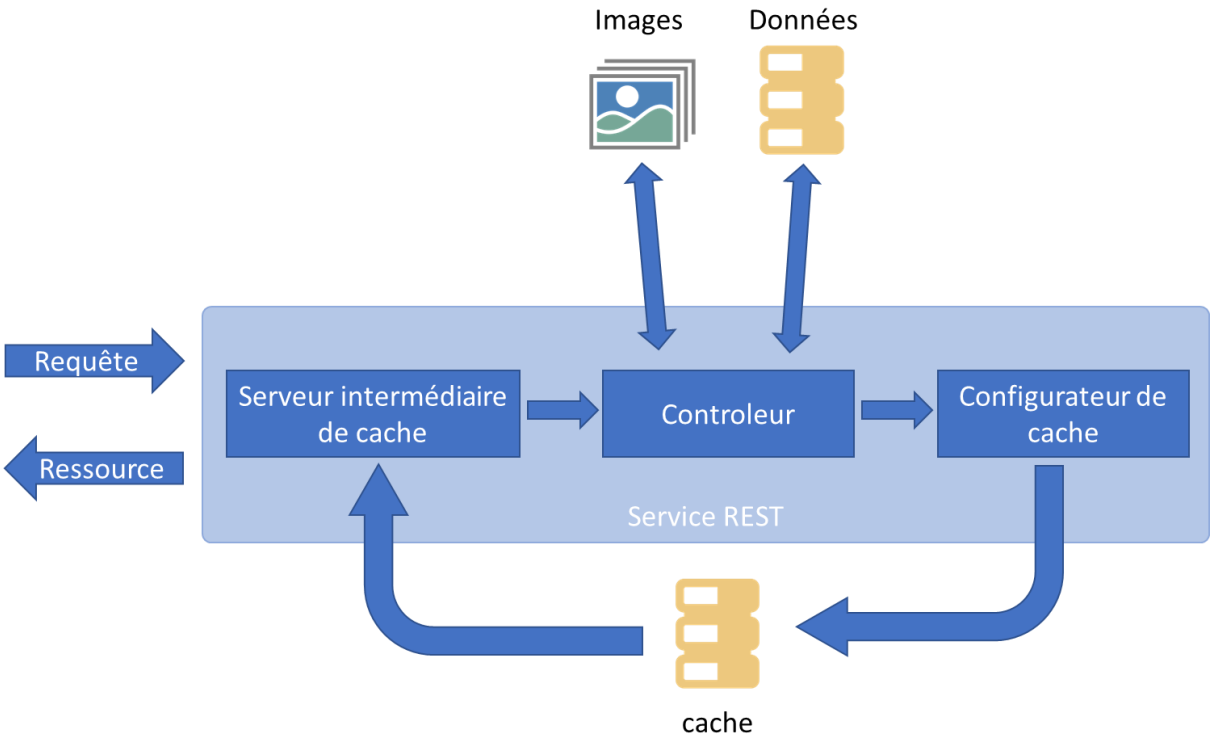
Une autre contrainte du style architectural REST est qu'il doit être implémenté en couche. Toute une série de serveur intermédiaire peuvent être utilisés. Tels qu'un serveur de répartition des charges ou un cache partagé. Cette implémentation en couche augmente la modularité et donc la maintenabilité.

Enfin, la contrainte d'interface uniforme simplifie et découple l'architecture. Ce qui permet à chaque composant d'évoluer séparément.

11.1.2 Performance

La contrainte de mise en cache des données du style architectural REST est essentielle dans le cadre de cette application. Les images ne changeront pas ou peu. Une image est un candidat idéal pour une mise en cache coté client. Les performances s'en verront donc grandement améliorées.

La contrainte architecturale d'implémentation en couche permet, quant à elle, d'implémenter un serveur intermédiaire de mise en cache partagée. On peut donc imaginer un système de cache partagé pour les images nouvellement traitées qui seraient plus sollicitées que les images plus anciennes.

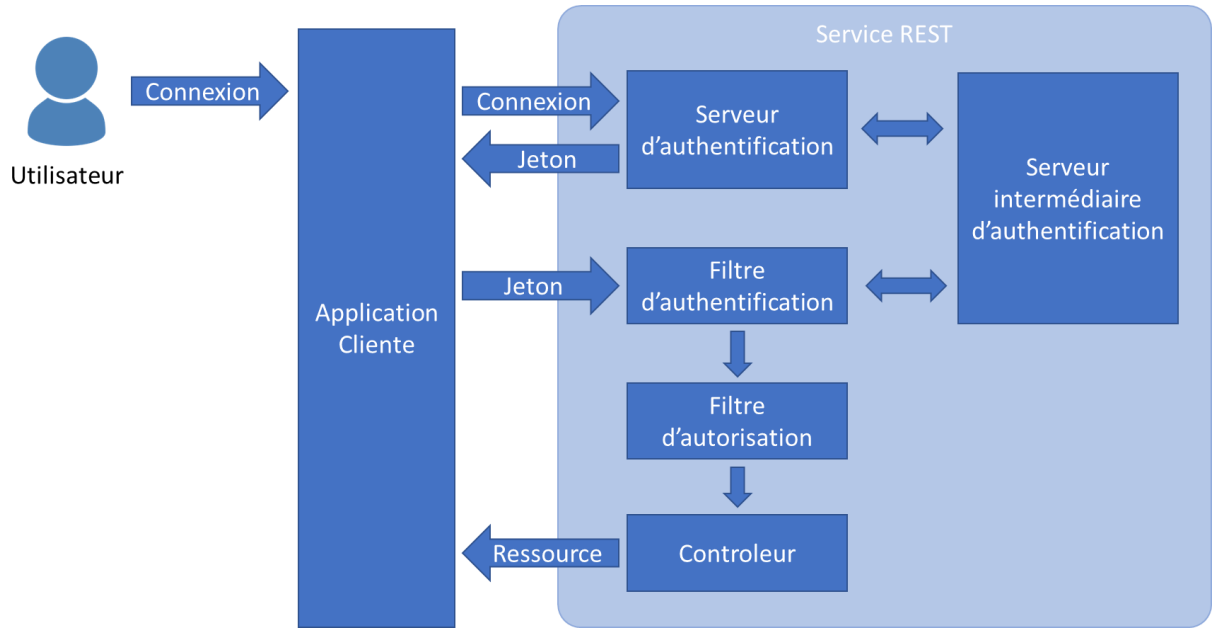


11.1.3 Tolérance aux échecs

La communication entre le client et le serveur se fait sans que le serveur ne conserve l'état de la session du client. L'état de la session est maintenu par le client et transféré au serveur à chaque requête. Les requêtes, entre le client et le serveur, sont donc complètes. Elles ne nécessitent pas de données supplémentaires. La contrainte architecturale sans état offre donc une meilleure tolérance aux échecs.

11.1.4 Sécurité

La contrainte architecturale d'implémentation en couche offre la possibilité d'implémenter des serveurs intermédiaires responsables de la sécurité. Une des couches de sécurité les plus communes est la gestion de l'authentification et la gestion des rôles (et la restriction des droits d'accès liée à ces rôles).



11.2 Le langage

Un des choix les plus cornélien que nous aurions à faire, est le choix du langage pour implémenter le service REST. Notre expérience nous pousserait vers ce que l'on connait, vers les technologies Microsoft, mais est vraiment la meilleure solution ?

11.2.1 Asp.net core

D'un point de vue pratique, nous aurions tendance à partir vers Asp.net core technologie Microsoft open source basée sur .Net core. Cette technologie offre une certaine portabilité, et a l'énorme avantage de pouvoir être développé sous Visual Studio, gratuit sous sa forme Community, qui est probablement l'IDE de développement le plus complet et le plus puissant jamais conçu.

Cependant Asp.net core est une technologie encore assez jeune, qui doit peut-être encore

faire ses preuves.

11.2.2 Java

Une autre possibilité serait d'utiliser Java. Bien sûr Java est portable. Une équipe de développement qui maîtriserait les technologie Java aurait sans doute intérêt à l'utiliser. Ce n'est malheureusement pas notre cas. L'énorme quantité de packages Java, la difficulté de les faire fonctionner entre eux et l'obligation d'utiliser des IDE qui nous paraissent préhistoriques face à Visual Studio sont autant de raisons qui nous fait écarter Java de la liste des langages envisageables.

Bien entendu, nous le répétons, une équipe de développement à l'aise avec ces technologies aurait tout intérêt à les utiliser tellement elles sont riches et diversifiées.

11.2.3 Python

Nous avons déjà abordé la possibilité d'utiliser Python dans les chapitres précédents. Le gros avantage de Python, c'est qu'il est très populaire en environnement scientifique grâce à des librairies dédiées de grandes qualités. Il est devenu un outil incontournable pour les Data Scientists. Nous avons pu nous en rendre compte lors de nos tests effectué sur un réseau neuronal convolutif. Python est aussi hautement portable.

Bien sûr Python offre la possibilité d'implémenter un service web Restful. Il existe plusieurs librairies qui permettent d'implémenter un service REST, dont Flask ou Eve.

Il existe également des wrappers à OpenCV utilisables en Python. Le gros avantage est que Python est un langage interprété, qui permettrait donc d'injecter du code à tout moment. Il serait donc envisageable de permettre à un administrateur de créer de nouveaux filtres sans avoir à recompiler l'application. Python offrirait donc une qualité non-fonctionnelle de maintenabilité intéressante, puisque nous pourrions faire évoluer l'application beaucoup plus simplement. Cette technique serait également possible en C# ou en Java, mais elle demanderait un peu plus d'effort pour être implémentée.

Un désavantage qui pourrait venir à l'esprit lorsque l'on sait que Python est un langage interprété, est qu'il pourrait être moins performant. Cependant Python offre la possibilité à qui le souhaite, d'optimiser certaines parties de code en l'écrivant directement en C++. Ce qui pourrait finalement rendre Python plus performant que Java et C#.

11.2.4 Conclusion

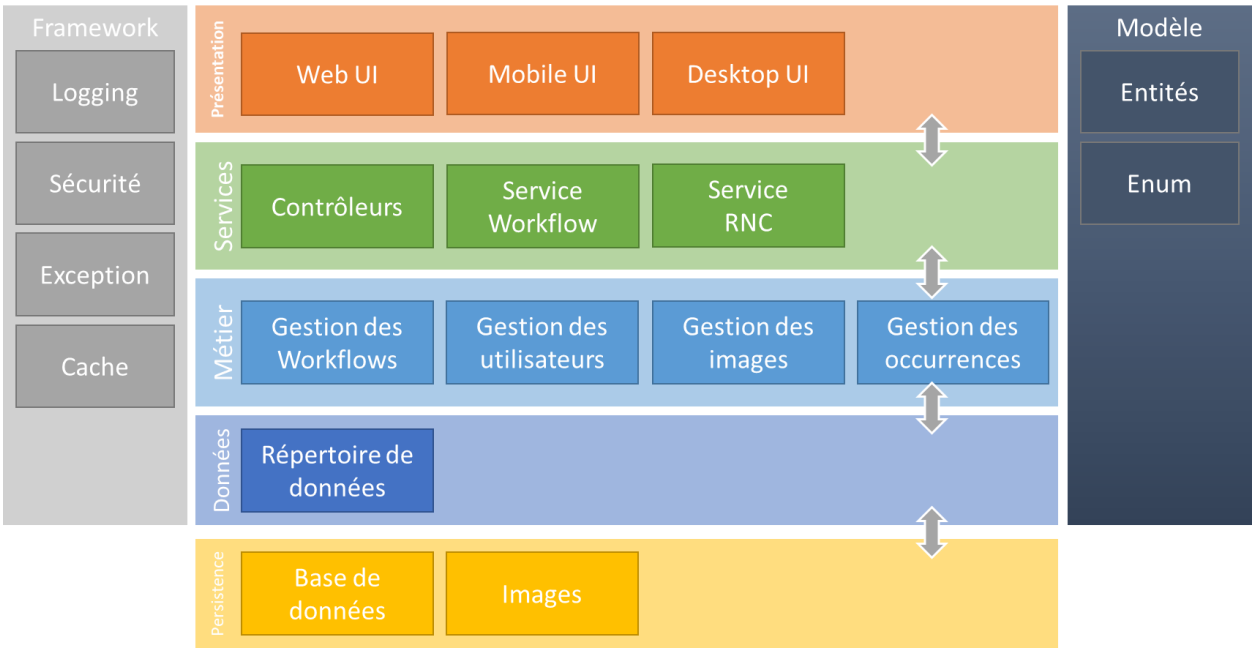
Nous pourrions poursuivre cette comparaison en ajoutant des langages comme C ou C++, mais il nous semble que le candidat idéal est Python, et ce malgré notre affection pour les technologies Microsoft, il parait le plus intéressant pour réaliser ce travail. Mais comme toujours, le choix du langage dépend essentiellement des compétences de l'équipe de développement.

11.3 Architecture en couche (N-Tier)

Comme déjà mentionné plus haut, le cadre académique implique d'avoir une maintenabilité optimale. Bien sûr, la maintenabilité n'est pas l'apanage de l'académique, mais le fait que le travail puisse être repris par une toute autre équipe en fait un point d'attention primordial.

L'une des méthodes les plus communes pour diminuer le couplage entre les différents composants d'une application est d'utiliser une architecture en couche (ou N-Tier). Le principe d'une telle architecture est assez simple. Chaque couche de l'application ne peut communiquer qu'avec la couche strictement inférieure à elle au travers des interfaces mises à disposition par celle-ci. Ainsi, dans notre exemple, la couche présentation ne peut communiquer qu'avec la couche services, qui elle-même ne peut communiquer qu'avec la couche métier…

Les rôles de chaque couche étant bien défini par le biais de leurs interfaces, une modification dans une des couches n'aura pas d'impact sur les autres. Cependant, l'ajout d'une nouvelle fonctionnalité impactera la plupart des couches.



Imaginons un expert domaine effectué une correction sur un dénombrement de cellules. Il effectue cette tache au travers une UI Web. L'UI va contacter le contrôleur (OccurrenceController) au travers du service web, ce contrôleur va appeler le composant en charge d'effectuer cette tâche (OccurrenceManager). Ce composant va ensuite valider les données en entrée et vérifier si la catégorie existe. Si cette catégorie n'existe pas il va appeler l'extracteur de catégories (CategoryExtractor) qui va générer une image sur base de la position de l'occurrence. Le gestionnaire d'occurrence va ensuite enregistrer cette nouvelle catégorie au travers du répertoire adéquat et enfin modifier l'occurrence en lui attribuant sa nouvelle catégorie.

11.4 Application web mono-page

11.4.1 Performance

La performance est une autre qualité non-fonctionnelle qui semble impérative pour cette application. Les images à traiter peuvent être très lourde à charger. Rajouter toute la lourdeur des rafraîchissements de pages web pourrait être catastrophique au niveau de l'expérience utilisateur.

Dans une application web mono-page (single page application), tout le code nécessaire au bon fonctionnement de l'application est chargé une seule fois. C'est ce code qui se charge de modifier l'interface lors des interventions de l'utilisateur. Cela permet à l'utilisateur d'avoir une expérience plus fluide et plus proche de ce que l'on peut trouver dans une application desktop.

11.4.2 Quelle technologie choisir

Les applications web mono-page ont la cote ces dernières années, c'est pourquoi une pléthore de technologies ont vu le jour. Nous n'allons pas ici faire un comparatif rébarbatif qui sortirait du cadre de ce travail. D'ailleurs, peu importe la technologie choisie, elle remplira parfaitement son rôle. Il faut donc en choisir une qui conviendra aux développeurs.

L'une d'entre elle sort, à notre sens, un peu du lot. Il s'agît d'Angular. Le plus gros avantage, à nos yeux, de cette technologie, est le langage TypeScript qui lui est associé. TypeScript est un langage, qui lorsqu'il est « transcompilé », génère du JavaScript. Tout code JavaScript reste d'ailleurs tout à fait valide au sein du code TypeScript. Dans ce langage, il est possible de « typer » les variables, de créer des classes et des interfaces, de recourir à l'injection de dépendances etc… Quelques avantages non négligeables pour des développeurs que nous qualifierons de plus traditionnels.

11.5 Site web adaptatif

11.5.1 Adaptabilité

Une autre qualité non-fonctionnelle apportée par les technologies web est la portabilité. Mais lorsque l'on parle de portabilité pour une interface utilisateur, il est impératif de penser adaptabilité.

Depuis l'avènement des smartphones et autres tablettes, l'importance d'avoir des pages web qui s'adaptent parfaitement à la résolution de l'appareil utilisé, est devenu primordial.

Même s'il est peu probable qu'une application de traitement d'image soit utilisée depuis un smartphone, il est difficile d'ignorer ce paramètre et il nous semble opportun d'en tenir compte.

11.5.2 Quelle technologie choisir

De nouveau, la technologie à choisir dépend principalement de qui va réaliser l'application. Un développeur à l'aise avec le design d'application web, pourrait très bien réaliser cela lui-même au travers des fichiers de style. Cela dépend aussi de la technologie choisie au paragraphe précédent. Angular, par exemple, semble mieux s'accorder avec Material ou Bootstrap. L'important est de tenir compte de cet aspect d'adaptabilité de l'interface utilisateur pour faire en sorte que son expérience soit bonne quel que soit le support qu'il utilise.

11.6 Conclusion

Tout comme l'analyse au chapitre précédent, le manque d'interaction avec les utilisateurs finaux ou les parties prenantes nous a obligé d'imaginer toute une série de besoins, fonctionnels et non-fonctionnels.

Nous faisons ici un brossage des points à prendre en compte en se basant sur les qualités non-fonctionnelles qui nous sembles importants.

La maintenabilité nous paraissait essentiel dans le cadre d'un travail académique, même si cette qualité devrait être un axiome à tout développement.

La portabilité également, pour permettre à toute personne désireuse de reprendre ce travail, de pouvoir le faire sous l'environnement qu'elle souhaite.

La performance est une autre qualité non-fonctionnelle qui a attiré notre attention. Les images à traiter peuvent être très volumineuses. Pour arriver au cadrage des éléments à catégoriser, il faut appliquer plusieurs filtres, dont certains nécessitent un temps de calcul non négligeable. Cependant, des tests plus approfondis et une meilleure définition de ce que l'on sous-entend par performant, auraient été nécessaire pour élaborer une architecture qui répond à ce besoin. Néanmoins, nous avons pu parcourir quelques solutions pour une performance lors des interactions utilisateurs face à l'application.

Chapitre 12
Conclusion

En tant que développeur il est possible de travailler sur des projets intéressants. Mais travailler sur un sujet qui compte et avoir l'impression que nos travaux ont une véritable utilité, c'est beaucoup plus rare.

La majeure partie de ce travail se concentre sur l'étude des techniques de traitement des images numériques, c'est un domaine très vaste. Dès le début nous avons été emballés par le sujet. Nous avions quand même un petit problème, ni l'un ni l'autre ne connaissions le traitement d'image numérique. Bien sûr, nous avions déjà utilisé des outils comme Photoshop ou Gimp pour améliorer ou recadrer une image mais sans plus.

Il nous fallait maitriser le sujet avant d'envisager de solutionner le problème. Nos premières recherches et nos premières lectures nous ont incité à réaliser un travail de synthétisation des différentes techniques. À la suite de ce constat, nous avons essayé de rédiger un état de l'art le plus complet possible. Bien entendu nous n'avons conservé que les techniques les plus populaires et celles qui à nos yeux étaient dignes d'intérêt.

En parallèle à ce travail de synthétisation, nous avons réalisé l'implémentation des différents filtres au fur et à mesure que nous les découvrions. Passer de la théorie à la pratique facilite grandement la compréhension du sujet et donc le travail de synthétisation.

Les chapitres 2 à 8 s'articulent autour de l'état de l'art. Les chapitres 2 et 3 nous a appris à manipuler une image et ses pixels. Les chapitres 4 et 5 nous ont montré comment lisser une image, atténuer ses défauts et enfin en faire ressortir les contours. A la fin de ces quatre chapitres, il était évident qu'il nous manquait quelque chose. Le chapitre 6 nous a permis de combler ces vides, améliorer l'image avant de trouver les contours, améliorer les contours avant de les segmenter. Ce qui nous amène au chapitre 7 concernant la segmentation. La segmentation qui permet grâce aux contours d'obtenir une liste d'éléments de cette image.

L'état de l'art nous a fourni les connaissances et les outils nous permettant de réaliser des expérimentations concrètes. Lors de la rédaction de l'état de l'art, nous avions déjà expérimenté les filtres et nous en avions assemblé quelques un. Nous nous en sommes servi pour illustre ce mémoire. Dans le chapitre 8 nous avons expérimenté différentes méthodes, différents assemblages de filtres et segmentations. Nous n'avons malheureusement pas obtenu de résultat satisfaisant. Nous avons réussi à segmenter l'image, des cellules et des pores sont sortis des résultats. De ces résultats, nous avons ensuite tenté de trouver les noyaux. La détection n'a pas été concluante, elle nous a donnés trop de résultats. Nous avons le sentiment qu'il est presque impossible de résoudre le problème dans l'état actuel des choses. Il faudrait maintenant développer un outil permettant d'assembler les filtres, modifier les paramètres des filtres et visualiser les résultats en temps réel.

Notre première intuition en visualisant les images était d'établir un lien entre le nombre de

cellules et la quantité de pixels mauves. Le chapitre 9 regroupe des expérimentations permettant de réfuter cette intuition. Nous avons expérimenté la régression linéaire sur 50 images et les résultats ne sont pas concluants. Il serait intéressant de relancer une campagne de tests avec dix fois plus d'images. Une information nous est parvenue tardivement. Une même boîte de Pétri est photographiée à de multiples reprises dans le temps. Avec cette information et la progression du nombre de cellules dans le temps, nous aurions pu tenter d'établir une différence du nombre de pixels et établir une corrélation entre cette différence et la différence du nombre de cellules.

La dernière partie du mémoire est consacré à l'étude d'un système d'aide à l'analyse d'image. Un premier jet d'analyse et d'architecture y sont présentés. Bien que nous soyons sur la fin de nos obligations, nous voudrions que l'application voie le jour. Il nous semble important de continuer pour terminer nos expérimentations et peut être offrir une solution au problème. En parallèle au système de détection traditionnel, nous voudrions implémenter une solution d'apprentissage profond.

Chapitre 13
Bibliographie

ARZI, J. (s.d.). *Transformée de Hough sans seuillage.* Récupéré sur TSD Conseil: http://www.tsdconseil.fr/log/opencv/ext/hough/index.html

Aujol , J.-F., Ladjal , S., & Masnou , S. (2013, 10 18). *Exemplar-based inpainting from a variational point of view.* Récupéré sur Institut Camille Jordan: http://math.univ-lyon1.fr/~masnou/fichiers/publications/InpaintingV2.pdf

Bergounioux, M. (2015). *Introduction au traitement mathématique des images - méthodes déterministes.* Orlean: Springer.

Bertalmıo, M., Caselles, V., Masnou, S., & Sapiro, G. (2011, 1 15). *Inpainting.* Récupéré sur Institut Camille Jordan: http://math.univ-lyon1.fr/~masnou/fichiers/publications/survey.pdf

BUSSEUIL, F., & MUTHELET, L. (2010). *Détection de formes par Transformée de Hough.* Récupéré sur MUTHELET Laurent: http://laurentmuthelet.free.fr/pres/doc/formes.pdf

Choqueuse, V. (s.d.). *Pourquoi Python ?* Récupéré sur Python pour les nuls: http://vincentchoqueuse.github.io/Python-pour-les-nuls/chapitre1/index.html

Corpus, M. (s.d.). *Présentation de l'écosystème Python scientifique*. Récupéré sur Makina Corpus: https://makina-corpus.com/blog/metier/2017/presentation-de-lecosysteme-python-scientifique

Do, V., Lebrun, G., Malapert, L., Smet, C., & Tschumperle, D. (2016, 1). *Inpainting d'Images Couleurs par Lissage Anisotrope et Synth`ese de Textures.* Récupéré sur David Tschumperlé - Research scientist in computer sciences: https://tschumperle.users.greyc.fr/publications/tschumperle_rfia06.pdf

Dumoulin, V., & Visin, F. (2018, 1 12). *A guide to convolution arithmetic for deep learning.* Récupéré sur arxiv: https://arxiv.org/pdf/1603.07285.pdf

Esterhuizen , D. (2013, 05 18). *C# How to: Difference Of Gaussians*. Récupéré sur Software by Default.com: https://softwarebydefault.com/2013/05/11/image-edge-detection/#comments

Esterhuizen , D. (2013, 4 26). *C# How to: Image Contrast*. Récupéré sur Software by default.com: https://softwarebydefault.com/2013/04/20/image-contrast/

Esterhuizen , D. (2013, 5 1). *C# How to: Image Convolution*. Récupéré sur Software by default.com: https://softwarebydefault.com/2013/05/01/image-convolution-filters/

Esterhuizen , D. (2013, 5 20). *C# How to: Image Edge Detection*. Récupéré sur Software by default.com: https://softwarebydefault.com/2013/05/11/image-edge-detection/

Esterhuizen , D. (2013, 6 30). *C# How to: Image Median Filter*. Récupéré sur Software by default.com: https://softwarebydefault.com/2013/05/18/image-median-filter/

Fisher, B. (2000, 4 5). *Bilateral Filtering for Gray and Color Images*. Récupéré sur CVonline: http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

Graf, B., & Espic, C. (s.d.). *Transformée de Hough.* Récupéré sur elynxsdk: http://elynxsdk.free.fr/ext-docs/Demosaicing/more/news1/article-graf-espic.pdf

Hasnaoui, Y., Hamad, B., & Fayala, K. (2012). *Détection des droites par la transformée de Hough.* Récupéré sur Ecole supérieur des Sciences de Tunis: https://fr.slideshare.net/khaledfayala/dtection-des-droites-par-la-transforme-de-hough?from_action=save

Körtje, K.-H. (2012, 7 5). *Image Processing for Widefield Microscopy*. Récupéré sur Leica microsystem: https://www.leica-microsystems.com/science-lab/deconvolution/

Laffly , D. (2006, 10). *Régression multiple : principes et exemples d'application.* Récupéré sur Université de Pau et des pays de l'Adour: https://web.univ-pau.fr/RECHERCHE/SET/LAFFLY/docs_laffly/Laffly_regression%20multiple.pdf

Legrand, F. (2014, 3 30). *Transformée de Hough.* Récupéré sur Informatique Appliquée aux Sciences Physiques: http://www.f-legrand.fr/scidoc/srcdoc/opencv/math/hough/hough-pdf.pdf

Leverger, C. (2016, 5 8). *Détectiondescontoursd'uneimage:le filtredeCanny.* Récupéré sur Colin leverger: https://colinleverger.fr/assets/projects/CANNY-COLIN-LEVERGER.pdf

Levkine , H. (s.d.). *http://www.hlevkin.com/articles/SobelScharrGradients5x5.pdf.* Récupéré sur http://www.hlevkin.com.

Lingrand, D. (2008). *Introduction au traitement d'images 2eme édition.* Paris: Vuibert.

Liu , G., Fitsum, A., Shih , K., Wang , T.-C., Tao , A., & Catanzaro, B. (2018, 12 15). *Image Inpainting for Irregular Holes Using Partial Convolutions.* Récupéré sur Cornell University: https://arxiv.org/pdf/1804.07723.pdf

Macanufo, J., Gray, D., & Brown, S. (2014). *Game storming, Jouer pour innover.* Diatenio.

Metz, D. (2010, 08 4). *Quelle différence entre sRGB et Adobe RGB ?* Récupéré sur Blog couleur: http://www.blog-couleur.com/?Quelle-difference-entre-sRGB-et

Mordvintsev, A., & Abid K. (2013). *Image Inpainting*. Récupéré sur OpenCV-Python Tutorials: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_photo/py_inpainting/py_inpainting.html

Ng, A. (2017, 8 1). *Deeplearning.ai*. Récupéré sur youtube: https://www.youtube.com/channel/UCcIXc5mJsHVYTZR1maL5l9w?reload=9

OpenCV Open Source Computer Vision. (s.d.). Récupéré sur OpenCV: https://docs.opencv.org/3.1.0/da/d22/tutorial_py_canny.html

Operation pixel. (s.d.). *La Détection de Contours*. Récupéré sur Opération pixel: http://operationpixel.free.fr/traitementniveaudegris_detection_contour.php

Perrier , V. (2016). *Filtrage non linéaire.* Récupéré sur Grenoble INP: http://chamilo2.grenet.fr/inp/courses/ENSIMAG4MMTI/document/TP/TP2/tp2.pdf

Poncelet, N., & Cornet, Y. (2010). *TRANSFORMÉE DE HOUGH ET DÉTECTION DE LINÉAMENTS SUR IMAGES SATELLITAIRES ET MODÈLES NUMÉRIQUES DE TERRAIN.* Récupéré sur Université de Liège: https://popups.uliege.be/0770-7576/index.php?id=1036&file=1

Pröve, P.-L. (2017, 7 22). *An Introduction to different Types of Convolutions in Deep Learning.* Récupéré sur Toward data science: https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d

Rey, P. (2014). *Traitement des images avec C#5 et WPF.* Paris: Book on Demand Gmbh.

Satya, M. (2015, 02 17). *Blob Detection Using OpenCV*. Récupéré sur https://www.learnopencv.com/: https://www.learnopencv.com/blob-detection-using-opencv-python-c/

Satya, M. (2017, 11 29). *Image Classification using Convolutional Neural Networks in Keras*. Récupéré sur Learn OpenCV: https://www.learnopencv.com/image-classification-using-convolutional-neural-networks-in-keras/

Sinha, U. (2016). *The Canny Edge Detector*. Récupéré sur AI Shack: http://aishack.in/tutorials/canny-edge-detector/

Suzuki, S., & Abe, K. (1983, 12 16). *Topological Structural Analysis of Digitized Binary.* Récupéré sur xuebalib.com: https://pdf-s2.xuebalib.com/xuebalib.com.17233.pdf

Telea , A. (2002, 10 24). *AnImageInpaintingTechniqueBasedon the Fast Marching Method.* Récupéré sur University of Groningen: http://www.cs.rug.nl/~alext/PAPERS/JGT04/paper.pdf

Trucco. (s.d.). *Edge detection* . Récupéré sur University of Nevada, Reno: https://www.cse.unr.edu/~bebis/CS791E/Notes/EdgeDetection.pdf

Université de Toulouse. (20116). *Régression linéaire simple.* Récupéré sur WikiStat: http://wikistat.fr/pdf/st-l-inf-regsim.pdf

Université de Toulouse. (2016). *Régression linéaire multiple.* Récupéré sur WikiStat: https://www.math.univ-toulouse.fr/~besse/Wikistat/pdf/st-l-inf-intRegmult.pdf

University of Crete. (s.d.). *Canny detection.* Récupéré sur The Intelligent Systems Laboratory

(InteLLigence). : http://www.intelligence.tuc.gr/~petrakis/courses/computervision/canny.pdf

Unknow. (s.d.). *Effets du seuillage à hystérésis sur la détection de contours.* Récupéré sur ultra.sdk: http://ultra.sdk.free.fr/docs/Image-Processing/filters/Edges%20Detection/Effets%20du%20seuillage%20%E0%20hyst%E9r%E9sis%20sur%20la%20d%E9tection%20de%20contours.pdf

Wikipedia. (s.d.). *Cercle chromatique*. Récupéré sur Wikipédia: https://fr.wikipedia.org/wiki/Cercle_chromatique

Wikipedia. (s.d.). *Correction gamma*. Récupéré sur Wikipedia: https://fr.wikipedia.org/wiki/Correction_gamma

Wikipedia. (s.d.). *Détection de contours*. Récupéré sur Wikipedia: https://fr.wikipedia.org/wiki/Détection_de_contours

Wikipedia. (s.d.). *Grayscale*. Récupéré sur Wikipedia: https://en.wikipedia.org/wiki/Grayscale

Wikipedia. (s.d.). *Inpainting*. Récupéré sur Wikipedia: https://en.wikipedia.org/wiki/Inpainting

Wikipedia. (s.d.). *Méthode de Otsu (ru)*. Récupéré sur Wikipedia: https://ru.wikipedia.org/wiki/Метод_Оцу

Wikipedia. (s.d.). *Méthode d'Otsu.* Récupéré sur Wikipedia: https://fr.wikipedia.org/wiki/Méthode_d%27Otsu

Wikipedia. (s.d.). *Morphologie mathématique*. Récupéré sur Wikipedia: https://fr.wikipedia.org/wiki/Morphologie_mathématique

Wikipedia. (s.d.). *Multitier architecture*. Récupéré sur Wikipedia: https://en.wikipedia.org/wiki/Multitier_architecture#Three-tier_architecture

Wikipedia. (s.d.). *Representational state transfer*. Récupéré sur Wikipedia: https://fr.wikipedia.org/wiki/Representational_state_transfer

Wikipedia. (s.d.). *Segmentation d'image*. Récupéré sur Wikipedia: https://fr.wikipedia.org/wiki/Segmentation_d%27image

Wikipedia. (s.d.). *Single-page application*. Récupéré sur Wikipedia: https://en.wikipedia.org/wiki/Single-page_application

Wikipedia. (s.d.). *Teinte Saturation Valeur*. Récupéré sur Wikipedia: https://fr.wikipedia.org/wiki/Teinte_Saturation_Valeur

Wikipedia. (s.d.). *Transformée de Hough.* Récupéré sur Wikipedia: https://fr.wikipedia.org/wiki/Transformée_de_Hough

Yuan, E. (2013, 10 5). *Bilateral Filtering*. Récupéré sur Eric Yuan's Blog: http://eric-

yuan.me/bilateral-filtering/

Chapitre 14
Annexes

14.1 Classes de convolution

Classe Convolution

```
public class Convolution
{
    public enum ConvolutionColorSpace
    {
        RGB,
        GrayScale,
        Auto
    }


    public static ConvolutionResult Convolve<T>(Bitmap sourceBitmap, T filter, bool
              computeDirection=false, ConvolutionColorSpace space = ConvolutionColorSpace.Auto)
        where T : ConvolutionFilterBase
    {
        if (space == ConvolutionColorSpace.Auto)
            space = TryToDetermineSpace(sourceBitmap);

        return space == ConvolutionColorSpace.RGB
            ? ConvolveRGB(sourceBitmap, filter, computeDirection)
            : ConvolveGrayScale(sourceBitmap, filter, computeDirection);
    }

    public static ConvolutionColorSpace TryToDetermineSpace(Bitmap source)
    {
        BitmapData data = source.LockBits(new Rectangle(0, 0, source.Width, source.Height),
                                  ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);

        IntPtr ptr = data.Scan0;
        int tenPercent = source.Height / 10;
        int gray = 0;

        // Declare an array to hold the bytes of the bitmap.
        int bytes = Math.Abs(data.Stride) * tenPercent;
        byte[] rgb = new byte[bytes];

        // Copy the RGB values into the array.
        Marshal.Copy(ptr, rgb, 0, bytes);

        var bidon = data.Stride - data.Width * 3;
        for (int i = 0; i < rgb.Length-bidon; i += 3)
        {
            if (rgb[i] == rgb[i + 1] && rgb[i + 1] == rgb[i + 2])
                gray += 3;
        }

        source.UnlockBits(data);
        return rgb.Length - bidon == gray
                ? ConvolutionColorSpace.GrayScale
                : ConvolutionColorSpace.RGB;
    }


    public static ConvolutionResult ConvolveRGB<T>(Bitmap sourceBitmap, T filter
                                                   , bool computeDirection)
              where T : ConvolutionFilterBase
    {
        BitmapData sourceData = sourceBitmap.LockBits(new Rectangle(0, 0,
```

```
                            sourceBitmap.Width, sourceBitmap.Height),
                            ImageLockMode.ReadOnly, PixelFormat.Format24bppRgb);

        byte[] pixelBuffer = new byte[sourceData.Stride * sourceData.Height];
        byte[] resultBuffer = new byte[sourceData.Stride * sourceData.Height];

        Marshal.Copy(sourceData.Scan0, pixelBuffer, 0, pixelBuffer.Length);
        sourceBitmap.UnlockBits(sourceData);

        double[] blue = new double[filter.Kernels.Count()];
        double[] green = new double[filter.Kernels.Count()];
        double[] red = new double[filter.Kernels.Count()];
        var res = new ConvolutionResult();

        if (computeDirection)
            res.Directions = new byte[sourceData.Stride * sourceData.Height];

        int padding = filter.Padding;
        int calcOffset = 0;
        int byteOffset = 0;
        int stride = sourceData.Stride;
        int height = sourceBitmap.Height;
        int width = sourceBitmap.Width;
        bool twoKernel = filter.Kernels.Count()==2;
        var kernelCount = filter.Kernels.Count();
        var kernels = filter.Kernels;

        //Foreach rows
        for (int rowIndex = padding; rowIndex < height - padding; rowIndex++)
        {
            //foreach lines
            for (int lineIndex = padding; lineIndex < width - padding; lineIndex++)
            {
                byteOffset = rowIndex * stride + lineIndex * 3;
                //foreach kernel
                for (int i = 0; i < kernelCount; i++)
                {
                    var kernel = kernels[i];
                    var k = kernel.Kernel;
                    var factor = kernel.Factor;
                    blue[i] = red[i] = green[i] = 0;
                    //foreach row in kernel
                    for (int filterRowIndex = -padding; filterRowIndex <= padding; filterRowIndex++)
                    {
                        //foreach line in kernel
                        for (int filterLineIndex = -padding; filterLineIndex <= padding;
                                filterLineIndex++)
                        {
                            calcOffset = byteOffset +
                                         (filterLineIndex * 3) +
                                         (filterRowIndex * stride);

                            blue[i] += (double)(pixelBuffer[calcOffset]) *
                                     k[filterRowIndex + padding, filterLineIndex + padding];

                            green[i] += (double)(pixelBuffer[calcOffset + 1]) *
                                      k[filterRowIndex + padding, filterLineIndex + padding];

                            red[i] += (double)(pixelBuffer[calcOffset + 2]) *
                                    k[filterRowIndex + padding, filterLineIndex + padding];
                        }
                    }

                    blue[i] = filter.ForceAbsoluteValue
                        ? Math.Abs(factor * blue[i])
                        : factor * blue[i];
                    blue[i] = blue[i] > 255 ? 255 : blue[i] < 0 ? 0 : blue[i];

                    green[i] = filter.ForceAbsoluteValue
                        ? Math.Abs(factor * green[i])
                        : factor * green[i];
                    green[i] = green[i] > 255 ? 255 : green[i] < 0 ? 0 : green[i];

                    red[i] = filter.ForceAbsoluteValue
```

```
                        ? Math.Abs(factor * red[i])
                        : factor * red[i];
                    red[i] = red[i] > 255 ? 255 : red[i] < 0 ? 0 : red[i];
                }

                if (twoKernel)
                {
                    if (computeDirection)
                    {
                        var x = (red[0] + green[0] + blue[0]) / 3;
                        var y = (red[1] + green[1] + blue[1]) / 3;
                        res.Directions[byteOffset] = ToDirection(x, y);
                    }

                    resultBuffer[byteOffset] = (byte)(Math.Sqrt(Math.Pow(blue[0], 2) +
                                                Math.Pow(blue[1], 2)) + filter.Offset);
                    resultBuffer[byteOffset + 1] = (byte)(Math.Sqrt(Math.Pow(green[0], 2) +
                                                    Math.Pow(green[1], 2)) + filter.Offset);
                    resultBuffer[byteOffset + 2] = (byte)(Math.Sqrt(Math.Pow(red[0], 2) +
                                                    Math.Pow(red[1], 2)) + filter.Offset);
                }
                else
                {
                    if (computeDirection)
                        res.Directions[byteOffset] = ToDirection(red, green, blue);

                    resultBuffer[byteOffset] = (byte)(Max(blue) + filter.Offset);
                    resultBuffer[byteOffset+1] = (byte)(Max(green) + filter.Offset);
                    resultBuffer[byteOffset+2] = (byte)(Max(red) + filter.Offset);
                }
            }
        }

        Bitmap resultBitmap = new Bitmap(sourceBitmap.Width, sourceBitmap.Height);

        BitmapData resultData = resultBitmap.LockBits(new Rectangle(0, 0,
                                resultBitmap.Width, resultBitmap.Height),
                                ImageLockMode.WriteOnly, PixelFormat.Format24bppRgb);

        Marshal.Copy(resultBuffer, 0, resultData.Scan0, resultBuffer.Length);
        resultBitmap.UnlockBits(resultData);

        res.Output = resultBitmap;
        return res;
    }

    private static double Max(double[] blue)
    {
        var l = blue.Length;
        double d = 0;
        for (int i = 0; i < l; i++)
        {
            if (d < blue[i])
                d = blue[i];
        }

        return d >= 0 ? d : d * -1;
    }

    private static byte ToDirection(double x, double y)
    {
        var dir = Math.Atan(Math.Abs(y) / Math.Abs(x))*100;

        if (Math2.Between(dir, 22.5, 67.5) || Math2.Between(dir, 202.5, 247.5)) return 45;
        if (Math2.Between(dir, 67.5, 112.5) || Math2.Between(dir, 247.5, 297.5)) return 90;
        if (Math2.Between(dir, 112.5, 157.5) || Math2.Between(dir, 297.5, 337.5)) return 135;
        if (dir == 0) return 255;

        return 0;
    }

    private static byte ToDirection(IEnumerable<KernelItem> kernels, double[] gray)
    {
        byte b=255;
```

```
        double maxima=0;
        var kernelItems = kernels as KernelItem[] ?? kernels.ToArray();
        for (int i = 0; i < kernelItems.Count(); i++ )
        {
            if (gray[i] > maxima)
            {
                maxima = gray[i];
                b = kernelItems[i].Orientation.ToValue();
            }
        }
        return b;
    }

    private static byte ToDirection(double[] red, double[] green, double[] blue)
    {
        var dir = new double[red.Length];
        for (int i = 0; i < red.Length; i++)
            dir[i] = (red[i] + green[i] + blue[i]) / 3;

        var index = dir.ToList().IndexOf(dir.Max());
        var x = index % 2 == 0 ? index : index - 1;
        var y = x + 1;
        return ToDirection(dir[x], dir[y]);
    }
}
```

Classe ConvolutionFilterBase

```
/**
* @overview Classe de base d'un filtre de convolution, IMMUTABLE
* @specfields Kernel:double[,] la matrice représentant le filtre
* @specfields name:string le nom du filtre
* @derivedfields Size:double la taille du kernel (largeur ou hauteur)
* @derivedfields Padding:double (la taille du kernel-1) divisé par deux
* @invariant Le kernel est carré, la hauteur et largeur sont identique
*/
public abstract class ConvolutionFilterBase
{
    // La fonction d'abstraction est
    // FA(c) = c.Kernel[i,j]  | 0 <= i,j < c.kernel.Lenght

    // Invariant de représentation
    //  kernel != null
    //  && kernel.width == kernel.height
    //  && Name != null
    //  && Mltiplier > 0
    //  && Multilier = ∀ int i,j Σ kernel[i,j]
    //  && Size == kernel.lenght
    //  && Padding > 0 && Padding == (kernel.Length - 1) / 2

    public abstract string Name { get; }

    private readonly List<KernelItem> kernels;
    public KernelItem[] Kernels => kernels.ToArray();

    public int Size => Kernels.First().Size;

    public int Padding => (this.Size - 1) / 2;

    public bool ForceAbsoluteValue { get; set; }

    public byte Offset { get; set; }

    protected ConvolutionFilterBase()
    {
        ForceAbsoluteValue = true;
        Offset = 0;
        this.kernels = new List<KernelItem>();
        InitKernels();
    }
```

```
    /**
    * @requires kernels!= null
    * @modifies Kernels est modifier pour contenir le(s) kernel(s)
    * @effects Initialise la liste de(s) kernel(s)
    */
    protected abstract void InitKernels();

    protected void AddKernel(double[,] kernel, double factor, KernelOrientation orientation)
    {
        KernelItem k = new KernelItem(orientation, factor, kernel);
        this.kernels.Add(k);
    }

    protected void ClearKernel()
    {
        this.kernels.Clear();
    }
}
```

Classe ConvolutionResult

```
public class ConvolutionResult
{
    public Bitmap Output { get; set; }
    public byte[] Directions { get; set; }

    public Bitmap DirectionToBitmap()
    {
       Bitmap output = new Bitmap(this.Output);
       BitmapData data = output.LockBits(new Rectangle(0, 0, output.Width, output.Height)
                                 , ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);
       IntPtr ptr = data.Scan0;

       // Declare an array to hold the bytes of the bitmap.
       int bytes = Math.Abs(data.Stride) * output.Height;
       byte[] rgb = new byte[bytes];

       // Copy the RGB values into the array.
       Marshal.Copy(ptr, rgb, 0, bytes);

       int j = 0;
       for (int i = 0; i < rgb.Length; i += 3)
       {
            rgb[i] = 0;//blue
            rgb[i + 1] = 0; //green
            rgb[i + 2] = 0;//red

            if (this.Directions[j] == 0)
            {
                rgb[i] = 255;//blue
                rgb[i + 1] = 255; //green
                rgb[i + 2] = 255;//red
            }

            if ( this.Directions[j] == 45) rgb[i + 2] = 255;//red
            if(this.Directions[j] == 90) rgb[i + 1] = 255; //green
            if(this.Directions[j] == 135) rgb[i] = 255;//blue

            j+=3;
        }

        //Copy changed RGB values back to bitmap
        Marshal.Copy(rgb, 0, ptr, bytes);
        output.UnlockBits(data);
        return output;
    }
}
```

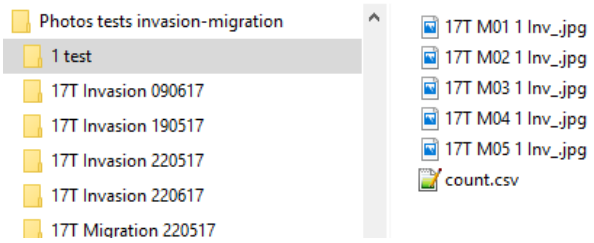
14.2 Tableau de données pour les régressions linéaires

Les données suivantes ont été obtenues grâce à l'ensemble des images et du décompte des cellules fournies par Mlle. Marie Fourrez de la Faculté de médecine. Les nombres de pixels ont été dénombrés après l'application de différents filtres.

Name	Count	ColorIsolation	ColorThreshold	Gray	kmean	type	original
M01 1	2750	5553466	1148752	3834137	1148755	invasion	140113344
M02 1	2910	6606010	1900908	4342579	1900901	invasion	138463324
M03 1	75	3095343	6100	1953069	6100	invasion	141585240
M04 1	11331	18521545	10515093	13015238	10515047	invasion	141414480
M05 1	8524	17061938	8936224	11221507	8936170	invasion	140698600
M01 2	1128	5071715	487737	3508507	487343	invasion	143555922
M02 2	596	4037970	56708	3018897	56708	invasion	142503444
M03 2	2442	5701681	1032044	3947586	1031565	invasion	142062512
M04 2	3262	7194678	1042535	4650376	1042540	invasion	144744960
M05 2	1973	5378725	925125	3245330	924768	invasion	141560640
M01 1	137	3707177	18416	2451940	18416	invasion	140154300
M02 1	1225	4952565	368230	3018007	368230	invasion	141034740
M03 1	5853	13025717	6456867	8948812	6456861	invasion	140995560
M04 1	389	3678159	58884	2405831	58884	invasion	140696640
M05 1	167	3864574	4085	2465414	4085	invasion	140985792
M01 3	249	4338101	105803	3069269	105803	invasion	139564512
M02 3	1209	4530484	714157	2895925	714153	invasion	140334012
M03 3	5878	11583644	5536734	7587122	5536438	invasion	138426178
M04 3	246	3404676	29498	2423515	29498	invasion	137324352
M05 3	761	4265700	425405	3004359	425326	invasion	139913400
M01 3	230	3730545	145146	2665664	145146	migration	139841280
M02 3	240	3036755	53620	1874837	53620	migration	139274208
M03 3	2558	7138031	3182218	4941381	3182061	migration	139853952
M04 3	598	4047085	173530	2835763	173530	migration	140817589
M05 3	494	3237816	211678	2067086	211678	migration	139663161
M01 2	491	2792760	9576	1762152	9576	invasion	139039186
M02 2	2968	5658527	1149415	3380159	1149401	invasion	138620000
M03 2	104	2641057	12459	1625968	12459	invasion	141229287
M04 2	7632	10888429	3918562	7113983	3917355	invasion	139304252
M05 2	3344	7662233	1917092	4784046	1916219	invasion	140985792
M01 1	852	3388349	675	1737135	675	invasion	141554304
M02 1	2796	7848522	2639494	5792567	2639284	invasion	138238360
M03 1	127	3557010	31426	2317248	31426	invasion	141845184
M04 1	4940	8589031	2935632	5263818	2935565	invasion	139989168
M05 1	863	4226138	142756	2630112	142756	invasion	138556440
M01 3	35	3677769	2065	1892518	1936	invasion	141400632
M02 3	3	3265652	0	1921483	0	invasion	141987456
M03 3	40	3918432	3662	1946923	3662	invasion	139849920
M04 3	232	3914688	18580	2090338	18580	invasion	140842368
M05 3	9	3588673	0	1769542	0	invasion	139934259
M01 1	128	1156194	15757	688263	15757	migration	138632560
M02 1	929	1294408	133478	773393	133478	migration	140418144
M03 1	25	1519970	0	1023146	0	migration	140530230
M04 1	758	1666969	53834	1118481	53834	migration	142273584
M05 1	1363	1987866	218329	1254061	218329	migration	140900398
M01 2	2347	6402242	1158773	4526734	1158705	migration	140171598
M02 2	5524	11962351	5964079	8583469	5963196	migration	139316760
M03 2	12232	21877272	11395928	15097695	11393440	migration	140303541
M04 2	5405	11193432	2623943	7054794	2621916	migration	140238644
M05 2	16708	33129481	24563838	24586133	24557295	migration	141254496

14.3 Script C# de création des données pour les régressions linéaires

Le code ci-dessous parcourt une structure de répertoire contenant une série d'images dont le nombre de cellules est connue et placé dans un fichier CSV.



Chaque image subit un filtre gaussien, et dans l'ordre :

- Une isolation de couleur rouge-bleu et un dénombrement
- Un seuillage de couleur et un dénombrement
- Un filtre en niveaux de gris et un dénombrement
- Un filtre KMean et un dénombrement

Implémentation OpenCV

```
[TestMethod]
public void CountPixelInDir()
{
    var path = @"D:\repos\Photos tests invasion-migration";

    var directories = Directory.GetDirectories(path);

    //parcourt les répertoires
    foreach (var directory in directories)
    {
        //Retrouve les fichiers
        var files = Directory.GetFiles(directory, "*_.jpg");

        //Lit le fichier csv contenant les résultats connus
        var imageCount = this.GetCsvLines(Path.Combine(directory, "count.csv"), false);
        File.Delete(Path.Combine(directory, "dataset.csv"));
        File.AppendAllText(Path.Combine(directory, "dataset.csv"),
              "Name,Count,ColorIsolation,ColorThreshold,Gray,kmean" + Environment.NewLine);

        foreach (KeyValuePair<string, int> pair in imageCount)
        {
            //Retrouve l'image à partir de la liste des fichiers connus
            var imageName = files.FirstOrDefault(f => f.Contains(pair.Key));

            if (imageName != null)
            {
                //Filtre gaussien
                Mat v = Cv2.ImRead(imageName);

                //Filtre gaussien 9x9
                Cv2.GaussianBlur(v, v, new OpenCvSharp.Size(5, 5), 0, 0, BorderTypes.Default);

                var res = new Mat();
                //Filtre rouge bleu
                res = this.ColorIsolationCount(v, directory, pair.Key);
                var cntCI = PixelCount.ParallelCount(res, Scalar.Black);

                //Seuillage couleur
                res = this.ColorThresholdCount(v, directory, pair.Key);
                var cntTHC = PixelCount.ParallelCount(res, Scalar.Black);

                //Niveaux de gris
                res = this.GrayThresholdCount(v, directory, pair.Key);
                var cntGray = PixelCount.ParallelCount(res, Scalar.Black);
```

```
                //Filtre kmean
                res = this.KMeanThresholdCount(v, directory, pair.Key);
                var cntkmean = PixelCount.ParallelCount(res, Scalar.Black);

                //Réalisation du dataset
                File.AppendAllText(Path.Combine(directory, "dataset.csv"), pair.Key + ","
                     + pair.Value + "," + cntCI+ "," + cntTHC + "," + cntGray + "," + cntkmean
                     + Environment.NewLine);
            }
        }
    }
}

private Mat ColorThresholdCount(Mat v, string directory, string name)
{
    Mat output = new Mat();
    Mat mask = new Mat();

    Mat hsv = new Mat();
    //Convertion RGB vers HSB
    Cv2.CvtColor(v, hsv, ColorConversionCodes.BGR2HSV);
    Cv2.ImWrite(Path.Combine(directory, @".\010" + name + ".png"), hsv);

    //Déclaration des seuil de couleurs HSB
    var lowerColor = new Scalar(0, 100, 0);
    var higherColor = new Scalar(180, 255, 255);

    //Seuillage par bande de couleur
    Cv2.InRange(hsv, lowerColor, higherColor, mask);
    Cv2.ImWrite(Path.Combine(directory, @".\020" + name + ".png"), mask);
    hsv.Dispose();

    //Erosion
    mask = mask.Erode(Cv2.GetStructuringElement(MorphShapes.Ellipse, new Size(13, 13)));
    Cv2.ImWrite(Path.Combine(directory, @".\030" + name + ".png"), mask);
    //Dilatation
    mask = mask.Dilate(Cv2.GetStructuringElement(MorphShapes.Ellipse, new Size(17, 17)));
    Cv2.ImWrite(Path.Combine(directory, @".\040" + name + ".png"), mask);

    //Intersection du masque et de l'image originale
    Cv2.BitwiseAnd(v, v, output, mask);
    mask.Dispose();

    //Enregistrement de l'image de sortie
    Cv2.ImWrite(Path.Combine(directory, @".\050" + name + ".png"), output);

    return output;
}

private Mat GrayThresholdCount(Mat v, string directory, string name)
{
    Mat output = new Mat();
    Mat mask = new Mat();

    Mat gray = new Mat();
    //Convertion RGB vers HSB
    Cv2.CvtColor(v, gray, ColorConversionCodes.BGR2GRAY);
    Cv2.ImWrite(Path.Combine(directory, @".\110"+name+".png"), gray);


    //Seuillage gris
    Cv2.Threshold(gray, output, 120, 255, ThresholdTypes.TozeroInv);
    Cv2.ImWrite(Path.Combine(directory, @".\120" + name + ".png"), output);
    gray.Dispose();

    return output;
}

private Mat KMeanThresholdCount(Mat v, string directory, string name)
{
    Mat output = new Mat();
    Mat mask = new Mat();
```

```
    Mat hsv = new Mat();
    //Convertion RGB vers HSB
    Cv2.CvtColor(v, hsv, ColorConversionCodes.BGR2HSV);
    Cv2.ImWrite(Path.Combine(directory, @".\210" + name + ".png"), hsv);

    //Déclaration des seuil de couleurs HSB
    var lowerColor = new Scalar(0, 100, 0);
    var higherColor = new Scalar(180, 255, 255);

    //Seuillage par bande de couleur
    Cv2.InRange(hsv, lowerColor, higherColor, mask);
    Cv2.ImWrite(Path.Combine(directory, @".\220" + name + ".png"), mask);
    hsv.Dispose();

    //Erosion
    mask = mask.Erode(Cv2.GetStructuringElement(MorphShapes.Ellipse, new Size(13, 13)));
    Cv2.ImWrite(Path.Combine(directory, @".\230" + name + ".png"), mask);
    //Dilatation
    mask = mask.Dilate(Cv2.GetStructuringElement(MorphShapes.Ellipse, new Size(17, 17)));
    Cv2.ImWrite(Path.Combine(directory, @".\240" + name + ".png"), mask);

    //Intersection du masque et de l'image originale
    Cv2.BitwiseAnd(v, v, output, mask);

    //Enregistrement de l'image de sortie
    Cv2.ImWrite(Path.Combine(directory, @".\250" + name + ".png"), output);

    //Calcul du Kmean
    KMeans.Proceed(output, mask, 2, true, Scalar.Black);

    //Enregistrement de l'image de sortie
    Cv2.ImWrite(Path.Combine(directory, @".\260" + name + ".png"), mask);

    return mask;
}

private Mat ColorIsolationCount(Mat v, string directory, string name)
{
    //MARCHE PAS...
    var res = ColorIsolation.ParallelIsolate(v, false, true, false);
    Cv2.ImWrite(Path.Combine(directory, @".\310" + name + ".png"), res);
    res = ZeroThresholdingFilter.ParallelApply(res, 160, true);
    Cv2.ImWrite(Path.Combine(directory, @".\320" + name + ".png"), res);

    return res;
}
```

14.4 Script C# des tests finaux

Implémentation OpenCV

```
public class FinalTest
{
    private int _counter = 0;
    public TestContext TestContext { get; set; }
    public string TestDir { get; set; }
    public string TestRoiDir { get; set; }
    Scalar[] Colors = {Scalar.Red, Scalar.Lime, Scalar.Cyan, Scalar.Gold, Scalar.Fuchsia};

    [TestInitialize]
    public void Init()
    {
        this.TestDir = @".\" + this.TestContext.TestName;
        this.TestRoiDir = @".\" + this.TestContext.TestName + @"\ROI";
        //if (Directory.Exists(this.TestRoiDir))
        //    Directory.Delete(this.TestRoiDir, true);
        if (Directory.Exists(this.TestDir))
            Directory.Delete(this.TestDir, true);

        //Directory.CreateDirectory(this.TestDir);
```

```
        Directory.CreateDirectory(this.TestRoiDir);
    }

    [TestMethod]
    public void ThresholdTest()
    {
        Bench.ComputeCpuMemoryUsage(() =>
        {
            //Lecture
            //Mat original = Cv2.ImRead(@".\Echantillon.png");
            Mat original = Cv2.ImRead(@"D:\repos\Photos tests invasion-migration\17T Invasion
090617\17T M01 1 Inv_.jpg");

            Mat output = new Mat();

            //Amélioration des contrastes
            output = this.ContrastCorrection(original, 1.1);

            //Correction gamma
            output = this.GammaCorrection(output, 0.8);

            //Filtre bilatéral de lissage
            output = this.BilateralFilter(output, 19);

            //Trouver les cerlces
            var circles = this.DetectCircle(output);
            //Create matrice for the mask
            Mat circleMask = new Mat(output.Size(), MatType.CV_8U);
            this.DrawCircle(circleMask, circles, 5);

            //Lissage par InPainting
            output = this.Inpaint(output, circleMask, 30);
            circleMask.Dispose();

            //Seuillage par bande de couleur
            Scalar lowerColor = new Scalar(0, 100, 0);
            Scalar higherColor = new Scalar(180, 255, 255);
            output = this.ColorThreshold(output, lowerColor, higherColor, 13, 17);

            //Trouver cellules ?
            var contours = this.FindContour(output);
            foreach (var contour in contours)
            {
                var x1 = contour.Min(c => c.X);
                var y1 = contour.Min(c => c.Y);
                var x2 = contour.Max(c => c.X);
                var y2 = contour.Max(c => c.Y);
                if (Math2.EuclideanDistance(x1, y1, x2, y2) > 25)
                {
                    var r = new Rect(x1, y1, x2 - x1, y2 - y1);
                    var cell = new Mat(output, r);
                    this.SaveROI(cell);
                }
            }

            var cnt1 = contours.Length;

            //Seuillage par bande de couleur
            lowerColor = new Scalar(130, 0, 0);
            higherColor = new Scalar(180, 255, 175);
            output = this.ColorThreshold(output, lowerColor, higherColor);

            //Dilatation et erosion
            this.Dilate(7, output);

            //Trouver les contours
            contours = this.FindContour(output);

            //Identifier les noyaux
            for (int i = 0; i < contours.Length; i++)
                Cv2.DrawContours(original, contours, i, this.Colors[i % 5], -1);

            this.Save(original);
```

```
            var cnt2 = contours.Length;

            //var p = Path.Combine(TestDir, "count.txt");
            File.AppendAllText(@".\count.txt", this.TestContext.TestName + Environment.NewLine);
            File.AppendAllText(@".\count.txt", "cellules : " + cnt1 + Environment.NewLine + "noyaux
: " + cnt2 + Environment.NewLine);

        }, 0);
    }
    [TestMethod]
    public void ThresholdSimpleInPaintTest()
    {
        Bench.ComputeCpuMemoryUsage(() =>
        {
            //Lecture
            //Mat original = Cv2.ImRead(@".\Echantillon.png");
            Mat original = Cv2.ImRead(@"D:\repos\Photos tests invasion-migration\17T Invasion
090617\17T M01 1 Inv_.jpg");

            Mat output = new Mat();

            //Amélioration des contrastes
            output = this.ContrastCorrection(original, 1.5);

            //Correction gamma
            //output = this.GammaCorrection(output, 0.8);

            //Filtre bilatéral de lissage
            output = this.BilateralFilter(output, 19);

            //Trouver les cerlces
            var circles = this.DetectCircle(output, 20, 8, 11);
            //Create matrice for the mask
            this.DrawCircle(output, circles, 0);

            //Seuillage par bande de couleur
            Scalar lowerColor = new Scalar(0, 100, 0);
            Scalar higherColor = new Scalar(180, 255, 255);
            output = this.ColorThreshold(output, lowerColor, higherColor, 13, 17);

            //Trouver cellules ?
            var contours = this.FindContour(output);
            foreach (var contour in contours)
            {
                var x1 = contour.Min(c => c.X);
                var y1 = contour.Min(c => c.Y);
                var x2 = contour.Max(c => c.X);
                var y2 = contour.Max(c => c.Y);
                //Cv2.Rectangle(v, new Point(x1, y1), new Point(x2, y2), Scalar.Red, 2);
                if (Math2.EuclideanDistance(x1, y1, x2, y2) > 25)
                {
                    var r = new Rect(x1, y1, x2 - x1, y2 - y1);
                    var cell = new Mat(output, r);
                    this.SaveROI(cell);
                }
            }

            var cnt1 = contours.Length;

            //Seuillage par bande de couleur
            lowerColor = new Scalar(130, 0, 0);
            higherColor = new Scalar(180, 255, 175);
            output = this.ColorThreshold(output, lowerColor, higherColor);

            //Dilatation et erosion
            this.Dilate(7, output);

            //Trouver les contours
            contours = this.FindContour(output);

            //Identifier les noyaux
            for (int i = 0; i < contours.Length; i++)
                Cv2.DrawContours(original, contours, i, this.Colors[i%5], 2);
```

```
            this.Save(original);

            var cnt2 = contours.Length;

            //var p = Path.Combine(TestDir, "count.txt");
            File.AppendAllText(@".\count.txt", this.TestContext.TestName + Environment.NewLine);
            File.AppendAllText(@".\count.txt", "cellules : " + cnt1 + Environment.NewLine + "noyaux
: " + cnt2 + Environment.NewLine);
        }, 0);
    }

    [TestMethod]
    public void ThresholdSimpleInPaintKmeanTest()
    {
        Bench.ComputeCpuMemoryUsage(() =>
        {
            //Lecture
            //Mat original = Cv2.ImRead(@".\Echantillon.png");
            Mat original = Cv2.ImRead(@"D:\repos\Photos tests invasion-migration\17T Invasion
090617\17T M01 1 Inv_.jpg");

            Mat output = original.Clone();

            //Trouver les cerlces
            //var circles = this.DetectCircle(output, 20, 8, 11);
            ////Create matrice for the mask
            //this.DrawCircle(output, circles, 0);

            //Filtre k-mean
            output = this.Kmean(output, 10);

            //Seuillage par bande de couleur
            //Scalar lowerColor = new Scalar(163, 0, 0);
            //Scalar higherColor = new Scalar(180, 255, 255);
            Scalar lowerColor = new Scalar(0, 100, 0);
            Scalar higherColor = new Scalar(180, 255, 255);
            output = this.ColorThreshold(output, lowerColor, higherColor, 7, 7);

            //Ouverture
            output = this.Erode(7, output);
            output = this.Dilate(7, output);

            //Trouver cellules ?
            var contours = this.FindContour(output);
            foreach (var contour in contours)
            {
                var x1 = contour.Min(c => c.X);
                var y1 = contour.Min(c => c.Y);
                var x2 = contour.Max(c => c.X);
                var y2 = contour.Max(c => c.Y);
                //Cv2.Rectangle(v, new Point(x1, y1), new Point(x2, y2), Scalar.Red, 2);
                if (Math2.EuclideanDistance(x1, y1, x2, y2) > 25)
                {
                    var r = new Rect(x1, y1, x2 - x1, y2 - y1);
                    var cell = new Mat(original, r);
                    this.SaveROI(cell);
                }
            }

            var cnt1 = contours.Length;

            //Seuillage par bande de couleur
            lowerColor = new Scalar(163, 148, 0);
            higherColor = new Scalar(179, 255, 255);
            output = this.ColorThreshold(output, lowerColor, higherColor);

            //Ouverture
            output = this.Erode(7, output);
            output = this.Dilate(7, output);

            //Trouver les contours
            contours = this.FindContour(output);

            //Identifier les noyaux
```

```
            for (int i = 0; i < contours.Length; i++)
                Cv2.DrawContours(original, contours, i, this.Colors[i % 5], -1);

            this.Save(original);

            var cnt2 = contours.Length;

            //var p = Path.Combine(TestDir, "count.txt");
            File.AppendAllText(@".\count.txt", this.TestContext.TestName + Environment.NewLine);
            File.AppendAllText(@".\count.txt", "cellules : " + cnt1 + Environment.NewLine + "noyaux
: " + cnt2 + Environment.NewLine);

        }, 0);
    }

    [TestMethod]
    public void KmeanTest()
    {
        Bench.ComputeCpuMemoryUsage(() =>
        {
            //Lecture
            //Mat original = Cv2.ImRead(@".\Echantillon.png");
            //Mat original =
Cv2.ImRead(@"D:\repos\Memoire\CancerCellDetection\ImageProcessingTests\bin\x64\Debug\0001Kmean.png")
;
            Mat original = Cv2.ImRead(@"D:\repos\Photos tests invasion-migration\17T Invasion
090617\17T M01 1 Inv_.jpg");

            Mat output = original.Clone();

            //Trouver les cerlces
            //var circles = this.DetectCircle(output, 20, 8, 11);
            ////Create matrice for the mask
            //this.DrawCircle(output, circles, 0);

            //Filtre k-mean
            output = this.Kmean(output, 10);

            //Seuillage par bande de couleur
            //Scalar lowerColor = new Scalar(163, 0, 0);
            //Scalar higherColor = new Scalar(180, 255, 255);
            Scalar lowerColor = new Scalar(0, 14, 0);
            Scalar higherColor = new Scalar(180, 255, 255);
            output = this.ColorThreshold(output, lowerColor, higherColor, 9, 11);


            //Trouver cellules ?
            var contours = this.FindContour(output);
            foreach (var contour in contours)
            {
                var x1 = contour.Min(c => c.X);
                var y1 = contour.Min(c => c.Y);
                var x2 = contour.Max(c => c.X);
                var y2 = contour.Max(c => c.Y);
                //Cv2.Rectangle(v, new Point(x1, y1), new Point(x2, y2), Scalar.Red, 2);
                if (Math2.EuclideanDistance(x1, y1, x2, y2) > 25)
                {
                    var r = new Rect(x1, y1, x2 - x1, y2 - y1);
                    var cell = new Mat(original, r);
                    this.SaveROI(cell);
                }
            }

            var cnt1 = contours.Length;

            //Seuillage par bande de couleur
            //lowerColor = new Scalar(163, 148, 0);
            //higherColor = new Scalar(179, 255, 255);
            lowerColor = new Scalar(0, 0, 38);
            higherColor = new Scalar(255, 255, 120);
            output = this.ColorThreshold(output, lowerColor, higherColor, 4, 4);


            //Trouver les contours
```

```
            contours = this.FindContour(output);

            //Identifier les noyaux
            for (int i = 0; i < contours.Length; i++)
                Cv2.DrawContours(original, contours, i, this.Colors[i % 5], -1);

            this.Save(original);

            var cnt2 = contours.Length;

            //var p = Path.Combine(TestDir, "count.txt");
            File.AppendAllText(@".\count.txt", this.TestContext.TestName + Environment.NewLine);
            File.AppendAllText(@".\count.txt", "cellules : " + cnt1 + Environment.NewLine + "noyaux
: " + cnt2 + Environment.NewLine);

        }, 0);
    }

    string Save(Mat v, [CallerMemberName]string function="")
    {
        this._counter++;
        string name = this._counter.ToString("D4") + function + ".png";
        var s = Path.Combine(this.TestDir, name);
        //Enregistrement de l'image de sortie
        Cv2.ImWrite(s, v);
        return s;
    }

    void SaveROI(Mat v)
    {
        string name = Guid.NewGuid() + ".png";
        var s = Path.Combine(this.TestRoiDir, name);

        Cv2.ImWrite(s, v);
    }

    public Mat ContrastCorrection(Mat v, double alpha)
    {
        //Chargement de l'image
        Mat output = new Mat();

        //Augmente le contraste de 10%
        v.ConvertTo(output, v.Depth(), alpha, 0);

        //Enregistrement de l'image de sortie
        this.Save(output);

        return output;
    }

    public Mat GammaCorrection(Mat v, double gamma)
    {
        //Chargement de l'image
        Mat output = new Mat();

        //Création de la table lut en fonction du facteur de correction gamma
        byte[] lookUpTable = new byte[256];
        for (int i = 0; i < 256; ++i)
            lookUpTable[i] = (byte)Math.Round(Math.Pow(i / 255.0, gamma) * 255.0);
        //Application de la correction gamma
        Cv2.LUT(v, lookUpTable, output);

        //Enregistrement de l'image de sortie
        this.Save(output);

        return output;
    }

    public Mat BilateralFilter(Mat v, int kernelSize)
    {
        Mat output = new Mat();

        //Bilateral blur
        Cv2.BilateralFilter(v, output, kernelSize, kernelSize * 2, kernelSize / 2);
```

```
        //Enregistrement de l'image de sortie
        this.Save(output);

        return output;
    }

    public CircleSegment[] DetectCircle(Mat v, int distanceMin=15, int radiusMin=13, int
radiusMax=15)
    {
        Mat gray = new Mat();

        //Convert in gray
        Cv2.CvtColor(v, gray, ColorConversionCodes.BGR2GRAY);

        //Get circles from the gray image
        //src_gray: Input image(grayscale)
        //circles: A vector that stores sets of 3 values: for each detected circle.
        //CV_HOUGH_GRADIENT: Define the detection method.Currently this is the only one available in
OpenCV
        //dp = 1: The inverse ratio of resolution
        //14.5 distance minimum entre les cercles
        //200 seuil haut pour la méthode de Canny
        //100 Seuil pour la détection de centre
        //13 et 15 tailles minimum et maximum du rayon des cercles à détecter
        return  Cv2.HoughCircles(gray, HoughMethods.Gradient, 1, distanceMin, 200, 10, radiusMin,
radiusMax);

    }

    public Mat DrawCircle(Mat v, CircleSegment[] circles, int increaseRadiusBy)
    {
        //Draw the circle in the mask
        foreach (var circle in circles)
            Cv2.Circle(v, (int)circle.Center.X, (int)circle.Center.Y, (int)circle.Radius +
increaseRadiusBy, new Scalar(255,255,255), -1);

        this.Save(v);

        return v;
    }

    public Mat Inpaint(Mat v, Mat mask, int radius)
    {
        Mat output = new Mat();

        //Taille de kernel
        Cv2.Inpaint(v, mask, output, 25, InpaintMethod.NS);

        //Enregistrement de l'image de sortie
        this.Save(output);

        return output;
    }

    private Mat ColorThreshold(Mat v, Scalar lowerColor, Scalar higherColor, int erodeSize=0, int
dilateSize=0)
    {
        Mat output = new Mat();
        Mat mask = new Mat();

        Mat hsv = new Mat();
        //Convertion RGB vers HSB
        Cv2.CvtColor(v, hsv, ColorConversionCodes.BGR2HSV);
        this.Save(hsv);

        //Seuillage par bande de couleur
        Cv2.InRange(hsv, lowerColor, higherColor, mask);
        this.Save(mask);
        hsv.Dispose();

        mask = this.Erode(erodeSize, mask);

        mask = this.Dilate(dilateSize, mask);
```

```
        //Intersection du masque et de l'image originale
        Cv2.BitwiseAnd(v, v, output, mask);
        mask.Dispose();

        //Enregistrement de l'image de sortie
        this.Save(output);

        return output;
    }

    private Mat Dilate(int dilateSize, Mat v)
    {
        //Dilatation
        if (dilateSize > 0)
        {
            v = v.Dilate(Cv2.GetStructuringElement(MorphShapes.Ellipse, new Size(dilateSize,
dilateSize)));
            this.Save(v);
        }

        return v;
    }

    private Mat Erode(int erodeSize, Mat v)
    {
        //Erosion
        if (erodeSize > 0)
        {
            v = v.Erode(Cv2.GetStructuringElement(MorphShapes.Ellipse, new Size(erodeSize,
erodeSize)));
            this.Save(v);
        }

        return v;
    }

    public Point[][] FindContour(Mat v)
    {
        Mat output = new Mat();
        Mat gray = new Mat();
        Point[][] contours;
        HierarchyIndex[] hierarchy;

        //Convert in gray
        Cv2.CvtColor(v, gray, ColorConversionCodes.BGR2GRAY);

        //Filtre gaussien
        Cv2.GaussianBlur(gray, output, new Size(7, 7), 5, 5, BorderTypes.Default);
        this.Save(v);

        //Méthode de Otsu
        int thOtsu = (int)OtsuThresholding.Compute(BitmapConverter.ToBitmap(v));

        //Détecteur de contours de Canny
        Cv2.Canny(output, output, thOtsu / 3, thOtsu);
        this.Save(output);

        //Enumération des contours
        Cv2.FindContours(output, out contours, out hierarchy, RetrievalModes.List,
ContourApproximationModes.ApproxTC89KCOS);

        ////Dessine les contours en vert
        //for (var i = 0; i < contours.Length; i++)
        //    Cv2.DrawContours(v, contours, i, new Scalar(0, 255, 0), 2);

        //this.Save(v);

        return contours;
    }

    public Mat Kmean(Mat v, int categoryCount)
    {
        Mat output = new Mat();
```

```
        //Réduction du nombre de couleur pour obtenir des "blobs"
        KMeans.Proceed(v, output, categoryCount, true, Scalar.Black);

        this.Save(output);

        return output;
    }
}
```