Faculté des Sciences de Montpellier

Mémoire de stage

Suivi des cinétiques de croissance de baies de raisin individuelles par analyse d'image

Maxence CAFIER

Encadrant scientifique : **Charles ROMIEU**

Tutrice pédagogique : **Séverine BERARD**

Master 2 - Bioinformatique

25 août 2022

Table des matières

Chapitre 1

Introduction

La vigne domestiquée représente un défi économique de taille, en particulier dans un pays tel que la France, deuxième pays mondial en terme de surface occupée par du vignoble. La volonté de produire des raisins de meilleure qualité, et le besoin de créer des variétés résistantes aux maladies de la vigne (e.g mildiou, oïdium) ont été des facteurs d'influence pour étudier cette plante. Aujourd'hui, un défi de plus en plus pressant réside dans le développement de variétés résistantes au stress hydrique, afin de faire face au réchauffement climatique. Un aspect de ces études concerne le phénotypage de la baie, qui porte aussi bien sur les concentrations de molécules d'intérêts (sucres, acides), que sur le volume et l'aspect visuel de la baie. Lors de la croissance de la baie, plus de 10 000 gènes sont exprimés, et c'est le stade de développement du raisin qui a le plus de poids sur leur expression. Il est donc important de pouvoir l'estimer. Justement, le suivi de la croissance du volume permet de caractériser ce stade de maturité (véraison), car celle-ci suit une courbe à double sigmoïde très reproductible. Autrement dit, savoir à quel endroit une baie se trouve sur cette courbe donne une indication sur la phase de véraison dans laquelle elle se situe.

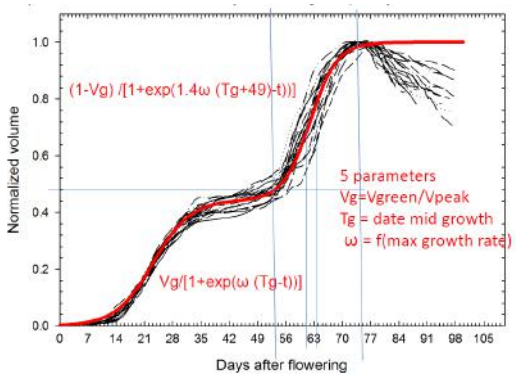


Figure 1 : 20 baies, 40 photos, Vitis vinifera cv Syrah, Image J (analyse « manuelle ») Savoi S., Torregrosa L. Romieu C.

Le phénotypage est traditionnellement réalisé manuellement, ce qui requiert temps, expertise, et souffre de subjectivité. Les méthodes de vision par ordinateur ont proliféré ces dernières décennies dans la viticulture, principalement pour le comptage et l'estimation de rendement, mais plus récemment beaucoup se sont portées sur le phénotypage automatique. Cependant, la méthodologie déployée par les biologistes emploie toujours un moyennage des variables observées sur les

différentes baies de l'étude, ce qui suppose une synchronicité des baies allant à l'encontre de la réalité terrain. **[1]** Ainsi, il existe une nécessité d'implémenter une solution de phénotypage automatique, robuste et non-destructive des baies, soutenue cette fois par une intuition méthodologique plus pertinente et précise qui est l'étude de baies individuelles.

La problématique de mon stage est donc de développer un programme permettant, à haut débit et à partir de photos uniquement, de suivre de manière automatique la cinétique de croissance du volume de baies individuelles.

Chapitre 2

Etat de l'art

Jusqu'à récemment, le calcul du volume des baies était effectué soit par une analyse d'image manuelle (en plaçant les contours d'une ellipse ou d'un cercle sur une baie) soit par l'utilisation d'algorithmes traditionnels dont l'efficacité est discutable, et dont l'amélioration reste marginale depuis plusieurs décennies. Dernièrement, l'impressionnante conquête des réseaux de neurones profonds sur la vision par ordinateur a changé le paysage du phénotypage par analyse d'image. Beaucoup s'attèlent au comptage du nombre de baies après une segmentation par deep learning (la segmentation est le fait classifier les pixels selon s'ils appartiennent à un objet - ici une baie - ou non), à l'instar de Zabawa**[2]**, ou de Nellithimaru **[3]**. Dans ces deux publications, la précision de la segmentation est suffisante pour compter les baies, mais trop faible pour déterminer convenablement le volume des baies. Luo **[4]** propose de combiner les fragments de contours de baies trouvés par un filtre de Canny (ce filtre sera présenté dans la suite du rapport) par un clustering, et de réaliser une régression circulaire sur ces fragments recollés. Cependant, Luo ne rentre pas dans le détail de son algorithme, le rendant difficile à implémenter, et semble assimiler toutes les baies à des sphères parfaites, ce qui n'est pas le cas. Roscher propose une méthode reposant sur la transformée de Hough (qui sera également expliquée dans ce rapport) pour la détection des formes circulaires dans ses images. Un protocole est employé pour l'affinage des les cercles détectés dans le but de les rendre plus précis, et enfin un classifieur examine chaque cercle pour trier ceux qui correspondent bien à une baie, et ceux qui ne sont que des faux positifs. De nouveau, les baies sont assimilées à des sphères, ce qui n'est pas le cas.

Miao **[5]** propose un pipeline complet en extrayant les contours saillants de l'image grâce à une version améliorée d'un modèle de deep learning spécialisé dans la détection de contours, HED **[6]**. Par la suite, chaque baie est isolée du reste de l'image grâce à une détection d'objets par deep learning. Enfin, un algorithme trouve itérativement les paramètres de l'ellipse collant le mieux aux contours trouvés par HED.

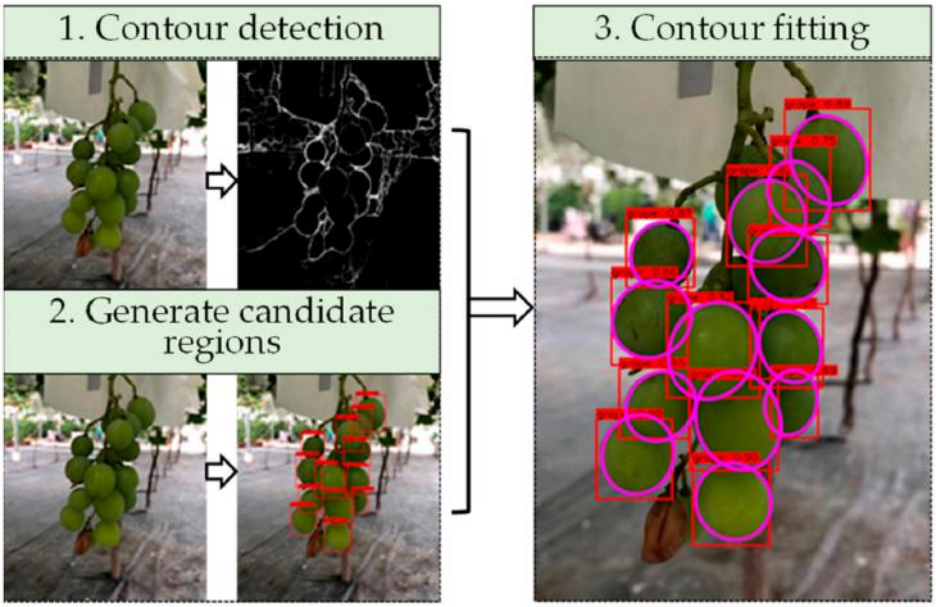


Figure 2 : Workflow de l'article de Miao en trois étapes clés.

Cette stratégie lui a permis d'observer sans bruit les variations quotidiennes du volume des baies. C'est en particulier de cet article que mon stage s'est inspiré, bien que de nombreuses autres voies aient été explorées.

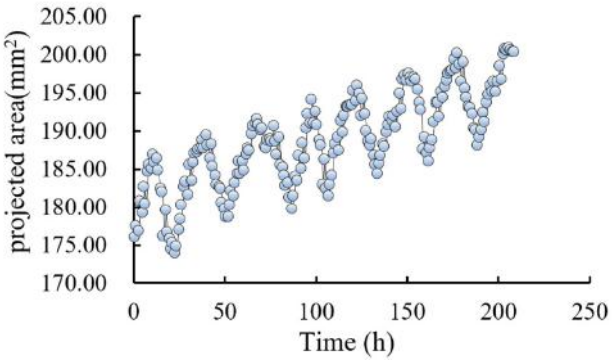


Figure 3 : Suivi de l'aire projetée d'une baie au cours du temps

Chapitre 3

Matériel et méthodes

3.1 Données dont nous disposons

Comme pour tout projet, il convient de bien comprendre nos données. Dans notre cas, il est important de saisir ce que sont nos photos du point de vue de l'ordinateur.

Nos photos sont des matrices, dont chaque cellule correspond à un pixel de notre image. La valeur inscrite dans une cellule varie de 0 à 255, et correspond à l'intensité lumineuse de ce pixel.

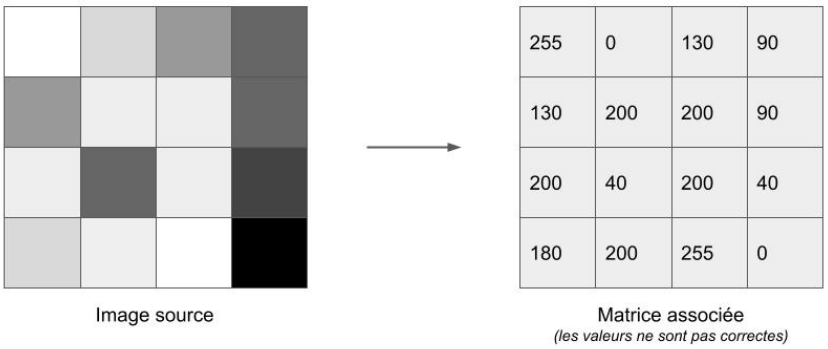


Figure 4 : Interprétation d'une image en noir et blanc par une matrice

Une telle matrice nous permet de décrire une image en noir et blanc, mais qu'en est-il des images en couleurs, telles que celles dont nous disposons ?

Par défaut, nos photos en couleur sont exprimées dans l'espace colorimétrique RGB, c'est-à-dire qu'il existe une matrice d'intensité lumineuse des pixels pour chacune des couleurs primaires (Rouge, Vert, Bleu). Nos photos sont donc des matrices tridimensionnelles, avec pour dimensions : Hauteur de la photo x Largeur de la photo x 3 canaux.

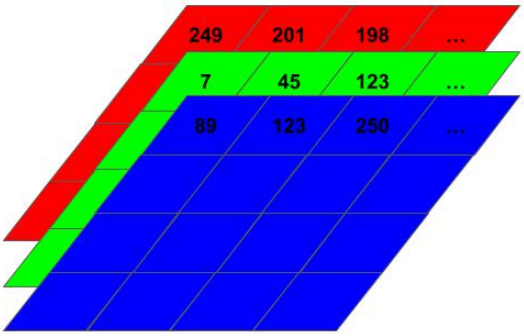


Figure 5 : Interprétation d'une image en couleur par une matrice.

Bien sûr, nos images peuvent être exprimées dans d'autres espaces colorimétriques que le RGB, ce sur quoi nous reviendrons dans la partie Détection de véraison.

3.1.1 Nos photos

Mon programme est destiné à être utilisé par plusieurs personnes, possédant des photos différentes. Voici un exemple de photos d'une même grappe à différents stade de maturité. Ces photos ont été prises par Elias Motelico-Heino, un autre stagiaire de mon équipe.

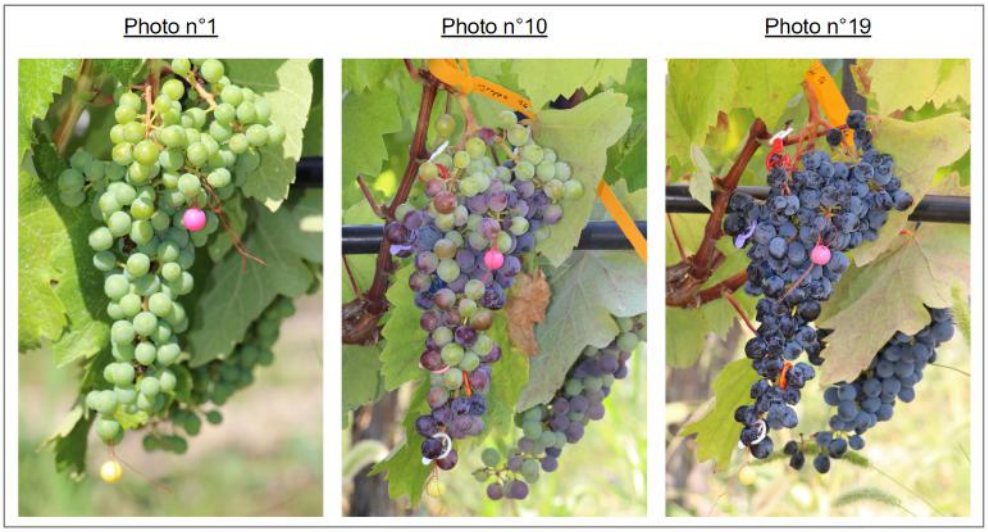


Figure 6 : Exemples de photos d'une grappe à différents stades de son développement. C'est principalement pour ces données que mon programme a été développé.

Les photos ont été prises dans la vigne pédagogique de l'InstitutAgro de Montpellier, avec une attention particulière portée sur la prise de vue, qui est voulue similaire sur toute la série temporelle. Des petites perles en plastique ont été disposées sur la grappe pour servir d'étalon lors du calcul du volume des baies. Au total, 18 grappes ont été suivies, appartenant à différentes variété de vigne. Sur la durée totale de la véraison d'une grappe, plus d'une vingtaine de photos sont prises.

3.2 La détection de baies de raisin

L'objectif de ce stage a été de phénotyper à haut débit les baies individuelles de raisins à partir de photos. Ces photos, qui sont nos données d'entrée, sont à nos yeux pleines de sens, car nous y reconnaissons les formes et les contextes qui nous sont familiers. Mais pour l'ordinateur, la sémantique de leur contenu est inconnue, elles ne sont que des matrices tridimensionnelles, dont les cellules contiennent la valeur d'intensité lumineuse de chaque pixel dans chaque canal (R, G et B). La première étape de notre programme consiste donc en la détection de baies, qui aura pour fonction de reconnaître chaque baie et de consigner dans un fichier ses coordonnées dans l'image. Cette étape nous permet par la suite d'étudier les baies de manière individuelle.

La solution la plus pertinente et la plus efficace pour cette tâche est la détection d'objets par intelligence artificielle. Depuis le programme Alexnet développé en 2012 par Alex Krizhevsky, Ilya Sutskever et Geoffrey Hinton, les réseaux de neurones ont largement dominé la vision par ordinateur. Et si Alexnet ne permettait alors que de classifier les images selon leur contenu (chien, chat, cheval, voiture, etc.), les modèles d'IA d'aujourd'hui permettent de reconnaître et localiser les nombreux objets présents dans les images. Parmi ces modèles, on retrouve notamment Fast-R-CNN, Faster-R-CNN, SSD, YOLO et d'autres encore. C'est YOLOv5 que j'ai choisi pour mon programme, pour ses performances au sommet de l'état de l'art sur les deux datasets benchmark classiques de la détection d'objets (Pascal VOC (visual object classes) [7] et Microsoft COCO (common objects in context) [8]) (voir le papier détection incendie), et pour le contrôle qu'il offre sur les paramètres d'entraînement et de détection. De plus, sa grande popularité permet d'avoir de nombreux outils d'annotation d'images adaptés au format YOLO.

3.2.1 YOLOv5

YOLO (You Only Look Once)[9] est un réseau de neurones de détection d'objets développé par Joseph Redmond en 2015. Sa précision et sa rapidité ont rapidement fait de cet algorithme un favori dans ce domaine de l'IA, et depuis sa création, le modèle a été modifié et amélioré plusieurs fois, avec ou sans collaboration de l'auteur original. C'est en 2020 que YOLOv5 a été proposé au public par Glenn Jocher, après avoir implémenté sous PyTorch (un framework facilitant le développement de réseaux de neurones sous Python) la dernière version de l'algorithme, YOLOv4.

Malgré la popularité de YOLOv5 dans le monde commercial et académique, aucun article scientifique n'est à ce jour paru de la part de son créateur. Pour cette raison, et parce que YOLOv5 lui-même se décline sous plusieurs versions, il est difficile de décrire précisément son architecture, mais sa structure générale est la suivante. Il est composé de trois parties successives :

1. La colonne vertébrale, qui est la partie cherchant à extraire de l'image les caractéristiques les plus significatives et riches de sens. Cette colonne reprend un réseau neuronal à part entière, nommé CSPNet, constitué de nombreuses couches de convolution.
2. La tête réalise la détection, c'est-à-dire la localisation d'objets par boîtes englobantes et classification.
3. La tête réalise la détection, c'est-à-dire la localisation d'objets par boîtes englobantes et classification.

De nombreux outils d'annotations d'images adaptés au format YOLO existent. J'ai choisi d'utiliser Ybat pour son ergonomie, sa simplicité d'utilisation et son fonctionnement hors ligne, sans nécessité de télécharger nos données sur un serveur. Le dataset final dataset contient

plus de 13 000 baies annotées. Ce dataset est composé de photos de raisin sous diverses conditions d'illumination, de différentes résolutions, en intérieur et en extérieur, avec un fond uni ou avec un fond naturel. Il contient également des photos de vignes sans raisin, pour éviter les faux positifs liés à la présence de feuilles ou de tuteurs.

Plusieurs modèles YOLOv5 ont été entraînés au cours de mon stage, généralement sous la nécessité d'étoffer mon set de données par des photos dans de nouvelles conditions. Le modèle final reprend l'architecture YOLOv5 "L", qui est plus lourde et plus longue à entraîner (car plus de paramètres), mais qui offre une détection de meilleure qualité. Ce modèle a été entraîné pendant 80 epochs sur Google collaboratory, un produit de Google Research qui permet à l'utilisateur d'exécuter du code Python en empruntant des ressources informatiques externes très conséquentes.

Le modèle entraîné a été récupéré sur ma machine locale en téléchargeant un fichier récapitulant les poids des neurones, autrement dit le réseau "éduqué". Grâce à cela, la détection d'objets, qui est computationnellement beaucoup moins lourde que l'entraînement, peut être réalisée localement sur ma machine personnelle.

La sortie d'une détection d'objets par YOLOv5 est un fichier texte pour chaque image analysée. Dans ce fichier, les coordonnées des objets détectés sont inscrites dans un format particulier à YOLO, comme vous pouvez le voir dans la figure X.

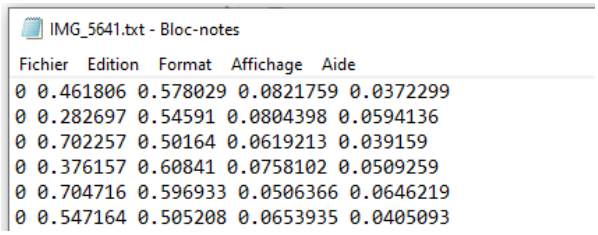


Figure 7 : Exemple de labels sortis par YOLOv5 après détection.

Une ligne représente un objet, et le format est le suivant :

Id
classe x_centre y_centre largeur_boite hauteur_boite

Dans la figure ci-dessus, toute les valeurs de la première colonne sont à zéro, car nous n'avons qu'une seule classe : Berry (baie de raisin), dont l'identifiant est 0. Les deuxième et troisième colonnes correspondent aux coordonnées x et y du centre de la boîte, et les quatrième et cinquième correspondent à la largeur et la hauteur de la boîte. Les valeurs x_centre et largeur_boite sont normalisées par la largeur de l'image, et les valeurs y_centre et hauteur_boite sont normalisées par la hauteur de l'image.

Il est aussi possible de produire une image composite, représentant toutes les boîtes trouvées.

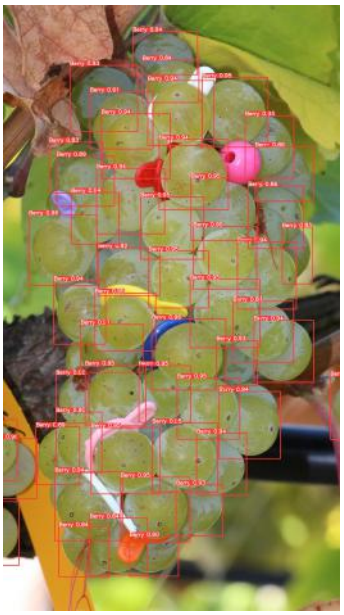


Figure 8 : Baies détectées par YOLOv5.

Les baies individuelles maintenant détectées, elles peuvent être isolées grâce à leur coordonnées. Pour ce faire, il suffit de faire un rognage de l'image d'origine selon les coordonnées des boîtes trouvées par YOLOv5. Certaines boîtes sont parfois trop étroites, j'ai donc estimé empiriquement qu'un agrandissement de 15

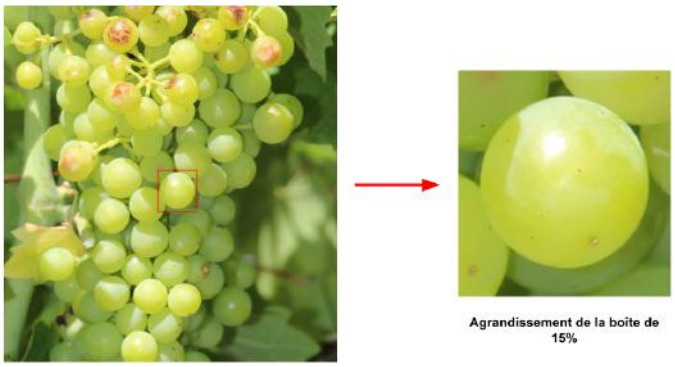


Figure 9 : Rognage de l'image autour de la baie, selon les coordonnées de la boîte

Ces dizaines de nouvelles petites images ne sont jamais sauvegardées, elles sont produites à la volée et disparaissent à l'itération suivante, elles sont en quelque sorte des plans de travail temporaires, ce qui épargne la mémoire du système. Le résultat des analyses réalisées sur elles est en revanche sauvegardé. Un des avantages de travailler sur des petits morceaux d'image est que l'on gagne une plus grande liberté sur les pré-traitements que l'on peut décider d'appliquer. Par exemple, essayer de changer le contraste de l'image complète pour faire ressortir les contours des baies est souvent problématique, parce que ce qui résulte en une amélioration en haut de la grappe de raisin peut aussi résulter en une détérioration en bas. Dès lors, c'est à un réglage manuel fin et fastidieux qu'il faut s'atteler. Sur une image rognée autour d'une baie, les pré-traitements sont beaucoup plus permissifs car les circonstances d'illumination et de flou optique sont localement

plus homogènes, et peuvent être réalisés de manière automatique selon les conditions de prise de vue.

3.3 La détection du volume des baies

3.3.1 La détection du contour de la baie

Pour calculer le volume d'une baie, il est nécessaire de comprendre où elle commence et où elle se termine. C'est donc la qualité de la détection de ses contours qui va déterminer la qualité de l'estimation de son volume. Au cours de mon stage, j'ai recherché et travaillé sur plusieurs méthodes de détection de contours ; je ne discuterai dans ce rapport que des trois méthodes implémentées dans mon programme final.

Le filtre de Canny

Le filtre de Canny (Canny edge detection) a été conçu en 1986 pour la détection des contours sur une image. Trente-six ans plus tard, cet algorithme est encore largement utilisé. Nous n'allons pas rentrer dans les détails, mais cet algorithme se déroule en 4 étapes :

(1) La réduction du bruit (lissage) grâce à un filtre Gaussien 2D.

(2) Le calcul du gradient d'intensité en chaque point, qui retourne l'intensité des contours, ainsi que leur direction.

(3) La suppression des non-maxima : seuls les maxima locaux sont conservés et considérés comme des contours.

(4) Le seuillage des contours par hystérésis : c'est un seuillage à deux valeurs. Les points de contours en dessous du seuil le plus petit sont éliminés, ceux au-dessus du seuil le plus grand sont conservés, et ceux entre les deux ne sont conservés que s'ils juxtaposent un point déjà validé.

La fonction Canny() est intégrée sur Python dans le package OpenCv, et permet d'appliquer les 4 étapes en nous laissant le contrôle sur les deux seuils de l'étape 4. La première étape de réduction de bruit a pour but de supprimer les signaux parasites (petits contours non désirés) tout en conservant les contours principaux (le contour des baies). Nous n'avons pas de contrôle sur le lissage qui est appliqué par cette fonction, cependant il semble plutôt inefficace, ce qui nous est favorable car cela nous permet d'appliquer notre propre lissage personnalisé en amont.

La fonction de lissage suggérée sur la majorité des tutoriels trouvés est la fonction blur() d'OpenCv. Cependant, des recherches plus poussées m'ont fait découvrir le filtre bilatéral, qui est capable de lisser une image tout en conservant particulièrement bien la définition des contours, ce qui en fait une fonction très adaptée à notre problème.

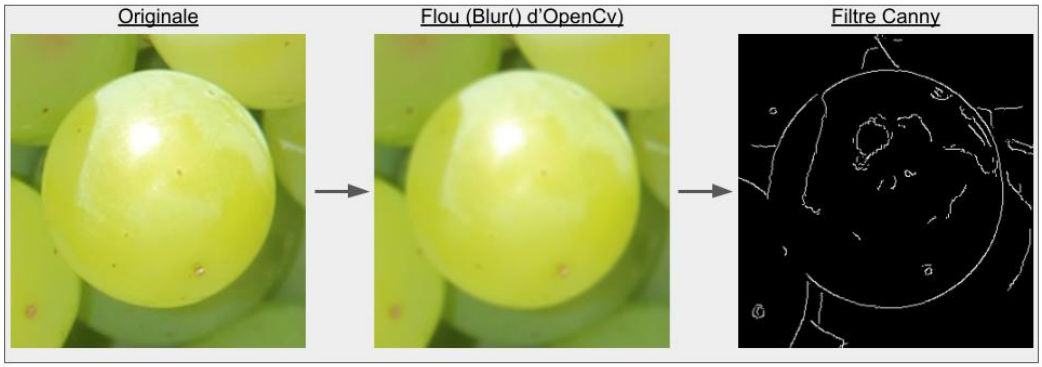


Figure 10 : Contours détectés par un Blur d'openCv suivi d'un filtre Canny

Comme le montre la figure X, le flou d'OpenCv est relativement efficace pour supprimer du bruit, mais les contours deviennent diffus. Supprimer davantage de bruit avec cette fonction se fait au coût d'une perte accrue de définition des contours de la baie.

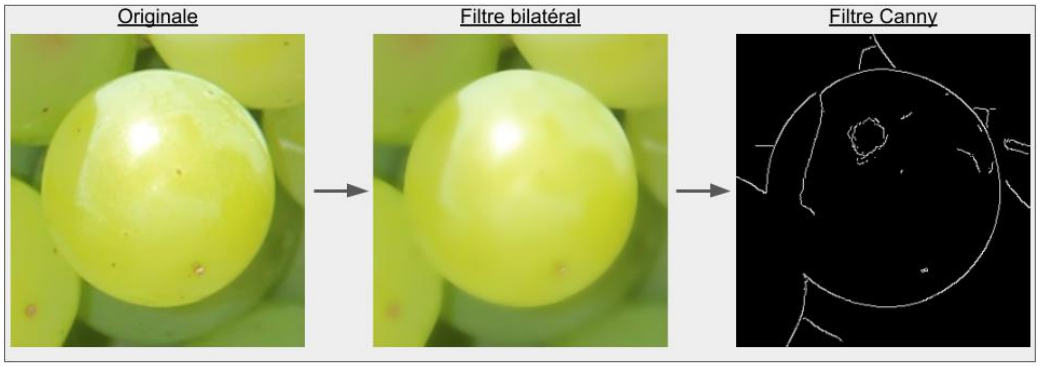


Figure 11 : Contours détectés par un filtre bilatéral suivi d'un filtre Canny

Le filtre bilatéral (Figure Y) nous permet de lisser davantage tout en conservant la définition des contours essentiels. C'est donc celui-ci qui a été intégré dans mon programme.

HED (Holistically-nested Edge Detection)

Dans le domaine de la détection de contours également, l'intelligence a su se faire une place de plus en plus prépondérante ces dernières années. De fait, mon stage a été librement inspiré par un article de Miao, Huang et Zhang **[5]**, mettant à l'oeuvre un pipeline de phénotypage des baies (de manière groupée et non individuelle), dans lequel la détection de contours à été réalisée par intelligence artificielle, grâce à l'amélioration d'un algorithme nommé Holistically-Nested Edge Detection (HED). Par soucis de simplicité et de concision, je ne décrirai que succinctement l'architecture et le fonctionnement de l'algorithme HED.

Le réseau neuronal HED reprend la structure d'un réseau VGG16, contenant 13 couches de convolutions divisées en 5 étages, chacun délivrant une carte des contours préliminaire, dont le contenu est de plus en plus concis au fil des étages. Une couche finale permet de fusionner ces 5 cartes de contours pour en donner une finale.

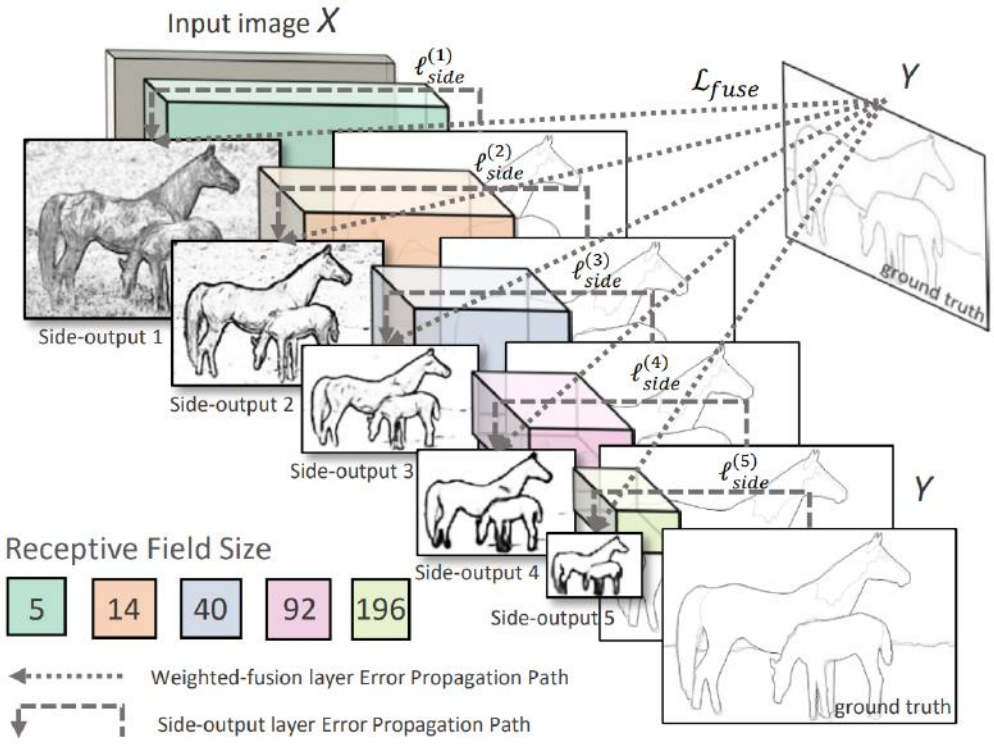


Figure 12 : Schématisation du fonctionnement de HED. Source :Xie **[6]**

Il est possible de récupérer n'importe quelle de ces 6 cartes de contours produites, mais c'est généralement la fusionnée qui est la meilleure, et c'est celle-ci que nous avons utilisée dans notre cas.

Le papier de Miao, Huang et Zhang décrit diverses améliorations faites au HED original, malheureusement ni les modifications de l'architecture ni un entraînement sur nos propres données n'ont été possibles, pour des raisons de compatibilité avec Windows. Néanmoins, j'ai pu télécharger un modèle pré-entraîné sur le BSDS500, un dataset qui sert de benchmark pour la détection de contours. Les résultats de ce modèle sont somme toute proches de ceux produits par le HED amélioré.

HED à la qualité de reconnaître même les contours les plus ténus, mais cela peut aussi poser problème : comment ne récupérer que ceux qui nous intéressent ? De plus, même ce qui semble apparaître noir sur la sortie n'est en réalité pas d'intensité 0. La solution qui s'impose à l'esprit est le seuillage (thresholding), mais les conditions de prises de vues variables rendent difficile l'utilisation d'un seuil fixe. C'est pourquoi j'ai proposé une méthode de seuillage des contours adaptable et à la volée. Celle-ci repose sur l'utilisation de la méthode Otsu, définissant un seuil de manière automatique à partir de l'histogramme de l'image. Mes essais ont montré qu'utiliser un seuil 20% plus grand que celui déterminé par Otsu permettait de séparer le bon grain de l'ivraie plus efficacement. Ainsi, seuls les contours d'intensité supérieure au seuil choisi sont conservés. Nous verrons par la suite que cette grande réduction du nombre de pixels retenus est très importante pour le temps de calculs des analyses futures.

Le problème majeur avec HED est que les contours produits sont épais. De plus, si les contours réels des baies (vérité terrain) se trouvent toujours sous les contours épais produits par

HED, plusieurs de mes essais ont révélé que l'on ne peut pas prédire précisément leur position : ils se trouvent tantôt au centre du trait épais, tantôt vers le haut, tantôt vers le bas. En conclusion, HED seul n'est pas suffisant pour offrir le phénotypage précis que nous souhaitons atteindre. Cependant, sa quasi exhaustivité dans les contours lui confère une utilité dont nous ne voulons pas nous priver. Ainsi, nous proposons une méthode combinant HED et le filtre Canny.

HED nettoyé par Canny

Pour affiner nos contours, nous avons proposé de mutualiser les contours Canny et les contours HED, pour ne retenir que les pixels qu'ils possèdent en commun. La combinaison des deux à l'avantage de resituer les contours sur la vérité terrain grâce à la précision de Canny, et de se débarrasser d'une partie du bruit restant grâce à un thresholding adapté sur la production HED. Le résultat est en quelque sorte un Canny nettoyé.

Detectron2

La dernière des solutions pour la détection de contours, et probablement la plus prometteuse, est l'implémentation de Mask R-CNN sous Detectron2 **[10]**. Développée par Facebook AI Research, Detectron2 est une bibliothèque qui pousse la détection d'objets un peu plus loin en proposant l'implémentation de nombreux modèles, adaptés à différentes tâches : localisation d'objets, segmentation d'objets, segmentation panoptique, détection de points clés (tête, bras gauche, jambe droite etc.). C'est la segmentation d'objet, et donc l'implémentation du réseau de neurones Mask-R-CNN sous Detectron2 qui nous intéressent. Bien plus que de reconnaître un objet et de le localiser par une boîte qui l'encadre, ce modèle permet de créer un masque sur l'objet. Par exemple, sur la photo d'une girafe, il reconnaîtra et identifiera la girafe, et classifiera chaque pixel selon s'il appartient à l'animal ou non. Logiquement, cela nous donne aussi les contours de ladite girafe qui correspondent aux pixels lui appartenant, en contact avec ceux qui ne lui appartiennent pas.

Ce modèle est doté en premier lieu d'un réseau RPN (Region Proposal Network), dont la fonction est de proposer des régions autour des potentiels objets trouvés. Il est ensuite composé d'un ensemble de trois branches, réalisant respectivement la classification de l'objet trouvé, la proposition d'une boîte englobante autour de cet objet, et la production d'un masque pixel par pixel de cet objet. C'est cette dernière branche qui est innovante, et qui nécessite l'ajout de diverses étapes dans le fonctionnement du modèle, que nous ne décrirons pas dans ce mémoire.

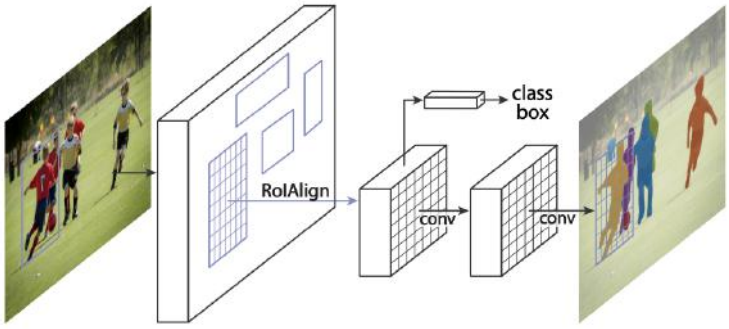


Figure 13 : Flux de travail de Mask R-CNN

J'ai créé mon propre set de données à partir de nos photos prises dans diverses conditions par mon équipe, en particulier celles d'Elias, grâce au fameux logiciel d'annotation LabelMe. Au total, plus de 550 images ont été annotées à la main. A l'instar de YOLOv5, j'ai réalisé l'entraînement sur Google collaboratory, et ai ensuite téléchargé le modèle entraîné à masquer les baies pour l'utiliser librement sur ma machine locale. Je peux alors appliquer une détection sur mes images rognées.

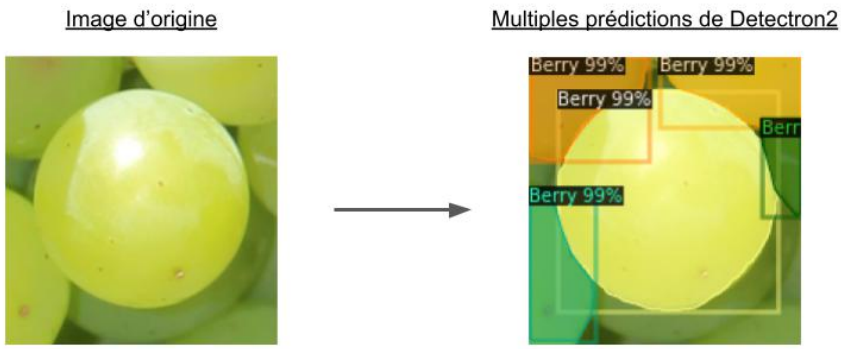


Figure 14 : Exemple d'output de notre modèle Detectron2 entraîné

Cette détection peut être réalisée en faisant varier différents paramètres, mais le principal est le seuil de confiance minimal pour qu'un masque soit émis. Généralement, nos images sont rognées autour d'une baie, mais d'autres baies peuvent apparaître aussi. Comment alors ne récupérer que le masque de la baie nous intéressant ? Les taux de confiances ne permettent pas de discriminer les masques entre eux, la solution a donc été d'ajouter une étape calculant la distance euclidienne entre le centre de chaque masque et le centre de l'image, que YOLOv5 parvient à centrer presque infailliblement autour de la baie d'intérêt.

La sortie de ce modèle est un dictionnaire de dictionnaires, contenant classes, boîtes et masques. Le masque est une matrice bidimensionnelle (noir et blanc) de dimensions identiques à l'image d'entrée, dont les pixels faisant partie du masque ont pour valeur True, et les autre False.

Néanmoins, ce sont les contours de ce masque qui nous intéressent, et non toute sa surface. Pour extraire ces contours, j'ai utilisé la fonction findContours() d'OpenCv, en la paramétrant pour qu'elle simplifie les segments droits par les deux points qui l'encadrent, par souci d'économie de mémoire.

3.3.2 La détection de cercle/ellipse

Nous savons donc retrouver via diverses méthodes les contours des baies. La suite logique de notre travail est d'essayer de trouver le cercle ou l'ellipse la plus probable passant par les contours de la baie étudiée, c'est ce qu'on appelle le Circle fitting ou Ellipse fitting. A partir des paramètres de cercle ou de cette ellipse, nous pourrons calculer le volume en voxels de la baie.

Transformée de Hough circulaire

La première solution avec laquelle j'ai expérimenté est la Transformée de Hough circulaire. Cet algorithme permet d'extraire d'une image les cercles qui y seraient présents. Il se lance sur une image ayant subi le filtre Canny, et peut être vulgarisé de la manière suivante :

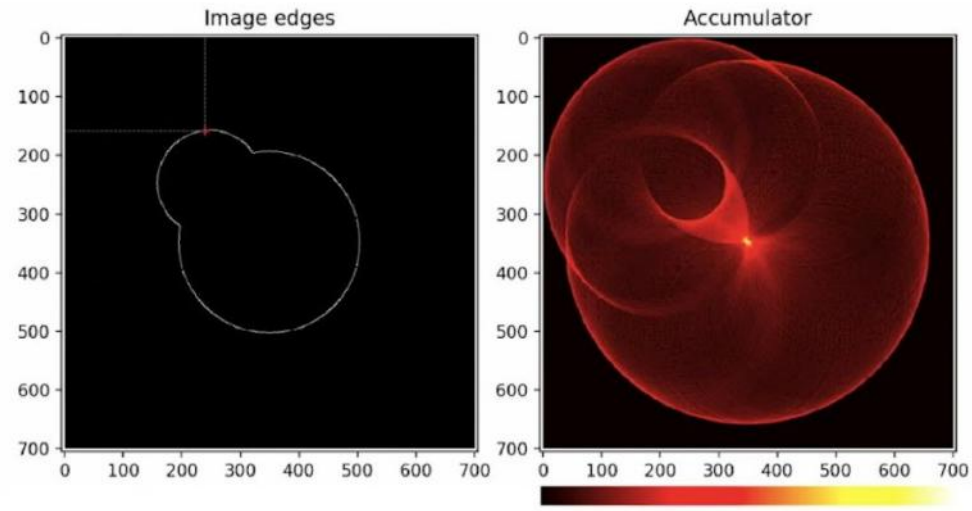


Figure 15 : A gauche, la carte des contours, à droite la matrice d'accumulation.

(A) L'utilisateur choisit plusieurs paramètres, dont une valeur minimale et maximale dans laquelle il estime que le rayon du cercle à trouver doit se situer. (en pixels)

(B) Une matrice d'accumulation de la même hauteur et largeur que l'image est créée. Nous pouvons l'imaginer comme une image complètement noire.

(C) Sur chaque point de contours trouvés par le filtre Canny, un cercle d'un rayon r+0 (valeur minimale) est créé. Pour chaque pixel appartenant à ce cercle, l'intensité du même pixel est incrémentée sur la matrice d'accumulation

(D) Le pixel où l'accumulation est la plus grande est retenu comme étant le meilleur candidat pour être le centre d'un cercle de rayon r.

(E) Le rayon r du cercle à créer est incrémenté à r+1.

(F) Répéter B, C, D et E jusqu'à ce que le rayon r atteigne la borne supérieure définie par l'utilisateur.

(G) Le point qui, parmi toutes les matrices d'accumulation, présente la plus grande intensité est voté comme étant le centre d'un cercle de rayon r. (rayon de l'itération en question).

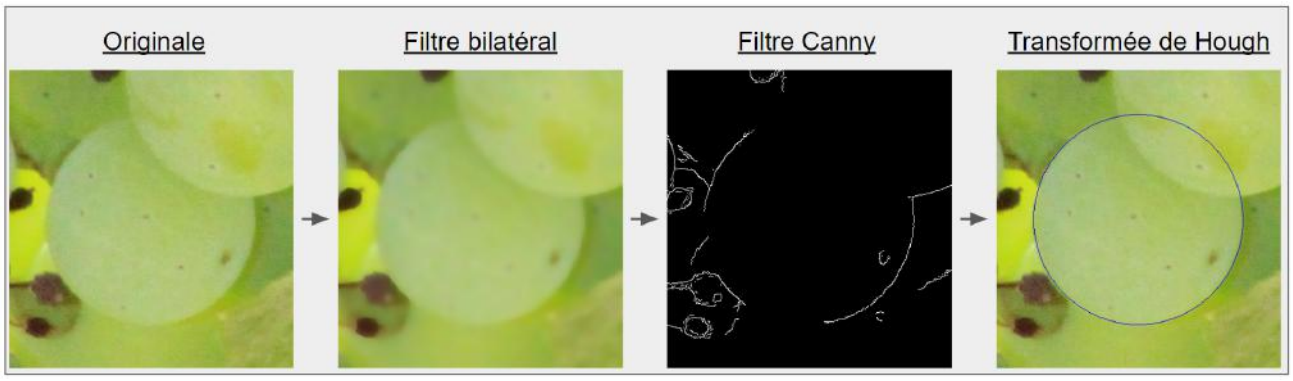


Figure 16 : Transformée de Hough circulaire sur les contours trouvés.

Cet algorithme a l'avantage de pouvoir retrouver un cercle en dépit de la présence de bruit et d'outliers. De plus, et en dépit des critiques qui lui sont faites, le temps de computation sur une baie individuelle est généralement de l'ordre de la fraction de seconde. Cependant, ses inconvénients l'emportent sur ses avantages :

— Il nécessite au moins une vague connaissance du rayon probable du cercle à trouver. (Même si j'ai su créer une fourchette de valeurs basée sur la taille de l'image rognée, ce n'est pas idéal)

— Il repose grandement sur la qualité des contours de Canny

— Il a un biais vers le plus grand contour de cercle trouvé, dans la mesure où le rayon de ce cercle resterait dans les bornes définies. Ainsi, dans le malencontreux cas où la baie d'intérêt (petite) serait en partie cachée par une baie secondaire (plus grande), il est probable que le maximum d'accumulation soit pour la baie secondaire.

— Et le plus gênant de tous, il ne peut trouver que des cercles, ce qui n'est pas congruent avec la forme naturelle des raisins qui sont rarement des sphères parfaites.

Pour cette dernière raison en particulier, j'ai décidé de ne pas retenir cet algorithme, qui m'avait pourtant mené dans de nombreuses directions les premiers mois de mon stage. L'alternative évidente était d'utiliser la transformée de Hough elliptique, un algorithme similaire, mais qui au lieu de créer sur chaque pixel des cercles d'un rayon r, il crée des ellipses, une forme bien plus proche de la réalité géométrique du raisin. Malheureusement, c'est ce même point qui rend cet algorithme inutilisable. Une ellipse est définie par 5 paramètres : les coordonnées x et y de son centre, les longueurs de ses axes majeur et mineur, et l'angle de rotation de l'ellipse. Par conséquent, pour chaque point de contours trouvé par Canny, la transformée de Hough doit essayer toutes les combinaisons de longueurs d'axes et d'angles possibles parmi les bornes établies, et le temps de computation devient alors très prohibitif. Pour cette raison, cette option a également été rejetée.

Régression elliptique et enveloppe convexe

OpenCV propose une fonction fitEllipse(), qui réalise une régression elliptique par la méthode des moindres distances carrées. L'ellipse trouvée est alors celle qui minimise la distance orthogonale carrée de tous les points de données à l'ellipse. Cette fonction est très rapide, mais a le défaut de prendre en compte TOUS les points d'entrée.

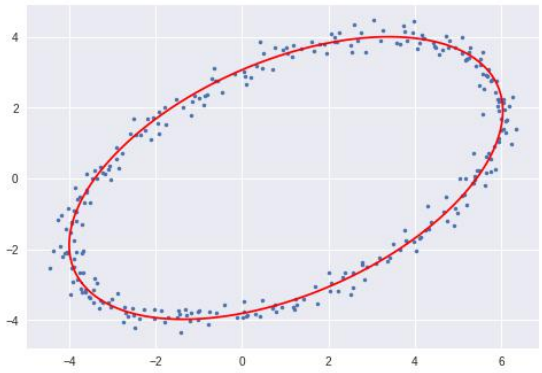


Figure 17 :L'ellipse trouvée cherche à satisfaire la totalité des points

Par conséquent, cette fonction ne peut pas être utilisée sur la production de Canny ou de HED, mais uniquement sur les contours du masque produit par Detectron2. Mais même dans les contours du masque peuvent poser problème si la baie était en partie cachée par une autre, car les contours ne sont plus tous représentatifs de la forme réelle de la baie, et l'ellipse produite sera "écrasée". Pour faire face à ce problème, j'ai proposé une solution innovante reposant sur l'enveloppe convexe des contours.

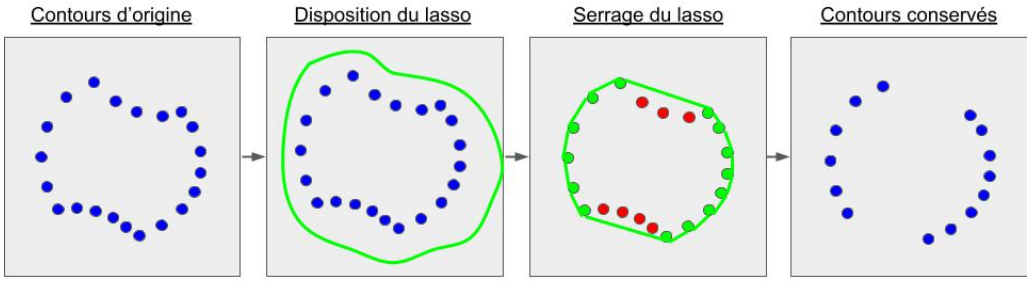


Figure 18 : Schéma du concept d'enveloppe convexe

Pour une baie qui serait partiellement cachée par deux autres baies, une en haut, et une en bas, nous obtiendrions le genre de contours que nous pouvons observer à gauche de la figure X. Dans ce cas, nous voulons nous débarrasser des contours dûs à l'occlusion de la baie (concaves), et conserver ceux qui représentent les limites réelles de la baie (convexes). Pour ce faire, j'ai déterminé l'enveloppe convexe des contours, qui est l'ensemble convexe le plus petit qui contient tous les points. Nous pouvons imaginer un lasso qui se resserre autour de nos points. Seuls les points en contact avec le lasso sont conservés. Cette méthode est efficace éliminer les contours gênants. Dans ces nouvelles conditions, une régression elliptique peut-être réalisée, donnant des résultats satisfaisants de manière très rapide.

RANSAC Ellipse fitting

RANSAC (RANdom SAmple Consensus) est une méthode itérative permettant d'estimer les paramètres d'un modèle mathématique particulièrement efficace pour ignorer les données qui ne nous intéressent pas (outliers), et ne coller qu'à celles pertinentes (inliers). Dans notre cas, les inliers sont uniquement les points du contour réel de la baie, et les outliers sont tous les autres points. Miao, Huang et Zhang **[5]** ont eux-mêmes implémenté l'algorithme RANSAC pour déterminer les paramètres des ellipses, sur la base des contours produits par leur HED amélioré.

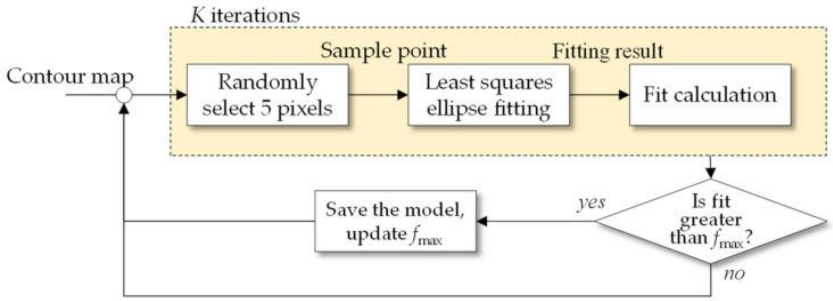


Figure 19 : Schéma récapitulant le workflow de RANSAC pour fitter une ellipse sur une carte des contours. Miao, Huang et Zhang.

A partir de la carte des contours, 5 points sont choisis au hasard, sur lesquels une régression elliptique est réalisée. Un score est calculé pour déterminer la pertinence de l'ellipse, et ses paramètres sont sauvegardés si c'est la meilleure ellipse trouvée jusqu'ici. Ce procédé est répété pour K itérations. Malheureusement, mise à part la figure ci-dessus, ils ne donnent pas davantage de détails sur l'implémentation de leur algorithme. Le zone d'ombre se trouve essentiellement sur la manière dont est calculée le score de pertinence de l'ellipse (Fit calculation).

J'ai donc programmé ma propre implémentation de l'algorithme RANSAC pour fitter des ellipses. Ma réflexion s'est surtout posée sur le calcul du score, que j'ai décidé de réaliser de la manière suivante :

A partir de X points prélevés au hasard sur ma carte des contours, je fit une ellipse. J'appose cette ellipse (en ajustant l'épaisseur du trait) sur ma carte des contours, et je compte le nombre de pixels de contours se trouvant sous l'ellipse. Mon score est la somme de ces pixels. Cependant, pour ne pas avoir un biais vers la plus ellipse possible, je pénalise son périmètre. De plus, afin de ne pas avoir d'ellipses très excentriques (très étirées) avec de grands scores, je pénalise l'excentricité de l'ellipse. Les pénalisations sont difficiles à ajuster, et je dirai que mon algorithme final à un léger biais vers les baies rondes. Sur les baies naturellement excentriques, il peut arriver à mon implémentation de raccourcir l'ellipse à une des extrémités. Dans mon implémentation, les points sélectionnés au hasard sont toujours au minimum de 8 contrairement à ce qui est proposé par Miao, afin d'éviter de piocher aléatoirement 5 pixels très proches qui pourraient donner une ellipse à très haut score. J'ai estimé empiriquement que la convergence vers l'ellipse quasi optimale nécessite entre 10000 et 15000 itérations. Voici un pseudo code de mon implémentation de RANSAC pour fit une ellipse sur une carte de contours.

```
Meilleur_score = 0
Meilleur_modele = None

Pour i dans K itérations :
    Prendre x points aléatoires sur carte des contours

    Fitter une ellipse sur ces points

    Modele = paramètres de ellipse

    Représenter ellipse sur carte des contours avec épaisseur = L pixels

    Compter nombre NB de pixels de contours présents sous cette ellipse

    Calculer son périmètre P et son excentricité EXC

    Pénalité = périmètre * excentricité

    Score = NB/Pénalité

    Si Score > Meilleur_score :
        Meilleur_score = Score
        Meilleur_modele = Modele
```

Figure 20 : Pseudocode récapitulant mon implémentation de RANSAC pour trouver une ellipse. L'entrée est une carte des contours, la sortie est le meilleur modèle (paramètres de la meilleure ellipse) et son score.

Dans mon programme final, trois des quatres méthodes disponibles reposent sur l'algorithme RANSAC.

3.4 Le tracking automatique

Le tracking automatique est le fait d'établir le suivi d'un objet au fil des images qui se suivent au cours d'une série temporelle. Ce procédé n'est appliqué à ma connaissance qu'aux vidéos, en conséquence les algorithmes existants ne sont adaptés qu'à celles-ci. La base de leur fonctionnement est le suivant : le programme essaie d'associer chaque boîte de la photo n à une boîte de la photo n+1, en se basant sur la minimisation de la distance euclidienne du centre de chaque boîte. Plus grossièrement, l'image essaie de retrouver d'une image à l'autre l'objet qui était à peu de choses au même endroit en coordonnées de pixels, ce qui a du sens pour les images d'une vidéo.

Dès lors, on comprend les implications : pour que le tracking automatique fonctionne, il faut que les coordonnées en pixels d'un même objet soient quasi-identiques d'une photo à l'autre. Cela signifie que la prise de vue doit être strictement la même, ce qui n'est pas réaliste pour un appareil photographique tenu à la main.

J'ai alors essayé plusieurs méthodes pour raccorder les coordonnées des photos de manière automatique, dont la clé de voûte était de trouver divers points clés, significatifs et uniques à travers les photos, afin d'estimer les déformations géométriques à appliquer pour faire superposer ces points entre chaque image. Malheureusement, ces procédés nécessitent des conditions presque opposées aux nôtres : il est préférable d'avoir une image hétérogène, avec du contenu variable, et des zones anguleuses pour placer des points clés efficaces. Nos photos de raisins sont homogènes et représentent un motif très répété et similaire (la baie de raisin), et il n'y a pas d'angles marqués car ce ne sont que des sphères ou des ellipsoïdes.

J'ai donc voulu inventer ma propre méthode de tracking, qui elle ne reposerait pas sur la position des objets dans l'image, mais sur les positions relatives des objets entre eux. (les baies étant relativement immobiles, la baie X sera toujours juste à gauche de la baie Y) Pour ce faire, j'ai exprimé les objets non pas par leurs coordonnées x et y dans l'image, mais par leur coordonnées polaires : à chaque baie est associée une liste de coordonnées polaires correspondant à la position des autres baies. En prenant la baie X pour sujet, la baie Y est située à un angle de 39° et à une distance de 560 pixels, la baie Z à (89°, 120 pixels), etc.

Je représentais ensuite cette liste de coordonnées associée à chaque baie par une courbe, que j'appelle la signature de la baie. Mon idée était de réaliser ensuite un appairage de signatures d'une photo à l'autre par clustering, en se basant sur la distance entre les courbes. Malheureusement, mon attention a été attirée sur d'autres défis et je n'ai pas pu mettre en place le clustering. La figure suivante montre un appairage fait à la main de quelques signatures entre deux photos.

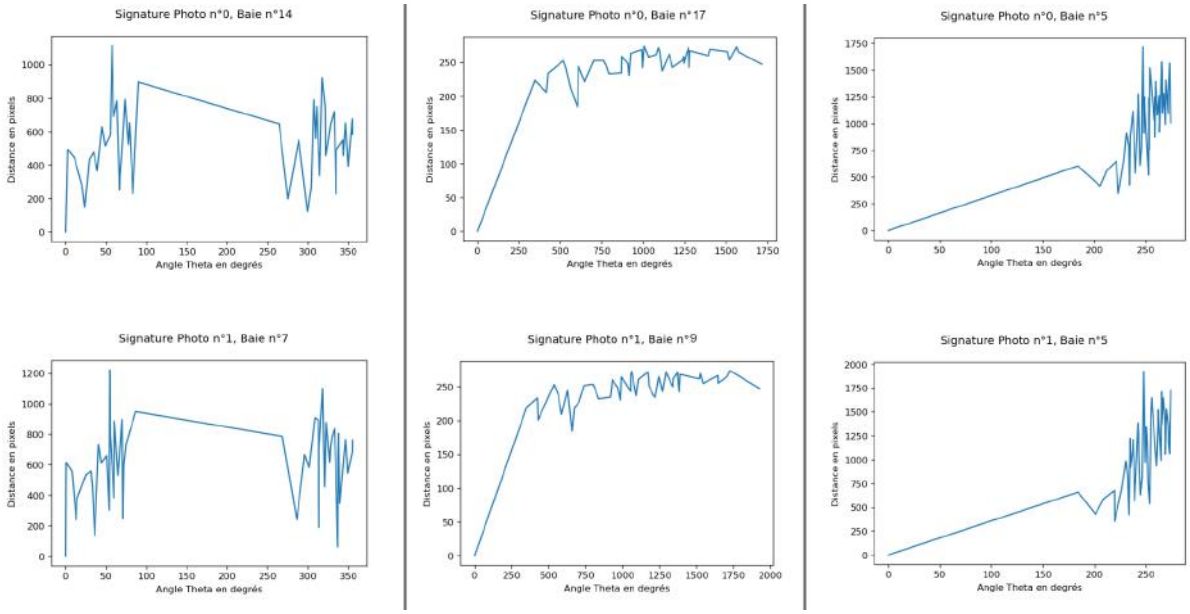


Figure 21 : Exemples de signatures appairées entre deux photos qui se suivent.

3.5 Le programme final : BerryTracker

Le tracking automatique n'ayant pas pu être réalisé lors de mon stage, et les diverses méthodes de calculs d'ellipses ne pouvant être jugées sans aide visuelle, le besoin est apparu de développer une interface graphique. J'ai donc développé, à l'aide du module Tkinter sous Python, un logiciel nommé BerryTracker permettant de réaliser la détection d'objets, proposant une aide visuelle bienvenue au tracking manuel des baies, et qui permet à l'utilisateur de comparer et sélectionner les meilleures ellipses sur les baies qu'il étudie. Voici le workflow de mon programme final :

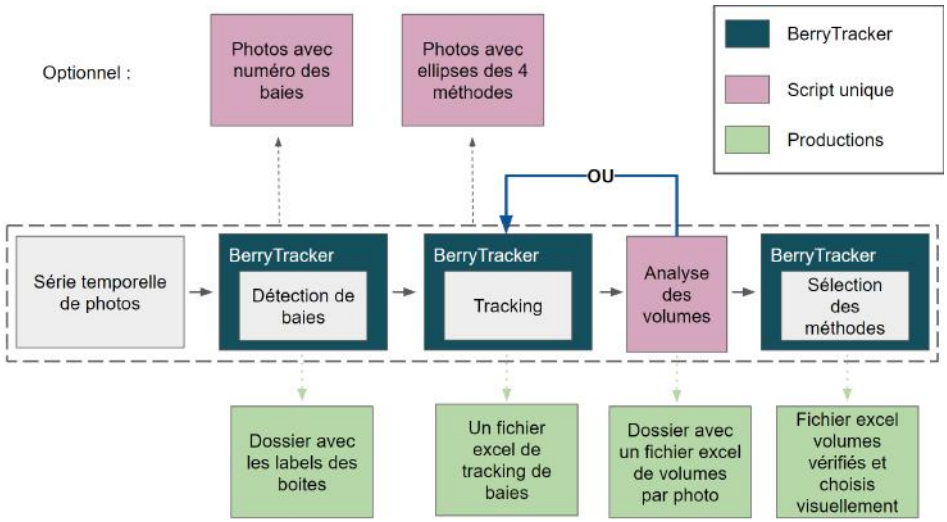


Figure 22 : Workflow de mon programme final.

L'utilisateur dispose d'un dossier de photos contenant une série temporelle. Il exécute sur BerryTracker la détection de baies, avant de passer au tracking des baies qui l'intéressent grâce à un module d'assistance manuelle (voir Annexe 2). Lorsque le tracking est terminé, l'utilisateur peut fermer le logiciel et lancer depuis un script l'analyse des baies suivies, c'est-à-dire le calcul d'ellipses par 4 méthodes différentes. Ces 4 méthodes sont récapitulées dans le tableau suivant :

Méthode	Détection de contours	Calcul d'ellipse
RHE	HED	RANSAC
RHCE	HED + Canny	RANSAC
D2HE	Masques Detectron2 nettoyé par enveloppe convexe	Régression elliptique
D2RE	Masques Detectron2	RANSAC

Figure 23 : Récapitulatif des méthodes de détection d'ellipse.

Ce script n'est pas inclus directement dans BerryTracker car il représente la partie la plus chronophage du workflow, et il m'a semblé plus pertinent de le laisser sous cette forme dans l'intention de l'exécuter sur un cluster de calcul à l'avenir.

Enfin, une fois les ellipses calculées, l'utilisateur peut retourner sur BerryTracker afin de sélectionner, pour chaque baie et chaque photo, l'ellipse qui lui convient le mieux. (voir Annexe 3) Il y a divers outputs à chaque stade du workflow, dont certains optionnels, et l'utilisateur a le

loisir de les analyser. Le produit final est un tableau excel, récapitulant pour chaque baie et chaque photo les paramètres de l'ellipse choisie et le volume associé.

3.6 La détection de véraison

Une autre de mes productions a été la détection automatique de la véraison des baies (leur stade de maturité) par analyse colorimétrique. Cet aspect de mon travail n'a pas été incorporé dans BerryTracker par manque de temps, mais il est parfaitement fonctionnel.

Mon approche a été celle de la science des données. J'ai d'abord créé un set d'entraînement avec nos images de baies classées selon 5 stades de véraison. A partir d'un échantillon de pixels prélevés au centre de la baie, j'ai extrait toute une batterie de variables issues de différents espaces colorimétriques, et après une étude de corrélation entre ces variables et la vérité terrain de la véraison, j'en ai retenu moins de dix. Puis, à partir de ces variables, j'ai entraîné divers modèles de classification et ai comparé leurs performances respectives, récapitulées dans le tableau suivant :

Model	Accuracy	Balanced Accuracy	F1 Score	Time Taken
LinearDiscriminantAnalysis	0,982	0,982	0,982	0,007
ExtraTreesClassifier	0,964	0,962	0,964	0,068
BaggingClassifier	0,945	0,948	0,946	0,019
RandomForestClassifier	0,945	0,944	0,946	0,105
DecisionTreeClassifier	0,927	0,928	0,927	0,006
LGBMClassifier	0,927	0,924	0,928	0,106
XGBClassifier	0,927	0,924	0,928	0,069
GaussianNB	0,927	0,922	0,926	0,006
LogisticRegression	0,909	0,917	0,909	0,017
LinearSVC	0,909	0,906	0,903	0,015
SVC	0,909	0,904	0,910	0,007
RidgeClassifierCV	0,909	0,902	0,904	0,008
PassiveAggressiveClassifier	0,891	0,889	0,892	0,008
SGDClassifier	0,873	0,873	0,868	0,008
LabelPropagation	0,873	0,867	0,870	0,007
LabelSpreading	0,873	0,867	0,870	0,007
CalibratedClassifierCV	0,873	0,862	0,857	0,062
ExtraTreeClassifier	0,855	0,848	0,841	0,006
RidgeClassifier	0,855	0,848	0,840	0,007
NuSVC	0,855	0,845	0,847	0,007
QuadraticDiscriminantAnalysis	0,855	0,845	0,848	0,007
NearestCentroid	0,818	0,824	0,817	0,006
Perceptron	0,818	0,819	0,814	0,008
KNeighborsClassifier	0,800	0,787	0,782	0,007
BernoulliNB	0,745	0,754	0,739	0,006
AdaBoostClassifier	0,673	0,635	0,594	0,055
DummyClassifier	0,145	0,134	0,143	0,006

Figure 24 : Tableau récapitulatif de tous les modèles de classifiers entrainés

Il est à noter que la précision n'est pas stable, et dépend de la séparation faite au vol des données en sets d'entraînement et de validations. Les précisions réelles sont légèrement inférieures à celles du tableau. Plutôt que de ne retenir que le modèle au sommet de la liste, j'ai créé une solution plus précise et plus robuste en utilisant une méthode d'ensembling : j'ai rassemblé

un nombre impair (7) de modèles dont le fonctionnement est suffisamment différent pour ne pas faire les mêmes erreurs aux mêmes endroits, et ai combiné leurs prédictions. Ainsi, le stade de véraison est déterminé par le vote démocratique de ces sept modèles, et la précision finale du modèle est constamment supérieure à 96%.

Chapitre 4

Résultats

L'analyse des temps de computation des diverses méthodes lors du calcul des ellipses révèle que la méthode D2EH est de loin la plus rapide. Le temps de calcul plus élevé sur les trois autres méthodes s'explique par le fait qu'elles utilisent RANSAC. La différence de temps de calcul entre ces trois méthodes, elle, s'explique par ce qui est donné en entrée à RANSAC : plus le nombre de points de contours est élevé, plus l'algorithme a besoin de temps. Pour cette même raison, plus une l'image d'une baie est résolue, plus le temps de computation est long.

Résolution \ Méthode	RHE	RHCE	D2EH	D2RE
400x400	~78s	~14s	~0.006s	~24s
200x200	~26s	~8s	~0.005s	~13s
100x100	~10s	~6s	~0.003s	~9s

Figure 25 : Temps de computation du calcul d'ellipse en fonction de la méthode, et de la taille de l'image rognée autour de la baie.

L'analyse de la "popularité" des méthodes (quelle méthode donne l'ellipse la plus souvent sélectionnée par l'utilisateur) montre une disparité :

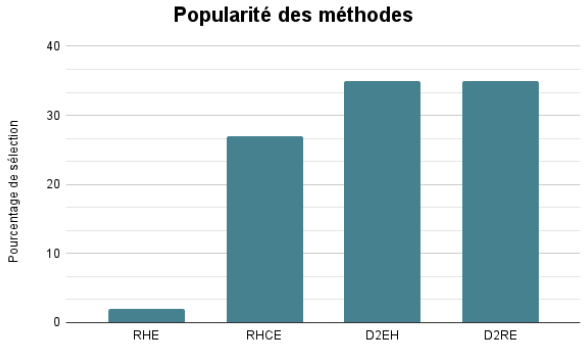


Figure 26 : Popularité des quatre méthodes en pourcentage.

Ces résultats sont un peu biaisés contre la méthode RHE, car je l'ai régulièrement désactivée à cause de son temps de calcul trop élevé. Néanmoins, j'ai pu constater qu'elle était de toutes façons très peu choisie, elle n'est que très rarement celle qui donne les meilleures ellipses. A la lumière de ces informations, j'ai intégré dans mon script 3 options de vitesse d'analyse de volumes :

— Normal = Toutes les méthodes sont actives

— Fast = RHE est désactivé (temps de calcul divisé par trois)

— Faster = RHE désactivé + baisse de la résolution des images si elles dépassent un certain seuil. (gain de temps supplémentaire au prix d'une précision moindre)

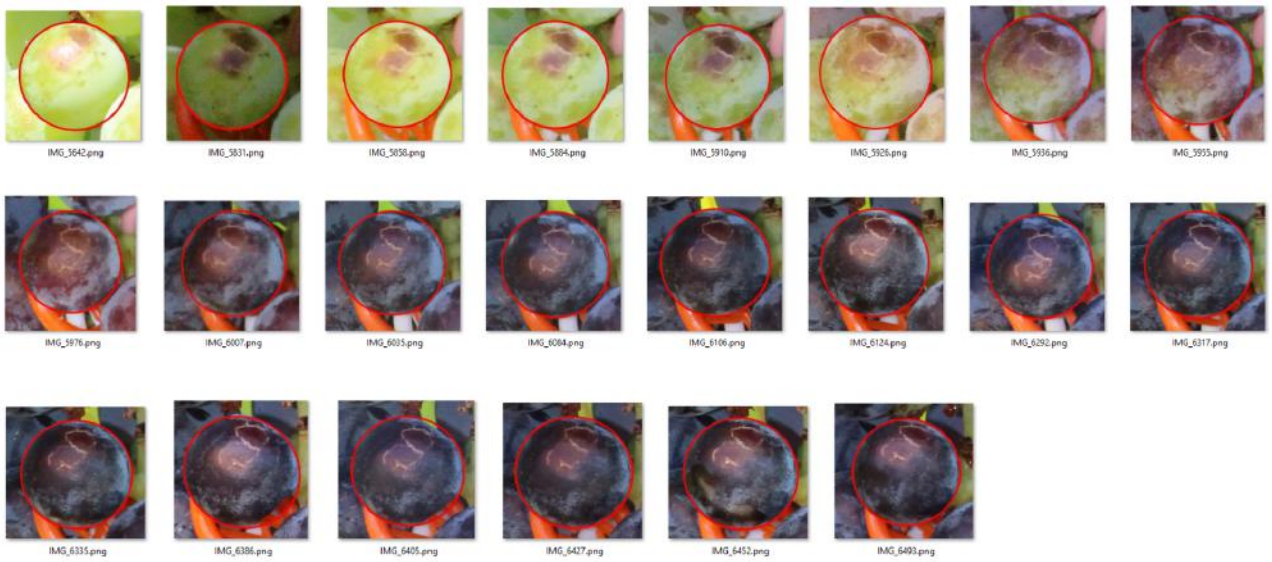


Figure 27 : Série temporelle d'une baie avec ellipses calculées.

Il s'agit ici de la représentation visuelle seulement, mais tous les paramètres et volumes des ellipses sont sauvegardés dans un fichier excel de manière automatique. Au cours d'une série, il est fréquent que sur une ou deux photos aucune des ellipses proposées ne soit satisfaisante. Dans ce cas, l'utilisateur devra placer sur une ellipse à la main par un autre moyen, par exemple avec le logiciel de traitement d'image ImageJ.

Il y a plusieurs choses qui peuvent faire faillir mon programme :

1. Une faible résolution
2. Un manque de contraste au niveau des contours des baies
3. Un flou trop prononcé
4. Une trop grande occlusion sur la baie observée (peut-être cachée par d'autres baies)
5. Une illumination très hétérogène (une partie de la baie au soleil, l'autre à l'ombre, et souvent une séparation entre les deux qui va être interprétée comme un contour saillant par le programme et fausser les résultats)

Quand trop de ces conditions s'accumulent, le programme ne donne pas de bons résultats.

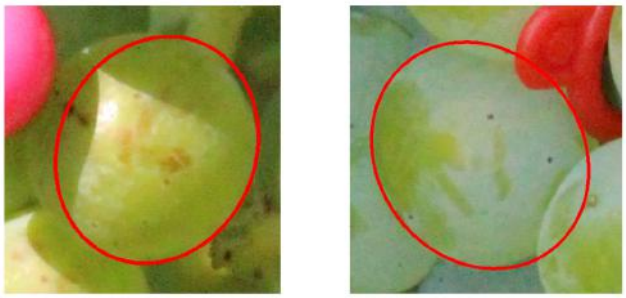


Figure 28 : Exemples de calculs d'ellipse ayant échoué. A gauche l'échec est probablement attribuable au manque de contraste général, si ce n'est entre la délimitation ombre/lumière. A droite, l'occlusion était trop importante.

Tout comme les baies de raisins ne sont pas des sphères, elles ne sont pas des ellipsoïdes parfaits non plus, ce qui complique parfois le fitting d'une ellipse. Par ailleurs, il réside une fragilité dans notre méthodologie : l'aire projetée d'un ellipsoïde dépend de la perspective de l'observateur. A l'instar d'un ballon de rugby, il peut apparaître rond sous un angle, et avoir la forme d'une olive sous un autre. Les baies de raisins sont généralement dans le même angle tout le long de leur croissance, mais si le pédicelle d'une baie se tord, cela pourrait fausser le suivi de son volume alors même que les ellipses trouvées seraient parfaites.

Chapitre 5

Conclusion

En conclusion, j'ai développé un flux de travail accompagnant le chercheur dans l'analyse de ses photos de raisins, de la détection de baies au calcul de leur volume, en passant par le tracking. La détection de baies intégrée sur BerryTracker est rapide et particulièrement robuste, grâce à un entraînement de YOLOv5 réalisé sur un set de données important. La détection du contour des baies est réalisée en parallèle par différents algorithmes (HED et Detectron2), et j'ai proposé de surcroît deux méthodes pour améliorer leur sorties (mutualisation de HED avec le filtre Canny, et nettoyage des contours de Detectron2 par enveloppe convexe). Le calcul des ellipses est réalisé par 4 méthodes, dont 3 reposent sur l'algorithme RANSAC que j'ai implémenté avec ma propre approche de scoring. La précision de ces ellipses demeure dépendante de certains facteurs de prise de vue, mais semble proche de l'état de l'art. La détection de véraison est réalisée par une approche d'apprentissage machine, et a une précision supérieure à 96%. Je n'ai pas réussi à implémenter un tracking automatique des baies, mais avec plus de temps, celui-ci aurait peut-être été possible. A défaut de cela, j'ai développé une assistance manuelle et visuelle via une interface graphique pour aider l'utilisateur à réaliser son tracking à la main. En outre, je pense avoir ouvert une voie innovante pour un tracking basé sur la position relative des objets entre eux, ce qui est tout à fait pertinent quand cette relation demeure identique au cours de la série temporelle. Avec plus de temps, j'aurais souhaité pouvoir installer certains scripts sur le cluster de calcul, afin de réduire le temps de computation et de faciliter l'utilisation de mon programme depuis les différentes machines de l'équipe.

Au cours de mon stage, j'ai pu découvrir le phénotypage à haut débit, un aspect de la bioinformatique qui m'était jusqu'alors inconnu, et qui dans l'ombre de ses cousins -omiques semble occuper une place de plus en plus prépondérante sur la scène viticole-vinicole, voire agricole dans sa globalité. Le développement de l'informatique et de la robotique a donné l'impulsion à cette caractérisation non-invasive et non-destructive des produits cultivés ces dernières années, et le bioinformaticien a une place à occuper dans ce contexte, non pas en analysant des séquence biologiques mais des informations visuelles.

Chapitre 6

Remerciements

Je remercie mon tuteur Charles ROMIEU pour son encadrement bienveillant, sa confiance et sa discussion toujours intéressante. Il m'a permis de travailler dans une grande autonomie, d'explorer et de rechercher, tout en me rappelant au pragmatisme de la recherche et à la nécessité d'atteindre des objectifs à court terme, tangibles et utiles à l'équipe. Je pense avoir eu grâce à lui un vrai aperçu de ce qu'est la recherche.

Chapitre 7

Références bibliographiques

[1] Rezk Shahood, Laurent Torregrosa, Stefania Savoi, Charles Romieu. First quantitative assessment of growth, sugar accumulation and malate breakdown in a single ripening berry. OENO One, Institut des Sciences de la Vigne et du Vin (Université de Bordeaux), 2020, 54 (4), pp.1077 - 1092. ⟨10.20870/oeno-one.2020.54.4.3787⟩. ⟨hal-03039736⟩

[2] Zabawa, Laura Kicherer, Anna Klingbeil, Lasse Toepfer, Reinhard Kuhlmann, Heiner Roscher, Ribana. (2020). Counting of Grapevine Berries in Images via Semantic Segmentation using Convolutional Neural Networks.

[3] Nellithimaru, A.K. ; Kantor, G.A. ROLS : Robust Object-level SLAM for grape counting. In Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPR), Long Beach, CA, USA, 15–20 June 2019 ; pp. 2648–2656.

[4] Luo, L. ; Liu,W. ; Lu, Q. ;Wang, J. ;Wen,W. ; Yan, D. ; Tang, Y. Grape Berry Detection and Size Measurement Based on Edge ImageProcessing and Geometric Morphology. Machines 2021, 9, 233. https ://doi.org/10.3390/machines9100233

[5]
Miao Y, Huang L, Zhang S. A Two-Step Phenotypic Parameter Measurement Strategy for Overlapped Grapes under Different Light Conditions. Sensors (Basel). 2021 Jul 1 ;21(13) :4532. doi : 10.3390/s21134532. PMID : 34283081 ; PMCID : PMC8272069.

[6] S. Xie and Z. Tu, "Holistically-Nested Edge Detection," 2015 IEEE International Conference on Computer Vision (ICCV), 2015, pp. 1395-1403, doi : 10.1109/ICCV.2015.164.

[7] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. 2010. The Pascal Visual Object Classes (VOC) Challenge. Int. J. Comput. Vision 88, 2 (June 2010), 303–338. https ://doi.org/10.1007/s11263-009-0275-4

[8] Lin, TY. et al. (2014). Microsoft COCO : Common Objects in Context. In : Fleet, D., Pajdla, T., Schiele, B., Tuytelaars, T. (eds) Computer Vision – ECCV 2014. ECCV 2014. Lecture Notes in Computer Science, vol 8693. Springer, Cham. https ://doi.org/10.1007/978-3-319-10602-$1_4$8

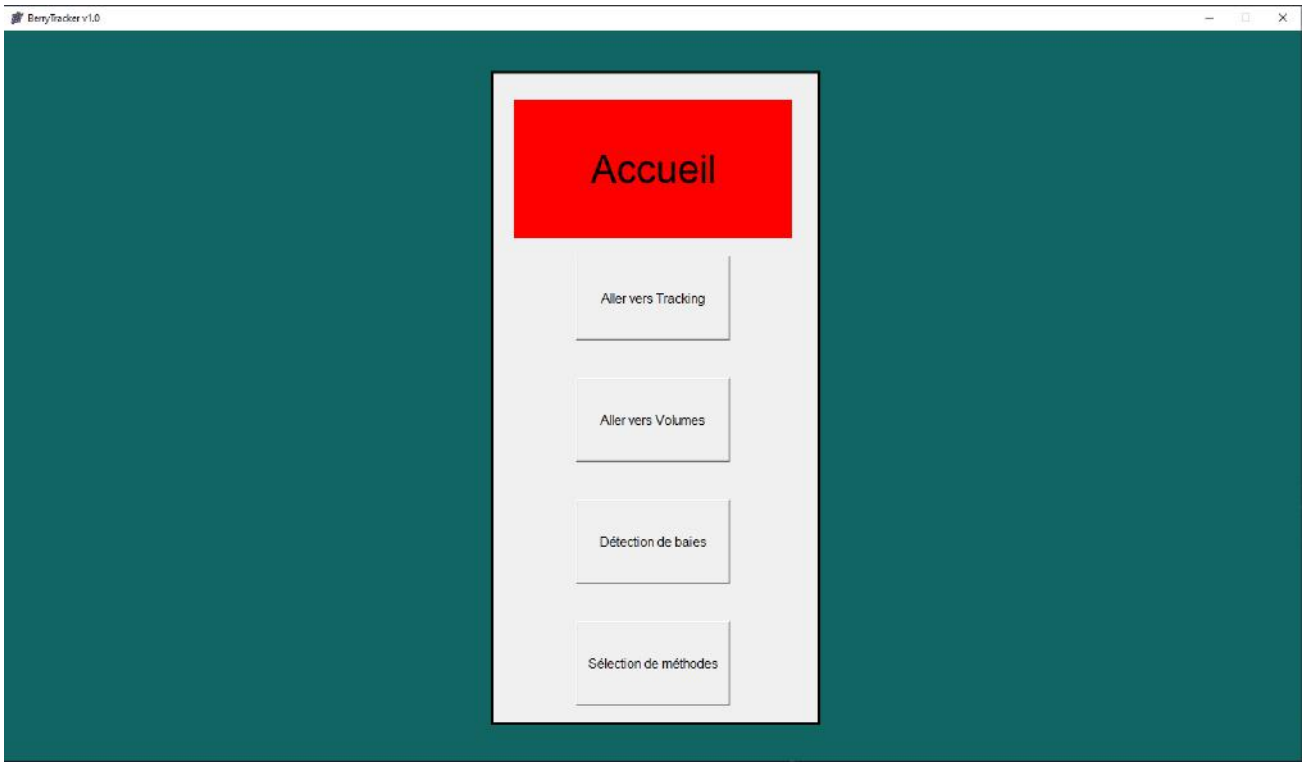
[9] You Only Look Once : Unified, Real-Time Object Detection Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi University of Washington, Allen Institute for AI,

Facebook AI Research http ://pjreddie.com/yolo/

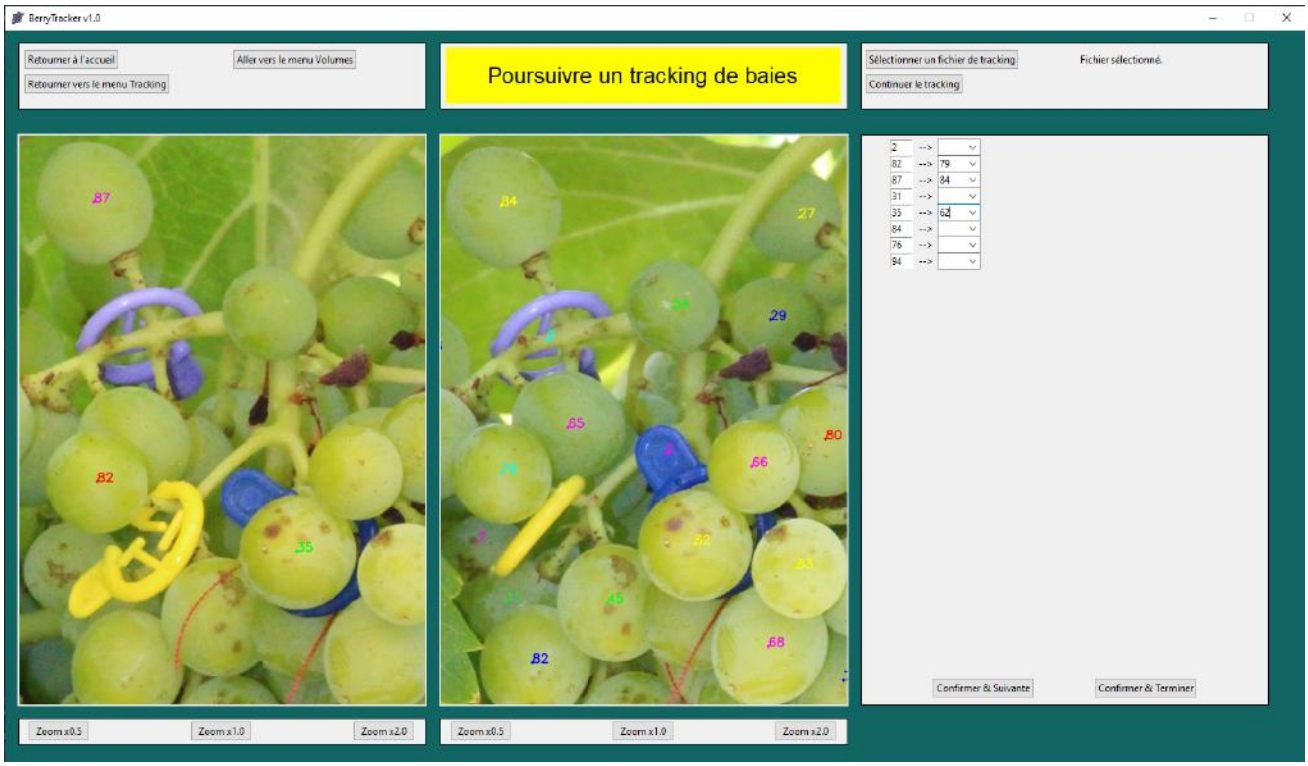
[10] Yuxin Wu and Alexander Kirillov and Francisco Massa and Wan-Yen Lo and Ross Girshick, title = Detectron2, howpublished = https ://github.com/facebookresearch/detectron2, year = 2019

Chapitre 8

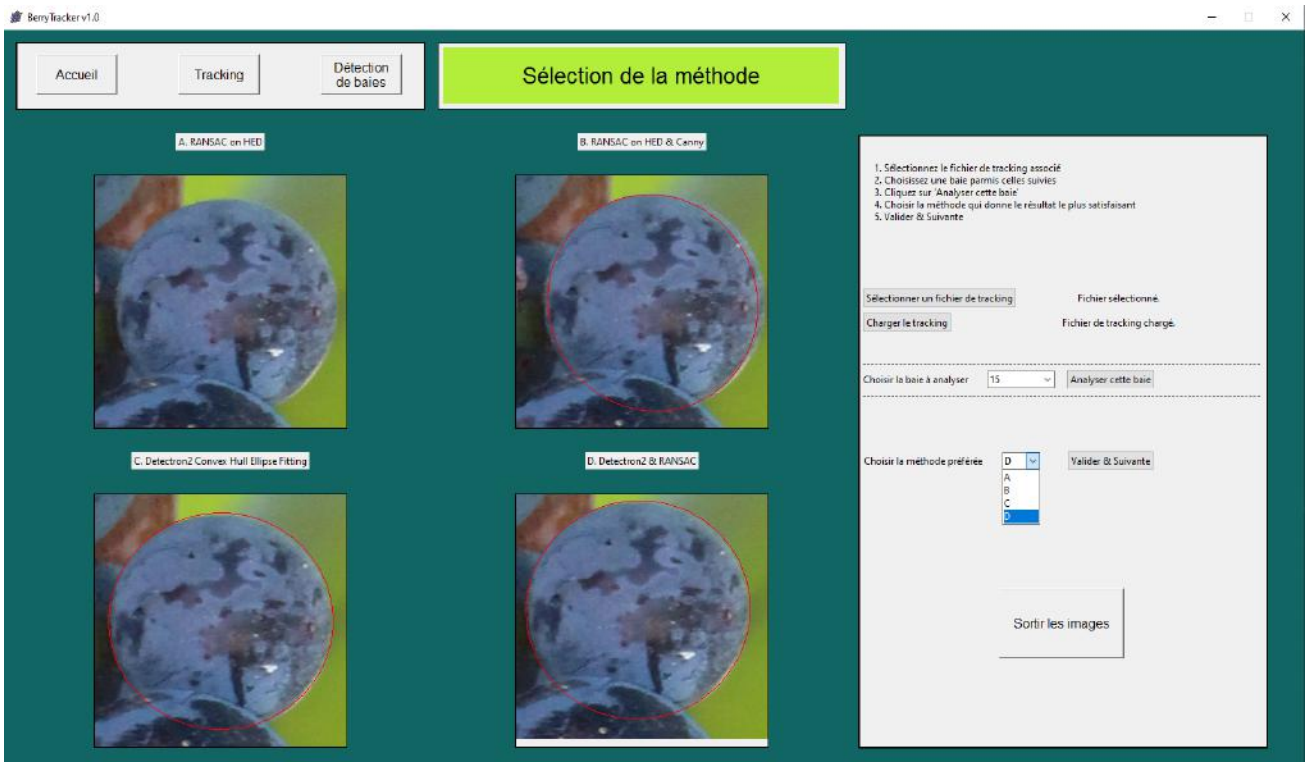
Annexe



Annexe 1 : Accueil de BerryTracker



Annexe 2 : Assistance visuelle au tracking sur BerryTracker



Annexe 3 : Sélection de la méthode préférée sur BerryTracker