Rapport de développement logiciel - IS1260
Détection de contours

Julien Duquesne

Table des matières

1 Introduction

La vision par ordinateur est une technologie en grande progression avec l'évolution des techniques d'intelligence artificielle telles que le machine learning. Elle permet par exemple de reconnaître de façon automatisée des formes et images, et a pour application la reconnaissance faciale par exemple.
La détection de contours est une brique essentielle des algorithmes de vision, pour reconnaître des objets et les compter notamment.
Dans ce rapport nous expliquerons les différentes étapes qui mènent à un algorithme de détection de contours, c'est à dire de variations rapides de l'image. Nous mettrons en oeuvre la méthode de Canny qui permet de pré-traiter et post-traiter l'image afin de la rendre moins sensible au bruit et améliorer l'implémentation d'algorithmes traitant automatiquement les images. La méthode consiste en 5 grandes étapes :
-Lissage de l'image pour en diminuer le bruit
-Estimation du gradient de l'image
-Amincissement du gradient en n'en gardant que les maximas
-Séparation des contours en faibles et forts selon la valeur de la norme
-Suppression des contours faibles non reliées à un contour fort
Ci-dessous on trouve l'image de départ ainsi que les résultats auxquels on arrivera après la mise en oeuvre de l'algorithme.
Avant de détailler la mise en place de la méthode de Canny, nous allons introduire les bases

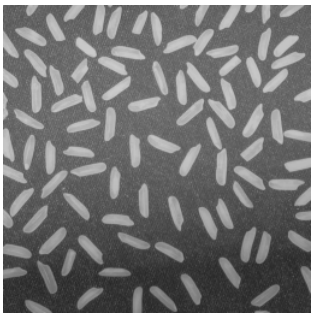

FIGURE 1 – Image de départ

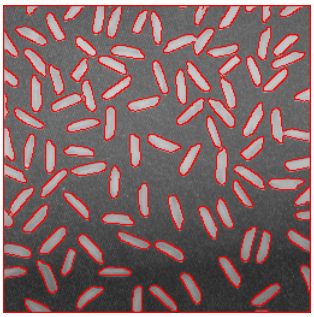

FIGURE 2 – Contours de l'image

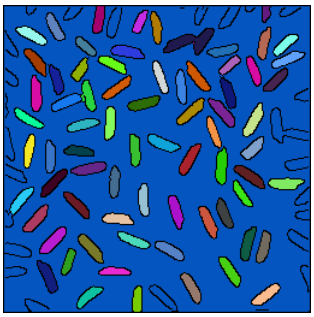

FIGURE 3 – Coloriages des grains de riz

du traitement d'image et un outil indispensable qu'est le filtrage.

2 Bases du traitement d'image

En informatique, une image est constituée de pixels qui sont des petits carrés (on en trouve en général de 10 à 36 millions dans les appareils photos classiques) restituant chacun de la lumière indépendamment des autres. Pour une image en noir et blanc, les pixels renvoient une intensité lumineuse codé entre 0 (noir) et 255 (blanc), c'est à dire sur 8 bits. Les images en noir et blanc sont donc des tableaux en deux dimensions sur Python. Pour une image en couleur, le pixel est constitué de 3 paramètres correspondant aux trois couleurs primaires que l'oeil est capable de capter à travers ses cônes , c'est le système RGB (Red Green Blue). Pour chacune de ces trois couleurs, le pixel contient renvoie une valeur entre 0 et 255. Les images en couleur sont donc des tableaux en trois dimensions sur Python.
On peut appliquer cette décomposition sur une image utilisée très fréquemment dans les

exemples de traitement d'image ci-dessous.
Pour notre application, nous traiterons l'image en noir et blanc, ce qui simplifiera les opé-

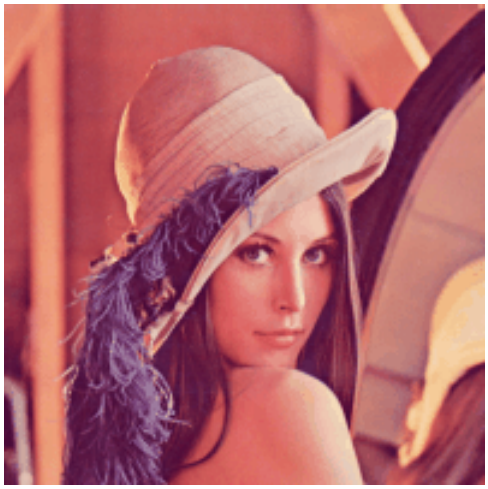

FIGURE 4 – Image coloré entièrement

FIGURE 5 – Décomposition selon les trois couleurs primaires

rations, au niveau du calcul du gradient notamment.

3 Filtrage d'une image

Le filtrage, opération de base du traitement des signaux, consiste à appliquer un opérateur linéaire et invariant par translation au signal. On peut alors écrire le résultat de l'application d'un filtre L à un signal f comme étant la convolution de f avec la réponse impulsionnelle de L, définie comme $h = L(\delta)$ où δ est la fonction valant 1 en 0 et 0 ailleurs.
La convolution de deux signaux f et g est définie par

$$f * g[j] = \sum f[i]g[j-i]$$

3.1 Filtrage en dimension 2

On définit la fonction filt2D qui prend en paramètres le signal et le noyau de la convolution (la réponse impulsionnelle du filtre).
Pour optimiser l'exécution de l'algorithme, on évite (et on évitera autant que l'on peut dans tout le code) les boucles, on privilégie ainsi ici la fonction signal.convolve2D du module scipy.

On choisit la valeur "same" pour le paramètre mode. Cela nous permet d'obtenir la sortie que l'on souhaite, i.e un signal de la même dimension que celui en entrée et centré par rapport au noyau. [1]

4 Implémentation de la méthode de Canny

4.1 Lissage de l'image

Le lissage permet d'éliminer le bruit, c'est à dire une variation brusque de l'intensité de l'image entre deux pixels adjacents introduite par l'acquisition de l'image. La détection de contours reposant sur l'identification de ces fortes variations d'intensité, la présence de bruit fausse la détection et doit donc être éliminée. Cela est réalisable en floutant l'image. Les pixels de bruit étant extrêmement localisés, le floutage va grandement en diminuer l'intensité alors que les contours seront certes moins nets mais tout de même identifiables.
On choisit de flouter l'image en appliquant un filtre gaussien définit par sa réponse impulsionelle

$$h[n,m] = \frac{1}{Z}\exp\left(-\frac{n^2+m^2}{2\sigma^2}\right)$$

où Z est tel que la somme des coefficients de h soit 1, et n et m varient entre $-k$ et k.
Il y a deux manières d'appliquer ce filtre : on peut appliquer la fonction de scipy ndimage.filters.gaussian_filter ou utiliser notre fonction filt2D avec un noyau gaussien défini dans la fonction gaussien.
On choisit la deuxième solution, car cela nous donne plus de latitude dans le choix des paramètres (σ et k), la fonction de scipy ne permettant de choisir que σ.
Pour choisir la largeur k du filtre, il faut trouver un compromis entre une grande largeur (qui augmente le temps de calcul) et un filtre suffisamment efficace. La valeur $\lfloor 2\sigma + 1 \rfloor$ semble être un bon compromis, puisque la matrice de h reste relativement petite, et qu'au delà de cette valeur, l'exponentielle devient négligeable.
Quant à la valeur de σ, plus elle augmente plus l'image est floutée, i.e plus on supprime le bruit mais les détails des contours disparaissent également. Il nous faut donc trouver un compromis. Pour cela on teste plusieurs valeurs de σ sur les figures ci-dessous.
On constate que la valeur 0.3 ne modifie quasiment pas l'image originale et ne supprime

FIGURE 6 – $\sigma = 0.3$

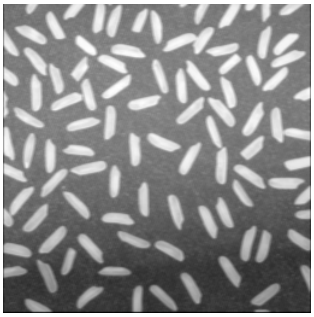

FIGURE 7 – $\sigma = 1$

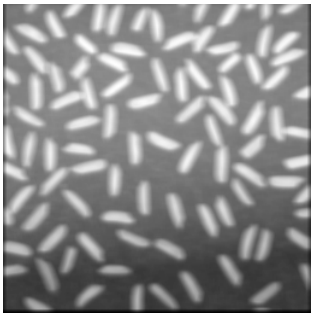

FIGURE 8 – $\sigma = 3$

donc que très peu le bruit présent. A l'inverse, la valeur $\sigma = 3$ gomme de nombreux détails

1. voir la documentation de python https ://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.signal.convolve2d.html

et rend les bords incertains, ce qui les rend très durs à détecter par la suite. Après plusieurs tests, la valeur $\sigma = 1$ semble être optimale, floutant suffisamment l'image pour en enlever la majorité du bruit mais gardant les contours très visibles.

4.2 Détection des contours de l'image

4.2.1 Calcul du gradient

En une dimension, on estime la dérivée numérique de l'image en un point par la fonction

$$D[n] = \frac{1}{2}(x[n+1] - x[n-1])$$

où x est l'intensité du pixel.
Ce n'est rien d'autre que l'approximation de la dérivée à l'ordre 2 (méthode des différences finies centrées). En effet, par les formules de Taylor-Young on a à l'ordre 2 :

$$x[n+1] = x[n] + x'[n] + \frac{1}{2}x''[n]$$

$$x[n-1] = x[n] - x'[n] + \frac{1}{2}x''[n]$$

En soustrayant les deux équations et en divisant par 2 on obtient le résultat cherché.
Avec l'expression précédente, il suffit d'appliquer le filtre de réponse impulsionelle $h = \begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix}$ pour obtenir la dérivée numérique selon x.
On va utiliser ici un filtre plus compliqué, plus robuste au bruit, qui consiste à la composition d'un filtre moyennant selon la direction y $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$ avec la dérivée numérique selon x vue précédemment. En multipliant les deux réponses impulsionnelles pour les composer, on obtient le filtre de Sobel.

$$S = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

En appliquant ce filtre et sa transposée à l'image, on obtient les matrices des dérivées numériques selon x et y. La norme et la direction du gradient s'obtiennent respectivement en prenant la racine de la somme des carrés des éléments des matrices et en calculant l'arctangente du quadrant formé par les dérivées selon x et y. Le calcul de l'arctangente se fait à l'aide de la fonction numpy arctan2.
On obtient alors la norme et la direction du gradient sur les figures 9 et 10.

4.2.2 Amincissement du gradient

Maintenant que l'on a la norme et la direction du gradient, il s'agit de ne conserver que ses maximums qui correspondent aux contours de l'image. Un pixel est un maximum du gradient si la norme du gradient en ce point est supérieur que la norme des pixels dans la direction du gradient et son opposé. Il faut donc discrétiser la direction du gradient pour savoir avec quels pixels comparer la norme du gradient en un point. On construit donc une fonction gradient_discrete qui prend en argument la valeur de l'angle de la direction du gradient et

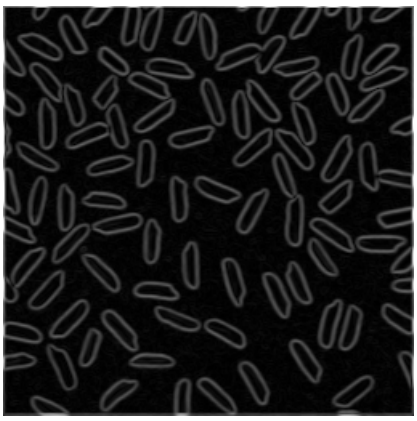

FIGURE 9 – Norme du gradient

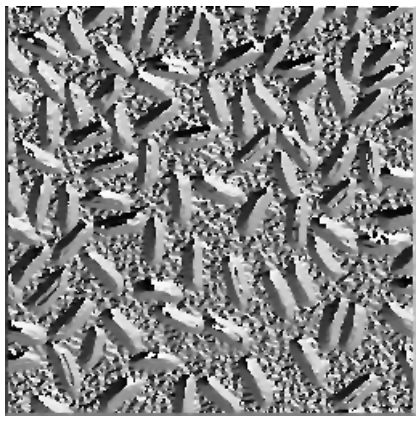

FIGURE 10 – Direction du gradient

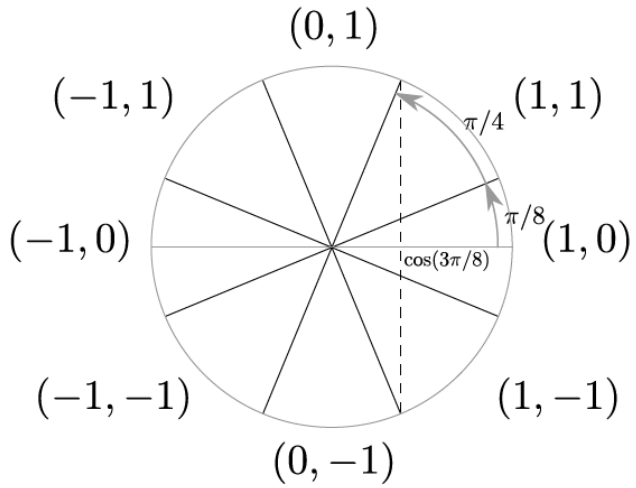

FIGURE 11 – Discrétisation du gradient - Image tirée du sujet d'IS1260 2016/2017 de Gilles Chardon

renvoie les coordonnées du pixel correspondant à cette direction.
Pour discrétiser la matrice du gradient, on a pas trouvé d'autres solution que parcourir la matrice avec des boucles for pour conserver dans une matrice contours les points où l'on est à un maximum du gradient. La fonction normalisation sert à ajouter des lignes et des colonnes de 0 aux extrémités de la matrice afin de ne pas avoir en prendre en compte dans la boucle for les points aux limites.

4.3 Séparation entre contours forts et contours faibles

On voit nettement sur l'image du gradient aminci les contours mais il y a encore du bruit, de faible intensité, qui doit être supprimé. On va alors procéder à un seuillage pour séparer les contours en 3 groupes distincts : contours forts, contours faibles et contours à éliminer. Pour cela on a besoin de 2 seuils :
-Un seuil supérieur qui déterminera les contours à considérer comme forts
-Un seuil inférieur qui déterminera les contours à considérer comme faibles

FIGURE 12 – Gradient aminci

Afin de faire ce triage, on définit la fonction tresholding et pour éviter l'usage de boucle, on utilise un sélecteur bouléen pour garder les pixels de l'image correspondants aux groupes contours forts et contours faibles. Enfin on uniformise la valeur de l'intensité de tout ces pixels en la fixant à 255 (blanc).
Il nous faut trouver les bonnes valeurs des seuils, on procède alors par tests, en choisissant d'abord le seuil inférieur pour éliminer les contours de bruit puis le seuil supérieur pour bien répartir les contours entre faibles et forts.
On constate que pour un seuil inférieur de 50, il y a beaucoup trop de bruit, il faut donc

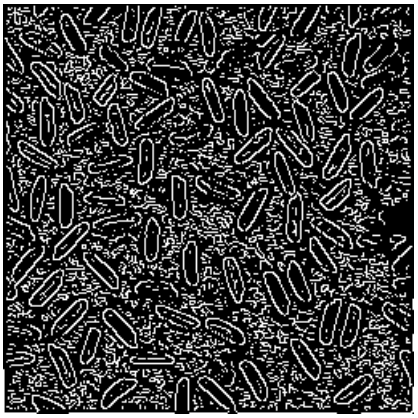

FIGURE 13 – Contours faibles pour un seuil inférieur de 50

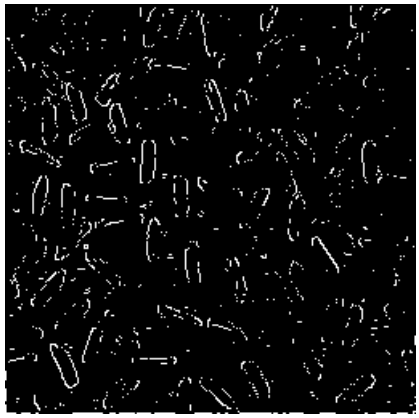

FIGURE 14 – Contours faibles pour un seuil inférieur de 120

augmenter le seuil, sans cependant supprimer des "vrais" contours. Une valeur de seuil de 120 semble être correcte, la majorité du bruit est éliminé et le reste le sera lors de l'hystérésis, l'étape suivante du filtrage.
Pour la valeur du seuil des contours forts, il faut que pour un contours donné, une partie suffisamment grande soit classée comme contour fort pour que celui-ci soit reconnu au final comme un contour. Le seuil doit alors être suffisamment bas pour accepter les contours bien marqués mais il doit également filtrer les bruits ce qui impose une valeur de seuil suffisamment

haute.
Sur les figures ci-dessus, on constate qu'une valeur de 150 est trop bas, puisqu'elle laisse

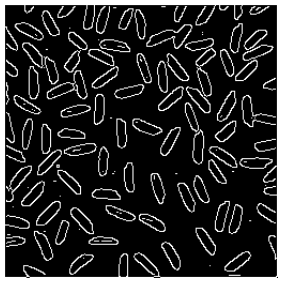

FIGURE 15 – Contours forts pour un seuil à 150

FIGURE 16 – Contours forts pour un seuil à 640

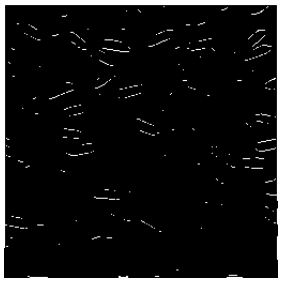

FIGURE 17 – Contours forts pour un seuil à 800

passer des contours de bruits (notamment à l'intérieur des objets). Le seuil à 800 est lui trop haut, très peu de contours y sont représentés, ce qui donne comme conséquences qu'ils manqueront dans la détection finale. La valeur de 640 semble être correcte, elle ne comporte pas de bruit, tous les contours y sont représentés et même si ils sont incomplets pour le moment, ils le seront lorsqu'ils seront reliés aux contours faibles.
Finalement on obtient la répartition des contours suivantes :

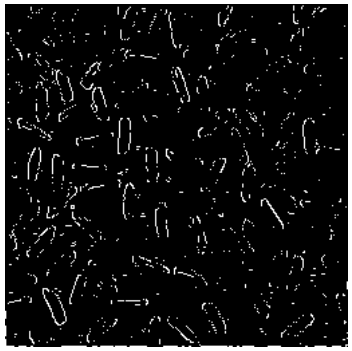

FIGURE 18 – Contours faibles

FIGURE 19 – Contours forts

4.4 Suppression des contours faibles isolés

Cette étape consiste à relier les contours forts et les contours faibles, en ne conservant que les contours faibles qui sont reliés à un pixel de contour fort. Chaque contour relié identifié est alors stocké comme une composante connexe de l'image. L'algorithme est alors proche de celui d'un parcours de graphe. Nous avons choisi de parcourir le graphe en profondeur avec une pile. On aurait pu également opter pour un algorithme récursif mais la taille de la pile de récursion de Python est assez limitée, ce qui rend la récursivité inutilisable lorsque l'image devient trop grande.
Le principe de l'algorithme est le suivant : on stocke dans une liste les pixels de contours forts, tant qu'elle n'est pas vide, on prend le premier pixel de la liste et on parcourt le graphe des pixels de contours (forts ou faibles) adjacents à l'aide d'une pile. Quand la pile est vide, on a trouvé tous les pixels adjacents, on ajoute la composante connexe que l'on vient de trouver à la liste des composantes connexes et l'on recommence avec le prochaine pixel de la liste des

contours forts (non déjà dans une composante connexe trouvée).
On peut alors superposer les contours trouvés avec l'image originale pour tracer les contours en rouge sur la figure suivante.

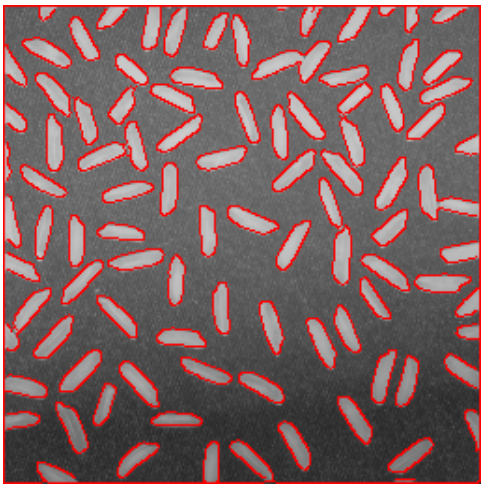

FIGURE 20 – Tracé des contours après filtrage de Canny

4.5 Segmentation

Maintenant que l'on a identifié les contours de l'image, on peut identifier les composantes connexes de l'image, qui sont les pixels non-contours de l'image adjacents entre eux (c'est à dire qu'on peut tracer un chemin de pixels non contours entre deux pixels d'une composante connexe).
Pour cela, on définit la fonction segmentation qui consiste une nouvelle fois en une parcours de graphe, en utilisant une pile. Les pixels adjacents d'un pixel d'une composante connexe sont ajoutées à cette composante connexe jusqu'à qu'il n'y en ait plus. On numérote ainsi les composantes connexes ce qui nous permet de les colorier par la suite dans des couleurs différentes (choisies aléatoirement, ce qui nous donne des assortiments plus ou moins réussis...).
On remarque qu'on a des problèmes sur les bords de l'image, les grains de riz ne sont pas

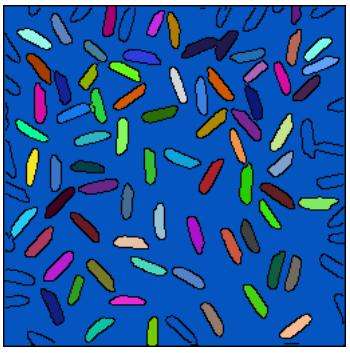

FIGURE 21 – Un assortiment de couleurs

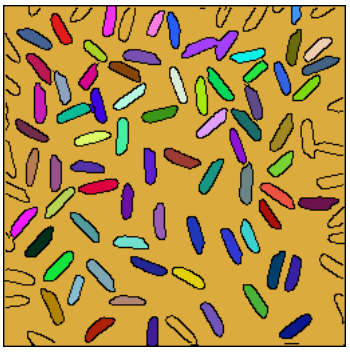

FIGURE 22 – Un autre

reconnus comme des composantes connexes. Cela est due au fait que les contours ne vont pas jusqu'au bord et qu'il y a quelques pixels entre le contour et le bord, ce qui fait que l'algorithme assimile le contour à la composante connexe du fond de l'image.

5 Conclusion

Avec ce projet, nous avons pu concevoir un algorithme de détection des contours qui fonctionne plutôt bien, bien que testé sur une image très basique. L'algorithme a donc pour le moment quelques limites :
L'image est très basique, les contours étant bien clairs et tranchés, les bruits étant assez faibles, il faudrait implémenter l'algorithme sur une image plus complexe comme celles que l'on trouve habituellement.
La définition des seuils est faîte à la main, il faudrait l'automatiser afin de pouvoir automatiser la détection de contours, quelque soit l'image en entrée pour que l'algorithme puisse fonctionner sur plusieurs images différentes, ce qui est le cas dans les technologies de reconnaissance d'image.
On a constaté qu'il y avait des problèmes sur les bords, pour le corriger, il faudrait appliquer un traitement spécial aux pixels placés sur le bord, ou bien ne pas en tenir compte et les supprimer pour ne pas fausser le rendu final.
On ne s'est intéressé qu'à une image en noir et blanc, il faut pouvoir l'élargir à une détection de contours sur des images en couleur. Une méthode pour réaliser cela serait sans doute d'appliquer l'algorithme que nous avons réalisé sur chacune des composantes RGB pour ensuite combiner les contours des trois composantes ce qui permettrait sans doute de retrouver les contours de l'image.