Projet noté

Détection des contours d'une image : le filtre de Canny

colinleverger [at] gmail [dot] com

Colin Leverger - ENSSAT Informatique, Multimédia et Réseaux
Promotion 2017

Destinataire : Benoit VOZEL

8 mai 2016

1 Introduction

Dans le cadre de la formation Informatique, Multimédia et Réseaux dispensé à l'ENSSAT de Lannion, nous avons étudié le traitement d'image en seconde année. Cette matière, qui s'inscrit dans la composante «multimédia» de notre cursus, nous a permis de comprendre les concepts clés de la manipulation d'images par programmation. Nous avons notamment travaillé à la création de filtres gaussiens, à la création de matrices de gradient, etc.

Pour pratiquer et appréhender au mieux les sujets évoqués en CM, nous avons effectué un TP noté. Il s'agissait d'écrire avec Scilab un programme permettant de trouver et d'isoler les contours des objets présents sur une image, de manière automatique. J'ai choisi d'utiliser un grand classique du traitement d'image, à savoir Lena, qui n'est autre qu'une pinup des années 70 qui se trouvait au bon endroit au bon moment (à savoir, dans les mains d'un chercheur dans le domaine de l'imagerie numérique lorsqu'il cherchait une image pour ses traitements).

Ce compte rendu présente la démarche suivie lors de la réalisation et du codage de ce TP. La structure du sujet sera suivie, car nous allons expliquer les méthodes et fonctions dans l'ordre chronologique de codage. Les fonctions et méthodes codées seront en effet très souvent réutilisées pour la suite du TP.

Le code associé à ce compte rendu pourra être trouvé dans l'archive jointe à ce rapport. Très peu de code sera directement exposé dans le rapport, mais les structures complexes et les choix d'implémentation des algorithmes y seront clairement explicités.

2 Préliminaires

Afin de travailler avec Scilab, quelques étapes préliminaires ont étés effectuées.

Tout d'abord, le choix de l'image était important pour pouvoir développer rapidement les premières fonctions. L'exécution des algorithmes prend un temps non négligeable, et il était important d'utiliser une petite image pour le début des opérations : pas plus de 64x64 pixels. Une image de 64x64 pixels représente par exemple une matrice de 64 par 64. Évidemment, plus l'image est grande (plus le nombre de pixels est important), plus la taille de la matrice à traiter est grande, et plus le traitement prend du temps.

La seconde chose à faire avant de se lancer dans le codage était l'installation du module Scilab nécessaire pour utiliser les fonctions basiques de chargement d'image. Il s'agit en effet de pouvoir transformer une image en matrice de valeurs, situées entre 0 et 255 (niveaux de gris). L'installation du module dans Scilab sur Mac OS est loin d'être triviale... L'utilisation du logiciel sous Windows est préférée.

3 Première étape : le rehaussement de Canny

La première étape de l'algorithme peut se décomposer en deux sous parties. Il s'agit tout d'abord de lisser l'image, afin d'en retirer les impuretés, et de calculer dans un second temps la norme du gradient et l'angle de la normale au gradient pour chaque pixel de l'image lissée.

3.1 Le filtrage de l'image

La présence d'impuretés sur une image est courante, et même très probable. Un exemple d'impuretés pourra être vu en figure 1 en page 2 [Phi]. Dans ce cas précis, il d'agit du bruit blanc gaussien, qui provoque une altération nettement visible de l'image.

On sent bien que la détection de contours demandée pourrait potentiellement être faussée par le bruit présent sur l'image : il est donc important de filtrer l'image, afin de supprimer le bruit et de pouvoir effectuer les traitements en limitant le biais.

FIGURE 1 – Le bruit blanc gaussien sur l'image Lena

Nous allons donc «lisser» l'image, en lui appliquant un filtre gaussien. Le filtre est lui même une matrice, et il faut l'appliquer en effectuant des produits de convolution entre l'image et le filtre. Typiquement, une boucle va parcourir toutes les valeurs de la matrice image de base et effectuer un traitement pour chacune de ces valeurs.

Concernant l'application du filtre, le principe est le suivant :

1. Il faut d'abord superposer le filtre et l'image,
2. Multiplier par les facteurs présents dans la matrice de filtrage chaque pixel de l'image par superposition,
3. Sommer les résultats de chaque multiplication,
4. Normaliser le résultat,
5. Affecter cette somme normalisée au pixel considéré.

Une nouvelle matrice de taille égale à l'image sera initialisée et remplie au fil du traitement.

Le traitement des valeurs au bord de l'image (les pixels qui délimitent l'image) est une problématique qui se pose : en effet, positionner le masque et faire un produit de convolution sur des valeurs non existantes est impossible ! Pour pallier ce problème, plusieurs méthodes plus ou moins complexes existent, par exemple :

— Augmenter la taille de l'image en créant des bords artificiels à valeur égale à 0.
— Faire une réflexion et utiliser les bords opposés quand nécessaires.

Le choix s'est porté sur la première solution évoquée ci-dessus lors de ce TP. Il s'est alors agi d'augmenter la taille l'image d'une demi-taille de filtre de chaque côté. Si le filtre fait une taille de 3x3, augmenter la taille de l'image de 1 à gauche et à droite, s'il fait 6x6 de 2 de chaque côté, etc. Notez que cette méthode peut ajouter un effet de bord, les valeurs ajoutées «0» étant en effet potentiellement fausses... Pour un exemple d'agrandissement de la matrice, voir figure 2 en page 3.

```
1.    2.    3.    4.    5.
1.    2.    3.    4.    5.
1.    2.    3.    4.    5.
1.    2.    3.    4.    5.
1.    2.    3.    4.    5.

0.    0.    0.    0.    0.    0.    0.
0.    0.    0.    0.    0.    0.    0.
0.    0.    0.    0.    0.    0.    0.
0.    0.    0.    0.    0.    0.    0.
0.    0.    0.    0.    0.    0.    0.
0.    0.    0.    0.    0.    0.    0.
0.    0.    0.    0.    0.    0.    0.

0.    0.    0.    0.    0.    0.    0.
0.    1.    2.    3.    4.    5.    0.
0.    1.    2.    3.    4.    5.    0.
0.    1.    2.    3.    4.    5.    0.
0.    1.    2.    3.    4.    5.    0.
0.    1.    2.    3.    4.    5.    0.
0.    0.    0.    0.    0.    0.    0.
```

FIGURE 2 – Agrandissement de l'image pour un filtre de taille 3x3. Première matrice : matrice de base, deuxième matrice : matrice temporaire faite de 0, 3e matrice : inclusion de la matrice de base dans la matrice temporaire.

Pour appliquer le filtre, il faudra donc commencer le traitement à l'indice *(XtailleDuDemiFiltre , YtailleDuDemiFiltre)* de la matrice image.

Un schéma de l'application de ce filtre pourra être trouvé en figure 3 en page 3 [VOZ16].

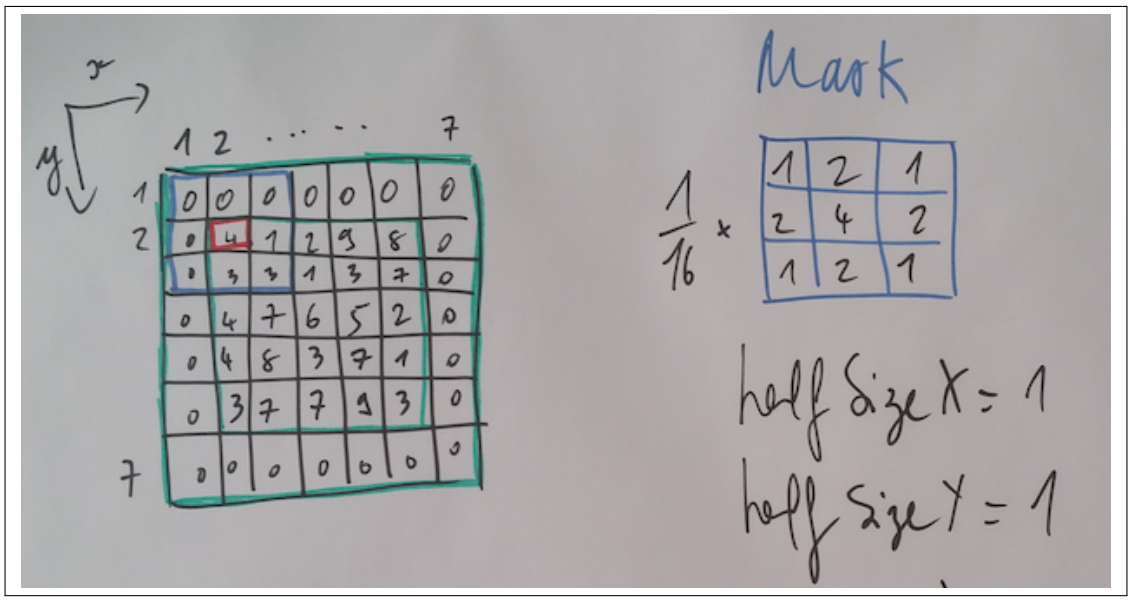

FIGURE 3 – Application du filtre par produit de convolution

Les étapes suivies si on suit la figure 3 seront donc (application pratique de l'algorithme) :

1. Placer le pixel rouge au début de notre image précédemment agrandie.

2. Faire les convolutions en faisant la somme de la multiplication par superposition les valeurs des deux matrices (ici, on va avoir 0 * 1 + 0 * 2 + ... + 4 * 4 + 1 * 2 + ... + 3 * 1)
3. Normaliser le résultat en le multipliant par l'inverse de la somme de toutes ses valeurs (ici, 1 / [1 + 2 + 1 + ... + 1] = 1 / 16)
4. Affecter la nouvelle valeur au pixel rouge dans un nouveau tableau.
5. Déplacer le pixel rouge sur l'image [1] en suivant les axes x et y pour traiter chacun des pixels.

La taille du filtre que nous appliquons sur notre image peut être plus ou moins grande. Évidemment, plus elle est grande plus le traitement est coûteux et plus il prend du temps. On note aussi que plus la taille du filtre gaussien est importante, plus l'image est floutée par le traitement. Deux filtres ont étés utilisés pendant les manipulations (voir figure 4 en page 4). Le premier filtre va lisser l'image convenablement, mais moins efficacement que le second.

On note la présence de coefficients devant les filtres : ceux-ci représentent la normalisation des valeurs. On va en effet diviser chaque pixel par la somme totale des éléments présents dans le filtre.

1	2	1			
2	4	2	x 1/16		
1	2	1			
2	4	5	4	2	
4	9	12	9	4	
5	12	15	12	5	x 1/159
4	9	12	9	4	
2	4	5	4	2	

FIGURE 4 – Les filtres utilisés pour expérimenter

La fonction d'application du filtre sera utilisée plusieurs fois le long de ce TP : c'est pour ça qu'elle a été codée de la façon la plus générique possible.

Un exemple d'image lissée pourra être trouvé en figure 5 en page 5.

3.2 Calcul de gradients et de normes

Afin d'effectuer des comparaisons entre pixels voisins, il faut désormais effectuer des calculs mathématiques, concernant la norme et la normale au gradient de chaque pixel de l'image filtrée.

Les formules à appliquer pour chaque pixel sont les suivantes :

$$e_s(i,j) = \sqrt{J_x^2(i,j) + J_y^2(i,j)}$$
$$e_o(i,j) = arctan(\frac{-J_x}{J_y})$$

Avec ici J_x qui représente les gradients en x et J_y les gradients en y. Pour calculer J_x et J_y, il s'agit simplement de réutiliser la fonction d'application d'un masque codée précédemment, avec les matrices du gradient (voir figure 6 en page 5).

1. L'image considérée sera toujours l'image de départ.

FIGURE 5 – Lena avant et après lissage gaussien

Une fois les masques appliqués et les matrices J_x et J_y prête à être exploitées, il suffit d'appliquer les formules mathématiques pour obtenir les valeurs pour chaque pixel (en utilisant comme d'habitude une double boucle pour parcourir toutes les valeurs de la matrice image). La fonction qui effectue ce traitement («*gradientNorm*» dans le code) retourne donc deux valeurs, e_s et e_o.

En x :	1	0	-1
En y :	1		
	0		
	-1		

FIGURE 6 – Masques à appliquer pour obtenir les gradients

Une subtilité pourra être soulignée : les valeurs de e_o ont en fait étés normalisées. Le but est de renvoyer un angle indiquant la direction approximée du gradient, afin de pouvoir trouver plus tard les voisins des pixels du contour (s'ils existent). Pour approximer, nous avons suivi le schéma 7 en page 6. Typiquement, si une valeur est située dans la zone dessinée en rouge (une valeur telle que 12 degrés par exemple) on pourra l'approximer à 0 degré pour simplifier.

Si une valeur est plus grande que 135 + 22,5 = 157,5 degrés, nous avons choisi de lui enlever 180 degrés afin de recommencer le traitement dans cette même fonction de normalisation (appel récursif). Même principe, si un angle est inférieur à -22,5, on lui ajoute 180 degrés et on recommence le traitement.

4 Deuxième étape : Suppression des non-maximums

La suppression des non-maximums est une étape assez rapide. Il s'agit de supprimer de la norme du gradient toutes les valeurs «faibles», les valeurs qui ne sont pas les plus grandes en suivant l'angle de leur gradient.

Les étapes de cette suppression pour chaque pixel sont les suivantes :

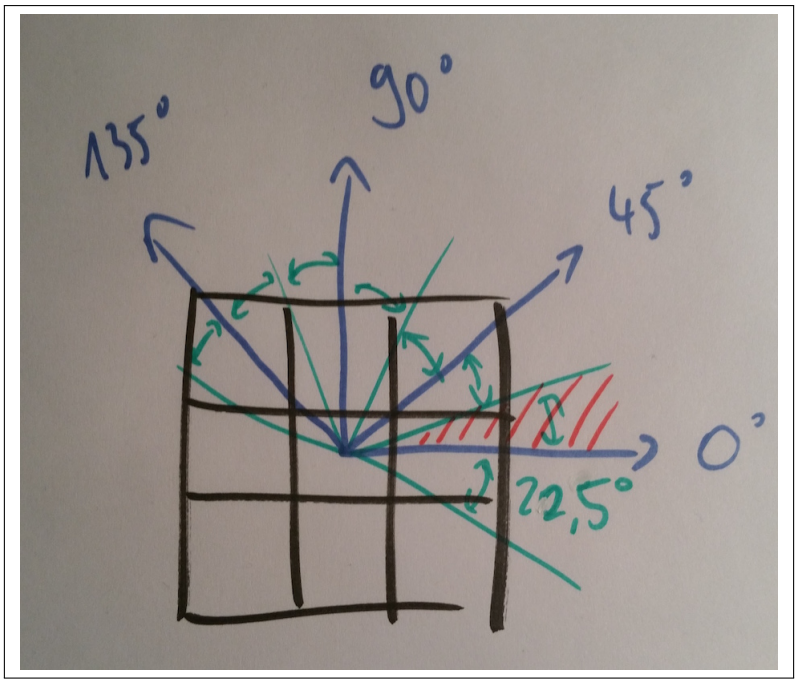

FIGURE 7 – Méthode de normalisation de l'angle du gradient

1. Trouver les voisins du pixel en suivant l'angle du gradient. Exemple : si l'angle est de 90 degrés, les pixels voisins seront le pixel juste au-dessus et juste en dessous du pixel courant. Ces pixels auront les coordonnées *(x,y-1)* et *(x,y+1)*.
2. Vérifier la valeur des voisins : si la valeur de l'un des voisins est plus grande que la valeur du pixel courant, supprimer le pixel courant (lui affecter la valeur 0), sinon le garder à sa valeur d'origine.

On note que la problématique du traitement des bords est aussi à considérer lors de cette étape. Ici, le choix a été fait de développer une fonction générique pour récupérer les valeurs d'un tableau à partir d'indices donnés, si elles existent. Si elles n'existent pas, la fonction retourne simplement une valeur égale à 0. La fonction codée pour ce faire s'appellera «*getMatValueIfExists*».

Afin de rendre générique la récupération des voisins d'un pixel, une fonction a également été codée ; cette fonction sera par ailleurs réutilisée plusieurs fois le long du TP. Cette fonction a pour nom «*getNeighborhoodCoords*», et retournera les coordonnées des voisins théoriques du pixel donné en paramètre.

5 Troisième étape : seuillage par hystérésis

Le seuillage par hystérésis était certainement la partie du TP le plus complexe à mettre en œuvre. Il s'agit ici d'affiner le filtrage des contours faibles et de garder uniquement les contours significatifs, en utilisant deux seuils. Les seuils que nous allons considérer sont le seuil haut (T_h) et le seuil bas (T_l).

Typiquement, si la valeur d'un contour est supérieure au seuil haut, nous allons naturellement le garder, cela signifie en effet qu'il est significatif. Si le seuil est inférieur au seuil bas, nous allons sur le même principe supprimer le pixel correspondant. La partie complexe de l'algorithme est lorsqu'un pixel à une valeur située entre T_l et T_h. Dans ce cas, il s'agit de vérifier si le pixel considéré est

connecté à un pixel de contour en regardant les voisins perpendiculaires à son gradient.

Statistiquement, il est prouvé que les contours se trouvent généralement orientés à 90 degrés de l'angle de gradient du pixel considéré. Voir figure 8 en page 7 (en vert, les voisins potentiels).

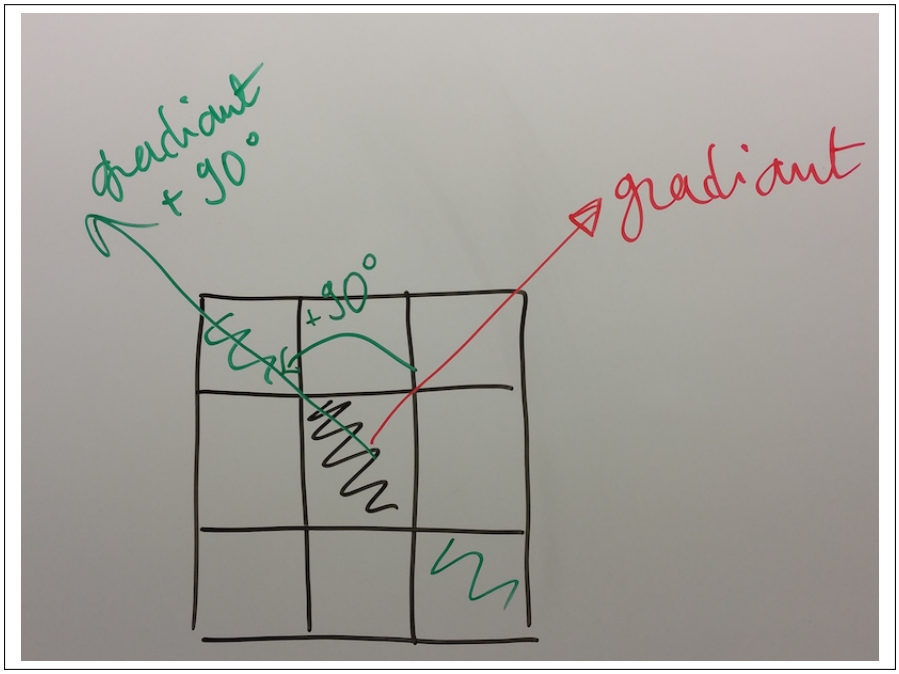

FIGURE 8 – Détection des contours par hystérésis

Pour commencer les expérimentations, le seuil haut T_h a été fixé manuellement et arbitrairement à une valeur de 40. Le seuil bas est quant à lui calculé à partir du seuil haut ($T_l = 1/2 * T_h$). Lors de cette étape, un pixel faible sera mis à 0 et un pixel fort à 255 ; 255 représente en effet la couleur blanche en niveaux de gris.

L'algorithme de l'hystérésis, qui s'appellera «*hysteresisThreshold*» dans le code, s'effectuera en deux passes :

1. Dans un premier temps/une première boucle, supprimer tous les pixels faibles inférieurs à T_l et allumer tous les pixels forts supérieurs à T_h.
2. Une fois cette première passe effectuée, une seconde itération permettra de traiter plus finement chaque pixel situé entre T_l et T_h avec la matrice des gradients. Lors de cette seconde itération, les fonctions «*getNeighborhoodCoords*» et «*getMatValueIfExists*» seront réutilisées pour récupérer les voisins du pixel. Il ne faut pas oublier d'ajouter 90 degrés à l'angle du gradient du pixel considéré avant de récupérer les voisins.

Le résultat de cette fonction «*hysteresisThreshold*» sera une image contenant les contours des objets de l'image de départ, ce qui était le but de ce TP.

Il est possible d'agir manuellement sur les seuils pour faire des expérimentations. Évidemment, plus le seuil T_h est haut, moins l'image résultat contiendra de contours, et inversement. Si le seuil T_h est réglé trop bas, le résultat ne sera pas bon et beaucoup de contours parasites pourront être trouvés sur l'image. Pour voir un exemple de résultat erroné à cause d'un seuil trop bas, se référer à la figure 9 en page 8.

FIGURE 9 – Trop de contours détectés à cause d'un seuil T_h trop bas

6 Dernière étape : trouver le seuil automatiquement

Lors de la dernière partie du TP, l'objectif était d'améliorer notre fonction hystérésis, et de trouver le seuil T_h automatiquement, en analysant le module du gradient e_s. Pour calculer T_h, il faut calculer la fonction de répartition du module du gradient de l'image, et trouver la valeur pour laquelle la fonction de répartition est inférieure à p_h, avec 70% < p_h < 95%.

La fonction «*hysteresisThreshold*» étant codée et fonctionnelle, il ne reste plus qu'à coder la fonction «*computeThreshold*» qui va calculer les deux seuils T_h et T_l.

Cette dernière fonction sera codée de la manière suivante :

1. Normalisation de la matrice e_s pour supprimer les valeurs à virgule,
2. Calcul du pas de l'histogramme,
3. Calcul de l'histogramme du module du gradient,
4. Normalisation de l'histogramme,
5. Calcul de la fonction de répartition,
6. Calcul de T_h en utilisant la fonction de répartition.

Pour le calcul du pas de l'histogramme (sa précision), il était important de trouver les valeurs maximum et minimum présentes dans la matrice e_s. Ensuite, en fonction du nombre de pas choisis en argument de la fonction, le calcul s'effectuera en faisant $step = (valueMax - valueMin)/histSize$;.

Cette fonction nécessite l'utilisation de deux variables, «*histogramIndexes*» et «*histogram*». La première variable «*histogramIndexes*» stocke le pas calculé précédemment. Si le pas est de 2, elle contiendra les valeurs $[0, 2, 4, 6, ..., valueMax]$ avec $histSize$ valeurs. La seconde, «*histogram*», sera un accumulateur : chaque fois qu'une valeur de e_s sera comprise entre une borne de «*histogramIndexes*» à l'index i, «*histogram[i]*» va s'incrémenter de 1.

Une fois l'histogramme calculé, il s'agit désormais d'en déduire la fonction de répartition dudit histogramme. Pour ce faire, il faut effectuer une somme cumulative de l'histogramme et stocker cette somme dans une nouvelle variable.

Un exemple d'histogramme et de fonction de répartition calculé pourra être trouvé en figure 10 en page 9.

Enfin, une fois toutes ces étapes validées, il s'agit de trouver la valeur de réparation présente dans «*histogramIndexes*» pour laquelle le pourcentage est supérieur à une valeur donnée. Le seuil T_h en sera directement déduit (et $T_l = 0.5 * T_h$).

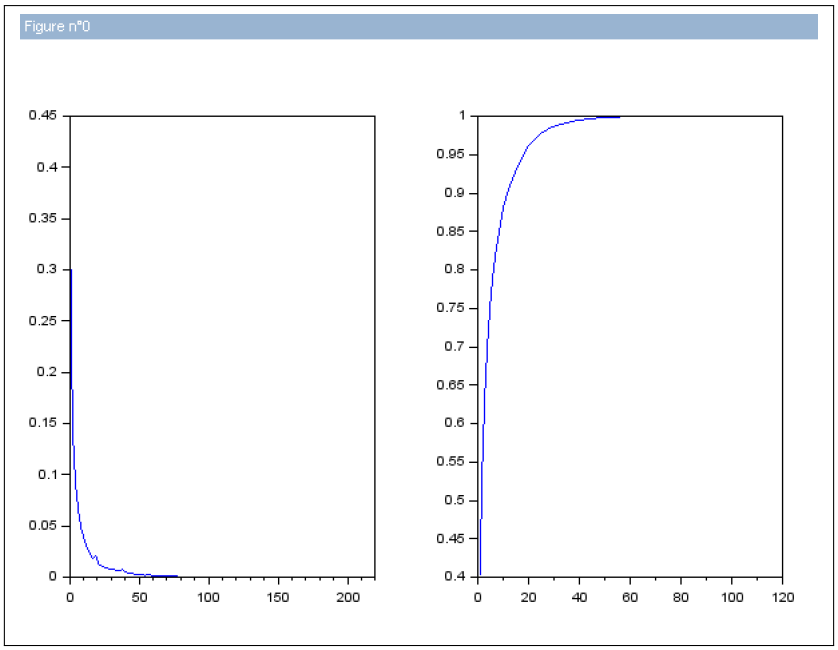

FIGURE 10 – Exemple d'histogramme calculé par «*computeThreshold*»

Dans la figure 10, les abscisses et ordonnées ont étés réglées pour observer en abscisse la valeur des pixels et en ordonnées le nombre de pixels.

7 Conclusion

7.1 Tests & Benchmarks

L'algorithme a été testé avec une image 6000x6000 afin de le benchmarquer. L'hypothèse émise était que cet algorithme est linéaire ; si traiter une image de 60x60 prend 20 secondes, traiter une image de 6000x6000 prendrait simplement 100 fois plus de temps, soit 2000 secondes = 33,3 minutes. Malheureusement, le test n'a pas pu se terminer : erreur de capacité pour l'ordinateur...

Dans l'optique de suivre une méthodologie TDD (Test Driven Developement), une fonction de test a été codée au début du TP. Il s'agissait de s'assurer que la fonction d'application d'un masque était fonctionnelle. Dans un premier temps, il s'est agi d'appliquer la fonction écrite à la main à une image. Puis, cette même image a été filtrée avec une fonction interne à Scilab permettant aussi d'appliquer un filtre. Cette dernière est évidemment une référence, car est déjà testée et approuvée par la communauté. Enfin, une comparaison entre les deux résultats de filtrage a été effectuée. En soustrayant les deux résultats, une majorité de 0 devrait apparaitre. Le développement TDD n'est pas tellement adapté à une utilisation avec le logiciel Scilab, cette fonction de test sera donc la seule fournie pour ce TP.

Des tests et expérimentations ont étés effectués avec différentes valeurs de seuil T_h et différent pas d'histogramme. Un exemple d'expérimentation pourra être trouvé en figure 11 en page 10. Le résultat se lira de gauche à droite, avec, dans l'ordre d'apparition : l'image originale, l'image filtrée, la norme des gradients, et le contour extrait. Différents jeux de tests seront disponibles dans l'archive associée à ce rapport, pour différentes valeurs de seuils et différentes images (avec un nommage explicite).

Par exemple, sur la figure ci-dessus, on observe que les plumes du chapeau de Lena sont gardées

FIGURE 11 – Lena avec p_h égal à 70%.

par le programme, car le seuil est relativement bas. Il est possible d'avoir un traitement plus précis en augmentant p_h !

7.2 Améliorations possibles

La décomposition des filtres carrés en deux filtres plus petits est plus efficace lors de l'exécution. Pour voir une décomposition qui rendrait le code plus rapide, se référer à la figure 12 en page 10. Cette fonction n'a pas été implémentée lors de ce projet, faute de temps.

	1						1	2	1	
1 / 4 x	2	* 1 / 4 x	1	2	1	\=	2	4	2	x 1/16
	1						1	2	1	

FIGURE 12 – Décomposition d'un masque carré en deux masques plus petits

7.3 Retour d'expérience

Ce TP était très intéressant, car il m'a vraiment permis de comprendre les enjeux et le fonctionnement des algorithmes. Les difficultés rencontrées se sont concentrées lors du calcul automatique du seuil T_h, les notions d'histogramme et de répartitions n'étant pas forcément faciles à appréhender d'un premier abord. Je suis content d'avoir pu remplir le cahier des charges et l'aspect visuel et ludique du développement était un vrai plus lors de ce TP.

Table des figures

1 Le bruit blanc gaussien sur l'image Lena 2
2 Agrandissement de l'image pour un filtre de taille 3x3. Première matrice : matrice de base, deuxième matrice : matrice temporaire faite de 0, 3e matrice : inclusion de la matrice de base dans la matrice temporaire. 3
3 Application du filtre par produit de convolution 3
4 Les filtres utilisés pour expérimenter 4
5 Lena avant et après lissage gaussien 5
6 Masques à appliquer pour obtenir les gradients 5
7 Méthode de normalisation de l'angle du gradient 6
8 Détection des contours par hystérésis 7
9 Trop de contours détectés à cause d'un seuil T_h trop bas 8
10 Exemple d'histogramme calculé par «*computeThreshold*» 9
11 Lena avec p_h égal à 70%. 10
12 Décomposition d'un masque carré en deux masques plus petits 10

Références

[Phi] Philippeau, Xavier : *Filtre Mean Shift.* http://xphilipp.developpez.com/articles/meanshift/

[VOZ16] VOZEL, Benoit : *CM traitement image*, 2016