REYMOND Guillaume
WAUTLET Sébastien

M2IM

Analyse d'image TP1: Détection de contours

Introduction

Dans le cadre de ce projet d'analyse d'image, nous devons implémenter une méthode de détection de contours, à l'aide des opérateurs différentiels de base vus en cours. Suite à cela, nous devrons également permettre une meilleur approximations des contours grâce à un affinage et un chaînage des contours qui nous permettront d'aboutir à des primitives de bases. Pour cela, nous avons utilisé la librairie de traitement d'images que nous avons commencé à développer en cours de mathématiques pour l'image. Dans la limite fixé par la contrainte de temps, nous avons réalisé une petite interface qui permet d'alléger le travail de l'utilisateur. Ce premier TP restreint la problématique de la détection de contours, car nous devons n'utiliser « que » les opérateurs du premier ordre. Néanmoins, le problème est loin d'être trivial, nous avons donc du mettre en place plusieurs structures de données et algorithmes bien définis que nous allons détailler…

Principes et algorithmes

Structure de données

Nous avons 4 grandes classes de données dans la librairie :

- Image
- Filtre
- Histogramme
- Contours

La classe Histogramme ne servira pas dans ce TP car elle est uniquement destinée à effectuer des traitements par histogrammes sur une image (cours mathématique pour l'image). Il est a noter que nous aurions pu l'utiliser pour effectuer l'opération de seuillage. Nous n'avons pas de classe 'pixel', ce qui peut sembler étrange au premier abord, mais nous avons décidé que créer une classe sur un objet instancié des milliers, voir des millions de fois, n'améliorerait pas les performances de notre librairie. Les pixels sont donc des types simples (unsigned char ou unsigned int) ou des structures de types simples dans le cas d'images couleurs.

La classe Image est la plus importante et représente bien entendu le cœur de la librairie. Nous la considérerons comme une « boite noire » dans ce TP, qui nous permet d'effectuer les opérations basiques comme la lecture, l'écriture d'images (au format pgm – ppm), la récupération d'informations sur l'image (taille, nombre de canaux, intensité…), et bien entendu, l'affectation d'un pixel à une couleur, et la récupération de la valeur d'un pixel.

La classe Filtre représente de manière générale une matrice M x M avec quelques opérations et méthodes élémentaires, comme la lecture d'un filtre depuis un fichier ou le remplissage avec une matrice entrée en « dur » dans la librairie (ex : Sobel, Prewitt, etc.…). Chaque filtre possède également un coefficient multiplicateur et une direction codé de 0 à 7 en suivant les directions d'une connexité V8. Cette notion est très utile pendant la détection de contours comme nous le verrons dans la prochaine section.

La classe Contours enfin, nous intéresse plus particulièrement dans ce TP. Elle contient 2 pointeurs sur des « Image », un pour la source, et un pour le résultat. L'utilisation de pointeur nous permet de ne pas stocker inutilement 2 images. Cependant,

de nombreux calculs sont effectués pendant la détection, ce qui implique de stocker les résultats intermédiaires entre les différente étapes. Pour cela, on va utiliser des vector de la librairie standard du C++. Leur avantage indéniable est une grande protection contre les erreurs de dépassement mémoire. On peut considérer ces vector comme des images tampons, toutes les informations de tailles, profondeur, seront prises depuis le pointeur sur l'image source. Nous utiliserons entre autres une image pour la norme des gradients, pour la direction des gradients, pour les chaînes de l'image, ainsi qu'une image de travail. Viennent ensuite les méthodes qui correspondent aux étapes de la détection et de l'affinage des contours. A noter la méthode AppliqueImage() qui permet de mettre le contenu de l'image de travail dans une vraie structure Image pour la visualiser. Suivant l'étape en cours, différents traitements sont effectués pour pouvoir rendre l'image visible et intéressante (ex : les valeurs de gradient sont passé en valeur absolue car elles peuvent être négatives - les extrémités des chaînes sont mise à une forte intensité tandis que le reste des pixels est gris après le chaînage, etc.…).

Maintenant que les structures de données sont bien définies, nous allons voir les différentes étapes de la détection de contours. La possibilité de se référer au code est bien entendu possible même si la relative complexité du programme ne rend pas forcément la compréhension plus aisée. Toutes les classes sont définies dans image.h. Les implémentations sont dans leur fichier respectif : image.cpp, histogramme.cpp, filtre.cpp et contours.cpp. En cas de problème technique ou de compilation cf. la partie 4.

Détection

Dans notre approche, un contour sera caractérisé par une transition plus ou moins brutale des niveaux de gris d'une image. L'analyse du vecteur gradient permet de mettre en évidence un contour. En effet l'amplitude du gradient indique une plus ou moins forte discontinuité et sa direction est par définition normal au contour. Le gradient est un opérateur directionnel. Mais toute dérivation amplifie le bruit. Voilà pourquoi on parle souvent d'opérateurs de base, car il existe des méthodes plus poussées (filtrage, second ordre) pour minimiser les répercutions du bruit dans l'image. On peut cependant utiliser une méthode multidirectionelle pour tenter de palier à ces problèmes.

```
bool AppliqueGradientBi(const Filtre& masque1,const Filtre& masque2);
```

Cette fonction permet de calculer le gradient bidirectionnel à partir de deux masques passé en paramètres. Pour calculer la norme du gradient en un point, nous effectuons le produit de convolution entre le voisinage de ce point et le masque1 passé en paramètre. Puis on effectue le même calcul avec le masque2. La norme finale du gradient vaut $\sqrt{gx^2 + gy^2}$. Pour déterminer la pente en ce point, nous utilisons la formule $\arctan(gy/gx)$. Grâce à la classe Filtre, nous pouvons aisément changer d'opérateur sans modification du code. La taille du voisinage exploré pendant la convolution s'adaptant automatiquement depuis la taille du filtre.

Typiquement, nous pourrons écrire :

```
filtre1->Sobel(Filtre::E); //matrice de sobel standard 3x3 codé en dur pour commodité
filtre2->Sobel(Filtre::N); //direction nord
contours->AppliqueGradientBi(*filtre1,*filtre2);
```

Note d'implémentation : Dans le but de rendre plus performant ce calcul, nous ne stockons pas la valeur du gradient dans la première direction puis dans la deuxième direction pour les additionner plus tard. Nous les sommons directement dans notre image de travail. De même, nous utilisons la fonction atan2 au lieu de atan pour calculer l'angle, car cette fonction renvoi un angle borné entre –pi et +pi en tenant compte du

cadran où l'on se trouve. Enfin, pour éviter de consommer trop de mémoire inutilement, nous ne gardons pas la valeur flottante de la pente (qui demanderait beaucoup de mémoire) mais uniquement une direction codée sur le modèle V8 (Nord, Est , Sud, Nord-Ouest, etc...).

L'avantage de cette méthode est que seul 2 filtres sont nécessaires pour calculer le gradient en 1 point. Cependant elle peut être plus sensible au bruit que la méthode multidirectionelle. Sa complexité est de l'ordre w * h avec w largeur de l'image et h hauteur de l'image.

```
bool AppliqueGradientMulti(const Filtre& masque1,const Filtre& masque2,const Filtre& masque3,const Filtre& masque4);
```

Cette fonction reprends le même principe que son homologue bidirectionnelle sauf que l'on ne calcule pas que deux valeurs pour le gradient cette fois. En chaque point, nous effectuons le produit de convolution pour chaque masque passé en paramètre. Dans ce cas, c'est le masque qui maximise le gradient qui sort vainqueur. La pente est donc déterminée suivant la direction du filtre gagnant uniquement. Seulement 4 masques sont nécessaires, car nous pouvons facilement approximer les autres par symétrie. Nous pouvons même nous permettre de ne calculer que 4 valeurs car suivant le signe du gradient obtenu, nous choisirons le masque en cours ou son opposé. Ceci n'est malheureusement vrai que pour les filtres symétriques. Nous avons donc une version surchargée de cette fonction acceptant 8 masques lorsqu'on utilise Prewitt ou Kirsch par exemple. Attention, il faut bien comparer les valeurs absolus des gradients pour choisir la valeur la plus élevée.

Note d'implémentation : Idem que pour le bidirectionnelle. Nous ne conservons pas tous les calculs pour ne pas encombrer la mémoire. La vérification est faite au moment du calcul.

Cette méthode permet de vraiment tenir compte du gradient qui apporte le plus d'information à l'image en ce point. Sa complexité est du même ordre que la méthode bidirectionnelle. Au niveau des résultats, nous conservons les normes des gradient en chaque point dans le vector norme_gradient et la direction dans le vector direction_gradient. Pour les bords de l'image, nous avons fait le choix de dire simplement que si un pixel est en dehors de l'image, alors il ne compte pas dans le calcul du produit de convolution. Ceci produit la plupart du temps un contour du bord de l'image, mais en contrepartie l'algorithme est plus simple. Idéalement, il faudrait considérer qu'un pixel hors de l'image à la même valeur que le pixel du bord.

Pour visualiser les résultats dans une image, quelques précautions sont à prendre. Pour visualiser l'intensité du gradient, il faut enlever les valeurs négatives (avec une valeur absolue), puis borner les résultats dans l'espace 0 - 255. Pour afficher les directions du gradient en chaque point, nous avons choisi d'affecter une couleur fixe par direction. Pour tester la validité de nos résultats, nous avons comparé ce que donne des logiciels professionnels avec nos résultats. Le résultat nous a semblé satisfaisant.

Voici quelques résultats :

image lenna originale

image du gradient multidirectionelle

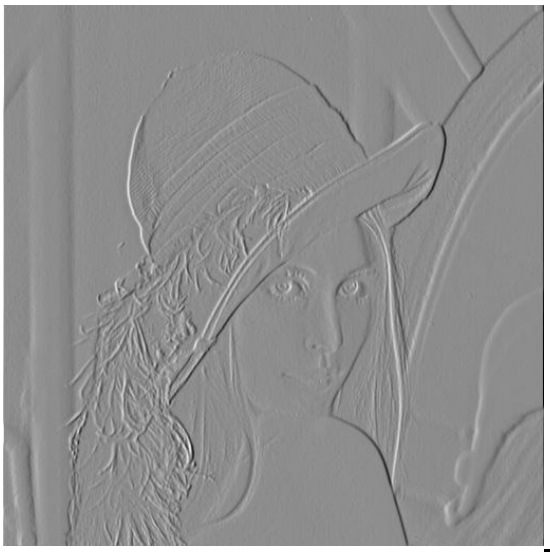


image du gradient x (normalisé)

image du gradient y (normalisé)

L'image du gradient multidirectionnelle est très sombre pour l'œil humain, mais elle n'est bien sûr pas normalisée (nous avons cependant utilisé les valeurs absolues), car c'est l'image sur laquelle les prochaines étapes s'effectueront. Les deux images de gradient individuel sont quand à elles normalisées. C'est à dire que nous avons redistribué les valeurs de niveaux de gris entre 0 et 255. Ceci dans le but de mieux voir l'influence de la direction du masque dans le calcul du gradient. Pour mieux voir le travail accompli par cette étape, nous pouvons observer le négatif de l'image du gradient multidirectionnel.

image du gradient en négatif

Bien que produisant une image agréable à l'œil, cette mise en valeur des contours est insuffisante car trop complexe. Nous cherchons à obtenir des contours francs. C'est pourquoi nous allons passer à l'étape de seuillage…

Seuillage

Nous souhaitons obtenir une réponse unique à la question : est-ce un contour ? Pour l'instant la réponse se borne à une intensité de gradient. Pour parvenir à nos fins nous devons donc diviser les pixels de l'image en deux catégories : Les pixels contours et les pixels pas contours. C'est ce que réalise l'opération binaire de seuillage. Bien que différentes méthodes de seuillage existe comme nous allons le voir, elles fonctionnent toutes sur le même principe : Séparer les pixels au dessus du seuil et ceux en dessous.

```
bool AppliqueSeuilFixe(unsigned int seuil);
```

Cette fonction applique un seuil fixe donné en paramètre. La complexité est toujours en O(x * y) la taille de l'image, car on effectue un seul parcours de l'image. Bien entendu, si le pixel est supérieur au seuil, nous lui attribuons la valeur 255, sinon 0. Nous obtenons donc une image binaire à la fin. Cette méthode seule est peu intéressante. Elle demande trop de connaissance a priori sur l'image que nous souhaitons traiter. Plusieurs essais ont été nécessaires pour trouver un seuil intéressant dans l'exemple ci dessous.

seuil fixe(45)

bool AppliqueSeuilGlobale();

Une approche plus automatique semble clairement plus intéressante. La première idée qui vient naturellement est la moyenne des valeurs des niveaux de gris dans l'image. Ainsi, on détermine cette moyenne et on appelle la méthode de seuil fixe avec la moyenne en paramètre. La complexité est plus grande car on effectue 2 parcours de l'image.

Seuil avec la moyenne

L'image contenant beaucoup de nuances de gris très sombre pour des contours, le résultat contient beaucoup de pixels « bruit » dans les contours.

bool AppliqueSeuilEcartType();

Plutôt que d'utiliser la moyenne, nous pouvons utiliser la notion d'écart type comme paramètre de seuil pour mieux représenter la distribution des niveaux de gris dans l'image.

seuil avec écart type

La complexité de l'algorithme est cependant plus grande car il faut trois parcours de l'image pour calculer l'écart type et l'appliquer.

bool AppliqueSeuilLocale(unsigned int fenetre);

Avec cette méthode, plutôt que d'utiliser l'information globale de l'image, nous allons nous intéresser aux phénomènes locaux. Nous définissons une taille de fenêtre dans laquelle nous allons calculer la moyenne des pixels. Cette méthode est de complexité plus importante car elle est en w * h * t^2 avec w la largeur de l'image, h la hauteur et t la taille de la fenêtre.

Note d'implémentation : Il est assez facile de voir qu'il s'agit en fait d'un produit de convolution par un filtre moyenne avec un coefficient de 1/nb_element. Une fois la moyenne effectué, on la compare à la valeur de notre pixel en cours. Cette méthode a donc été implémenté très facilement et la généricité de nos structures de données s'est avérée payante, car le filtre construit est plutôt utilisé pour flouter une image.

seuil local(taille fenêtre voisinage à 15)

Nous n'avons pas pour l'instant de méthode automatique de seuillage qui préserve les contours et efface le bruit. Nous allons donc voir une dernière méthode de seuillage qui vient du monde de la physique.

```
bool AppliqueSeuilHysteresis(int haut,int bas,int fenetre);
```

Hystérésis commence comme un seuillage normal, avec un double seuil, c'est à dire que si la valeur du pixel est plus grande que le seuil haut alors on met ce pixel à 255 et inversement si il est plus bas que le seuil bas. La différence survient pour toute les valeurs comprise entre ces deux seuils. Nous allons effectuer un parcours dans le voisinage du pixel à la découverte d'un pixel supérieur au seuil haut. Si ce pixel existe alors le point considéré est gardé, sinon il est mis à 0. Ce phénomène empêche de former trop de trous dans les contours et évite le bruit local. C'est de loin le meilleur seuillage que nous ayons qui reste efficace dans tous les cas.

Nous pouvons déterminer de manière automatique des seuils haut et bas. En effet, l'utilisation de l'écart type nous a montré de bons résultats. Nous proposons donc d'utiliser la valeur moyenne + écart type comme seuil haut et moyenne – écart type comme seuil bas. Si le seuil bas est inférieur à 0, alors la valeur est tronqué à 0.

L'algorithme effectue donc un calcul de la moyenne puis un calcul de l'écart type tout comme la méthode de seuillage global par écart type. Enfin elle applique la méthode de seuillage par hystérésis manuelle avec ces deux seuils ainsi déterminés. Nous obtenons des résultats assez satisfaisant avec cette méthode, c'est donc la méthode utilisée par défaut. Sa complexité est légèrement supérieur aux autres méthodes de seuillage.

seuil par hysteresis automatique

Nous avons donc maintenant réussi à classifier les pixels dans notre image en deux catégories. Le problème qui se pose est de pouvoir obtenir des contours d'épaisseur 1 afin d'enlever toute ambiguïté.

Affinage

Lors de cette étape, nous allons réutiliser les direction de gradient que nous avions stocké. En effet, pour « amincir » les contours, nous devons faire en sorte que seul le maximum local dans une direction de gradient soit présent. Les pixels d'un même contour partagent la même direction de gradient. L'algorithme consiste donc à parcourir l'image à la recherche d'un contour et de regarder dans le voisinage défini par le sens du gradient, si ses voisins ont une norme de gradient supérieur. Si c'est le cas, alors nous pouvons supprimer ce point car ce n'est pas un maximum local pour ce sens du gradient. C'est grâce à cette méthode que nous éliminons entre autre les points doubles générés par l'étape de détection. La complexité de l'algorithme d'affinage revient toujours à l'ordre d'un parcours de l'image.

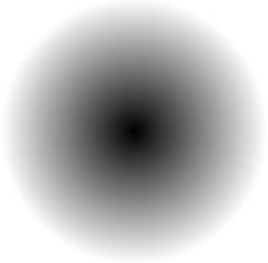


image d'un fort dégradé

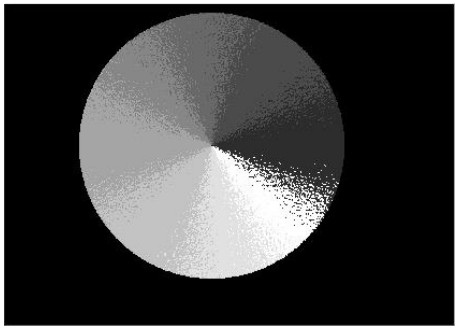


image de direction de gradient associé

image des directions du gradient

image affiné (épaisseur contours 1 pixel)

Note d'implémentation : Les images de directions de gradient sont obtenues en distribuant 8 valeurs différentes de niveaux de gris suivant la direction en cours. Nous avons utilisé ce genre d'image pour déterminer la validité de nos calculs de pente. Nous pouvons notamment nous assurer que si 2 pixels appartiennent au même contour alors il partage le même gradient localement.

Cette étape nous permet d'obtenir une image satisfaisante, qui possède relativement peu de bruit avec des contours assez précis. Nous pouvons cependant réaliser une dernière opération avant de rendre une image finale des contours. Nous avons crée quelques discontinuités dans l'image des contours pour arriver à ce résultat. Notre objectif est de réaliser le chaînage et la fermeture des contours.

Chaînage

L'opération de chaînage consiste à parcourir l'ensemble des contours dans l'image, pour former des chaînes de pixels connexes. Nous devons également trouver les extrémités des chaînes pour préparer le terrain pour la fermeture.

Devant une telle tâche, la première question importante que nous nous sommes posé concerne le codage de nos chaînes. Comment coder efficacement les chaînes en gardant leurs positions et leurs identifiants ? La réponse nous est venu assez naturellement, il suffit de les coder directement dans l'image. En effet, coder dans l'image reviens à coder dans une matrice. Nous avons donc « gratuitement » une structure de données qui connaît la position de chaque pixel de la chaîne. Le codage en lui même se fera simplement sur l'identifiant de la chaîne auquel appartient ce pixel. Pour coder les extrémités, nous ne pouvons pas utiliser de chiffres spéciaux pour les distinguer, car nous pourrions virtuellement ouvrir une image possédant des milliers de chaînes. La solution que nous avons retenue est de prendre une valeur négative quand on est une extrémité. Si nous détectons que nous sommes une extrémité de la chaîne x, alors ce pixel prends la valeur – x. Bien entendu, la prudence sera de mise quand il nous faudra afficher le résultat…

Bien que relativement simple, ce codage nous permet en une instruction de savoir si un pixel appartient à une chaîne et s'il est une extrémité.

```
bool Chainage()
```

Cette fonction effectue le chaînage des contours, c'est à dire qu'elle lance un parcours de chaîne sur chaque pixel contour de l'image qui n'appartient pas encore à une chaîne. Une fois tous les pixels explorés, nous supprimons les chaînes de longueur 1, ou autrement dit, les pixels isolés. Nous avons en effet jugé qu'ils ne représentaient pas une information forte dans les contours. Enfin la fonction annonce le nombre de chaîne trouvées.

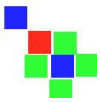
```
int Parcours(unsigned int i,unsigned int j)
```

Cette fonction effectue le parcours d'une chaîne de pixel. C'est une fonction récursive appelée par Chainage. A chaque nouveau pixel dans le voisinage qui n'appartient pas encore à une chaîne, nous explorons ce pixel. Cette fonction, surtout sous sa forme récursive n'est pas compliqué, cependant le passage délicat est la détection des extrémités de chaînes. Nous sommes confronté à deux cas distincts. Soit je possède un unique voisin (V8) et en ce cas, je suis une extrémité. Soit je possède deux voisins et je peux être une extrémité. Dans tous les autres cas non.

Comment faire dans le cas ou nous avons deux voisins ? Soit nous déterminons que nous sommes le pixel « milieu » entre deux autres :

Soit nous sommes le pixel extrémité :

Pour distinguer ces deux cas, il suffit de regarder la connexité V4 d'un voisin. Si on trouve l'autre pixel voisin dans le V4 alors on se trouve dans le deuxième cas et on est une extrémité.



Dans ce cas par exemple, nous avons en rouge le pixel exploré et en bleu ses voisins. Les carrées vert représente la connexité V4 d'un voisin. On voit bien sur l'image qu'elle n'atteint pas l'autre voisin bleu, donc notre algorithme en déduit que le pixel rouge n'est pas une extrémité. Attention, si on prends la connexité V8, alors l'algorithme est faux, comme on peut le voir dans l'exemple ci dessous :

Le pixel rouge à deux voisins. Le voisin du centre regarde son voisinage V8. Il trouve un autre pixel voisin donc l'algorithme en déduit que le pixel rouge est une extrémité, ce qui est bien évidemment faux. Nous voyons bien sur cet exemple que l'on utilise le V4 car les pixels voisins doivent être collés entre eux. Le résultat produit par cet algorithme est satisfaisant, il nécessite cependant une étape d'affinage bien réalisé (1 pixel d'épaisseur partout !) sous peine de mal détecter les extrémités. Sa complexité est de l'ordre d'un parcours d'image O(w * h), car pour chaque pixel on regarde son voisinage immédiat.

image des chaînes en gris foncé et des extrémités en blanc(644 chaînes)

Maintenant que nous avons nos chaînes, nous pouvons passer à la dernière étape de ce premier TP, le raccord des différents contours.

Fermeture

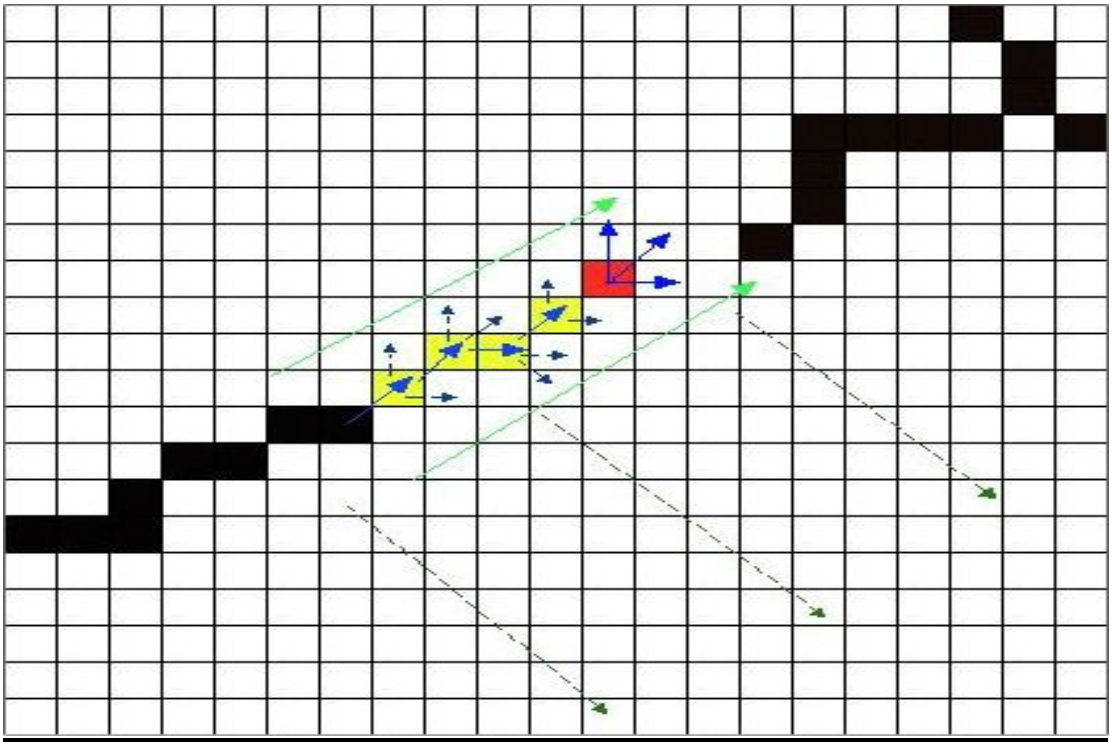
Le principe de la fermeture de contours peut s'aborder de deux grande manières différentes. La première consiste à sélectionner les extrémités que l'on veut tenter de joindre. Pour cela, on regarde dans une fenêtre défini par l'utilisateur autour de l'extrémité en cours. Le premier problème se pose alors : comment choisir efficacement le prochain bout à joindre si plusieurs extrémités sont présentes ? Une approche est de choisir l'extrémité la plus proche possible, ou celle possédant le sens de gradient le plus proche de l'autre extrémité. Une fois les deux point à relier déterminés, on peut essayer de les joindre avec une droite en utilisant l'algorithme de Bresenham. Une autre alternative qui nous semble plus judicieuse, est de former une nouvelle chaîne qui maximise la valeur de gradient entre les deux points. On obtient ainsi une image avec des contours plus régulier.

Nous avons choisi une autre approche quoique fondamentalement identique à la précédente. Si nous n'utilisons pas la méthode Bresenham, alors pourquoi chercher une extrémité à joindre ? De par la nature du gradient, si nous suivons la direction et maximisons la valeur de la chaîne, alors nous ne pouvons que tomber sur la bonne extrémité (ou sur un voisin très proche dans des cas extrêmes). C'est à partir de ce principe que nous avons implémenté :

```
bool FermetureChaine(unsigned int taille_fenetre)
```

Cette fonction effectue une recherche en aveugle à partir d'une extrémité pour fabriquer une chaîne joignant un autre contour. Le principe est le suivant : On parcours l'image. Pour chaque pixel qui est une extrémité, on détermine le sens de déplacement du contour auquel appartient ce pixel grâce à la pente du gradient. Tant qu'on n'a pas dépassé la taille de la fenêtre donnée par l'utilisateur, on se déplace dans cette direction. Si on atteint un autre contour, alors on s'arrête. Sinon, on cherche dans quel direction se déplacer pour suivre le contour au mieux pour la prochaine itération. C'est la partie la plus délicate de l'algorithme.

Comment trouver la direction optimal de déplacement ? Il suffit pour cela de tester son voisinage à la recherche du gradient le plus élevé. Le pixel qui maximise le gradient se trouve dans une certaine direction de gradient. On doit changer de direction pendant la construction de la chaîne sous peine de construire des lignes droites, mais on ne doit pas revenir sur nous même par exemple, ou prendre des virages trop brusques. En effet, le gradient n'est pas franc sur un contour, il possède des valeurs élevées de part et d'autre du contour. Un algorithme trop naïf pourrait ainsi boucler sur lui même en prenant des virages trop serrés si il regarde juste la valeur du gradient. Toutes ses conditions se traduisent ainsi : Nous allons chercher dans le voisinage du pixel en cours, le voisin qui maximise, et le gradient, et le degré de similitude avec la direction en cours. A chaque itération, nous avons donc le choix entre 3 direction possibles. Nous gardons également en mémoire des pixels visités et non choisis, donc interdits. Observons une fermeture sur un exemple :



principe de l'algorithme de fermeture des contours

En noir nous avons des contours francs, bien détectés par les précédentes étapes. Les flèches vert foncé indiquent la direction du gradient (perpendiculaire au contour). Les flèches vert clair indiquent la direction de déplacement qu'on en déduit. L'exemple est simple ici, car on travaille sur une chaîne de direction constante ou presque. En jaune, les pixels déjà trouvés par notre algorithme. En rouge, le pixel en cours. Les flèches bleue représentent la direction de déplacement possible. Celles à l'intérieur des pixels sont les choix que nous avons fait.

Nous voyons bien entre autre, que si la direction précédente était l'est, alors nous pouvons nous diriger à l'est, au nord-est ou au sud-est. On choisit la direction qui maximise le gradient parmi ces trois pour avoir la meilleur chaîne possible entre les deux extrémités.

On construit ainsi notre chaîne au fur et à mesure. Si jamais nous trouvons un pixel contour dans notre voisinage alors on arrête l'itération. Idem si nous dépassons la taille fixé par l'utilisateur pour la chaîne. Pour retrouver facilement des contours troués mais évidents (discontinuité de 2 ou 3 pixels seulement), nous utilisons l'information de l'étape d'affinage. Les pixels supprimé par l'affinage sont placés dans un vector pour être réutilisés ici : Si nous nous trouvons sur un pixel qui a été enlevé précédemment alors nous considérons que ce pixel est « gratuit ». Il ne compte pas dans la taille de la chaîne. En effet, comme nous l'avons vu précédemment, nous prenons déjà en compte deux caractéristiques du gradient, si en plus ce pixel était toujours présent après le seuillage, alors nous pouvons affirmer être dans la bonne voie à suivre. On commence donc le décompte dés que l'on se promène réellement en aveugle.

L'algorithme de fermeture de chaîne nous a semblé donner des résultats satisfaisant. Sa complexité est de l'ordre de la taille de l'image, mais difficile à formaliser.

image de fin de traitement (méthode automatique sans adaptation à l'image) avec affichage des chaînes et extrémités (76 chaînes)

Conclusion

Nous avons finalement réussi à détecter les contours dans une image en utilisant des opérateurs de base dans ce domaine. Quelques imperfections subsistent, mais ces opérateurs nous ont déjà démontré leur efficacité. Pendant ce TP, nous avons implémenté beaucoup d'algorithmes qui ont une complexité de l'ordre de la taille de l'image. Ceci reflète l'importance d'avoir de bonnes structures de données quand on travaille dans le domaine de l'image.

Compilation et utilisation du programme

Si la commande « make gtk » ne marche pas, alors il peut être intéressant de lire cette partie !

Ce programme est constitué de plusieurs modules dont certains appartiennent à la matière mathématiques pour l'image et d'autres à l'analyse d'image. Le but est d'obtenir une librairie de traitement d'image. Une module de test d'interface a été rajouté pour

pouvoir observer de manière plus convivial les résultats de notre librairie. Tout le code écrit est portable, même pour l'interface, car nous utilisons GTK+. Le code est donc parfaitement compilable sous Windows ou Linux. L'ajout de l'interface a cependant compliqué les choses. Voici donc la démarche à suivre pour bien compiler ce programme.

Sous n'importe quelle machine possédant la commande make et g++, taper la commande « make » pour créer la librairie sans interface. Cela devrait toujours fonctionner. Ceci va créer un exécutable « libimage » qui prend en paramètre l'image sur laquelle faire le traitement.

```
G:\TP\cvs\libimage>make
Building console interface...
g++ -O3 -Wall -c main.cpp -o main.o
Building Image module...
g++ -O3 -Wall -c image.cpp -o image.o
Building Histogramme module...
g++ -O3 -Wall -c histogramme.cpp -o histogramme.o
Building Filtre module...
g++ -O3 -Wall -c filtre.cpp -o filtre.o
Building Contours module...
g++ -O3 -Wall -c contours.cpp -o contours.o
Libimage is up-to-date
g++ -O3 -Wall main.o image.o histogramme.o filtre.o contours.o -o libimage
G:\TP\cvs\libimage>libimage lena512.pgm
[Image::1] Le fichier lena512.pgm semble etre un pgm
[Image::1] Header OK
[PGM] ** Chargement **
etc…
```

Compilation et lancement du programme sous windows (avec MINGW) ou Linux

Si vous désirez compiler avec l'interface graphique, il faudra taper la commande « make gtk ». Suivant la machine dont vous disposez, plusieurs scénarios peuvent se produire. Sur une machine Linux du bâtiment 710, tout devrait se compiler sans problème, car les librairies et les fichiers d'en-tête de GTK+ sont bien disponibles sur les machines. Sur une machine Linux autre, il faudra simplement veiller à ce que les packages nécessaires soient installés. Sous Windows, il faudra installer les librairies et en-têtes de GTK+ pour Windows disponible ici : *http://www.gimp.org/~tml/gimp/win32/downloads.html*.

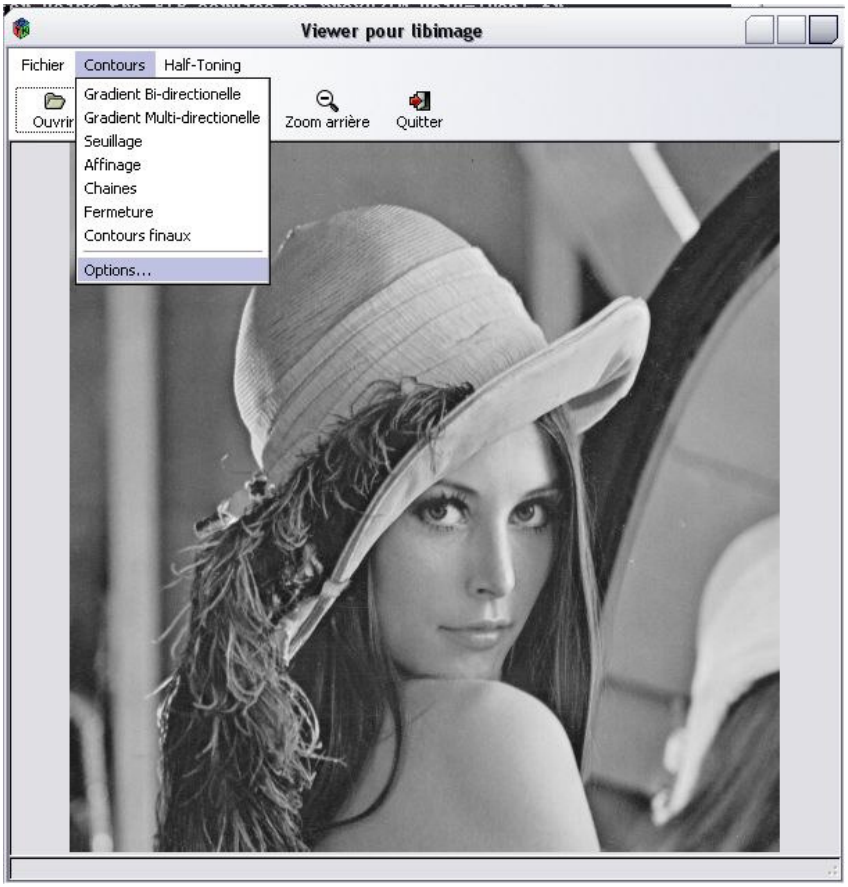
La commande « make gtk » va lancer un script shell qui test si l'environnement de travail est sain pour la compilation de l'interface. Si ce programme trouve tous les fichiers nécessaires alors la compilation a lieu, sinon elle échoue. La commande « make reconfig » permet de relancer le script de détection si on a changé de machine ou si les fichiers ont été installé depuis. Si tout se passe bien, un exécutable « viewer » est crée. Il doit être lancé sans arguments.

```
proxy710:~/mi/$ make reconfig
rm -f config.in
Analyse OS... LINUX
J'ai besoin de GTK ou de OpenCV pour l'interface graphique...
[GTK] Cherche GTK+ headers /usr/include/gtk-2.0... OK
[GTK] Cherche GTK+ headers /usr/lib/gtk-2.0/include... OK
[GTK] Cherche GLib headers /usr/include/glib-2.0... OK
[GTK] Cherche GLib headers /usr/lib/glib-2.0/include... OK
[GTK] Cherche Pango headers /usr/include/pango-1.0... OK
[GTK] Cherche Atk headers /usr/include/atk-1.0... OK
[GTK] Cherche GTK+ lib /usr/lib/libgtk-x11-2.0.so... OK
[GTK] Cherche GDK lib /usr/lib/libgdk-x11-2.0.so... OK
```

```
[GTK] Cherche GDK lib /usr/lib/libgdk_pixbuf-2.0.so... OK
[GTK] Cherche GLib lib /usr/lib/libglib-2.0.so... OK
[GTK] Cherche GLib lib /usr/lib/libgobject-2.0.so... OK
[GTK] Cherche GLib lib /usr/lib/libgmodule-2.0.so... OK
[GTK] Cherche Pango lib /usr/lib/libpango-1.0.so... OK
[GTK] Cherche Pango lib /usr/lib/libpangoxft-1.0.so... OK
[GTK] Cherche Atk lib /usr/lib/libatk-1.0.so... OK
Il est possible d'utiliser GTK sur ce système.
proxy710:~/mi/$ make gtk
Building GTK interface...
g++ -O3 -Wall -I"/usr/include/atk-1.0" -I"/usr/include/glib-2.0" -I"/usr/lib/glib-2.0/include" -I"/usr/include/gtk-2.0" -I"/usr/lib/gtk-2.0/include" -I"/usr/X11R6/include" -I"/usr/include/pango-1.0" -I"/usr/include/freetype2" -D_USE_GTK_ -o interface.o -c interface.cpp
Building Contours module...
g++ -O3 -Wall -c contours.cpp -o contours.o
Libimage is up-to-date
g++ -O3 -Wall -o viewer interface.o image.o histogramme.o filtre.o contours.o -L"/usr/lib" -lgtk-x11-2.0 -lgdk-x11-2.0 -latk-1.0 -lgdk_pixbuf-2.0 -lm -lpangoxft-1.0 -lpangox-1.0 -lpango-1.0 -lgobject-2.0 -lgmodule-2.0 -ldl -lglib-2.0
```

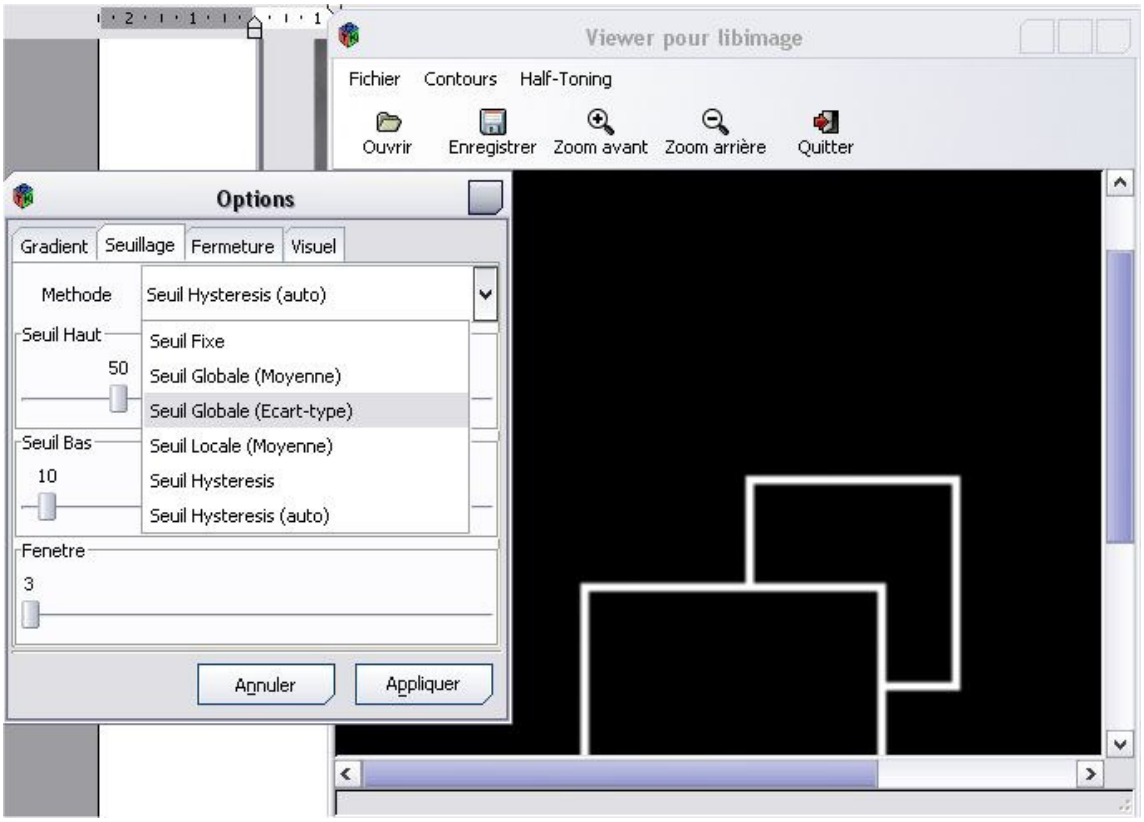
compilation de l'interface sous Linux(sous Windows il n'y a pas de chemin par défaut pour les librairies donc on demande le chemin d'installation de GTK)

Nous espérons que la compilation n'a pas été trop dur…vous pourrez trouver une version précompilé pour windows ici *: http://lambda.man.free.fr/M2IM/AAI*



Une fois l'image chargée, on peut appliquer les étapes à l'image en les sélectionnant depuis le menu « Contours ». Les 7 premiers menus sont des méthodes

automatiques sans contrôle. C'est une manière d'appliquer rapidement une étape sans se poser de question. Pour pouvoir rentrer les paramètres de seuil, matrice etc… il faut aller dans Contours -> Options…



Il suffit alors d'appuyer sur « Appliquer » pour voir le résultat. Attention l'onglet Visuel est très important car il permet de choisir quelle image (de quelle étape) on va afficher. Par défaut, on affiche l'image original avec les chaînes par dessus, mais pour visualiser l'image de l'affinage, il faut aller explicitement dans l'onglet visuel.