Rapport d'Analyse d'Image

Détection de contours

PARIS Axel, RICHE Vincent

Introduction

Nous allons présenter dans ce rapport nos résultats sur le sujet du TP “Détection de contours”. Chaque chapitre abordera une partie du TP et présentera notre méthode, nos résultats sous forme de plusieurs figures et une analyse de notre part.
Il est important de préciser que nous n’avons utilisé aucunes fonctions d’OpenCV dans le travail demandé par le TP : nous les avons seulement testé pour voir les résultats et les comparer avec les nôtres.
La dernière partie expliquera comment compiler et tester notre programme et expliquera la répartition du travail entre les deux membres du groupe.

Pipeline global

La figure 1 présente le pipeline global de notre application. Le lecteur peut ainsi se faire une idée de notre méthode. Nous allons maintenant détailler chaque étape dans les chapitres suivants.

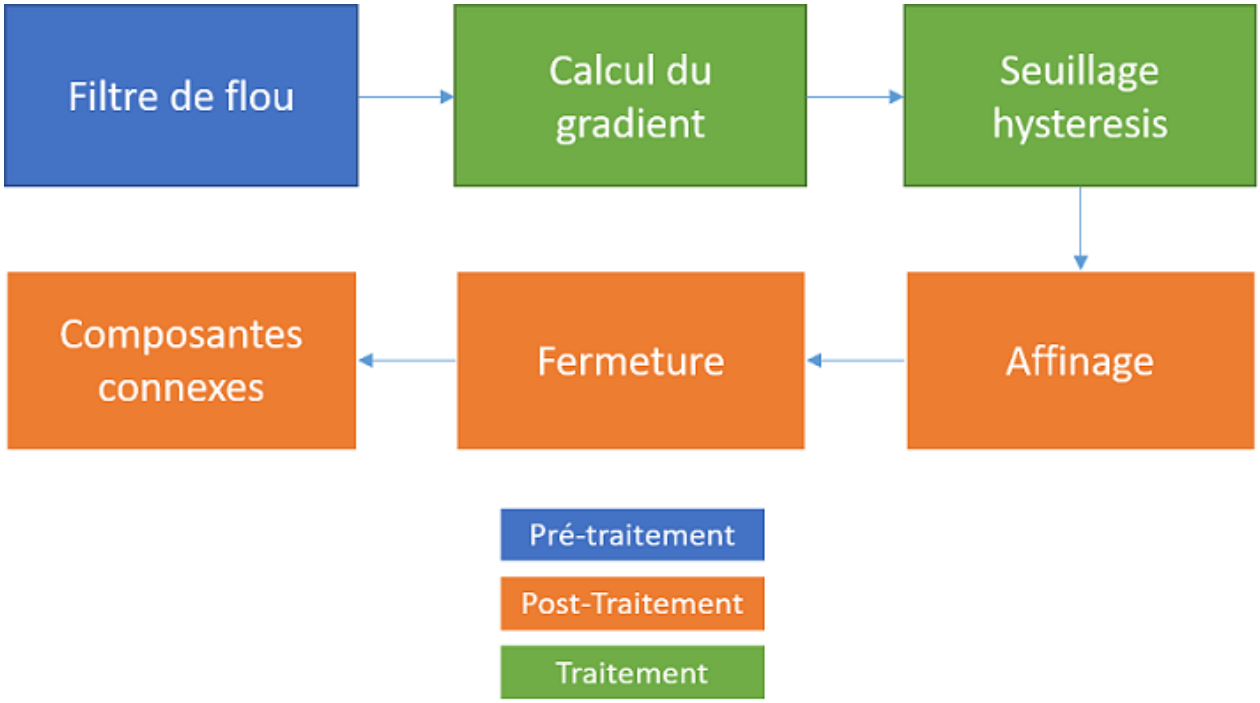

Figure 1 : Pipeline global

Application d'un filtre

Filtre de détection de contours

Nous avons développé une fonction nous permettant d'appliquer un filtre de la forme d'une matrice *3x3* sur une image. Nous avons donc codé le produit de convolution du filtre avec un point de l'image.
Nous avons testé les filtres *Prewitt*, *Sobel* et *Kirch*. Les résultats du calcul du gradient sont présenté dans la figure 2.
Nos gradient bidirectionnel et multidirectionnel sont calculés selon les formules suivantes:

```
Bidireictionnel   = [abs(gX) abs(gY)] * 0.5
Multidirectionnel = [abs(gX) + abs(gY) + abs(gXY) + abs(gYX)] * 0.25
```

L'utilisation de la fonction *abs* est une approximation de la formule donnée en cours.

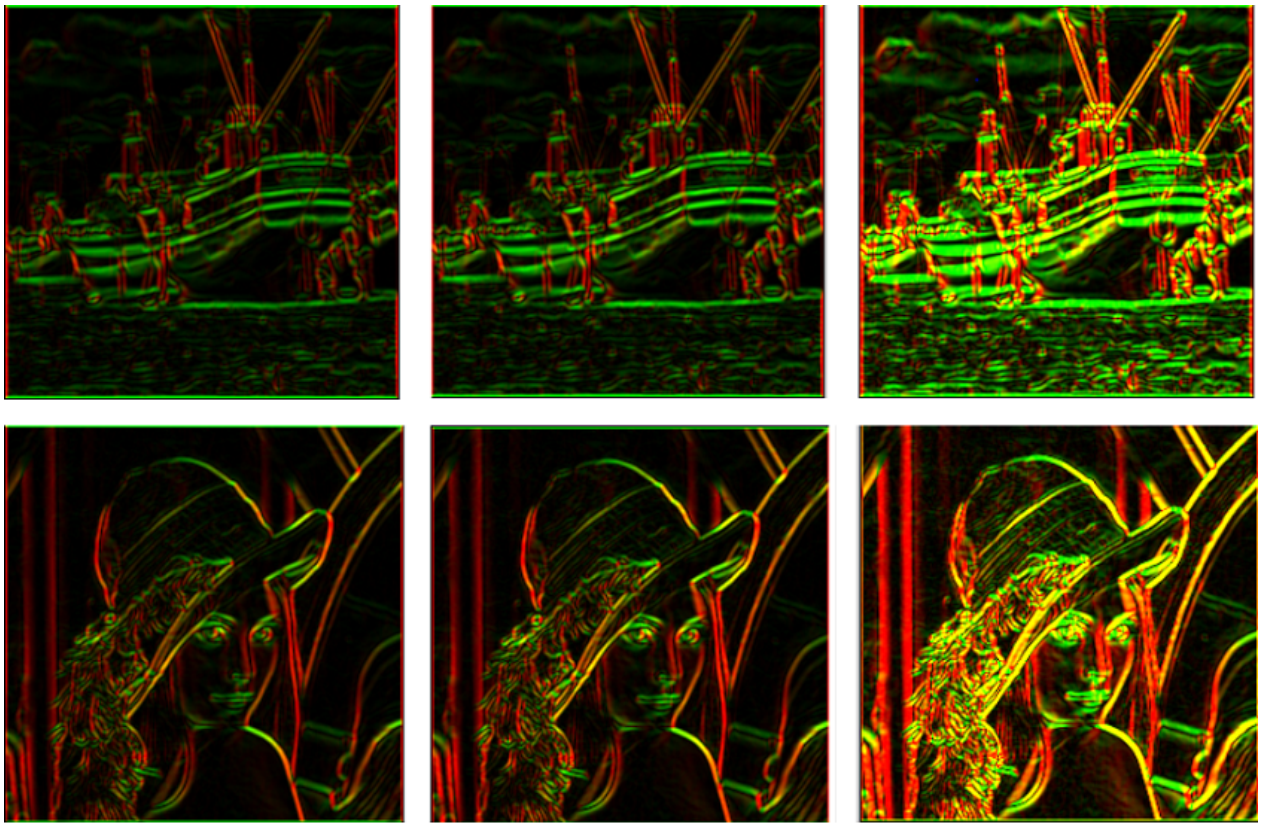

Figure 2 : Gradient bidirectionnel avec différents noyaux. De gauche à droite : Prewitt, Sobel et Kirsch

Filtre gaussien

Nous avons également testé l'application d'un filtre de flou. Les résultats sont visibles sur la figure 3. L'utilisation de ce filtre nous permet notamment de supprimer le *"bruit"* des images de bases que nous pouvons utiliser : c'est donc la première étape de notre méthode.

Figure 3 : Application d'un filtre de flou, 3 itérations

Seuillage

Seuillage global

La première méthode de seuillage que nous avons implémenté est un seuillage global. Une première méthode consiste à calculer une valeur moyenne sur l'ensemble de l'image et à l'utiliser comme un seuil.
Nous avons également développé une interface nous permettant de tester les différentes valeurs d'un seuil dans le but de comparer et de trouver les meilleurs paramètres pour nos images tests.
Nos images de seuillage sont des images binaires. On colore le point si le gradient est supérieur au seuil donné. Les résultats sont décrits sur la figure 4.

Figure 4 : Seuillage global. A gauche, avec un seuil automatiquement calculé sur l'image source. A droite, avec un seuil réglé manuellement

Seuillage hystérésis

Nous nous sommes ensuite tournés vers le seuillage par hystérésis dans le but d'obtenir de meilleurs résultats. Les valeurs des seuils haut et seuils bas sont définis à la main dans l'interface de l'application, on peut donc voir l'impact en temps réel des sliders sur l'étape de seuillages.
Il est également possible de spécifier la taille du voisinage dans lequel on va chercher les points du second seuil. Plus on prend de voisins plus les contours ont de chances d'être fermés directement à l'étape de seuillage.
Les résultats du seuillage hystérésis avec différents paramètres sont décrit sur la figure 5.

Figure 5 : Seuillage hysteresis. Les seuils haut et bas sont définis manuellement pour chaque image

Post-traitements

Affinage des contours

Une fois notre filtrage et seuillage effectué, nous avons développé un algorithme d'affinage des contours.
Pour chaque pixel de l'image, nous calculons les modules des gradients aux points *M1* et *M2* (qui sont à équidistance du pixel en cours, dans la direction du gradient). Pour calculer *gM1* et

gM2, on interpole le valeurs des gradients calculés pour deux points dans le voisinage de *M1* et *M2* selon ces formules :

```
Si gX > gY et que gY > 0:
  gM1 = (gX / gY) * G(i - 1, j) + [(gY - gX)  / gY] * G(i - 1, j + 1)
  gM2 = (gX / gY) * G(i + 1, j - 1) + [(gY - gX)  / gY] * G(i - 1, j)

Si gY > gX et que gX > 0:
  gM1 = (gY / gX) * G(i, j + 1) + [(gX - gY)  / gX] * G(i - 1, j + 1)
  gM2 = (gX / gY) * G(i, j - 1) + [(gX - gY)  / gX] * G(i + 1, j - 1)
```

On considère un pixel comme maxima local si sa norme G est supérieur à celles des points *M1* et *M2*, donc si `G > GM1 et G > GM2` .

Les résultats de cette étapes sont visibles dans la figure 6. Les résultats sont variables : on peut remarquer la création de trous à plusieurs endroits dans chaque image. Néanmoins, les contours bien que ne faisant pas 1 pixel de largeur sont beaucoup plus fins et précis.

Figure 6 : Affinage avec la technique des maxima locaux.

Fermeture des contours

Notre algorithme de fermeture fonctionne comme celui d'OpenCV : la première étape est celle dite de dilatation ou l'on élargit les contours. L'algorithme est le suivant :

```
Copie image_affinage => image_dilatation
Pour chaque pixel de l'image résultat de l'affinage
    Pour chaque pixel du voisinage 3x3 du pixel regardé
        On cherche le point avec la norme de gradient maximale
    Si cette norme est supérieur au seuil haut de l'hysteresis
        On met le point à blanc dans image_dilatation
```

Cette étape nous permet de fermer les contours de manquant. L'étape suivante est celle d'érosion avec l'algorithme suivant :

```
Pour chaque pixel de l'image résultat de la dilatation
    Si ce pixel a X voisins non coloré
        On le décolore
```

Les résultats de cette étapes dépendent de la variable *X* représentant le nombre de voisins *seuil*.
Nos résultats avec cette algorithme de fermeture (étape dilatation et érosion) ne sont pas excellent visuellement : les résultats en sortie de l'étape d'affinage semblent meilleurs.
Nous n'avons pas eu le temps de tester d'autres algorithmes.

Composantes connexes

Dans cette dernière étape, nous essayons de détecter les composantes connexes de notre image : le but est de créer des clusters de points qui pourraient être utilisés dans d'autres opérations. Nous utilisons un algorithme de *labeling* que nous avons inventé à partir de nos recherches. Le pseudo-code est le suivant :

```
A partir de l'image affinée
Initialiser une matrice label de la taille de l'image
currentLabel = 0;
Pour chaque pixel de l'image résultat de l'affinage
    Si ce pixel est noir ou déjà labellisé
        continue;
```

```
Sinon
    Assigner currentLabel à ce pixel
    Pour chacun de ses voisins en NxN colorés
            Assigner currentLabel à ce voisin
    Fin Sinon
currentLabel++;
```

On se déplace donc dans le voisinage d'un premier pixel jusqu'à ne plus trouver de voisin : on clôture l'ensemble si on en trouve plus et on passe à un autre pixel. On obtient donc une image avec *currentLabel* couleurs. Les résultats de cette algorithme sont présentés sur la figure suivante.

Figure 7 : Détection des composantes connexes avec notre algorithme de labelling.

Possibilités d'améliorations

Comme nous l'avons déjà dit, notre algorithme de fermeture ne donne pas de résultats visuellement satisfaisants : c'est une première piste d'amélioration pour notre programme. Il serait intéressant de tester d'autres algorithmes et de les intégrer à notre pipeline. L'étape d'affinage supprime également beaucoup de pixels : cela crée parfois beaucoup de trous dans

les images. Il serait bon d'exposer des paramètres pour cette étape, de manière à pouvoir faire un affinage plus ou moins important.

L'étape de recherche de composantes connexes est également assez "simpliste" et pourrait être amélioré en recherchant les voisins dans une direction plus précise au lieu du voisinage en "carré" du pixel.

Une autre amélioration pratique serait de rendre le choix de l'image configurable directement dans l'interface de l'application. C'est le seul paramètre "en dur" qu'il nous reste.

Utilisation, compilation, tests et répartition du travail

Nous avons développé notre programme sous *Windows* et *Visual Studio*. Nous avons inclus dans l'archive la solution complète qui fonctionne sur les machines de l'université avec *Visual Studio 2015*. Le projet est également accessible depuis github.

Nous avons essayé d'exposer le plus de paramètres possibles dans l'interface du programme. Il est ainsi possible de régler le type de gradient *(bi/multi directionnel*), le kernel utilisé pour le calcul du gradient (*Prewit / Sobel / Kirsch*), le seuil haut et le seuil bas de l'hystérésis ainsi que la taille de la matrice de voisinage pour les étapes hystérésis et recherche de composantes connexes.

Le seul paramètre "en dur" du programme concerne l'image de test. Pour la choisir, des *defines* sont présents dans le fichier *inputs.h*, il suffit de commenter/décommenter les lignes en haut du fichier. Au lancement, nous avons fait en sorte que les meilleurs paramètres pour nos algorithmes soient enregistrés pour chaque image.

Concernant la répartition du travail, nous avons découpé le projet comme ceci :

- Vincent s'est occupé de l'algorithme d'affinage, du gradient multidirectionnel, de la création de l'interface graphique et du paramétrage du programme.
- Axel a travaillé sur le seuillage hysteresis, la fermeture et la recherche de composantes connexes.
- Les opérations de bases (gradient bi directionnel, filtre de flou, convolution) ont été codées ensemble.

Conclusion

Nous avons beaucoup apprécié ce projet et nous pensons avoir des résultats corrects dans l'ensemble. La prise en main des composantes d'OpenCV et le codage des opérations existantes a été une expérience pédagogique et enrichissante. Nous savons maintenant ce qui se cache derrière les appels des fonctions d'OpenCV a priori.

Les parties concernant les composantes connexes et la fermeture furent également intéressantes dans le sens où nous avons dû faire des recherches et trouver des algorithmes qui donnent de bons résultats. Néanmoins, le temps nous a manqué pour faire tout ce que nous voulions faire à la base et nous aurions aimé poussé plus loin cet aspect recherche.

Bibliographie et Sitographie

http://master-ivi.univ-lille1.fr/fichiers/Cours/pje-semaine-5-analyse-cc.pdf

https://docs.opencv.org/2.4/

https://github.com/FlorianLance/Contour-detection

http://www.optique-ingenieur.org/fr/cours/OPI_fr_M04_C05/co/Contenu_07.html

http://www.programming-techniques.com/2013/03/sobel-and-prewitt-edge-detector-in-c.html