

# Image, Acquisition, Analyse et Traitement de l'Image

Master 2 Recherche IGI (Informatique Graphique et Image)

Université de Lyon 1

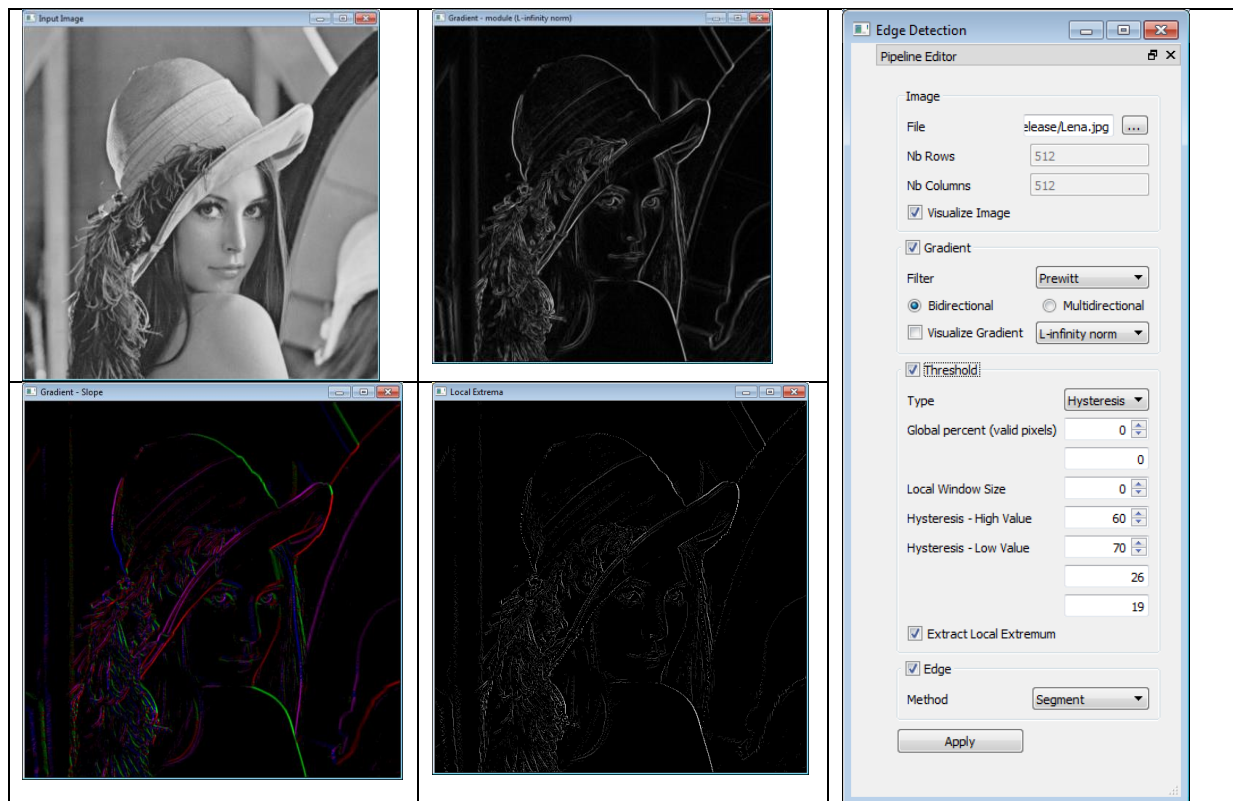
Elèves :

Promo : 2015 – 2016

Pascal GUEHL – p1511749

Clément PICQ –

## TP : Détection de contours



## Contenu

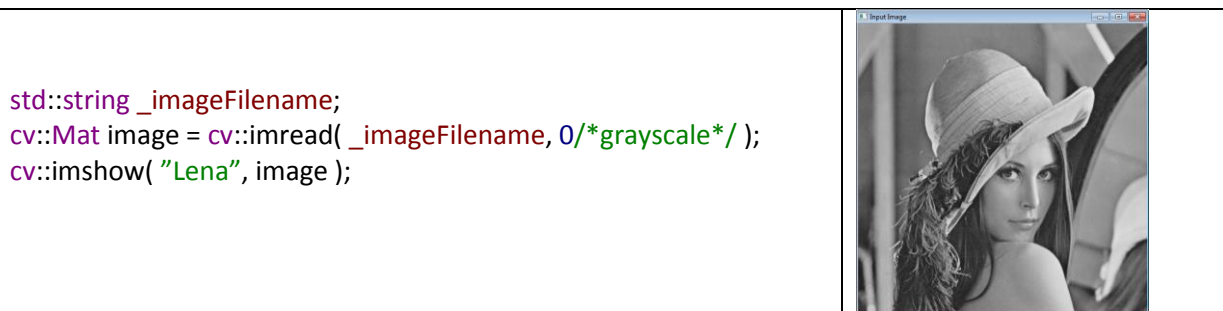
Introduction.....	3
Contexte .....	3
I. Détection de contours par utilisation d'opérateurs différentiels .....	4
1. Gradient.....	4
a. Cas Bi-directionnel.....	4
b. Cas Multi-directionnel .....	6
c. Interface paramétrable .....	7
2. Seuillage .....	8
a. Méthode à seuil unique.....	8
b. Seuillage par hystérésis .....	11
c. Autres méthodes d'estimation des seuils .....	13
d. Interface paramétrable .....	14
3. Affinage de contours .....	15
a. Extraction des maxima locaux.....	15
b. Autres méthodes ? .....	16
II. Post-traitement pour affiner les contours .....	17
1. Organisation des contours / Structuration des données .....	17
2. Extraction des composantes connexes .....	20
3. Fermeture de contours .....	21
a. Méthodes .....	21
b. Réflections .....	21
c. Résultats .....	22
III. Performances .....	23
1. Affichage des résultats .....	23
IV. Programme informatique.....	24
1. Dépendances .....	24
1. Environnement de développement .....	24
2. Compilation / Execution .....	24
3. Utilisation .....	25
Conclusion .....	26

# Introduction

## Contexte

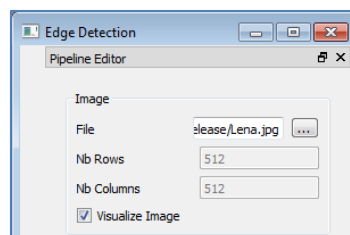
Ce TP d'Analyse d'Images comporte 2 parties. La première partie concerne la détection de contours par utilisation d'opérateurs différentiels. L'obtention du module et la pente du Gradient constitue la base du pipeline de détection des contours. Des étapes de post-traitements permettront de récupérer plus précisément ces contours (affinage) par seuillage et extraction des maxima locaux. La fin du pipeline concerne l'extraction topologique/géométrique des contours.

Nous avons choisi d'assurer la gestion des images et des matrices par la librairie OpenCV. Elle fournit le type de base `cv::Mat`. Il permet de stocker des données sous forme matricielle, parfaitement adapté aux images. En outre, il donne accès à des fonctionnalités de lecture, écriture d'images et visualisation.



L'environnement de programmation multiplateforme sera basé sur qtCreator. Le programme a été testé sur Linux et Windows.

L'interface graphique a été réalisée avec la librairie Qt via qtCreator. Ci-dessous, un screenshot de la partie de gestion des images d'entrée :



L'interface permet de saisir les fichiers images et en affiche les dimensions. L'API du programme accepte des tailles non symétriques, des images en niveau de gris ou couleur, aux formats .jpg et .png

On détaillera les autres fonctionnalités de l'IHM dans chacune des séquences relatives aux différentes parties du pipeline de traitement.

# I. Détection de contours par utilisation d'opérateurs différentiels

## 1. Gradient

On procède ici au calcul du vecteur gradient en chaque point de l'image. On applique différents types de masques de convolution comme Prewitt, Sobel et Kirsch. Deux cas sont traités : bidirectionnel et multidirectionnel.

Tous les algorithmes bas niveaux sont gérés par une classe utilitaire « Algorithm » contenant des méthodes « static ».

### a. Cas Bi-directionnel

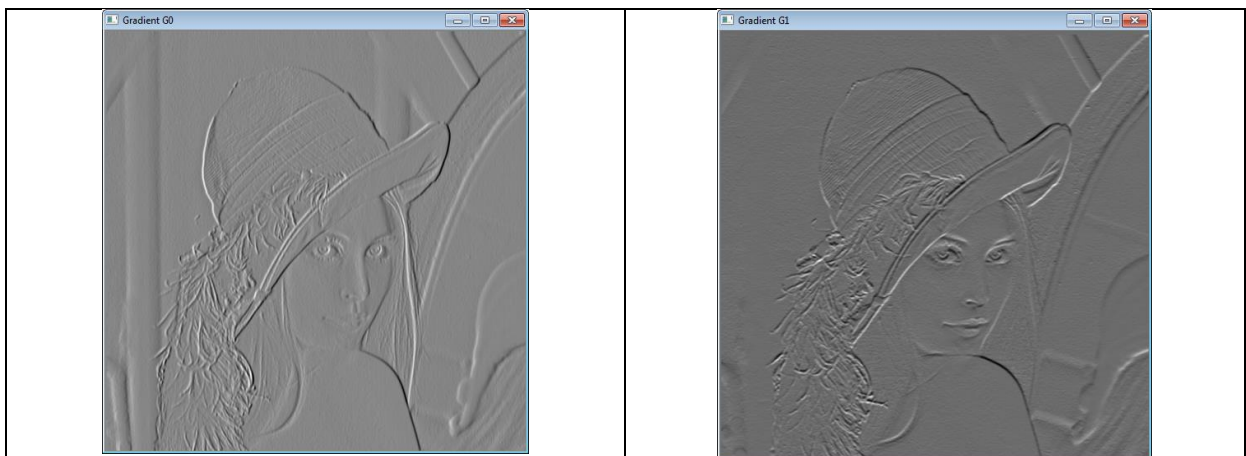
A partir d'une image en entrée, on calcule les composantes horizontales et verticales du Gradient.

```
static cv::Mat algorithm::filter( const cv::Mat& src, const cv::Mat& filter ) ;
```

Cette fonction effectue la convolution classique d'une image avec un noyau de filtre. Elle prend en entrée une image (`const cv::Mat& src`) et un filtre comme Prewitt, Sobel ou Kirsch (`const cv::Mat& filter`), et retourne une matrice (image) filtrée.

Notes : Nous avons opté pour des noyaux de convolution de taille 3x3, mais le code de la fonction reste générique.

On obtient les composantes horizontales Gx et verticales Gy du gradient. Chacun donne des informations de directions des contours locaux. Le gradient est perpendiculaire à la surface des objets (ou formes). Ci-dessous, un exemple d'application d'un filtre de Prewitt 3x3 horizontal (G0) et vertical (G1) sur l'image de Lena. Les résultats sont re-normalisés sur 0-255 :



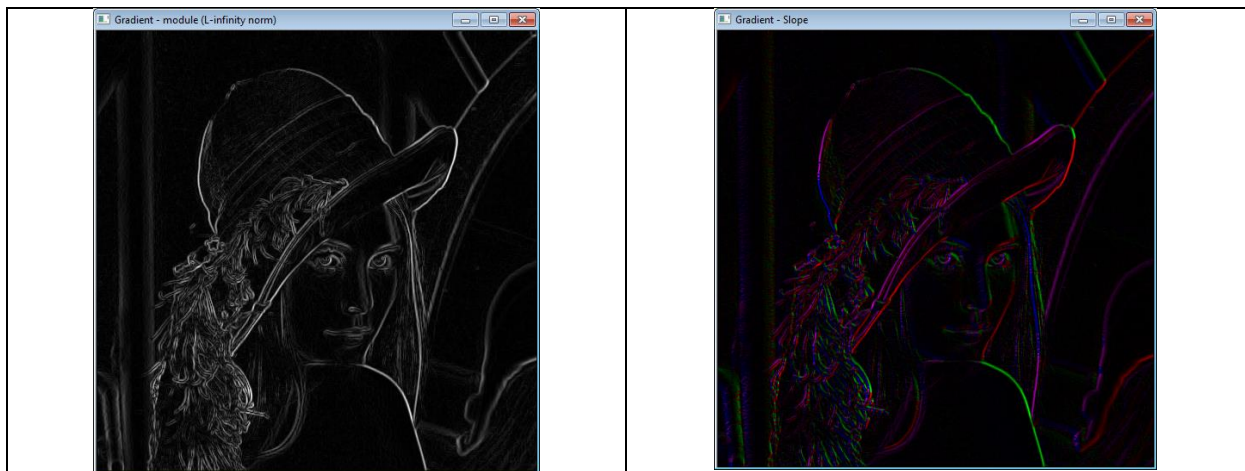
Pour reconstruire le gradient sous la forme d'un module et d'une pente, on utilise les formules suivantes :

$$\text{Module} = \sqrt{Gx^2 + Gy^2}$$

$$\text{Theta} = \arctan( Gy / Gx )$$

La norme est ici la norme L2. Pour la pente Theta, on utilise la fonction `atan2()` de la librairie mathématique standard. En effet, comparé à `atan()`, elle permet de prendre en compte toutes les possibilités de positionnement dans les différents quadrants du cercle.

Ci-dessous, un exemple d'application d'un filtre de Prewitt 3x3 horizontal (G0), vertical (G1), à 45 degré (G2) et à 135 degrés (G3) sur l'image de Lena. Les résultats sont re-normalisés sur 0-255 :



Concernant la pente, nous avons choisi de colorer les valeurs selon 3 cas. Etant donné que la pente Theta donnée par la fonction  $\text{atan2}()$  varie entre  $-\pi$  et  $\pi$ , on divise l'espace des valeurs en 4 en attribuant une couleur RGB :

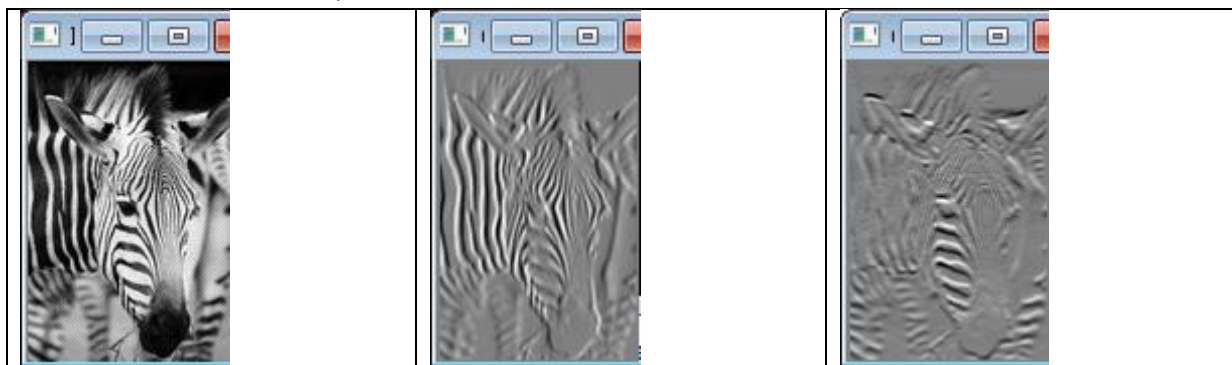
$[-\pi ; \pi/2] \Rightarrow$  rouge

$[-\pi/2 ; 0] \Rightarrow$  vert

$[0 ; \pi/2] \Rightarrow$  bleu

$[\pi/2 ; \pi] \Rightarrow$  violet (combinaison rouge/bleu)

Ci-dessous, un autre exemple montrant la détection des contours horizontaux et verticaux :



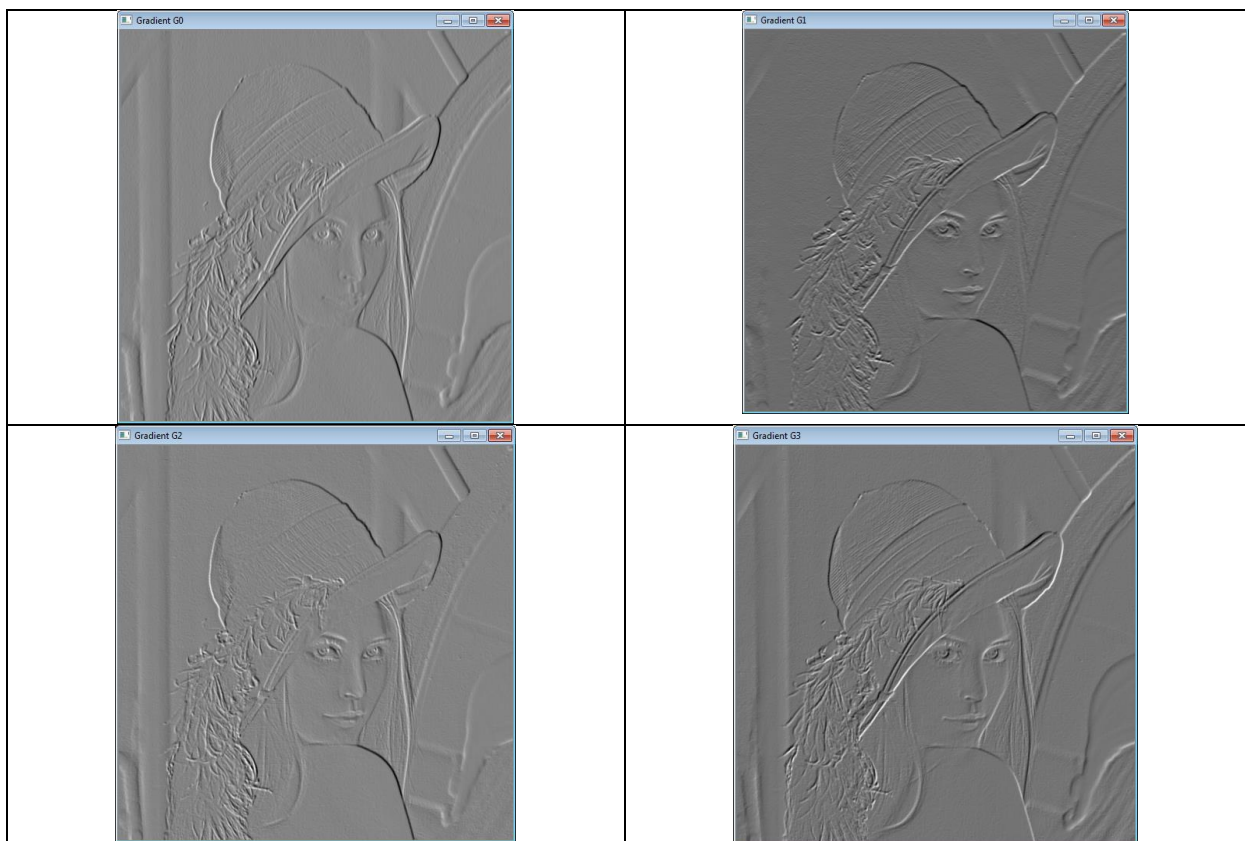
### b. Cas Multi-directionnel

Quatre directions ont été sélectionnées : 0, 45, 90 et 135. Ci-dessous, les différents masques des filtres multidirectionnels :

Filtres / directions	0	45	90	135
Prewitt	$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -1 \end{bmatrix}$
Sobel	$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$	$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix}$
Kirsch	$\begin{bmatrix} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & 5 \end{bmatrix}$	$\begin{bmatrix} -3 & 5 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & -3 \end{bmatrix}$	$\begin{bmatrix} 5 & 5 & 5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix}$	$\begin{bmatrix} 5 & 5 & -3 \\ 5 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix}$

On obtient les composantes horizontales G0 et verticales G1 du gradient, ainsi que des composantes diagonales G2 et G3, respectivement à 45 et 135 degrés.

Ci-dessous, un exemple d'application d'un filtre de Prewitt 3x3 horizontal (G0), vertical (G1) et diagonaux (G2 et G3) sur l'image de Lena. Les résultats sont re-normalisés sur 0-255 :



La méthode suivante prend en entrée une liste de composantes du gradient et renvoie son module.

```
static cv::Mat algorithm::moduleL1( cv::Mat* gradientComponents, int nbComponents )
```

Dans le cas multidirectionnel, le gradient se compose de 4 composantes G0, G1, G2, G3 permettant de détecter les contours orientés à 0, 45, 90 et 135 degrés. On utilise soit la norme L1, soit L-infinie.



Norme L-infinie :

$$\text{Module} = \max( G_i )$$

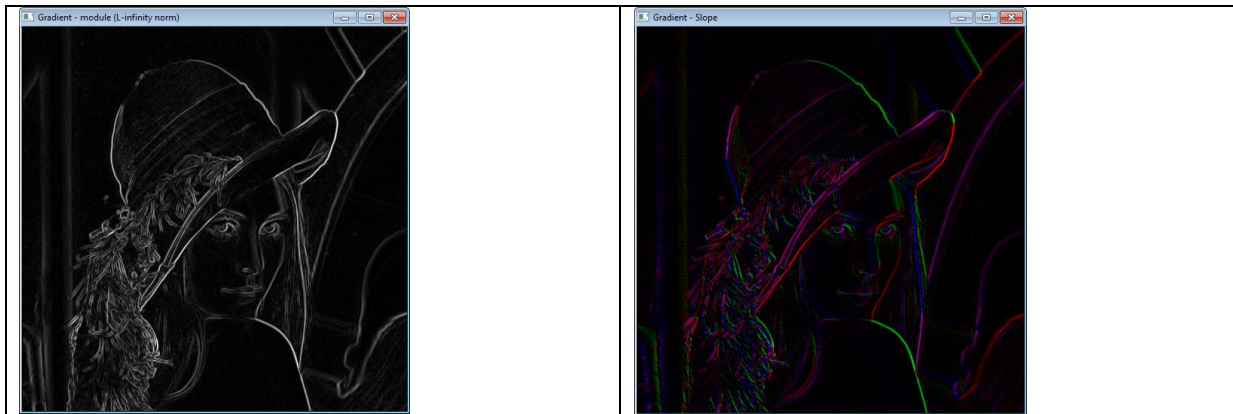
$$\text{Theta} = \max( \arctan( G_1 / G_0 ), \arctan( G_3 / G_2 ) )$$

Norme L1 : on cherche les deux plus grandes valeurs  $G_a$  et  $G_b$  des  $G_i$  et on en prend la norme L1

$$\text{Module} = \sum( G_a + G_b ) ;$$

$$\text{Theta} = \arctan( G_a / G_b )$$

Ci-dessous, un exemple d'application d'un filtre de Prewitt 3x3 horizontal ( $G_0$ ), vertical ( $G_1$ ) et diagonaux ( $G_2$  et  $G_3$ ) sur l'image de Lena. Les résultats sont re-normalisés sur 0-255 :



Les filtres multidirectionnels sont plus gourmands en mémoire, demande plus de calculs mais permettent de détecter un plus grands nombres de configuration de contours grâce aux directions diagonales.

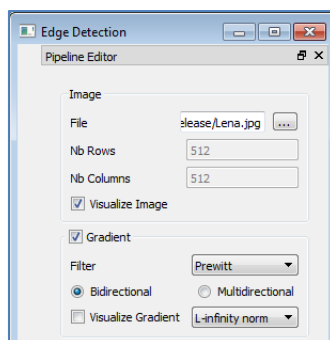
NOTE :

Il existe aussi d'autres méthodes de détection de contours, notamment avec l'opérateur Laplacien qui opère sur les dérivées secondes. Nous ne les avons pas implémentées par manque de temps.

### c. Interface paramétrable

L'utilisateur peut activer ou non la génération du gradient :

- type de filtre : Prewitt, Sobel, Kirsch
- mode bidirectionnel ou multidirectionnel
- choix de la norme L1, L-infinie
- visualiser ou non les composantes du gradient (horizontales, verticales, diagonales)



## 2. Seuillage

Le seuillage d'image est une méthode de segmentation d'image. À partir d'une image en niveau de gris, le seuillage d'image peut être utilisé pour créer une image comportant uniquement deux valeurs, noir ou blanc (monochrome).

L'étape de filtrage précédente permet d'estimer la probabilité qu'un pixel soit un point contour candidat. L'information est stockée sous forme d'intensité du gradient (module). Il reste à décider si un pixel est effectivement un point contour. Il existe plusieurs méthodes de seuillage. Dans la suite nous allons voir des méthodes se basant sur des informations globales à l'image, locales, utilisant un ou plusieurs seuils. Le résultat obtenu est une image binaire permettant de classer les pixels en deux catégories. Le principe est simple :

Si  $||\text{Gradient}|| > \text{seuil}$  , le pixel est un point contour candidat

Si  $||\text{Gradient}|| \leq \text{seuil}$  , le pixel n'est pas un point contour candidat.

En pratique, on pourra utiliser des fonctions du type :

```
static void algorithm::applyThreshold( cv::Mat& data, int threshold )
```

```
static cv::Mat algorithm::applyThreshold( const cv::Mat& data, int threshold )
```

Pour chaque pixel ( i, j ) du module du gradient :

Si  $\text{Gradient}( i, j ) > \text{threshold}$  , pixel = 1 (ou 255)

Si  $\text{Gradient}( i, j ) \leq \text{threshold}$  , pixel = 0

Il restera ensuite à utiliser les points obtenus pour former les contours proprement dits. Cela nécessitera des étapes supplémentaires, car les contours formés par ces points sont parfois : épais, bruités, interrompus (non fermés).

Pratiquement, cette méthode est peu intéressante car elle nécessite de connaître la valeur de seuil à appliquer pour chaque image. Or, pour obtenir des résultats utilisables, elle nécessite plusieurs ajustements et varie en fonction de l'image en entrée. Pour pallier à cela, nous allons voir trois méthodes qui permettent de déterminer des valeurs de seuils « automatiques ».

### a. Méthode à seuil unique

#### i. Méthode globale (histogramme)

Il s'agit ici de donner de déterminer le seuil en se basant sur l'analyse de l'histogramme des niveaux de gris de l'image. L'histogramme est un outil classique d'exploration de données. C'est un moyen rapide pour étudier la répartition d'une variable : il donne une estimation statistique de la distribution sous-jacente de des valeurs de niveaux de gris d'une image. L'idée est de retenir un pourcentage des données les plus significatives (c'est-à-dire les contours). Etant donné un paramètre utilisateur de « pourcentage de données à conserver », l'histogramme permet de déterminer une valeur de seuil associé. Par exemple, l'utilisateur va dire « je souhaite conserver 80% des données », l'algorithme va alors déterminer la valeur de seuil associé à cette probabilité à partir de l'histogramme cumulée. Elle sera ensuite utilisée dans la binarisation de l'image.

Afin de déterminer cette valeur de seuillage, l'API propose la fonction « globalThreshold » suivante :

```
int algorithm::globalThreshold( const cv::Mat& src, float percentFilter )
```

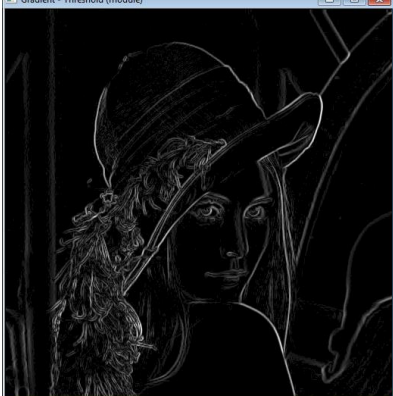
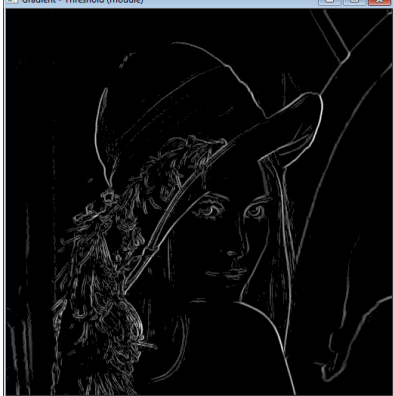
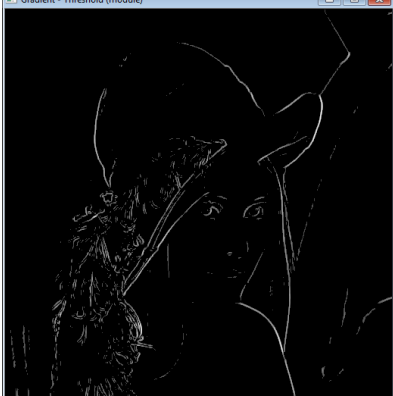



Elle prend en entrée une image ainsi qu'un pourcentage de pixels valides à conserver et retourne le seuil associé à l'étude de l'histogramme des données. Pratiquement, l'histogramme est implémenté sous forme d'un tableau à 256 valeurs représentant le nombre d'occurrence des diverses valeurs de niveaux de gris de l'image en entrée. Ensuite, on se base sur l'histogramme cumulé afin de déterminer le seuil pour lequel la probabilité en entrée est atteinte. Le code ne construit pas l'histogramme cumulé sous forme de tableau mais teste les données à la volée en calculant la somme cumulée et en testant la valeur avec la probabilité en entrée.

La valeur de seuil retournée est ensuite envoyée aux méthodes de seuillage standard vu précédemment :

```
static void algorithm::applyThreshold( cv::Mat& data, int threshold )
static cv::Mat algorithm::applyThreshold( const cv::Mat& data, int threshold )
```

Ci-dessous, 4 résultats de seuillage global étant donné les valeurs de pourcentage de pixels à conserver suivantes : 75%, 50%, 25% et 15%

	
75% de pixels valides => seuil = 16	50% de pixels valides => seuil = 36
	
25% de pixels valides => seuil = 69	15% de pixels valides => seuil = 93

## ii. Méthode locale (voisinage)

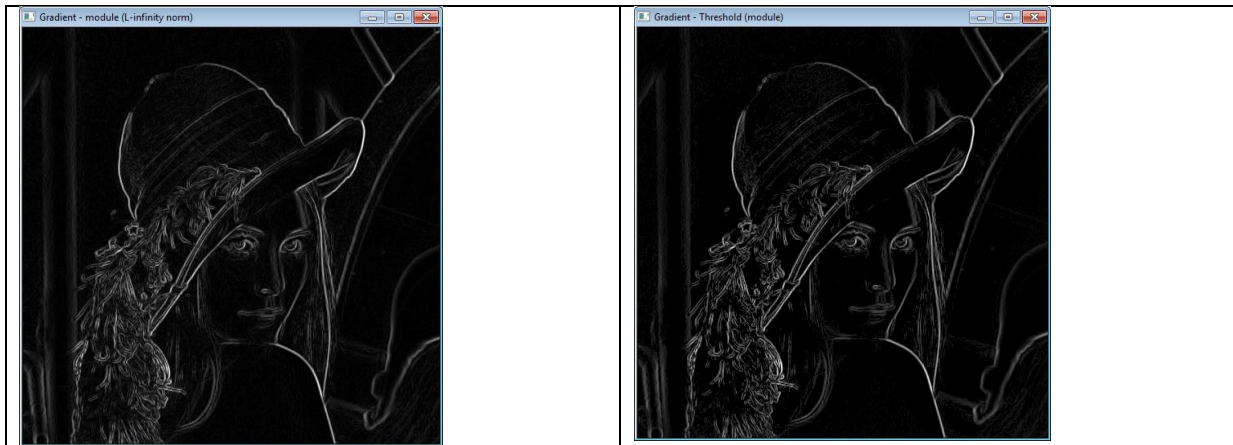
On ne s'intéresse plus ici à aux informations globales contenues dans l'image mais plutôt aux phénomènes locaux. La valeur du seuil est alors déterminée localement en fonction d'un nombre restreint de points, dans un voisinage plus ou moins restreint du point considéré. Le principe est de calculer la moyenne dans une fenêtre de taille déterminée par l'utilisateur. Concrètement, l'API fournit la fonction suivante :

```
static cv::Mat algorithm::localThreshold( cv::Mat& src, int windowSize )
```

Elle prend en entrée une image source "src" ainsi qu'une taille de fenêtre locale et retourne l'image seuillée. L'utilisateur indique la taille de fenêtre dans laquelle calculer la moyenne des pixels. Elle indique en fait le nombre de points du voisinage à considérer. Cette méthode correspond à une convolution par un filtre moyenneur dont le coefficient se calcule comme suite :

$\text{coeff} = 1 / \text{nbElements}$  avec  $\text{nbElements} = (2 * \text{windowSize} + 1)^2$

Une fois la moyenne calculée sur la fenêtre locale, cette valeur va faire de servir de seuil. L'image va être binarisée en comparant cette moyenne à la valeur du pixel en cours comme vu précédemment. Ci-dessous, un exemple de seuillage locale avec une fenêtre de 50 voisins :



## iii. Commentaires

La méthode globale produit des résultats intéressants mais parfois trop violents : on souhaiterait pouvoir conserver certaines valeurs. On n'a pas assez de contrôle pour s'adapter aux situations différentes de diverses images.

La méthode de seuillage locale est plus longue à calculer car elle est de complexité supérieure. Le seuil (c'est-à-dire la moyenne) est à déterminer pour chaque pixel de l'image. Le nombre d'opération peut vite être très pénalisant :  $\text{nbColumnsImage} * \text{nbRowsImage} * \text{nbColumnsMeanFilter} * \text{nbColumnsMeanFilter}$ . De plus, les résultats obtenus sur l'image de Lena ne sont pas très probants, il faut un grand voisinage pour arriver à supprimer des valeurs. Ici, 50, soit un filtre moyenneur de taille 101x101.

En outre, seuiller avec un unique seuil de décision pour toute une image peut causer des problèmes. On risque de ne pas détecter des points contours situés dans des zones de faible contraste, ou encore de sélectionner à tort des points contours dans les zones bruitées. La prochaine méthode, par hystérésis, va permettre de récupérer des valeurs écartées dans la première étape.

## b. Seuillage par hystérésis

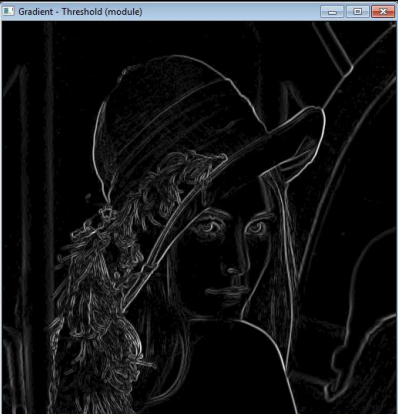
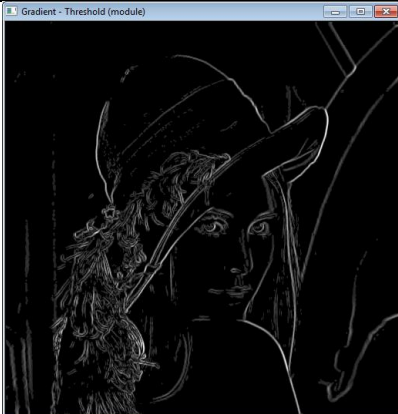
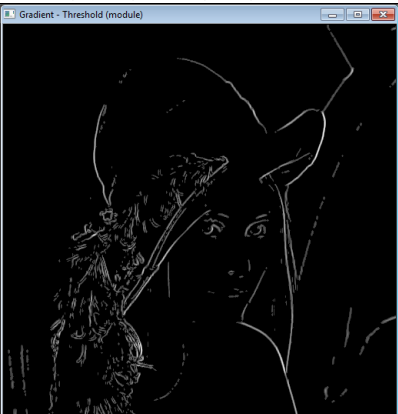
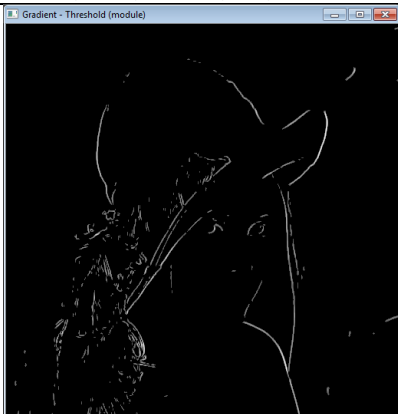
### i. Méthode générale

Dans cette méthode, on va appliquer un premier seuillage identique au seuillage global vu précédemment, en indiquant un pourcentage de pixels valides. Ce premier seuillage va permettre de sélectionner des « points de contours sûrs ». On va ensuite récupérer certains points laissés de côté par ce 1<sup>er</sup> traitement en appliquant un 2<sup>ème</sup> seuillage globale mais cette fois-ci avec un pourcentage de pixels valides plus élevés, ce qui aura comme conséquence de générer un seuil plus bas. Il laissera ainsi passer plus de points. Ce 2<sup>ème</sup> seuillage sera cependant « contraint ». Un pixel sera déterminé comme contour, si et seulement si il passe le 2<sup>ème</sup> seuil et si le même point avait passé le 1<sup>er</sup> seuillage.

Les données sont ainsi seuillées avec un seuil haut et un seuil bas. La particularité de l'algorithmique se situe pour les valeurs situées entre ces deux seuils, pour lesquelles on recherche une connectivité géométrique. Concrètement, l'API fournit la fonction suivante :

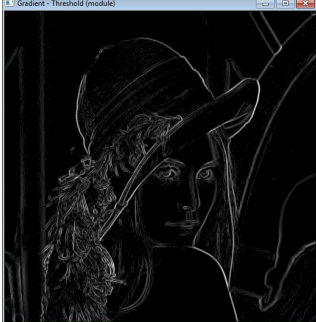
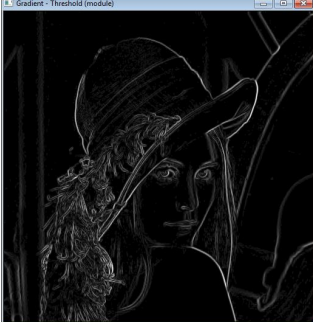
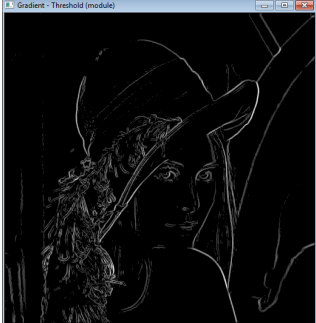
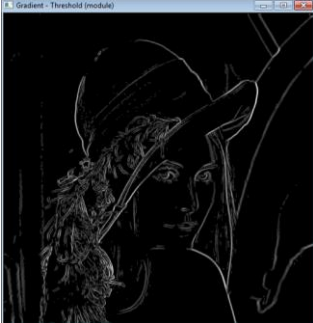
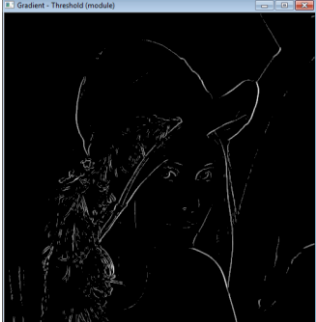
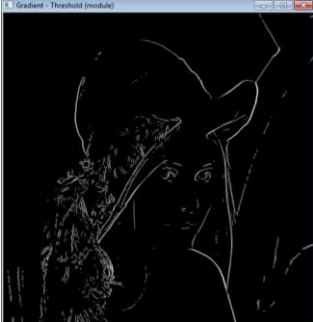
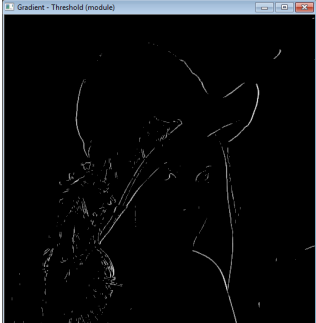
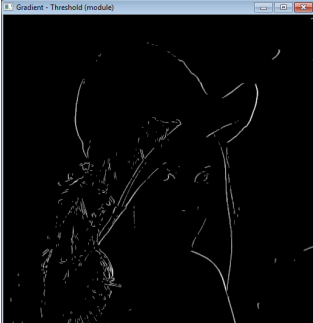
```
static cv::Mat algorithm::hysteresis( const cv::Mat& src, int& pHysteresisHighThreshold,  
                                     int& pHysteresisLowThreshold )
```

Cette fonction prend en entrée une image source "src" et applique l'algorithme de seuillage par hystérésis. En interne, elle utilise deux paramètres statiques de pourcentage de pixels valide « haut » et « bas », sur le même principe que le seuillage globale. Ci-dessous, le résultat des diverses combinaisons de pourcentages de pixels valides haut et bas :

	
% de pixels valides : 75% (haut) et 99% (bas) => seuils : 16 (haut) et 3 (bas)	% de pixels valides : 50% (haut) et 75% (bas) => seuils : 36 (haut) et 16 (bas)
	
% de pixels valides : 25% (haut) et 50% (bas) => seuils : 69 (haut) et 36 (bas)	% de pixels valides : 15% (haut) et 25% (bas) => seuils : 93 (haut) et 69 (bas)

## ii. Comparaisons : global vs hystérésis

Sur la gauche les résultats de la méthode global, sur la droite celle par hystérésis :

	
75% de pixels valides => seuil = 16	% de pixels valides : 75% (haut) et 99% (bas) => seuils : 16 (haut) et 3 (bas)
	
50% de pixels valides => seuil = 36	% de pixels valides : 50% (haut) et 75% (bas) => seuils : 36 (haut) et 16 (bas)
	
25% de pixels valides => seuil = 69	% de pixels valides : 25% (haut) et 50% (bas) => seuils : 69 (haut) et 36 (bas)
	
15% de pixels valides => seuil = 93	% de pixels valides : 15% (haut) et 25% (bas) => seuils : 93 (haut) et 69 (bas)

Cette méthode par hystérésis est plus coûteuse (voir partie « Performances ») mais offre les meilleurs résultats. On a plus de contrôle pour essayer de « classifier » les pixels comme points contours ou non. Le 2<sup>ème</sup> seuillage permet de récupérer des valeurs écartées dans la 1<sup>ère</sup> étape de seuillage.

### c. Autres méthodes d'estimation des seuils

Il existe d'autres méthodes de seuillage automatique. On présente ici les informations sous forme de veille technologique ou Related Works comme dans les articles scientifiques.

#### i. *Le seuillage adaptatif ( « Adaptive Thresholding » )*

Cette classe de méthodes utilise la valeur d'un seuil qui est dynamiquement changé afin de rendre compte de l'intensité locale des différentes zones de l'image d'entrée (en niveau de gris). Deux méthodes, le « Dynamic thresholding » et le « Constrained average thresholding » permettent de réaliser ce seuillage adaptatif. Ces méthodes ont la particularité de rehausser le contraste ainsi que les contours, elles ont en contrepartie le désavantage de représenter les grandes surfaces de pixels à un niveau d'intensité constant.

SOURCE : <http://www.tsi.telecom-paristech.fr/pages/enseignement/ressources/beti/dither/arnaud/adapt.htm>

#### ii. *La méthode d'Otsu*

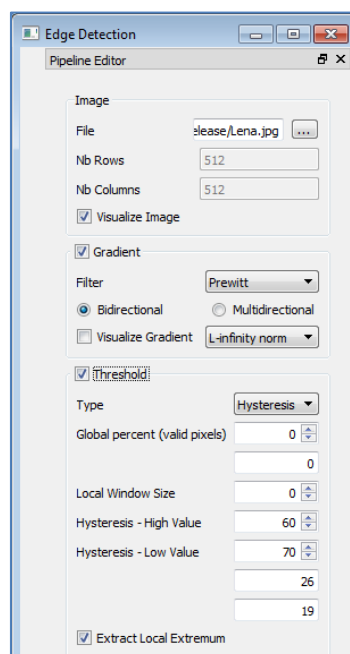
En vision par ordinateur et traitement d'image, la méthode d'Otsu est utilisée pour effectuer un seuillage automatique à partir de la forme de l'histogramme d'une image, ou la réduction d'une image à niveaux de gris en une image binaire. L'algorithme suppose alors que l'image à binariser ne contient que deux classes de pixels, (c'est-à-dire le premier plan et l'arrière-plan) puis calcule le seuil optimal qui sépare ces deux classes afin que leur variance intra-classe soit minimale.

SOURCE : [https://fr.wikipedia.org/wiki/M%C3%A9thode\\_d%27Otsu](https://fr.wikipedia.org/wiki/M%C3%A9thode_d%27Otsu)

#### d. Interface paramétrable

L'utilisateur peut activer ou non l'utilisation du seuillage :

- type de méthodes : globale, locale, par hystérésis
- méthode globale : la valeur du pourcentage valide de pixels à conserver (le seuil calculé sera affiché dans l'interface)
- méthode locale : taille de la fenêtre pour la prise en compte du voisinage (nombre de points voisins considérés autour de chaque pixel => fenêtre symétrique centrée sur le pixel en cours)
- méthode par hystérésis : la valeur des pourcentages haut et bas valides de pixels à conserver (les seuils calculés seront affichés dans l'interface)



Au terme de cette étape de seuillage, nous avons obtenu une classification des pixels de l'image en deux catégories : ceux appartenant aux contours et les autres. Cependant, les contours obtenus sont souvent épais ou dupliqués. Nous allons maintenant voir comment obtenir des contours d'épaisseur affinée afin de préciser les résultats et permettre leurs extractions.



### 3. Affinage de contours

Dans cette partie on présente une méthode pour affiner les contours obtenus après l'étape de seuillage : l'extraction des minima locaux.

#### a. Extraction des maxima locaux

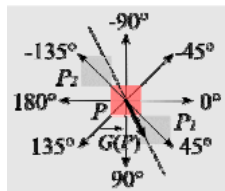
L'API fournit une méthode d'extraction des minima locaux :

```
static cv::Mat localExtremum( const cv::Mat& slope, const cv::Mat& module );
```

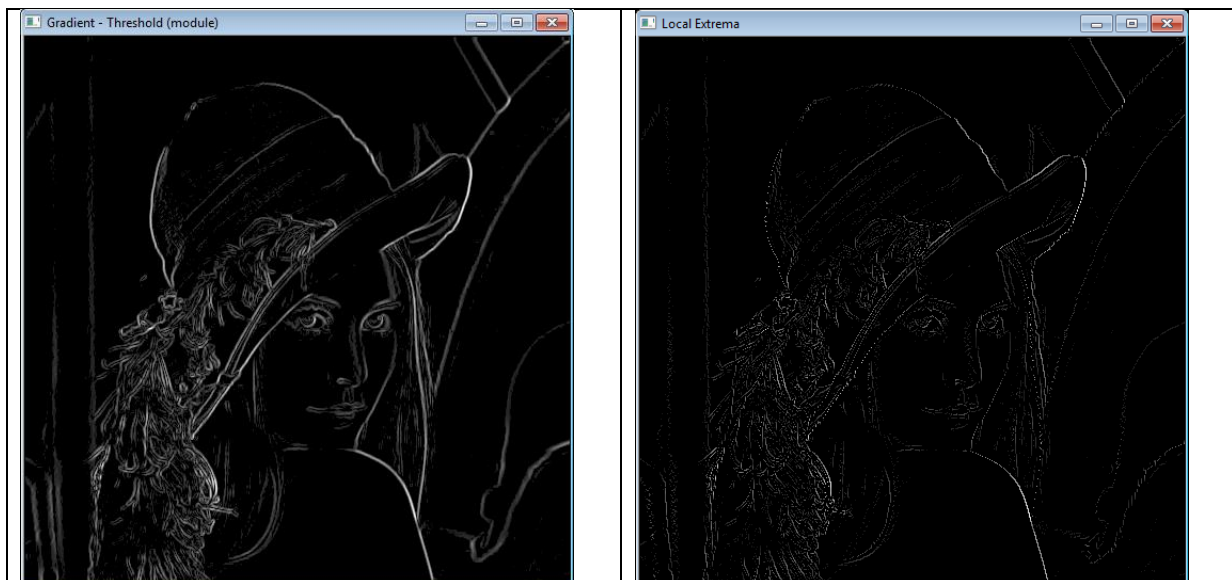
Elle prend en entrée les modules et pente du gradient.

Pour chaque pixel, le but est de déterminer si, dans la direction du gradient, les 2 voisins situés de part et d'autre du pixel en cours ont une valeur de module de gradient inférieure. Si oui, le pixel est un extrema local et on le conserve, sinon sa valeur est mise à 0 (il est écarté).

Afin de déterminer les 2 voisins sur la grille des pixels, on discrétise la direction du gradient selon 8 directions possibles espacées de  $\pi/4$ . Cela correspond à la notion de 8-connexité comme ci-dessous :



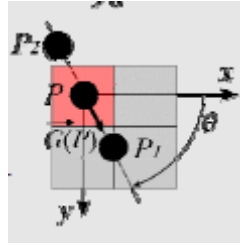
Ci-dessous, un exemple de résultats de suppression de non-maxima locaux :



Par rapport au résultat du seuillage (à gauche), les contours (à droite) ont été affinés jusqu'à obtenir une épaisseur de 1 pixel. La méthode n'est pas exacte puisque les directions sont discrétisées grossièrement, mais le résultat reste cohérent.

### b. Autres méthodes ?

Il est également possible d'utiliser une méthode d'interpolation. Il s'agit de déterminer les 2 points « virtuels » situés de part et d'autre du pixel en cours selon la « vraie » direction du gradient, et non pas une direction discrétisée comme précédemment. Ensuite, il s'agit d'interpoler les valeurs selon les 4 pixels d'une cellule de la grille image, à la manière du filtrage réalisé par OpenGL sur les textures (le filtrage linéaire).



Nous n'avons pas eu le temps d'implémenter cette méthode.

Nous allons maintenant voir comment récupérer et structurer les informations obtenus pour en déduire une notion géométrique/topologique.

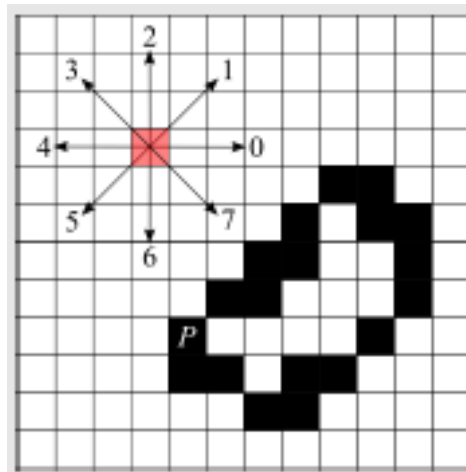
## II. Post-traitement pour affiner les contours

### 1. Organisation des contours / Structuration des données

Le but principal de cette partie est de retrouver les composantes connexes du graphe. Dans notre cas, cela revient essentiellement à identifier et stocker les différentes courbes composant les contours. Pour cela, nous avons choisi d'utiliser le codage de Freeman pour encoder chacune des composantes connexes. On rappelle ici son principe :

#### Codage de Freeman :

On encode les changements de directions basé sur une position de départ ( pixel  $[i,j]$  ). Huit déplacements sont possibles, correspondant à une connexité 8. Voici un exemple. A partir du point P, on peut encoder le contour sous la forme  $\{ 6, 0, 7, 0, 2, 0, 1, 1, 2, 2, 2, 4, 2, 4, 5, 6, 4, 6, 4, 5 \}$  :



Pour optimiser les temps de calculs, on utilise un tableau encodant les 8 mouvements possibles correspondant à chacune des directions. Voici ce tableau indexé par 8 directions et stockant les offsets de mouvements pour atteindre un voisin en 8-connexité :

```
//Freeman directions encoding  
static const int freemanDirections[ 8 ][ 2 ] = { {0,1}, {-1,1}, {-1,0}, {-1,-1}, {0,-1}, {1,-1}, {1,0}, {1,1} };
```

NOTE :

En cas de portage sur GPU, il sera alors possible de le stocker dans des mémoires optimisées de type Read-Only disposant d'un cache (CUDA : texture ou mémoire constante).

Pour stocker ces informations, nous utilisons une structure « Edge » contenant la position du pixel ainsi qu'un vecteur d'entiers encodant les différentes directions. Une structure minimaliste permettant d'optimiser la place en mémoire.

```
/**
 * @brief The Edge struct
 */
struct Edge
{
    Edge() : _directions() {}

    /**
     * Pixel start position
     */
    int s_x, s_y;

    /**
     * Pixel end position
     */
    int e_x, e_y;

    /**
     * Freeman code : encode the change of directions
     */
    std::vector< unsigned short > _directions;
};
```

La technique de Freeman permet de suivre un contour. Le remplissage des informations de direction se fait en se positionnant sur le point de contour le plus à gauche et le plus haut. On en déduit les directions possibles de déplacements. Tant qu'on ne tombe pas sur un pixel « déjà visité », on avance de 2 directions (modulo les 8 directions) dans le tableau, c'est-à-dire en tournant de  $\pi/2$ , sinon on tourne de  $-\pi/2$ . On entretient pour cela une carte des pixels déjà visités.

```
//direction to follow
int dir = 1;
while(true){

    new_x = x + freemanDirections[dir][0];
    new_y = y + freemanDirections[dir][1];

    if ( map.dejaVue(...,...) ){
        dir = ( dir + 2 ) % 8;
    }else{
        dir = ( dir + 7 ) % 8;
    }
}
```

Une fois la détection de tous ces contours effectués nous les stockons dans un vecteur de « Edge ».

Afin de pouvoir visualiser/tester les résultats, nous avons créé une fonction qui prend en paramètre un vecteur de Edge et retourne une matrice binaire correspondant à tous les contours de l'image.

```
static cv::Mat traceEdges(const std::vector<Edge>& listEdges, int height, int width);
```



## 2. Extraction des composantes connexes

Pour pouvoir identifier efficacement chacun des pixels, nous effectuons un « parcours de graphe » qui, à chaque nouveau pixel non précédemment rencontré, va déterminer tous les autres points appartenant à cette même composante connexe.

Pour faire ce parcours de graphe, nous nous déplaçons toujours de gauche à droite et de haut en bas. Ainsi donc, on est certain que lorsque l'on tombera sur un point non visité, ce sera le point d'une nouvelle composante correspondant au pixel le plus à gauche et le plus en haut. Grâce à cette méthode, le choix pour la direction du prochain pixel de cette même composante est réduit à 4.

Une fois le premier pixel de la composante trouvé, nous cherchons itérativement tous les points suivant en testant dans toutes les directions possibles, les unes après les autres. Puisque l'extraction des maxima locaux a permis d'enlever les points côte à côte, on peut donc réduire le nombre de direction possible à 5 (interdisant les 3 directions de retour arrière).

A chaque itération, nous vérifions aussi que nous restons bien dans les limites de la taille de l'image. Si une direction suivie nous amène en dehors de ces limites, nous passons tout de suite à la suivante.

Il y a deux conditions d'arrêt pour cet algorithme itératif :

- Parmi les 5 directions testées, aucune ne nous amène vers un nouveau point appartenant au contour
- On retombe sur le point de départ de notre composante. Dans ce cas, le contour est fermé et tous les points de la composante ont été enregistrés. On peut donc sortir de l'itération.

A chaque fois que l'on rencontre un nouveau point, il est inutile d'y retourner pour la suite de l'algorithme. Pour nous assurer de cela, nous utilisons une matrice binaire initialisé à 0 et de même taille que la matrice de départ. A chaque fois que l'on visite un nouveau point dans la matrice de départ, on met le point correspondant dans la matrice binaire à 1. Ainsi donc, il sera facile de tester si un point a déjà été visité ou non, tant que cette matrice est bien mise à jour.



### 3. Fermeture de contours

Le but de cette partie est de reconstituer des bouts de courbes du contour qui ont été perdu pendant le seuillage ou pendant l'extraction des maxima locaux. Le but est de pouvoir rejoindre entre elles, deux parties de courbes situées à proximité.

#### a. Méthodes

Pour effectuer cela, nous avons utilisé une méthode assez proche de celle vue en cours et utilisant notre modèle de données pour stocker les courbes. Le principe reste le même, faire grandir chaque courbe pour voir si elle en atteint une autre.

Pour cela, on se base essentiellement sur deux éléments principaux :

- Notre modèle de données Vecteur de Edge qui possèdent toutes les courbes, indépendamment les uns des autres. Comme la structure possède des listes de directions, il est donc facile de faire grandir une courbe, en ajoutant simplement une nouvelle direction en début et fin de courbe.
- La matrice `_slope` qui contient la valeur de la pente du gradient pour chacun des points.

#### b. Réflexions

L'idée que l'on a eu, et que l'on a tenté de mettre en place, est d'utiliser la pente du gradient pour nous donner une idée de la direction pour l'expansion de la courbe. Comme la pente du gradient est perpendiculaire à la courbe, il suffit d'ajouter ou d'enlever  $90^\circ$  pour connaître approximativement la direction dans laquelle la courbe va croître.

La valeur de  $\theta$  représentant l'angle du gradient pour chaque pixel est une valeur comprise entre  $[-\pi, \pi]$ . Pour nous ramener à des valeurs que l'on peut facilement utiliser, on fait :

```
Thêta = (int)((Thêta + PI)/(2*PI))*8.0;
```

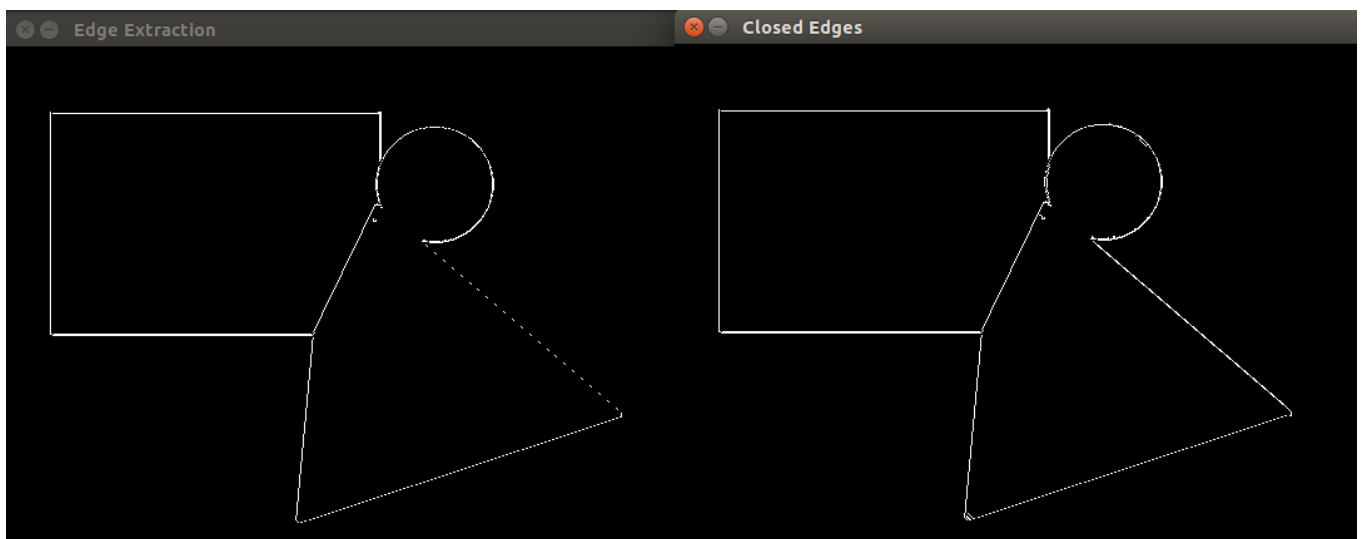
On obtient alors des valeurs comprises entre  $[0, 7]$  qui vont correspondre aux directions du codage de Freeman précédemment utilisé.

L'algorithme prend également un paramètre  $N$  qui va représenter le nombre d'itérations à effectuer avant de s'arrêter. Au bout de  $N$  itérations, si la croissance de la courbe n'a intersecté aucune autre courbe, alors on n'effectue aucun changement. En revanche, si on rencontre une autre courbe, on arrête la croissance et on ajoute les nouvelles directions dans le vecteur.

De plus, comme l'algorithme se base uniquement sur l'expansion des courbes, indépendantes les unes des autres, l'algorithme est donc parallélisable. Ce parallélisme est possible dans notre cas mais peut être utile. En effet, vu la petite quantité de données à traiter, le coût de mise en place du parallélisme serait probablement supérieur au gain lui-même. Mais dans d'autres circonstances, l'algorithme pourrait être appelé en parallèle si les données sont plus importantes.

### c. Résultats

Ci-dessous, un exemple avant et après la fermeture de contours, appliqué sur une image après un seuillage global et une extraction des maxima locaux :



On voit que sur l'image de gauche, une partie des informations a été perdue et que l'on a des pointillés sur ce qui était censé être une droite. Grâce à la fermeture de contour (image de droite), on arrive à relier les pointillés entre eux. Cependant, on voit aussi apparaître de nouveaux segments qui ne sont pas censés être présents.

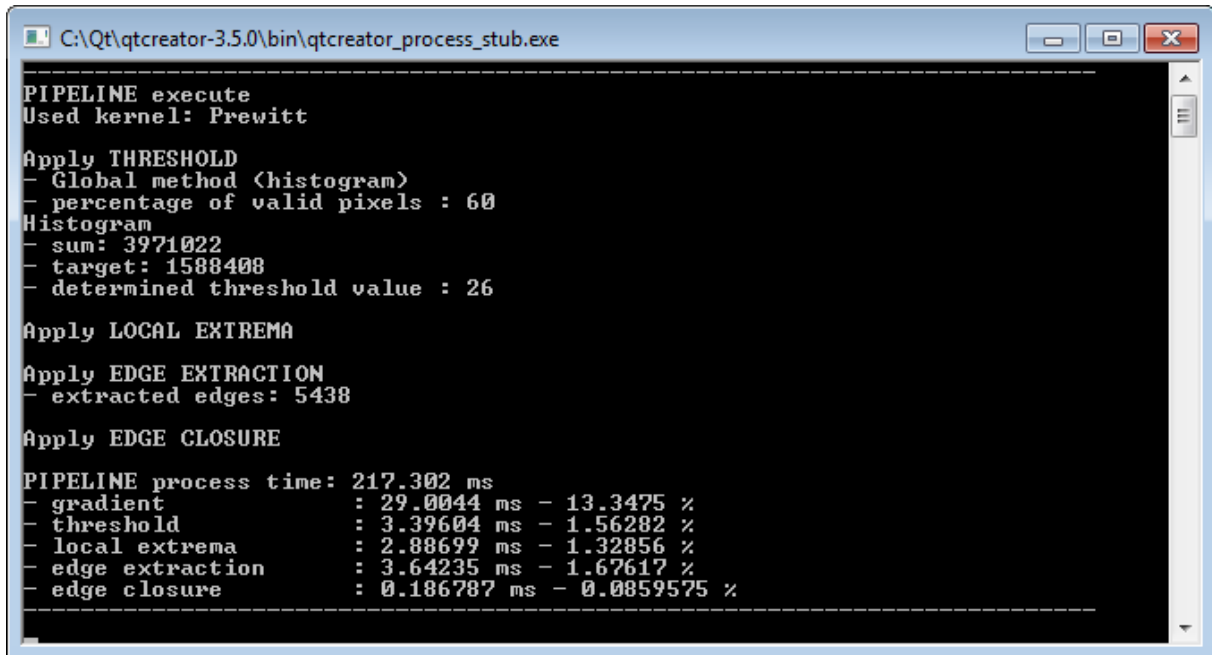
Lors du choix du nombre d'itérations, il faut faire attention à la valeur que l'on choisit. Une valeur trop grande va nous rajouter des segments indésirables, mais une valeur trop petite ne pourra pas du tout relier les courbes entre elles.

### III. Performances

#### 1. Affichage des résultats

L'API fournit une classe spécialisée dans la gestion des performances en utilisant des timers. Le code multi-plateforme gère le cas de Windows et Linux.

Ci-dessous un exemple de résultats obtenus par notre gestionnaire d'évènements :



```
C:\Qt\qtcreator-3.5.0\bin\qtcreator_process_stub.exe

PIPELINE execute
Used kernel: Prewitt

Apply THRESHOLD
- Global method <histogram>
- percentage of valid pixels : 60
Histogram
- sum: 3971022
- target: 1588408
- determined threshold value : 26

Apply LOCAL EXTREMA

Apply EDGE EXTRACTION
- extracted edges: 5438

Apply EDGE CLOSURE

PIPELINE process time: 217.302 ms
- gradient      : 29.0044 ms - 13.3475 %
- threshold    : 3.39604 ms - 1.56282 %
- local extrema : 2.88699 ms - 1.32856 %
- edge extraction : 3.64235 ms - 1.67617 %
- edge closure  : 0.186787 ms - 0.0859575 %
```

Les informations affichées concernent la durée totale du traitement, puis le détail de chaque étape. Le temps est indiqué en milliseconde et en pourcentage du temps total.

Il est à noter que la somme des étapes est inférieure au temps total. En effet, la bibliothèque OpenCV utilise des méthodes de visualisation qui prennent du temps. Suite à une réorganisation, puis refonte du code, nous n'avons pas pu terminer l'extraction des méthodes de visualisation du pipeline de traitement. Cependant, le gestionnaire d'évènements et performances ne mesure que les temps des fonctions, pas de la visualisation et des éventuelles constructions de matrices intermédiaires.

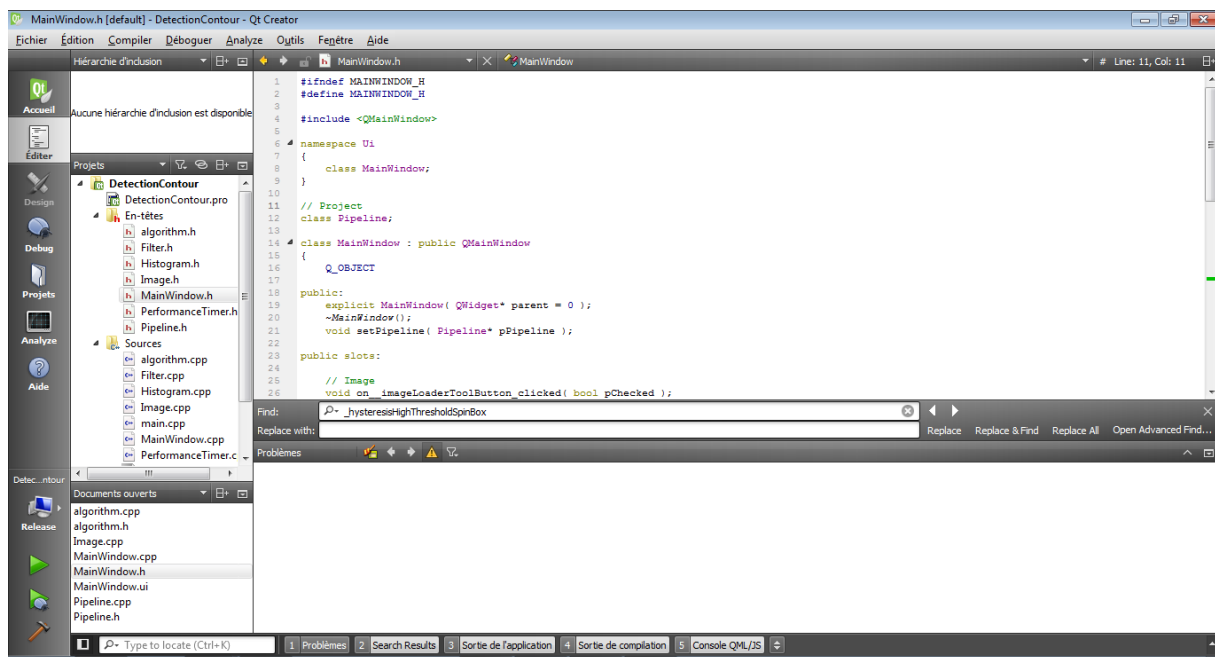
## IV. Programme informatique

### 1. Dépendances

Le programme nécessite l'installation de la librairie OpenCV et de l'environnement de développement qtCreator.

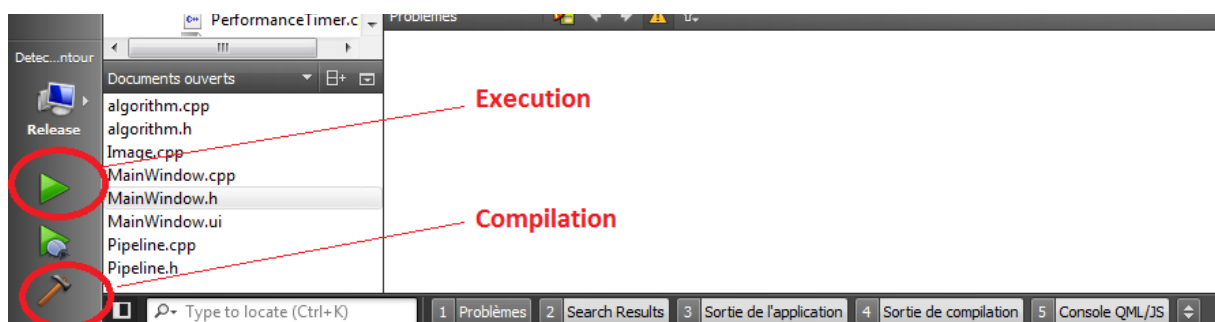
### 1. Environnement de développement

L'environnement de développement qtCreator permet de se libérer de la contrainte des lignes de commandes et des Makefiles. Il suffit d'ouvrir le fichier projet « DetectionContour.pro » avec qtCreator. L'environnement de développement se configure automatiquement :



### 2. Compilation / Execution

Pour compiler puis lancer le programme, il suffit de cliquer sur les deux boutons suivants situés en bas à gauche de l'interface :



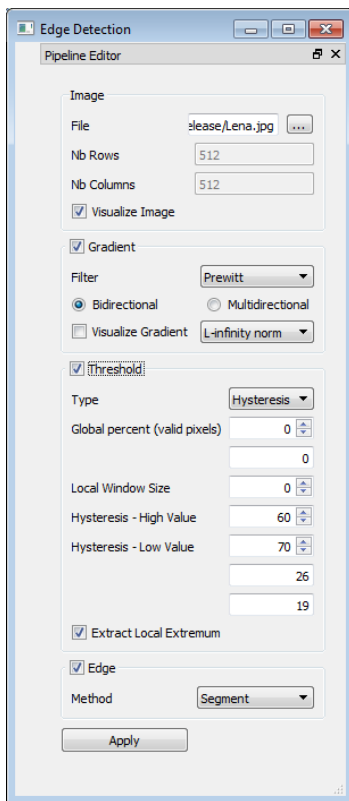
### 3. Utilisation

Tous les paramètres du pipeline de traitement sont paramétrables via un editeur Qt :

Un bouton APPLY situé tout en bas permet de lancer les calculs. On peut le lancer autant de fois qu'on le souhaite en modifiant tous les paramètres, les images, etc...

Les performances sont affichées dans une console.

Les images sont affichées dans des images OpenCV.



NOTES : un soin particulier a été porté sur la documentation du code. Le code a été séparé dans diverses classes C++ .

## Conclusion

Ce TP nous a permis d'implémenter et mettre en place un pipeline de traitement d'images spécialisé pour la détection de contours.

L'application d'opérateurs différentiels générant le gradient sous forme de module et de pente permet d'obtenir une première estimation des contours d'une image. Il est ensuite nécessaire de seuiller les données pour mieux classifier les données en tant que contour. Diverses techniques automatiques de seuillage ont été implémentées : en se basant sur des informations globales à l'image, localement et par hystérésis. Ces traitements génèrent des contours plus ou moins épais, dédoublés ou avec des trous. La sélection des extrema locaux permet d'affiner les contours jusqu'à une épaisseur de 1 pixel. Il est alors possible d'extraire les contours obtenus dans une structure de données à l'aide d'un codage de Freeman, une chaîne de valeurs encodant les différents changements de direction le long d'un contour donné. L'étape finale consiste à relier et fermer les contours.

Il n'existe pas de méthodes génériques pour détecter les contours. Les résultats dépendent des images et du choix des opérateurs de filtrage du gradient, des valeurs de seuillage et algorithmes utilisés ; notamment pour la fermeture. Il faut donc opérer au cas par cas.

Ce TP a aussi été l'occasion d'apprendre à maîtriser et utiliser la librairie OpenCV très pratique pour gérer la lecture, la sauvegarde, la visualisation et le traitement matriciel.