



## Détection de contours par approches du premier ordre

Kasperek Gautier  
Sparrow Alice

Le but de cette première partie est d'examiner l'adaptation d'un seuillage global de la norme d'un gradient à la détection des contours dans une image.

### 1. Seuillage de la norme d'un gradient

#### Représentation de la norme d'un gradient


#### Calcul de la norme d'un gradient par convolution



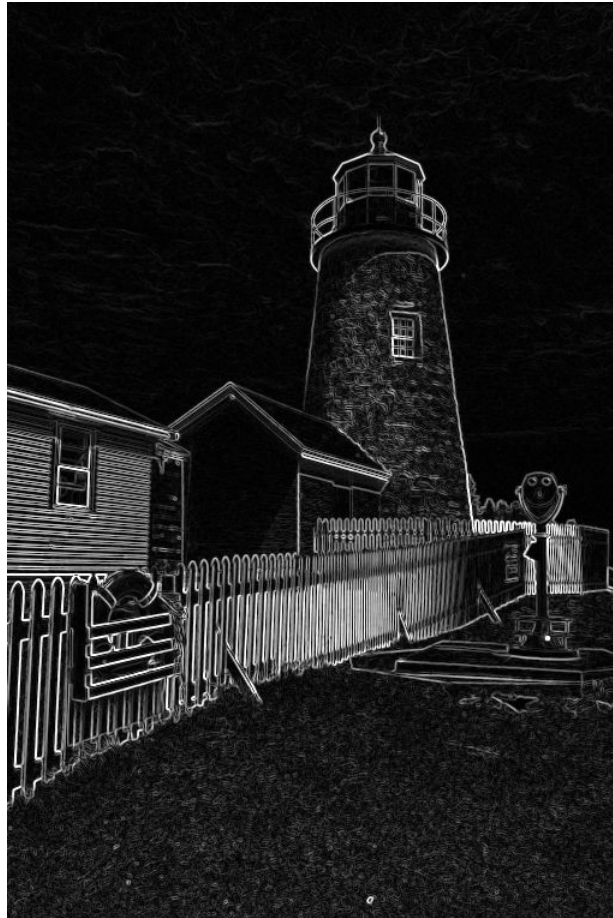
Figure 1 - Image "lighthouse\_8bits.png"

Nous avons calculé la norme du gradient de la figure 1 par convolution avec les masques de Sobel suivant :

$$\vec{\nabla} f(x, y) = \begin{pmatrix} (I * H_x^s)(x, y) \\ (I * H_y^s)(x, y) \end{pmatrix}, \text{ avec } H_x^s = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ et } H_y^s = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Nous obtenons alors l'image suivante. 

**A compléter**



***A commenter***

Figure 2 - Norme du gradient de l'image "lighthouse\_8bits.png"

Comparaison avec le résultat fourni par ImageJ.

Les images utilisées doivent avoir une profondeur de 32 bit car cela permet de coder des valeurs négatives. En effet les dérivées peuvent être négatives.

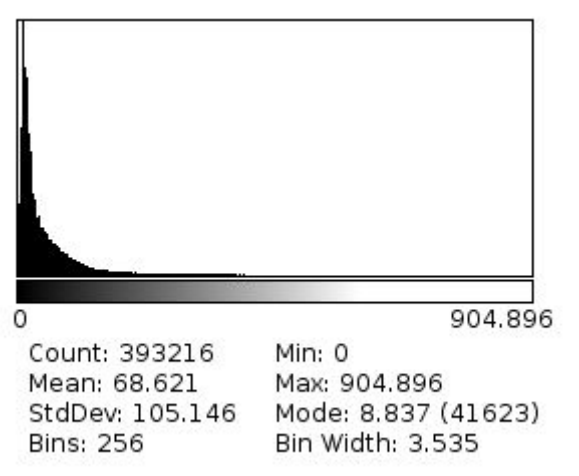


Figure 3 - Histogramme de la norme du gradient

On remarque sur la figure 3 que la norme du gradient a pour valeur maximale 904 et pour valeur minimale 0.

**A expliquer**

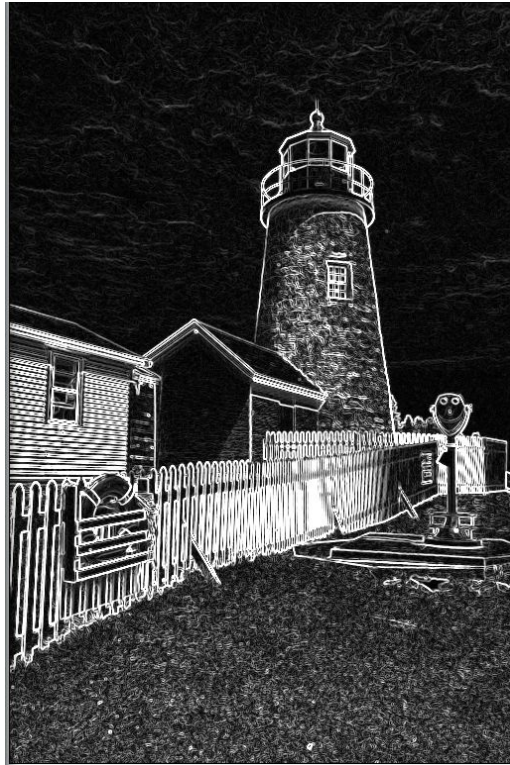


Figure 4 - Norme du gradient obtenue par le calcul intégré dans ImageJ

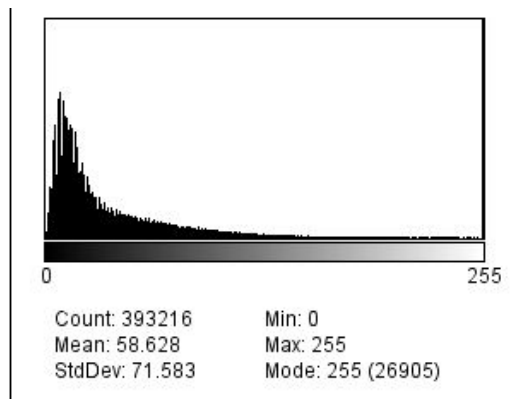


Figure 5 - Histogramme de la norme du gradient de ImageJ

On remarque une différence entre les deux images de la norme du gradient. En effet les contours de la figure 4 apparaissent beaucoup plus lumineux. On peut voir sur l'histogramme de la figure 5 que les pixels ont des valeurs comprises entre 0 et 255. Pour obtenir un résultat similaire sur l'image issue de notre calcul il faut faire une saturation.



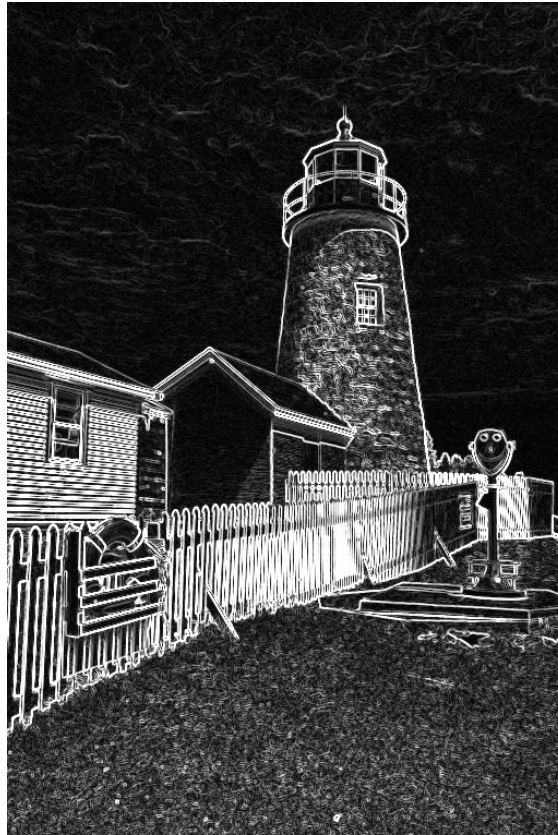


Figure 6 - Norme du gradient de l'image "lighthouse\_8bits.png" saturée

Après avoir appliqué la saturation avec l'instruction *Process/Math/Max(255)*, nous avons transformé l'image correspondante en niveaux sur 8 bits. Pour comparer l'image issue de notre calcul (saturée et reportée sur 8 bits) et celle calculée par ImageJ nous procédons à une soustraction entre ces deux dernières.

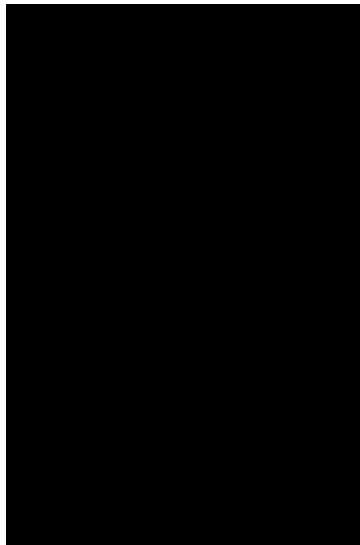


Figure 7 - Soustraction entre la norme du gradient de ImageJ et la nôtre.

On remarque que la soustraction des deux images produit une image noire. Cela signifie que les deux images sont **identiques**. En faisant ces manipulations, nous retrouvons l'image calculée par ImageJ.



```
// Calcul de la norme du gradient par masque de Sobel
//
requires("1.41i");    // requis par substring(string, index)
setBatchMode(true);    // false pour déboguer

/***** Création des images *****/
sourceImage = getImageID();
filename = getTitle();
extension = "";
if (lastIndexOf(filename, ".") > 0) {
    extension = substring(filename, lastIndexOf(filename, "."));
    filename = substring(filename, 0, lastIndexOf(filename, "."));
}
filenameDerX = filename+"_der_x"+extension; // images des
filenameDerY = filename+"_der_y"+extension; // dérivées
run("Duplicate...", "title="+filenameDerX);
run("32-bit");    // conversion en Float avant calcul des dérivées !!
run("Duplicate...", "title="+filenameDerY);
run("32-bit");

/***** Calcul de la norme du gradient *****/

// récupération de la taille de l'image
w = getWidth();
h = getHeight();
// Calculs pour chaque pixel
run("Convolve...", "text1=[-1 -2 -1\n 0 0 0\n1 2 1\n] normalize");

selectWindow(filenameDerX);
run("Convolve...", "text1=[-1 0 1\n-2 0 2\n-1 0 1\n] normalize");

run("Square");
selectWindow(filenameDerY);
run("Square");
imageCalculator("Add create 32-bit", filenameDerY, filenameDerX);
selectWindow("Result of "+filenameDerY);
run("Square Root");

setBatchMode("exit and display");
```



### Seuillage de la norme du gradient précédemment calculée

On utilise *Image/Adjust/Threshold...* pour seuiller la norme du gradient calculée, il n'est pas possible de trouver ainsi un seuil global pour l'image qui mette en évidence les pixels contours de manière satisfaisante.



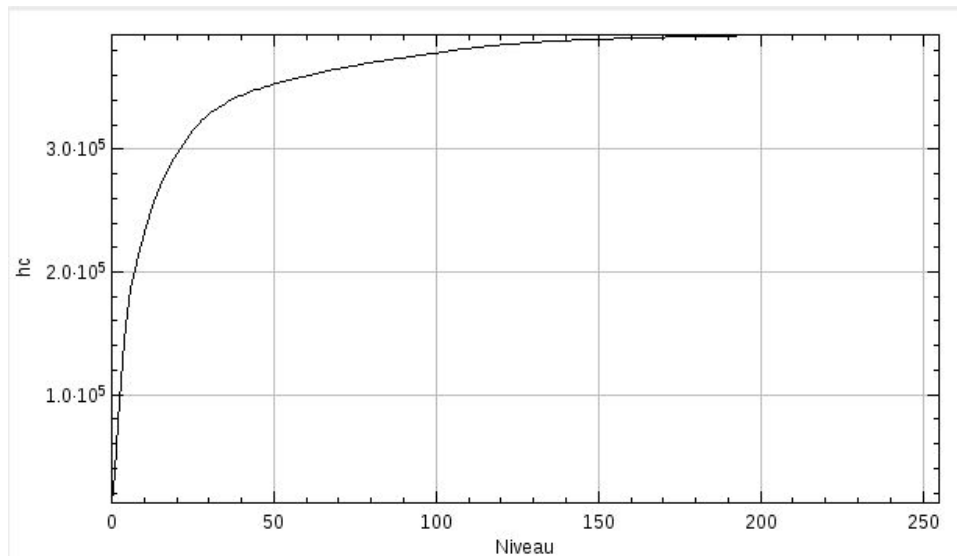
Figure 8 - Norme du gradient calculée et seuillée

En effet lorsqu'on applique des réglages sur le seuil, on se rend compte que des points points du sol et de l'intérieur du phare sont considérés comme des points de contours. Nous avons testé plusieurs valeurs et celles-ci ne permettent pas de faire disparaître ces points parasites. Si on augmente le seuil, les points parasites s'effacent en partie cependant un grand nombre des points de contours disparaissent également. Nous trouvons comme valeur optimale 191 pour ce seuil.

Nous allons déterminer ce seuil de manière semi-automatique grâce à l'histogramme cumulé de l'image de la norme du gradient.







*A revoir*

Figure 9 - Histogramme cumulé de la norme du gradient

Grâce à l'histogramme cumulé nous pouvons trouver à partir de quelle valeur les pixels représente 20% de l'image. Cela nous donnera une approximation d'un seuil permettant de mettre en évidence les contours de l'image.

$$512 \times 758 = 388\,096$$

$$388\,096 - (388\,096 \times 0.2) = 310\,476$$

On reporte cette valeur sur l'histogramme et on obtient graphiquement un seuil à 26. On obtient le résultat suivant.



*A revoir*

Figure 10 - Image seuillée avec le seuil obtenu graphiquement

On peut remarquer que le résultat n'est pas satisfaisant pour extraire les contours de l'image.

## 2. Détection des maxima locaux de la norme d'un gradient

### Calcul de la direction du gradient

Le but de cette partie est de réaliser la représentation des maxima locaux de la norme du gradient donnée par l'application des masques de Sobel. Pour cela nous allons dans un premier temps calculer l'angle du gradient pour chaque pixel. Nous allons pour chaque étape expliquer la partie de la macro correspondante, le but n'étant pas de faire une revue de code mais d'expliquer comment chaque résultat est obtenu. La macro complète sera disponible en annexe de ce rapport.

```
newImage("direction", "32-bit black", 512, 768, 1);
direction = getImageID();

for(i = 0; i < w; i++){
    for(j = 0; j < h; j++){
        selectImage(dupX);
        pixX = getPixel(i,j);
        selectImage(dupY);
        pixY = getPixel(i,j);
        angle = atan2(pixX,pixY)*(180/PI);
        selectImage(direction);
        setPixel(i,j,angle);
    }
}
```

Le code ci-dessus **calcul** la direction du gradient pour chaque pixel de l'image. Les images *dupX* et *dupY* sont respectivement les images obtenues par application des masques de Sobel  $H_x$  et  $H_y$  déjà présentés dans la première partie. L'image *direction* est l'image résultant dans laquelle la valeur de la direction sera enregistré. Pour chaque pixel on calcule la direction du gradient par la relation

$$\text{direction}(x,y) = \arctan(\text{dupX}(x,y) / \text{dupY}(x,y))$$

La valeur de l'angle est reporté en degré pour faciliter les calculs de la partie suivante. On obtient ainsi l'image *direction* ci-dessous. On peut remarquer que l'on obtient une image qui pourra être **exploité**, en effet d'autres opérations seront à effectuer mais on peut d'**hors** et déjà discerner les pixels de contour de l'image.





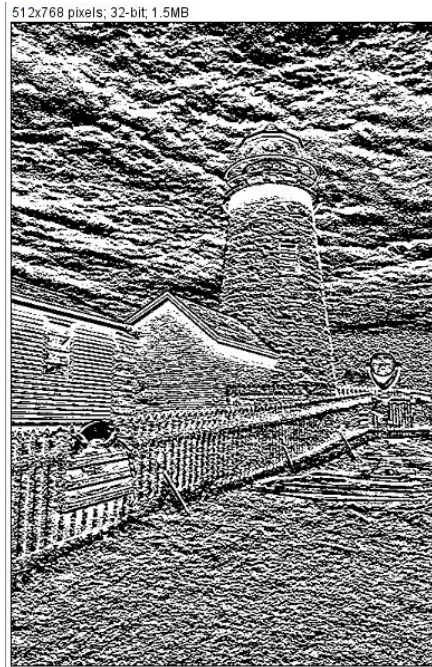


Figure 11 - Direction du gradient de l'image

De plus, lorsque l'on analyse plus en détail les valeurs de la direction on remarque une propriété intéressante. En effet, on peut constater que la direction du gradient suit les contours sur plusieurs pixels consécutifs. Par exemple, si l'on inspecte les pixels formant le toit de la maison devant le phare, on remarque que de nombreux pixels de valeurs  $55^\circ$  sont alignés sur un axe à  $55^\circ$  par rapport à la verticale. De plus, les diagonales de valeurs 0 permettent de facilement distinguer les différents contour.

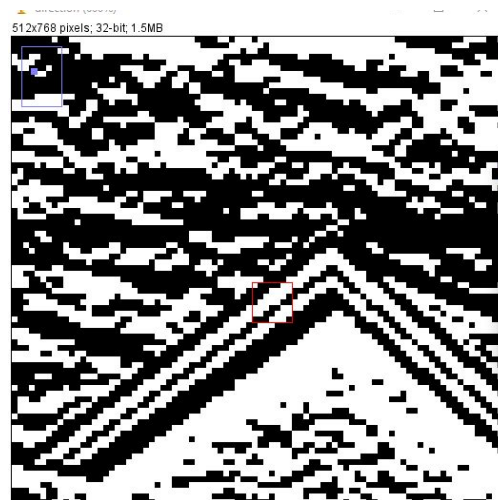


Figure 12 - Zoom sur la direction du gradient

*A revoir*

Prefs	172	173	174	175	176	177	178
327	90.0000	0.0000	0.0000	0.0000	0.0000	56.5700	55.4599
328	0.0000	0.0000	0.0000	82.4054	59.2026	56.1920	54.0507
329	0.0000	0.0000	56.7934	55.1140	57.4543	59.6567	0.0000
330	0.0000	56.1649	57.0738	54.1073	47.7263	0.0000	0.0000
331	53.7462	54.0395	57.7453	0.0000	0.0000	45.0000	56.9761
332	55.8987	59.3493	0.0000	0.0000	68.8059	57.3625	54.0395
333	0.0000	0.0000	0.0000	51.7662	60.8324	69.2277	0.0000

### Détection des maxima locaux

L'utilisation d'un seuil sur la norme du gradient pose un problème. En effet avec un seuil de détection unique pour toute l'image risque d'engendrer la non détection de points de contours en zone de faible contraste ou encore de sélectionner des points parasites dans des zones bruitées. Pour remédier à ce problème, nous allons utiliser la méthode des maxima locaux. Cela consiste à sélectionner les points auxquels la norme du gradient est maximale dans la direction du gradient.



**A expliquer**

```
grad1 = 0;
grad2 = 0;
newImage("contour", "32-bit black", w, h, 1);
contour = getImageID();

for(i = 0; i < w; i++){
    for(j = 0; j < h; j++){

        selectImage(direction);
        angle = round(getPixel(i,j)/45)*45;
        selectImage(norme);
        pixel = getPixel(i,j);
        if(angle ==180 || angle ==-180){
            grad1 = getPixel(i,j + 1);
            grad2 = getPixel(i,j - 1);
        }
        else if(angle ==135 || angle ==-45){
            grad1 = getPixel(i-1,j + 1);
            grad2 = getPixel(i+1,j - 1);
        }
        else if(angle ==90 || angle ==-90){
            grad1 = getPixel(i-1,j);
            grad2 = getPixel(i+1,j);
        }
    }
}
```



**A revoir**

```

else if(angle ==45 || angle ==-135){
    grad1 = getPixel(i-1,j-1);
    grad2 = getPixel(i+1,j + 1);
}
else{
    grad1 = getPixel(i,j - 1);
    grad2 = getPixel(i,j + 1);
}

selectImage(contour);
if(((pixel< grad1) & (pixel < grad2)) ) pixel = 0;
setPixel(i,j,pixel);
}
}

```

**A revoir**

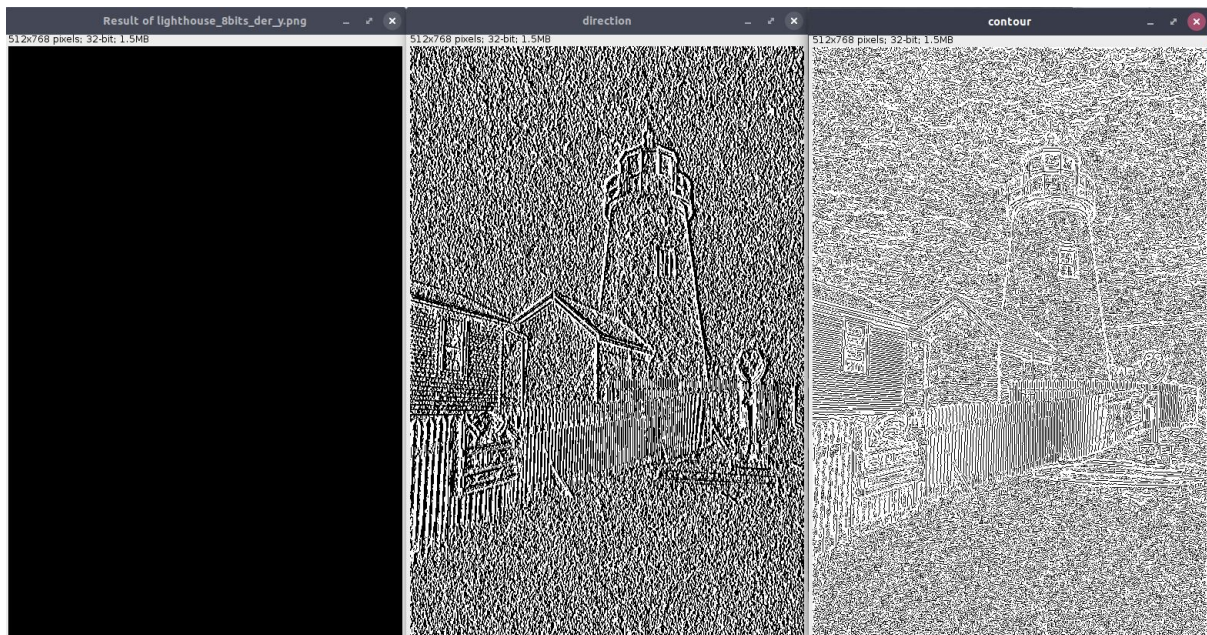


Figure 13 - A gauche le gradient, au centre la direction du gradient et à gauche les contours

Le threshold automatique en fin de macro ne change pas les valeurs mais seulement leur représentation afin d'avoir un résultat plus visuel.

```

selectImage(contour);
setAutoThreshold("Default");
resetThreshold();

```

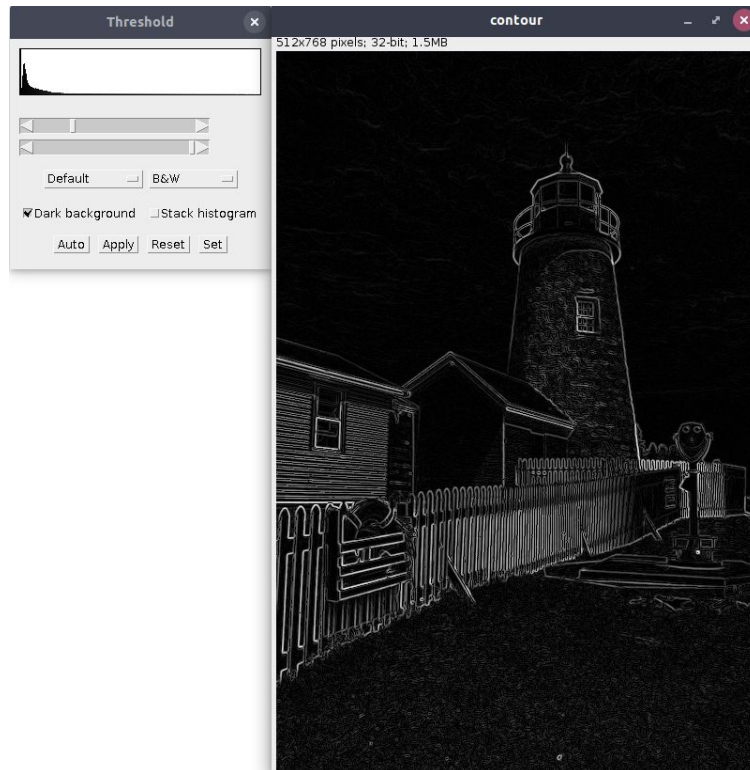


Figure 14 - Contours obtenus par détection des maxima locaux

*A revoir*

### Seuillage des maxima locaux par hystérésis

#### Analyse d'un plugin existant

Nous allons maintenant chercher à seuiller l'image pour aboutir aux pixels de contours. Le plugin fourni dans le sujet du tp implémente le seuillage par hystérésis proposé par Canny. Cette méthode utilise deux seuils afin d'améliorer la flexibilité du seuillage.



Les figures 15 et 16 qui suivent sont des exemples de résultat de l'utilisation du plugin. Pour la figure 15, on utilise la valeur 30 pour le seuil haut et le seuil bas. On constate qu'une grande partie des points de contours sont visibles mais il en manque certains. On peut par exemple le voir sur le contour gauche du phare. Sur la seconde figure, on constate que les contours sont bien visibles cependant, de nombreux points parasites apparaissent notamment sur le sol. Pour cette figure nous avons choisis 17 pour le seuil bas et 46 pour le seuil haut.





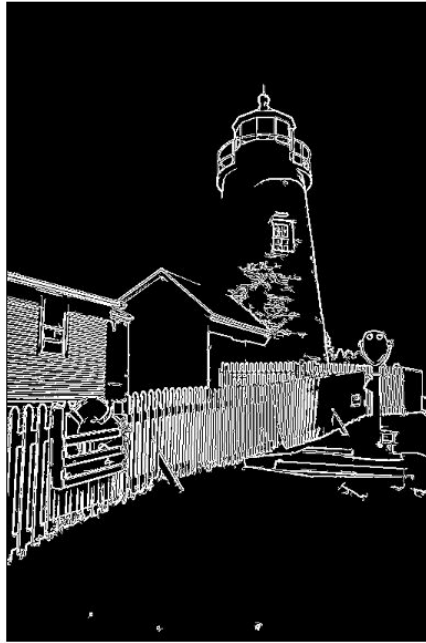
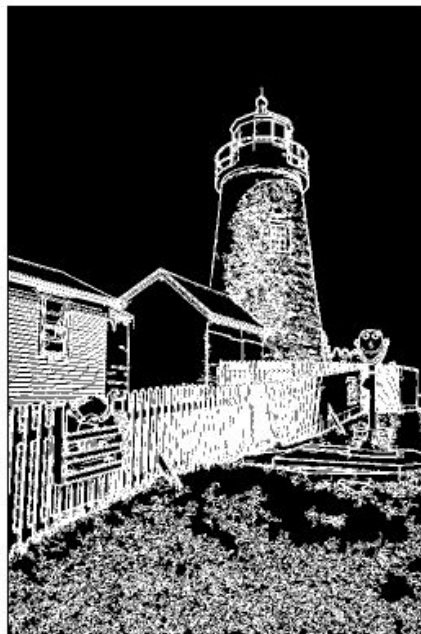


Figure 15 - Seuillage par hystérésis seuil bas = 30 et seuil haut = 30



*A revoir*

Figure 16 - Seuillage par hystérésis seuil bas = 17 et seuil haut = 46

Nous avons commenté le plugin fourni afin de mieux le comprendre.

```
public ByteProcessor hystIter(ImageProcessor imNormeG, int
seuilBas, int seuilHaut) {
    // Taille de l'image des maxima locaux de la norme du
```



gradient

```
        int width = imNormeG.getWidth();
        int height = imNormeG.getHeight();

        // Nouvelle image binaire
        ByteProcessor maxLoc = new ByteProcessor(width,height);
        // Tableau des pixels contours
        List<int[]> highpixels = new ArrayList<int[]>();

        // On parcourt l'image
        for (int y=0; y<height; y++) {
            for (int x=0; x<width; x++) {
                // On récupère le gradient g aux
                // coordonnées x y de l'image d'entrées
                int g = imNormeG.getPixel(x, y)&0xFF;
                // Si g est inférieur au seuil
                // bas highpixel[x,y] <= vide (0)
                if (g<seuilBas) continue;

                // Si g est supérieur au seuil
                // haut hihpixel[x,y] = 255 (c'est un contour)
                if (g>seuilHaut) {
                    // Pixel de l'image
                    // résultat en blanc
                    maxLoc.set(x,y,255);
                    highpixels.add(new int[]{x,y});
                    continue;
                }

                // Si g compris entre le seuil
                // haut et bas highppixel[x,y] est un contour si à côté d'un contour
                maxLoc.set(x,y,128);
            }
        }

        int[] dx8 = new int[] {-1, 0, 1,-1, 1,-1, 0, 1};
        int[] dy8 = new int[] {-1,-1,-1, 0, 0, 1, 1, 1};

        // tant que le tableau des contours n'est pas vide
        while(!highpixels.isEmpty()) {
            // Nouveau tab de contours
            List<int[]> newhighpixels = new ArrayList<int[]>();
            // Pour chaque pixel contour (coordonnées x
            // y)
            for(int[] pixel : highpixels) {
                int x=pixel[0], y=pixel[1];
```





```

// Pour chaque coordonnées
voisines (si elles sont dans l'image)
    for(int k=0;k<8;k++) {
        int xk=x+dx8[k], yk=y+dy8[k];
        if (xk<0 || xk>=width) continue;
        if (yk<0 || yk>=height) continue;
        // Si le voisin est
        enregistré comme un gradient compris entre les deux seuils
        // On le considère
        un contour car le pixel courant est un contour connexe
        // (Propagation des
        contours)

        if (maxLoc.get(xk, yk)==128) {
            maxLoc.set(xk, yk, 255);
            newhighpixels.add(new int[]{xk, yk});
        }
    }

    highpixels = newhighpixels;
}

// Pour chaque pixel :
// Si le pixel est à 128 (gradient compris entre les
deux seuils) mais pas voisin d'un contour
// Alors on le mets à 0
for (int y=0; y<height; y++) {
    for (int x=0; x<width; x++) {
        if (maxLoc.get(x, y)!=255) maxLoc.set(x,y,0);
    }
}
return maxLoc;
}

```



### Implémentation récursive du seuillage par hystérésis

Nous n'avons pas réussi à implémenter ce plugin de façon récursive, cependant voici comment nous aurions fait :

Pour tous les pixels de coordonnées (x,y)

Si le pixel correspondant dans l'image de contour n'est pas initialisé

Continuer

Sinon

~~Si la norme du gradient de ce pixel < seuil~~

~~Alors le pixel de l'image de contour (x,y) est affecté à 0~~

~~Sinon~~ si la norme du gradient > seuil haut

Alors le pixel de l'image de contour (x,y) est affecté à 255

~~Sinon~~

Pour tous les voisins :

Propager -> appel récursif à la fonction.

***A préciser***

Annexe :

```
// Calcul de la norme du gradient par masque de Sobel
//
requires("1.41i");      // requis par substring(string, index)
setBatchMode(true);     // false pour déboguer

/***** Création des images *****/
sourceImage = getImageID();
filename = getTitle();
extension = "";
if (lastIndexOf(filename, ".") > 0) {
    extension = substring(filename, lastIndexOf(filename, "."));
    filename = substring(filename, 0, lastIndexOf(filename, "."));
}
filenameDerX = filename+"_der_x"+extension; // images des
filenameDerY = filename+"_der_y"+extension; // dérivées
run("Duplicate...", "title="+filenameDerX);
run("32-bit");      // conversion en Float avant calcul des dérivées !!
run("Duplicate...", "title="+filenameDerY);
run("32-bit");

/***** Calcul de la norme du gradient *****/

// récupération de la taille de l'image
w = getWidth();
h = getHeight();
// Calculs pour chaque pixel
run("Convolve...", "text1=[-1 -2 -1\n 0 0 0\n1 2 1\n] normalize");

selectWindow(filenameDerX);
run("Convolve...", "text1=[-1 0 1\n-2 0 2\n-1 0 1\n] normalize");

run("Square");
selectWindow(filenameDerY);
run("Square");
imageCalculator("Add create 32-bit", filenameDerY, filenameDerX);
run("Square Root");
//run("Max...", "value=255");
norme = getImageID();

selectWindow(filenameDerY);
run("Duplicate...", "title=dupY");
```

```

dupY = getImageID();
run("Convolve...", "text1=[-1 -2 -1\n 0 0 0\n1 2 1\n] normalize");

selectWindow(filenameDerX);
run("Duplicate...", "title=dupX");
dupX = getImageID();
run("Convolve...", "text1=[-1 0 1\n-2 0 2\n-1 0 1\n] normalize");

newImage("direction", "32-bit black", 512, 768, 1);
direction = getImageID();

for(i = 0; i < w; i++){
    for(j = 0; j < h; j++){
        selectImage(dupX);
        pixX = getPixel(i,j);
        selectImage(dupY);
        pixY = getPixel(i,j);
        angle = atan2(pixX,pixY)*(180/PI);
        //if(angle >= 180) angle = 180;
        //if(angle < 0) angle = 180;
        selectImage(direction);
        setPixel(i,j,angle);
    }
}
grad1 = 0;
grad2 = 0;
newImage("contour", "32-bit black", w, h, 1);
contour = getImageID();

for(i = 0; i < w; i++){
    for(j = 0; j < h; j++){

        selectImage(direction);
        angle = round(getPixel(i,j)/45)*45;
        selectImage(norme);
        pixel = getPixel(i,j);
        if(angle ==180 || angle ==-180){
            grad1 = getPixel(i,j + 1);
            grad2 = getPixel(i,j - 1);
        }
        else if(angle ==135 || angle ==-45){
            grad1 = getPixel(i-1,j + 1);
            grad2 = getPixel(i+1,j - 1);
        }
        else if(angle ==90 || angle ==-90){

```

```

        grad1 = getPixel(i-1,j);
        grad2 = getPixel(i+1,j);
    }
    else if(angle ==45 || angle ==-135){
        grad1 = getPixel(i-1,j-1);
        grad2 = getPixel(i+1,j + 1);
    }
    else{
        grad1 = getPixel(i,j - 1);
        grad2 = getPixel(i,j + 1);
    }

    selectImage(contour);
    if(((pixel< grad1) & (pixel < grad2)) ) pixel = 0;
    setPixel(i,j,pixel);
}
}

selectWindow(filenameDerX);
close();
selectWindow(filenameDerY);
close();
selectImage(dupY);
close();
selectImage(dupX);
close();

selectImage(contour);
setAutoThreshold("Default");
resetThreshold();

setBatchMode("exit and display");

```