Semaine 9 : Détection de contours par approches du premier ordre

TP9

Maxime CATTEAU
Léane TEXIER

1ère partie : Seuillage de la norme d'un gradient

Compléter la macro donnée de façon à calculer une approximation des dérivées partielles de la fonction image dans chaque direction principale (x,y) par convolution avec les masques de Sobel donnés.

La détection de contours par approches du premier ordre passe tout d'abord par le seuillage de la norme d'un gradient. Afin de se mettre dans les meilleurs conditions pour la suite du TP, il est nécessaire de transformer l'image source en 32-bits afin de réaliser les convolutions nécessaires, notamment avec les filtres de Sobel qui peuvent données des résultats négatifs et flottants. L'image source utilisée est une image 8-bits, il s'agit de '*lighthouse_8bits.png'.*

Image source 'lightouse_8bits.png utilisée pour la suite du TP

Dans le but de déterminer les dérivées partielles suivant les direction principales *x* et *y*, nous décidons de dupliquer l'image originale (transformée en 32-bits au préalable) deux fois. L'une de ces duplications contiendra par la suite la dérivée partielle en *x* ('*lightouse_8bits_der_x.png*') et l'autre celle de la dérivée partielle en *y* ('*lightouse_8bits_der_y.png*'). La manipulation consiste à appliquer par convolution les filtres de Sobel.

Filtre de Sobel pour dériver suivant *x* :

-1	0	1
-2	0	2
-1	0	1

Filtre de Sobel pour dériver suivant *y* :

-1	-2	-1
0	0	0
1	2	1

En terme de code, cette opération peut être définie par les quelques lignes ci-dessous :

```
run("32-bit");

filenameDerX = filename+"_der_x"+extension; // images des
filenameDerY = filename+"_der_y"+extension; // dérivées
run("Duplicate...", "title="+filenameDerX);
run("32-bit");    // conversion avant calcul des dérivées !!

// Application du filtre de Sobel en X
run("Convolve...", "text1=[-1 0 1\n-2 0 2\n-1 0 1\n] normalize");

selectImage(sourceImage);
run("Duplicate...", "title="+filenameDerY);
run("32-bit");    // conversion avant calcul des dérivées !!

// Application du filtre de Sobel en Y
run("Convolve...", "text1=[-1 -2 -1\n0 0 0\n1 2 1\n] normalize");
```

Code correspondant au calcul des dérivées partielles suivant les directions principales (x et y)

L'exécution de ce code nous donne le résultat suivant :

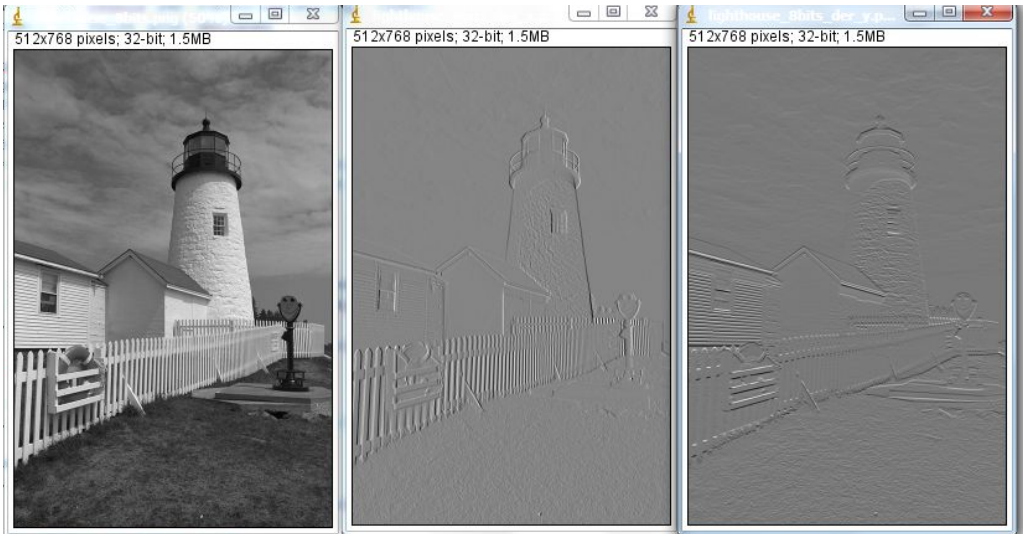


Image originale 'lightouse_8bits.png' en 32-bits, 'lightouse_8bits_der_x et 'lightouse_8bits_der_y'

Nous constatons que nous obtenons d'une part la dérivée partielle de l'image suivant *x* qui révèle les contours verticaux et d'autre part la dérivée partielle de l'image suivant *y* qui révèle les contours horizontaux. Ainsi, grâce à ces deux images, nous pourrons calculer la norme du gradient de l'image '*lightouse_8bits.png*'.

Compléter la macro donnée de façon à calculer la norme du gradient dans une nouvelle image. Cette image peut être simplement créée par duplication de l'image de ∂f / ∂x (ci-dessus nommée **filenameDerX**, de profondeur 32 bits), puis modifiée en chaque pixel grâce à la fonction **setPixel**.

L'étape qui suit le calcul des dérivées partielles dans les directions principales *x* et *y* consiste à évaluer la norme du gradient. Pour cela, nous dupliquons l'image afin de stocker le résultat sous le nom '*norme_du_gradient.png*'. Il s'agira d'une image en 32-bits afin de pouvoir effectuer des calculs plus tard. D'après le cours, la formule permettant le calcul de la norme du gradient est :

$$\|\vec{\nabla} f(x,y)\| = \sqrt{\left(\frac{\partial f}{\partial x}(x,y)\right)^2 + \left(\frac{\partial f}{\partial y}(x,y)\right)^2}$$

Formule permettant de calculer la norme du gradient d'un pixel de coordonnées (x, y)

Après avoir mis en place cette formule dans le code, il suffit de modifier chaque pixel de la nouvelle image '*norme_du_gradient.png*' afin de prendre en compte la nouvelle valeur calculer. Voici le code correspondant à cette étape :

```
w = getWidth();
h = getHeight();
selectImage(filenameDerX);
name_grad_norm = "norme_du_gradient"+extension;
run("Duplicate...", "title="+name_grad_norm);
run("32-bit");

for(i = 0; i < h; i++){
        for(j = 0; j < w; j++){
                selectImage(filenameDerX);
                  p_x = getPixel(j, i);          // Valeur du pixel sur l'image de la dérivée en x
                selectImage(filenameDerY);
                p_y = getPixel(j, i);          // Valeur du pixel sur l'image de la dérivée en y
                p = sqrt(p_x*p_x + p_y * p_y);   // Application de la formule
                selectImage(name_grad_norm);
                setPixel(j, i, p);
        }
}
```

Morceau de l'algorithme permettant de récupérer la norme du gradient de l'image 'lightouse_8bits.png'

L'exécution de ce code nous donne le résultat suivant :

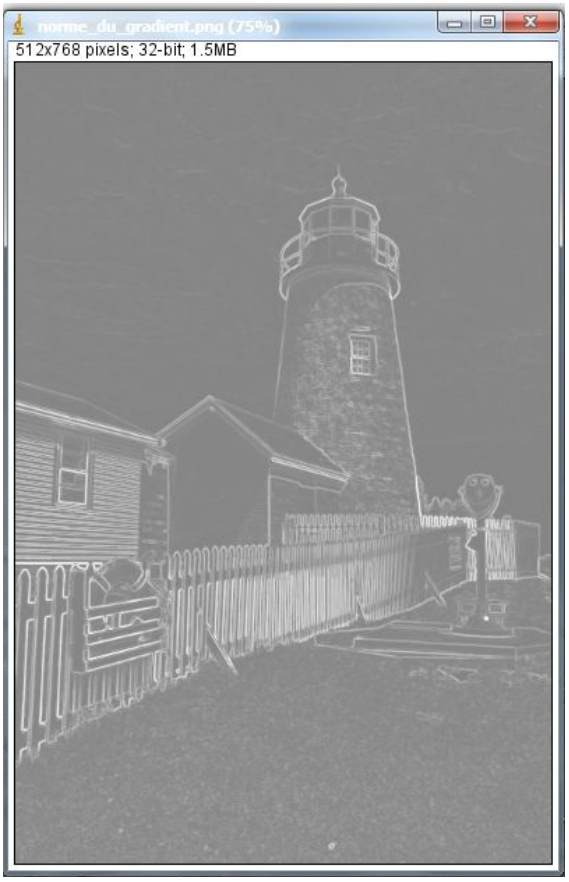
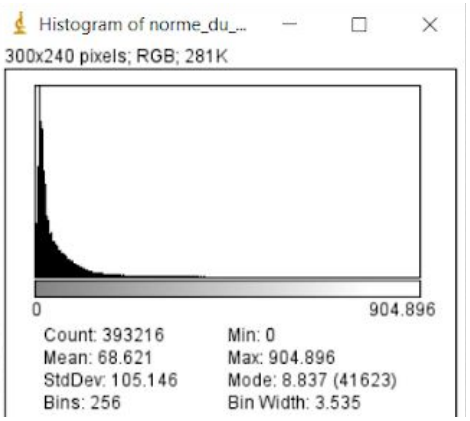


Image 'norme_du_gradient.png' correspondant à la norme du gradient de l'image source 'lighthouse_8bits.png'

Pourquoi les images utilisées ici doivent-elles avoir une profondeur de 32 bits ? Quelle est la valeur minimale et la valeur maximale de la norme du gradient (utiliser le menu **Analyse/Histogram** ou **Analyse/Measure**) et comment les expliquez-vous ?

Les images utilisées doivent avoir une profondeur de 32-bits. En effet, cela nous permet alors d'avoir des valeurs négatives et flottantes. Cela nous est très utile étant donné que l'application des masques de Sobel nous donne des valeurs aussi bien positives que négatives. Nous rappelons que sur l'image de base qui est en 8-bits, les valeurs des pixels s'étendent entre 0 (minimum) et 255 (maximum) ce qui n'est pas suffisant pour avoir les bonnes valeurs de la norme. Analysons la valeur minimale et la valeur maximale de la norme du gradient. Pour cela, analysons son histogramme.



Histogramme de la norme du gradient

Grâce à l'histogramme, on peut remarquer que la valeur minimale est de 0 et la valeur maximale est de 904,896. (Cela confirme bien l'utilité de convertir nos images en 32-bits). Grâce à l'analyse des pixels, on remarque que plus la valeur est proche de 0 (valeur minimale), plus le contour est faible. Au contraire, plus elle est proche du maximum, plus le contour est fort.

Comparer cette image avec le résultat obtenu en appliquant sur l'image source la fonctionnalité de calcul de la norme du gradient intégrée à ImageJ (menu **Process/Find Edges**). Quelle transformation faut-il opérer sur l'image issue de votre calcul pour obtenir un résultat similaire ? Appliquer effectivement cette transformation grâce au menu Image/Adjust/Brightness/Contrast ..., transformer l'image correspondante en niveaux sur 8 bits, puis comparer numériquement (grâce au menu **Process/Image Calculator ...**) les deux images ainsi obtenues pour la norme du gradient.

Nous avons, par la suite, appliqué la commande '**Find Edges**' à notre image source afin de la comparer avec l'image obtenue grâce à notre macro. Le résultat obtenu est une

image sur 8-bits étant donné que l'image source est une image 8-bits. Voici le résultat obtenu :

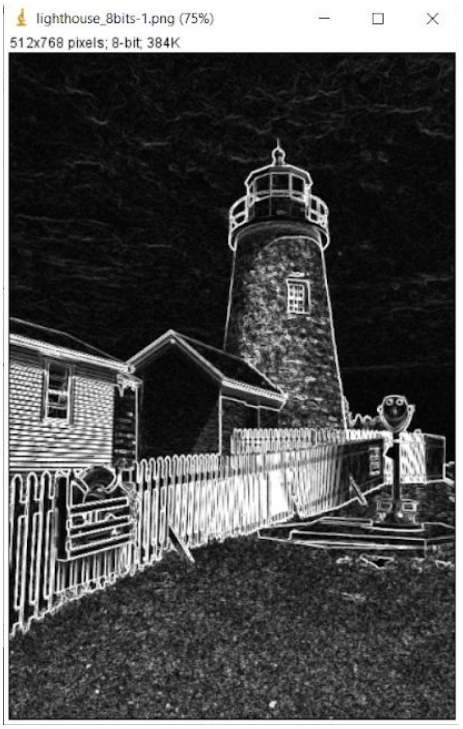


Image source 'lighthouse_8bits.png' avec application de la commande 'Find edges'

Afin d'obtenir un résultat similaire il faut modifier le contraste de notre image issue de notre calcul afin que les valeurs sont comprises entre 0 et 255 comme c'est le cas pour l'image obtenu avec la commande '**Find Edges**'.

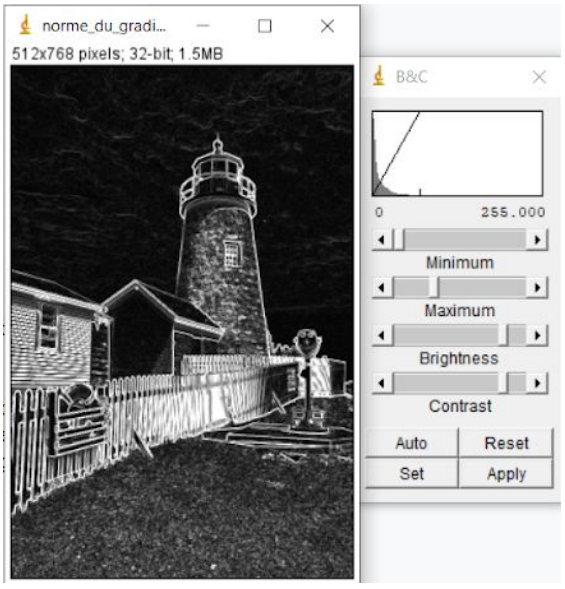


Image 'norme_du_gradient.png' avec valeurs comprises entre 0 et 255

Nous remarquons que l'image obtenu semble être la même que celle précédemment obtenue. Nous allons transformer l'image obtenu ci-dessus en image 8-bits. Cela nous permettra ainsi de comparer parfaitement les 2 images. Suite à cette opération, nous avons

donc soustrait une image à l'autre. Le résultat obtenu est une image noire. Grâce à l'histogramme de cette soustraction, nous remarquons que tous les pixels sont à 0. Cela signifie donc que les images sont similaires.

Utiliser le menu **Image/Adjust/Threshold...** pour seuiller la norme du gradient calculée. Est-il possible de trouver ainsi un seuil global pour l'image qui mette en évidence les pixels contours de manière satisfaisante ? Sur la base de ce principe, proposer un nouveau seuil pour la norme du gradient en ne retenant que 20% des pixels contours. Ce seuil est-il plus satisfaisant que celui déterminé à la question précédente ?

Nous allons maintenant effectuer un seuillage de notre image de la norme du gradient calculée. Nous allons essayer de trouver un seuil global pour l'image qui mette en évidence les pixels contours de manière satisfaisante. Pour cela, nous avons donc testé différents seuillages. Un seuil de 135 nous donne un résultat satisfaisant à notre sens puisqu'il concilie le bruit et les contours.

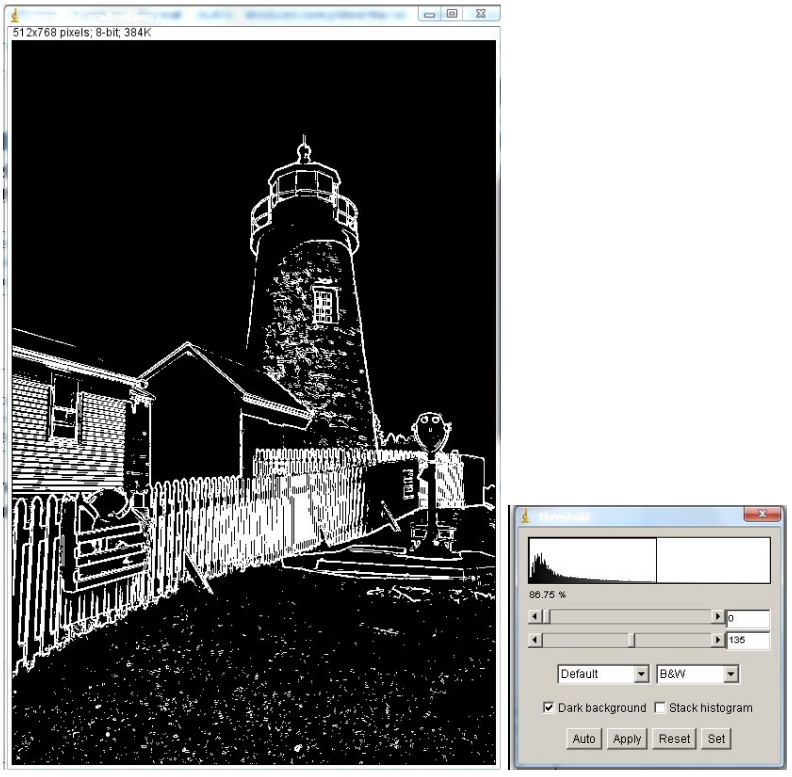


Image 'norme_du_gradient.png' avec un seuil de 135

Nous avons également pu remarquer qu'un seuil de 75 permet de voir tous les contours mais il y a beaucoup de bruit.

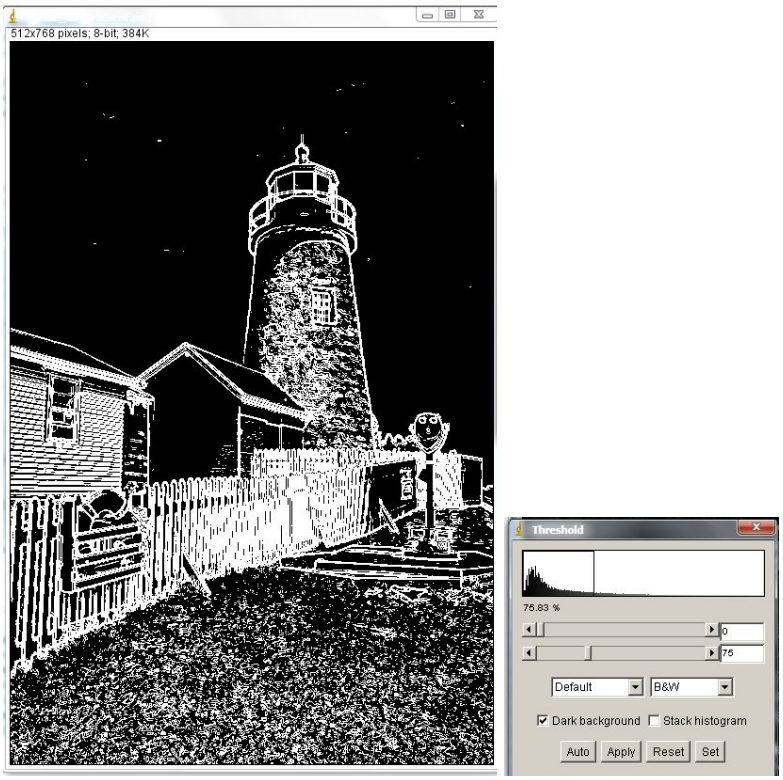


Image ‘norme_du_gradient.png’ avec un seuil de 75

Avec un seuil de 244 (valeur la plus élevée qui ne donne pas une image totalement noire), il y très peu de bruit même s’il y en a déjà notamment au niveau de la pelouse mais il y a également peu de contours mis en évidence.

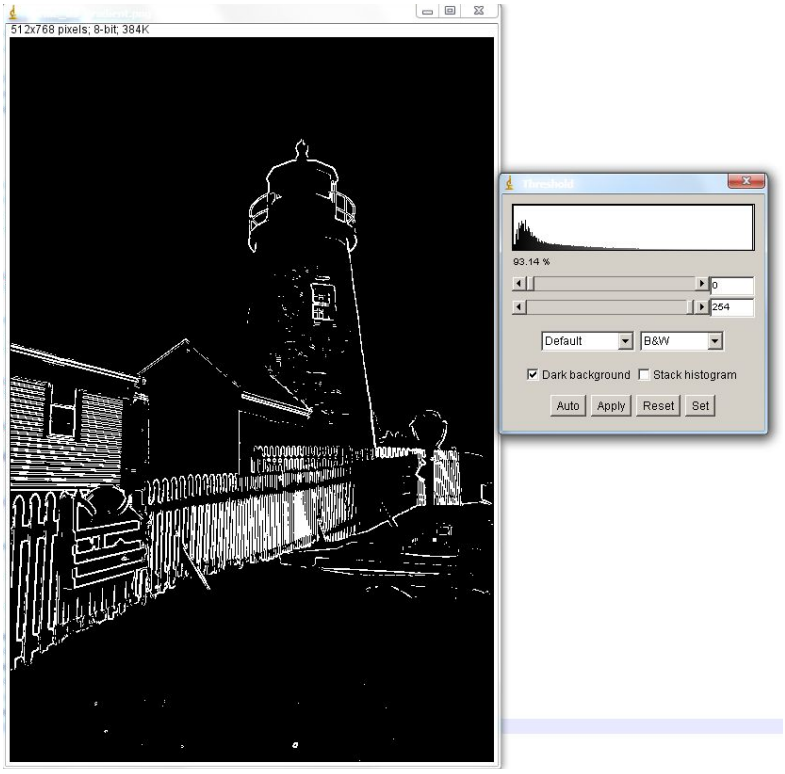
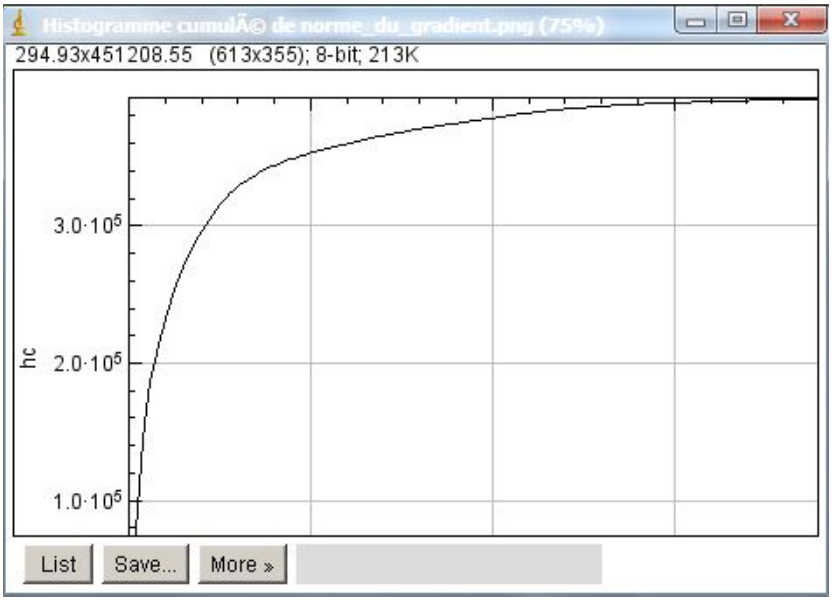


Image ‘norme_du_gradient.png’ avec un seuil de 244

Une alternative est de déterminer ce seuil de manière semi-automatique grâce à l'histogramme cumulé de l'image de la norme du gradient. L'idée est de sélectionner seulement un certain pourcentage de pixels contours : ceux pour lesquels la norme est la plus élevée, c'est-à-dire les n% de pixels contours les plus significatifs.

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



Histogramme cumulé de l'image 'norme_du_gradient.png'

Nous allons retenir seulement les 20% des pixels contours les plus significatifs (= pixels dont la norme est la plus élevée).

Pour rappel, notre image fait 512 * 768 pixels. Si on veut que 20% des pixels, il faudra alors n'avoir que 78 643 pixels environ. Ce nombre a été obtenu grâce au calcul suivant : (512*768)*20/100 = 78 643, 2. Cela signifie qu'environ 314 573 pixels vont donc être "mis de côté".

Grâce à l'étude de l'histogramme cumulé, nous remarquons que les pixels, au dessous de la valeur 25 sont les 20% des pixels qui nous intéressent. Un nouveau seuil serait donc 25.

2ème partie : Détection des maxima locaux de la norme d'un gradient

Ajouter à votre macro le calcul, dans une nouvelle image, de la direction du gradient estimé par les masques de Sobel. Pour calculer l'angle du gradient, on utilisera la fonction **atan2**, qui calcule une arc-tangente entre -π et π, et que l'on pourra convertir en degrés.

Dans cette deuxième partie, nous allons tout d'abord représenter sur une nouvelle image la direction du gradient estimée par les masques de Sobel. D'après le cours, la formule permettant le calcul de la direction du gradient est :

$$\theta(x, y) = \arctan\left(\frac{\partial f}{\partial y}(x, y) / \frac{\partial f}{\partial x}(x, y)\right)$$

Formule permettant de calculer la direction du gradient d'un pixel de coordonnées (x, y)

Afin de calculer la direction du gradient, nous avons ajouté dans notre boucle qui parcourt nos pixels, la formule permettant de calculer la direction. Nous avons alors multiplié ce résultat par 180 et divisé par **π** pour chaque pixel. Cela permet alors d'avoir sa direction en degrés. Nous avons alors affiché le résultat sur une nouvelle image. Voici le code correspondant à cette étape :

```
for(i = 0; i < h; i++){
        for(j = 0; j < w; j++){
                selectImage(filenameDerX);
                p_x = getPixel(j, i);           // Valeur du pixel sur l'image de la dérivée en x
                selectImage(filenameDerY);
                p_y = getPixel(j, i);           // Valeur du pixel sur l'image de la dérivée en y
                p = sqrt(p_x*p_x + p_y * p_y);  // Application de la formule
                // calcul de la direction en radian
                direction = atan2(p_y, p_x);
                // conversion en degres
                direction = direction * (180 / PI);

                selectImage(name_grad_norm);
                setPixel(j, i, p);
                selectImage(name_grad_direction);
                setPixel(j, i, direction);
        }
}
```

*Morceau de l'algorithme permettant de calculer la norme et la **direction** du gradient*

L’exécution de ce code nous donne le résultat suivant :

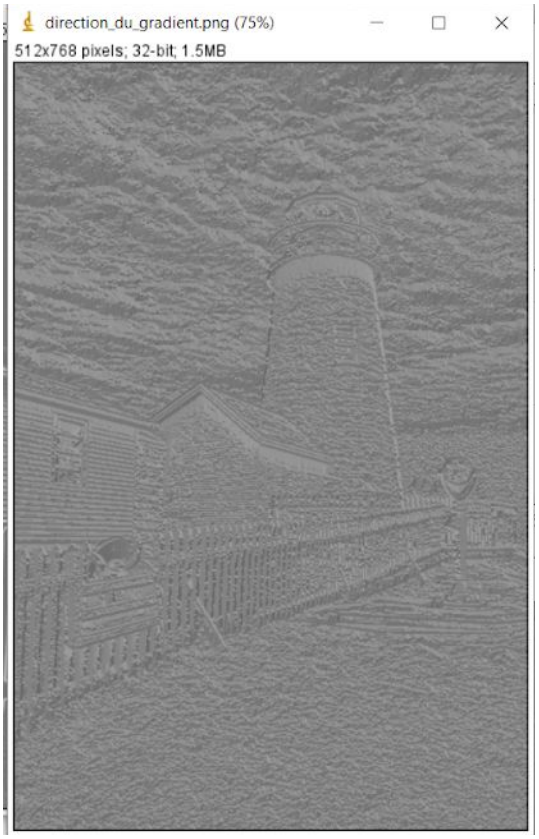
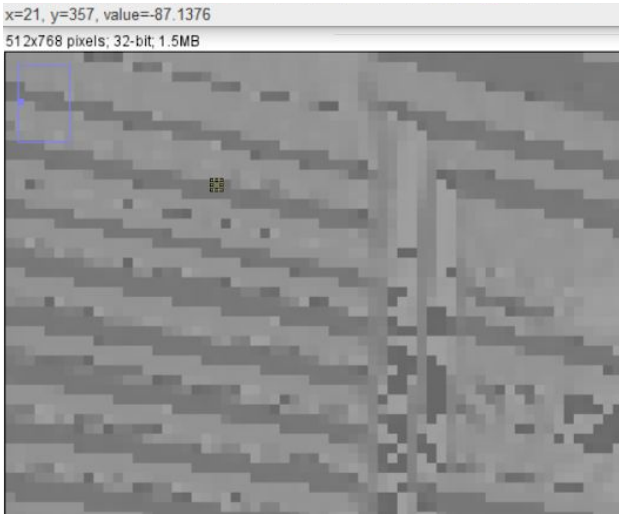


Image ‘direction_du_gradient.png’ correspondant à la direction du gradient de l’image source ‘lighthouse_8bits.png’

Examiner l'image représentant la direction du gradient de Sobel et l'interpréter en quelques pixels judicieusement choisis. Pour examiner les valeurs des pixels, on peut utiliser l'outil **Pixel Inspector** (à installer éventuellement par le bouton >>, puis disponible par le bouton Px).

Examinons maintenant quelques pixels. Analysons tout d’abord un pixel représentant un contour qui est quasiment horizontal tel qu’un des pixels de la maison.



Analyse du pixel (21,357) de l'image 'direction_du_gradient.png' : valeur d'environ 87

Nous remarquons que la valeur de ce pixel est proche de -90. Cela est logique car c'est un contour presque horizontal donc il est perpendiculaire d'où la présence du -90 (voir schéma du sens de rotation de la direction des pixels sur le schéma ci-dessous).

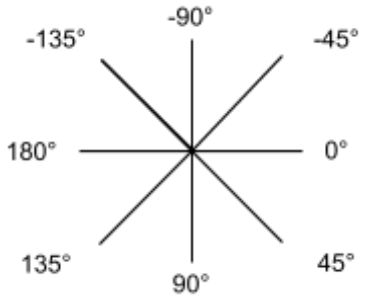
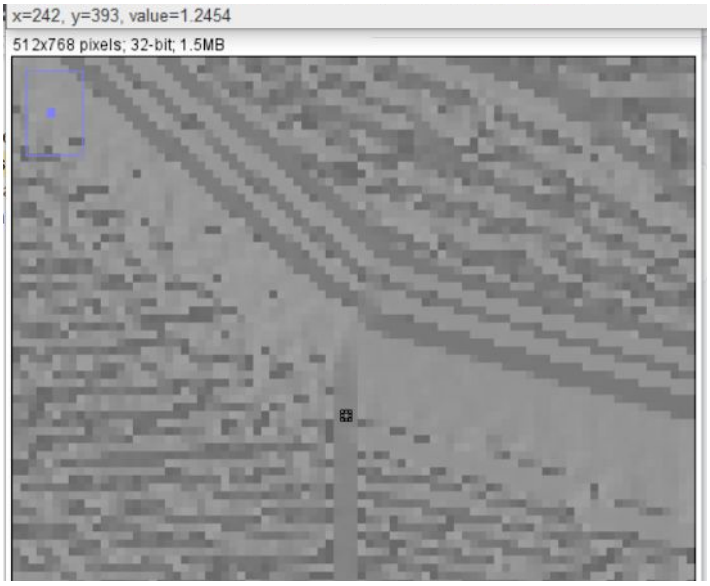


Schéma représentant le sens de rotation de la direction d'un pixel

Analysons maintenant un pixel représentant un contour qui est quasiment vertical tel qu'un des pixels représentant la jonction entre les deux murs de la maison.



Analyse du pixel (242,393) de l'image 'direction_du_gradient.png' : valeur d'environ 1

Nous remarquons que la valeur de ce pixel est proche de 0. Cela est logique car c'est un contour presque vertical donc il est parallèle.

La valeur de la direction des pixels de l'image nous permet de savoir la direction (cf schéma ci-dessus).

Compléter votre macro pour créer une nouvelle image de la norme du gradient, dans laquelle les non-maxima locaux sont éliminés (mis à 0). Les deux voisins à prendre en compte dans la direction du gradient seront déterminés par discrétisation de cette direction.

Afin d'avoir des contours plus nets (= d'épaisseur 1 pixel), nous allons créer une nouvelle image de la norme du gradient en supprimant les non-maxima locaux. Pour cela, il faut pour chaque pixel, comparer la valeur de sa norme avec les deux pixels situés de part et d'autre de ce point dans la direction du gradient. Le point *p1* est le pixel voisin situé dans la même direction que le pixel étudié. Le point *p2* est le pixel opposé par rapport à ce pixel. On compare alors la norme du gradient de notre pixel *p* avec celle de *p1* et de *p2*. Si on a la norme du gradient de *p* supérieure ou égale à *p1* et supérieure à *p2*, alors *p* un maximum local. Dans le cas contraire, c'est un non-maximum local, il faut donc mettre sa valeur à 0 afin de l'éliminer. Voici le code correspondant à cette étape :

```
non_maxima_locaux_supp = "norme_sans_maxima_locaux"+extension;
selectImage(name_grad_norm);
run("Duplicate...", "title="+non_maxima_locaux_supp); // Création de la nouvelle image

for(i = 0; i < h; i++){
   for(j = 0; j < w; j++){
            selectImage(name_grad_direction);
            direction = getPixel(j, i); // récupération de la direction
            selectImage(name_grad_norm);
            p = getPixel(j, i);    // récupération de la norme du gradient

            // Choix pixels p1 et p2 suivant l'angle (cf p. 12 du cours)
            if(direction >= 22.5 && direction < 67.5){
                     p1 = getPixel(j+1, i+1);
                     p2 = getPixel(j-1, i-1);
            } else if (direction >= -157.5 && direction < -112.5 ){
                     p1 = getPixel(j-1, i-1);
                     p2 = getPixel(j+1, i+1);
            } else if (direction >= 67.5 && direction < 112.5){
                     p1 = getPixel(j, i+1);
                     p2 = getPixel(j, i-1);
            } else if (direction >= -112.5 && direction < -67.5 ){
                     p1 = getPixel(j, i-1);
                     p2 = getPixel(j, i+1);
            } else if (direction >= 112.5 && direction < 157.5){
                     p1 = getPixel(j-1, i+1);
                     p2 = getPixel(j+1, i-1);
            } else if (direction >= -67.5 && direction < -22.5 ){
                     p1 = getPixel(j+1, i-1);
                     p2 = getPixel(j-1, i+1);
            }else if (direction >= -22.5 && direction < 22.5 ){
                     p1 = getPixel(j+1, i);
                     p2 = getPixel(j-1, i);
            }else{
                     p1 = getPixel(j-1, i);
                     p2 = getPixel(j+1, i);
            }

            selectImage(non_maxima_locaux_supp);
            //Mise à 0 la norme du gradient pour les pixels non maxima locaux
            if (p >= p1 && p > p2) {
                     setPixel(j, i, p);
            }
            else {
                     setPixel(j, i, 0);
            }
   }
}
```

Morceau de l'algorithme permettant de créer une nouvelle image de la norme du gradient dans laquelle les non-maxima locaux sont éliminés

Comparer cette nouvelle image de la norme du gradient et celle obtenue initialement, dans laquelle les non-maxima locaux n'étaient pas encore supprimés.

Comparons maintenant l'image obtenu avec ce nouveau code avec l'image de la norme du gradient obtenu au début de ce TP.

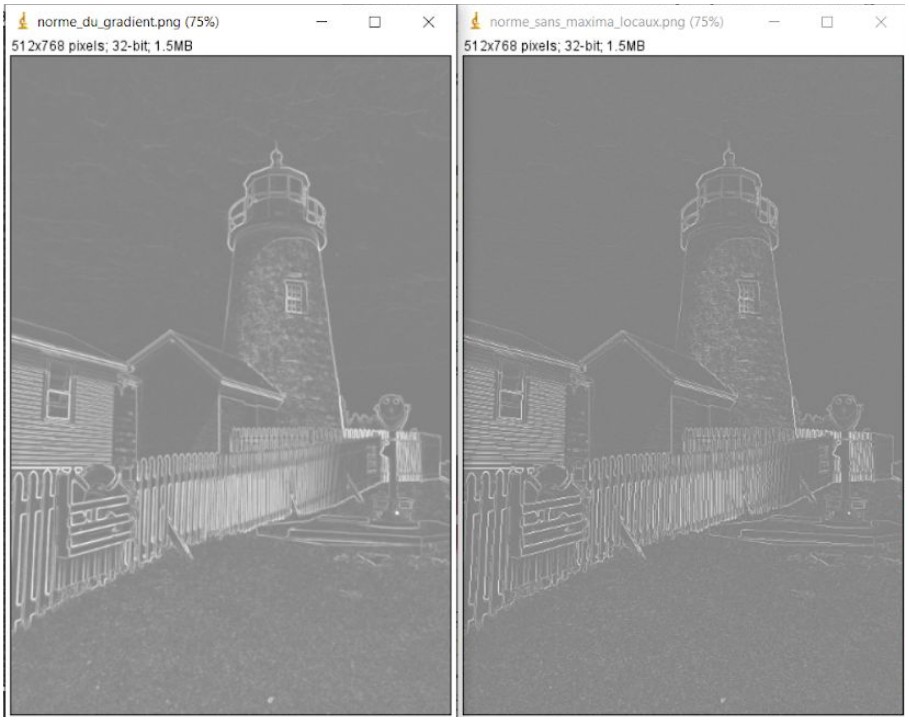


Image de la norme du gradient et image de la norme du gradient avec élimination des non-maxima locaux

On remarque que les deux images sont semblables. La différence est que la nouvelle image à ses contours beaucoup plus fins (1 d'épaisseur).

3ème partie : Seuillage des maxima locaux par hystérésis

Intégrer cette méthode dans un plugin imageJ et tester celui-ci.

Dans cette dernière partie, nous allons faire un seuillage des maxima locaux par hystérésis. Commençons tout d'abord par appliquer le plugin donné sur notre image de phare. Nous savons que l'algorithme de seuillage par hystérésis prend en paramètre une image G_M des maxima locaux de la norme du gradient. Nous récupérons donc l'image trouvé en fin de partie 2 afin de lui appliquer le plugin. Voici le résultat obtenu :

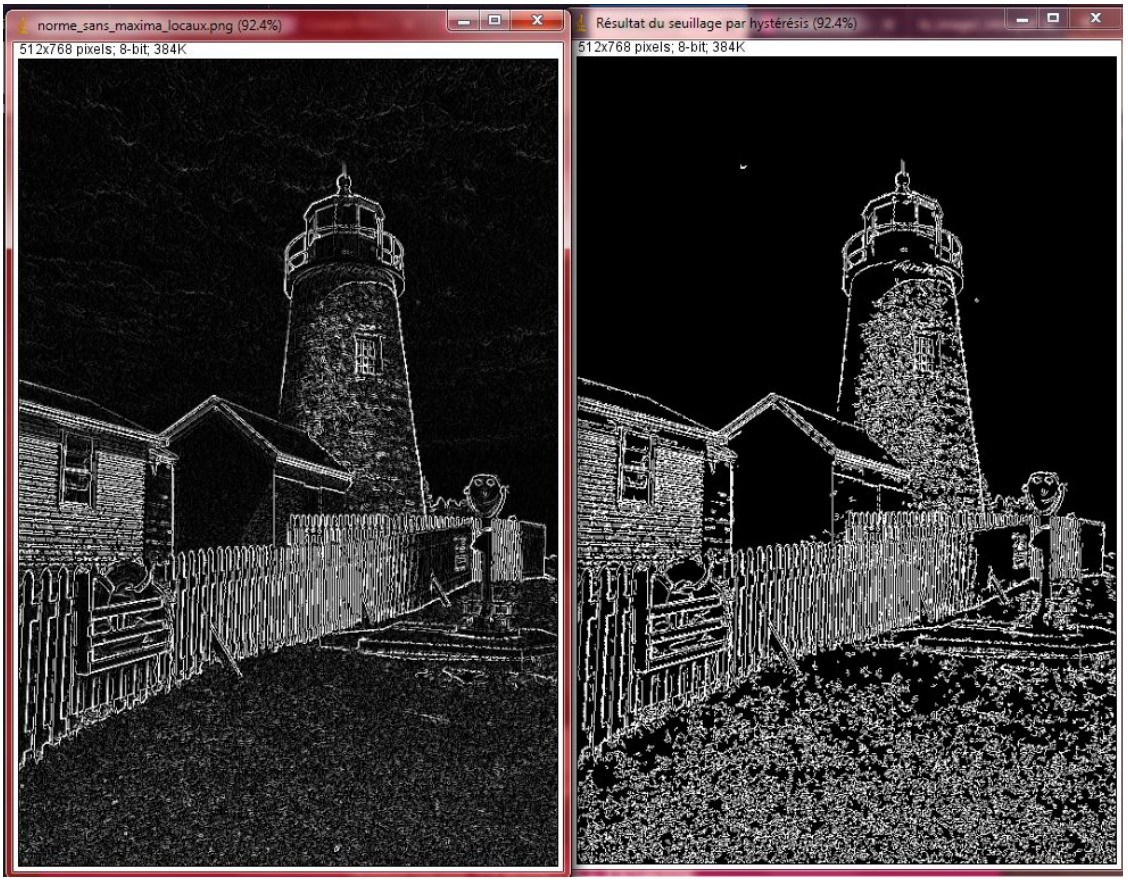


Image de la norme du gradient avec élimination des non-maxima locaux avant et après application du plugin de seuillage local par hystérésis (seuil haut = 100, seuil bas = 30)

Nous constatons que les contours de l'image de la norme du gradient après le seuillage par hystérésis sont beaucoup plus marqués et cela a globalement bien fonctionné avec le seuillage que nous avons défini. Toutefois, nous pouvons émettre quelques réserves sur le résultat puisqu'on distingue énormément de contours indésirables tels que l'herbe ou les briques du phare. Malheureusement, si nous réduisons le seuillage haut, beaucoup de contours "utiles" sont amenés à disparaître. Dans le cas d'une augmentation du seuillage bas, le bruit serait encore plus visible. Il s'agit d'un compromis à faire.

Analyser le code de la méthode **hystIter**, décrire globalement son fonctionnement, puis lui ajouter tous les commentaires nécessaires à sa compréhension.

La méthode **hystIter** prend en paramètre un imageProcessor et deux entiers (un pour le seuil bas et un pour le seuil haut) et elle retourne un ByteProcessor. Elle permet de mettre en évidence les pixels dont le seuil est compris entre les valeurs données tout en prenant en compte ses voisins.

Voici le code commenté :

```
public ByteProcessor hystIter(ImageProcessor imNormeG, int seuilBas, int seuilHaut) {
    // Récpération de la hauteur et de la largeur de l'imageProcessor donné en paramètre
    int width = imNormeG.getWidth();
    int height = imNormeG.getHeight();

    // Initialisation des structures de données utiles pour la suite
   // ByteProcessor sous-classe d'ImageProcessor pour les image 8 bits.
    ByteProcessor maxLoc = new ByteProcessor(width,height);
    List<int[]> highpixels = new ArrayList<int[]>();

    // Pour chaque pixel de l'image
    for (int y=0; y<height; y++) {
        for (int x=0; x<width; x++) {
            // Récupération de la valeur du pixel
            int g = imNormeG.getPixel(x, y)&0xFF;

            // Si la valeur du pixel est inférieur au seuil bas entré en paramètre, on ne fait rien
            if (g<seuilBas) continue;

            // Si la valeur du pixel est supérieur au seuil haut entré en paramètre alors ...
            if (g>seuilHaut) {
                // on met la valeur de notre pixel sur notre nouvelle image à 255 et ...
              maxLoc.set(x,y,255);
               // …on ajoute les coordonnées de notre pixel à notre liste d'entiers qui contient les couples (x,y) dont la
              // valeur est supérieur au seuil haut, puis continuer
                highpixels.add(new int[]{x,y});
                continue;
            }
           // Sinon on met la valeur de notre pixel sur notre nouvelle image à 128
            maxLoc.set(x,y,128);
        }
    }

    // Tableaux pour avoir les voisins du pixels traités
    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 notre liste de couple de pixel dont la valeur était supérieur au seuil haut n'est pas vide
    while(!highpixels.isEmpty()) {

        // Initialisation structure de données pour la suite
        List<int[]> newhighpixels = new ArrayList<int[]>();

         // Pour chaque couple de pixel dont la valeur était supérieur au seuil haut i.e. couple dans la liste highpixels
...
        for(int[] pixel : highpixels) {
            // Enregistrement du x et du y dans des variables
            int x=pixel[0], y=pixel[1];

            // Pour chaque voisin à distance de 1 (diagonale comprise) de notre pixel
            for(int k=0;k<8;k++) {
                int xk=x+dx8[k], yk=y+dy8[k];
               // Si c'est un bord, on ne fait rien
                if (xk<0 || xk>=width) continue;
```

```
            if (yk<0 || yk>=height) continue;
               // Si la valeur de la nouvelle image du pixel traité est à 128 (i.e. pixel à 128 de base ou pixel dont sa
valeur
            // est compris entre le seuil bas et le seuil haut compris alors ..
             if (maxLoc.get(xk, yk)==128) {
                // .. on met sa valeur à 255 et ...
                maxLoc.set(xk, yk, 255);
               // .. on ajoute ce couple à notre nouvelle liste qui va contenir tous les couples ainsi modifiés
                newhighpixels.add(new int[]{xk, yk});
             }
          }
       }

        // L'ancienne liste est remplacée par la nouvelle. Donc si on a changé au moins une valeur, on a au moins
un
      // couple dans cette liste donc on rerentre dans le while
       highpixels = newhighpixels;
   }

   // Pour chaque pixel
   for (int y=0; y<height; y++) {
      for (int x=0; x<width; x++) {
         // Si la valeur du pixel de la nouvelle image est différent de 255 le mettre à 0
          if (maxLoc.get(x, y)!=255) maxLoc.set(x,y,0);
      }
   }
  // retourne la nouvelle image
   return maxLoc;
}
```

Morceau de l'algorithme permettant de créer une nouvelle image de la norme du gradient dans laquelle les non-maxima locaux sont éliminés

Écrire une version récursive **hystRec** de la méthode de seuillage précédente.

Nous proposons une version récursive **hystRec** de la méthode **hystIter** précédente. Pour cela, nous proposons une signature de fonction telle que :

private void hystRec(ImageProcessor imNormeG, ImageProcessor imContours, boolean[][] visited, int seuilBas, int seuilHaut, int x, int y);

avec :

- *imNormeG* l'image de la norme du gradient.
- *imContours* l'image sur laquelle nous observons les contours.
- *visited* la matrice de l'image correspondant aux pixels déjà visités
- *seuilBas* le seuil bas de la méthode de seuillage
- *seuilHaut* le seuil haut de la méthode de seuillage
- *x* et *y* les coordonnées du pixel courant

Le code de cette fonction peut être divisé en deux parties, une première partie qui ne fait que traiter le pixel afin de transformer sa couleur (255 si sa valeur est supérieure au seuil haut, 0 si sa valeur est inférieure au seuil bas) et ajouter le pixel courant à la matrice des pixels visités.

```
int width = imNormeG.getWidth();
int height = imNormeG.getHeight();
int g = imNormeG.getPixel(x, y)&0xFF;
// 1° traiter le pixel
// > seuil bas -> on met le pixel à 255
if(g > seuilHaut){
        imContours.set(x,y,255);
} else {
        imContours.set(x,y,0);
}
visited[x][y] = true;
```

Morceau de l'algorithme permettant de modifier la valeur d'un pixel selon les cas

Ensuite, il ne nous reste plus qu'à appeler récursivement notre fonction dans chacune des 8 directions afin de propager le seuillage par hystérésis. Voici le morceau d'algorithme associé :

```
// 2° propager l'algo
// vers la droite
if(x < width-1 && !visited[x+1][y]){
hystRec(imNormeG, imContours, visited, seuilBas, seuilHaut, x+1, y);
}
// vers le bas
if(y < height-1 && !visited[x][y+1]){
hystRec(imNormeG, imContours, visited, seuilBas, seuilHaut, x, y+1);
}
// vers la gauche
if(x > 0 && !visited[x-1][y]){
hystRec(imNormeG, imContours, visited, seuilBas, seuilHaut, x-1, y);
}
// vers le haut
if(y > 0 && !visited[x][y-1]){
hystRec(imNormeG, imContours, visited, seuilBas, seuilHaut, x, y-1);
}
// vers le haut gauche
if(y > 0 && x > 0 && !visited[x-1][y-1]){
hystRec(imNormeG, imContours, visited, seuilBas, seuilHaut, x-1, y-1);
}
// vers le haut droit
if(y > 0 && x < width-1 && !visited[x+1][y-1]){
hystRec(imNormeG, imContours, visited, seuilBas, seuilHaut, x+1, y-1);
}
// vers le bas droit
if(y < height-1 && x < width-1 && !visited[x+1][y+1]){
hystRec(imNormeG, imContours, visited, seuilBas, seuilHaut, x+1, y+1);
}
// vers le bas gauche
if(y < height-1 && x > 0 && !visited[x-1][y+1]){
hystRec(imNormeG, imContours, visited, seuilBas, seuilHaut, x-1, y+1);
}
```

Morceau de l'algorithme permettant de propager le seuillage aux 8 voisins d'un pixel

Les seuils qui concilient le mieux le bruit et l'apparition des contours sont 17 pour le seuil bas et 145 pour le seuil haut. Voici ce que l'on obtient :

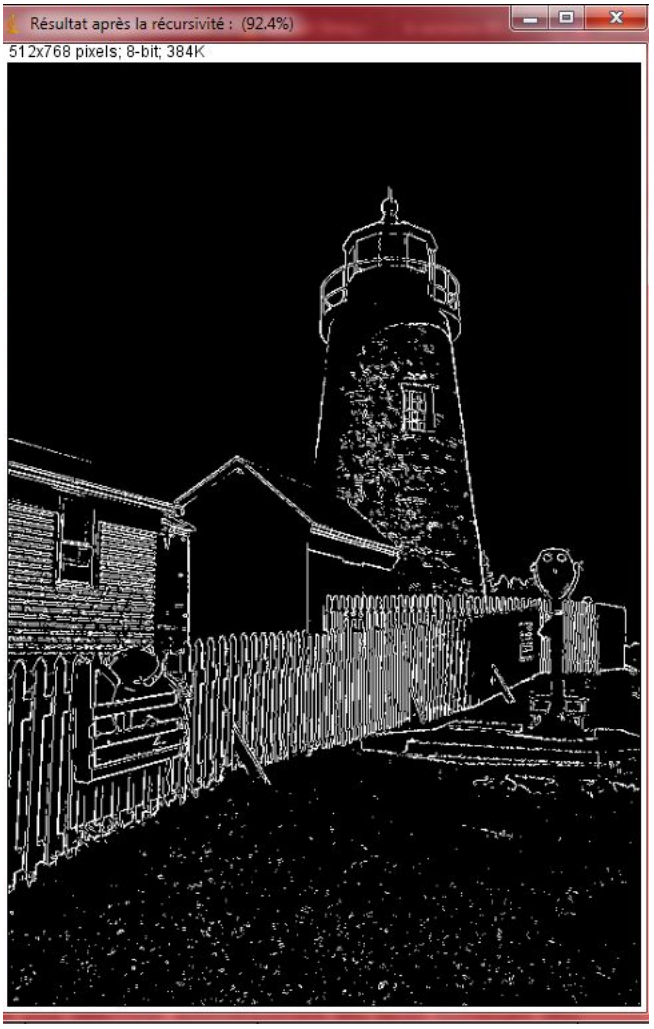


Image de la norme du gradient après un seuillage par hystérésis récursif (seuil haut = 145, seuil bas = 17).

Il n'existe pas de méthode pour déterminer automatiquement des seuils fournissant des résultats satisfaisants sur tous types d'images. Déterminer donc empiriquement, pour notre image de travail, quels sont les seuils que vous préconisez d'adopter. Bien que basés sur une analyse subjective, ceux-ci devront être justifiés au regard des résultats obtenus.

Après avoir étudié plusieurs méthodes de seuillage, qui pour la plupart nous ont fourni des résultats satisfaisants, nous avons pu constater que, selon les valeurs des seuils haut et bas, nous avions des propriétés différentes sur nos images résultantes. Par exemple, un seuillage bas nous permet de faire disparaître, parfois entièrement, le bruit (et donc les mauvais contours) d'une image mais fait disparaître certains "vrais" contours. Au contraire, un seuillage haut nous permet d'afficher tous les contours mais également ce qui n'en sont pas (par exemple l'herbe sur l'image '*lighthouse_8bits.png*' ou les briques qui ressortent du phare sur cette même image). Ainsi, il est nécessaire de concilier le seuillage haut et le seuillage bas afin d'obtenir le plus de contours et le moins de bruit. Dans le cadre de nos recherches, le seuillage bas le plus pertinent fut 17 et le seuillage haut le plus approprié fut 145.