Ivanina Ivanova, Preshtibye Raggoo MIN-INT

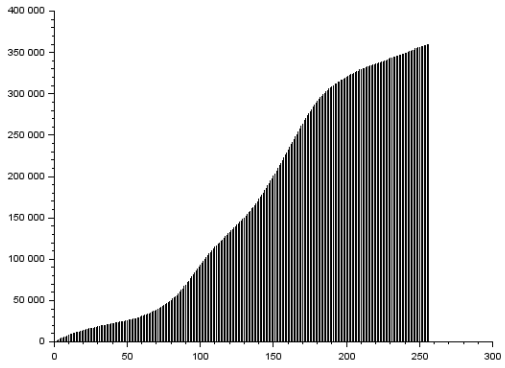
Compte-rendu TP4
Filtres différentiels, détection de contours (1ere partie)

Exercice 1

Normal

Avec T_affine

On voit très bien qu'on a un visuel bien mieux avec la fonction T_affine, c'est-à-dire qu'on voit les contours de l'image bien plus claire. Cette fonction permet, en général, d'afficher les contours de l'image qu'on est en train d'analyser avec le code.



Histogramme cumulé

Image initial

Ici, grâce à la fonction hist_cumul, qu'on a vu dans les TPs précédents, on affiche l'histogramme cumulé de l'image. De cet histogramme, on choisit les valeurs de p0 et p1. Ici , on a choisi p0 = 120 et p1=190, car du graphe, on voit que la variation des intensités des pixels se fait entre primordialement entre les valeurs 120 et 190.

On a essayé d'exécuter la fonction avec la valeur de p0 = 70, on a cependant remarqué qu'on a un meilleur affichage de l'image avec p0= 120. On estime que c'est parce que entre les valeurs 70 et 120, la variation des intensités des pixels n'est pas aussi linéaire que l'intervalle qu'on a choisi nous.

Ci-dessous on retrouve le code qu'on a utilisé pour avoir l'affichage des deux images et de l'histogramme .

```
exec('init_tp_image.sce');
im = lire_imageBMPgris('sweets.bmp');
Hist = hist_cumul(im);
plot2d3(Hist);
p0 = 120; // a modifier
p1 = 190; // a modifier
im2 = T_affine(im, p0, p1);
afficher_image(im);
imn2 = norme_gradient(im2);
afficher_image(int(imn2));
imn = norme_gradient(im);
afficher_image(int(imn));
```

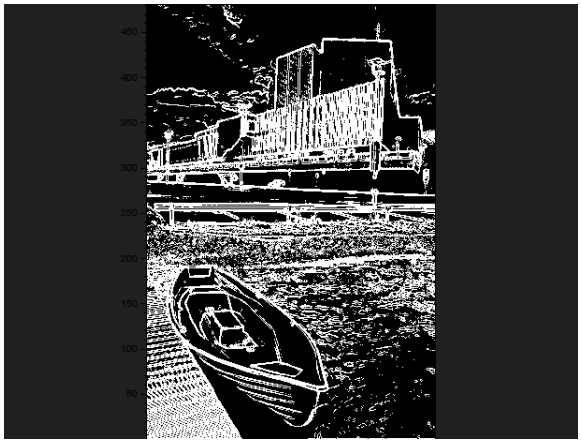
Et ci dessus le code source de la fonction norme_gradient:

```
function imn=norme_gradient(im)
   // tableau imn initialise a la meme
   // taille que l'image
   Dx = 0.5*[-1, 0, 1]
   Dy = 0.5*[1; 0; -1]
   imn = zeros(im);
   imx = conv2(im, Dx, "same");
   imy = conv2(im, Dy, "same");
   [M,N] = size(im);
   for u = 1:M
      for v = 1:N
         imn(u,v) = sqrt(imx(u,v).^2+imy(u,v).^2);;
      end
   end
endfunction
```

Exercice 2

Image Initial

Seuil - 20



Seuil - 10

Seuil - 40

Quand on fait varier la valeur du paramètre seuil on constate que quand le seuil est plus petit que 20 (ici on a choisi le seuil 10) il y a beaucoup plus des détails dans l'image donc il y a beaucoup plus des contours des différents objets. Par contre, quand on fait augmenter le seuil (ici on choisit le seuil 40) on voit qu' on perd de plus en plus des détails et les objets les plus éclairés sont mieux définis que les objets sombres.

Ci-dessus le code source de la fonction demandée:

```
function imc=contours_seuil(im, seuil)
  imn = norme_gradient(im);
  imc = zeros(im);
  //// A COMPLETER ////
  [M,N] = size(im);
  for u = 1:M
     for v = 1:N
```

```
        if  imn(u,v) < seuil
            then
                imc(u,v)=0
        else
            imc(u,v)=255
        end
    end
  end
endfunction
```

Exercice 3

Le code source de la fonction trouver_seuil:

```
function seuil=trouver_seuil(im, p)
  G = norme_gradient(im);
  H = hist_cumul(G);
  seuil = 1;
  for v =1:256
    if (H(v)/H(256) > p)
      then
        seuil=v
        break
    end
    v=v+1
  end
endfunction
```

Pour l'exercice 3, on a utilisé le break dans notre code. Cela est parce qu'on veut que le code arrête son exécution dès qu'on trouve la plus petite valeur de v qui satisfait la condition. Ainsi, on évite que notre programme fasse des opérations redondantes. On sort de la boucle dès qu'on trouve cette valeur du seuil.

En observant les deux images ci-dessus, on voit que cette fonction nous permet de d'afficher l'image originale, avec une transformation pour mieux voir les contours de l'image. La fonction trouver_seuil effectue ici une recherche automatique de seuil en fonction d'un pourcentage p et renvoie sa valeur, pour qu'on puisse ensuite afficher l'image avec ce valeur de seuil comme paramètre.

Dans les exercices précédents, c'est nous qui faisons varier la valeur de seuil. Bien que cela nous donne des bons affichages avec certaines valeurs de seuil choisi, on trouve que cela peut bien devenir problématique de faire plusieurs jeux d'essais.
Ici, le calcul automatique de p, nous permet d'avoir une valeur de p qui soit très cohérent et qui donne ainsi un valeur de seuil qui nous donne par conséquence, un très bon affichage de

l'image avec les contours bien visibles sans trop de perte d'information (bien que cela soit inévitable).

On observe aussi un très bon contraste dans toutes les images qu'on affiche ci-dessous.

Image Original

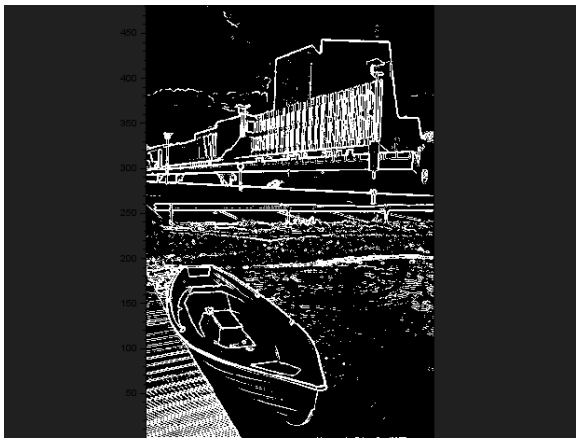


Image après transformation avec la function trouver_seuil

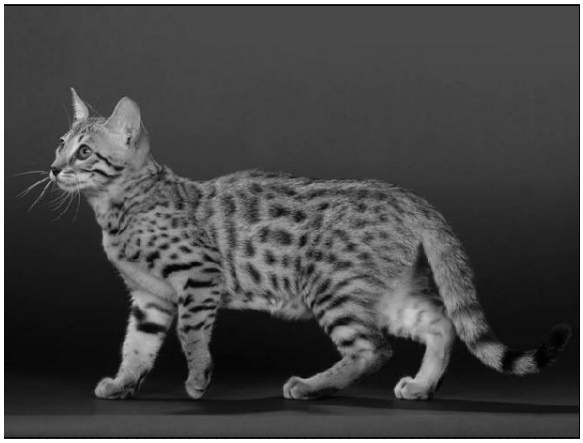


Image Original

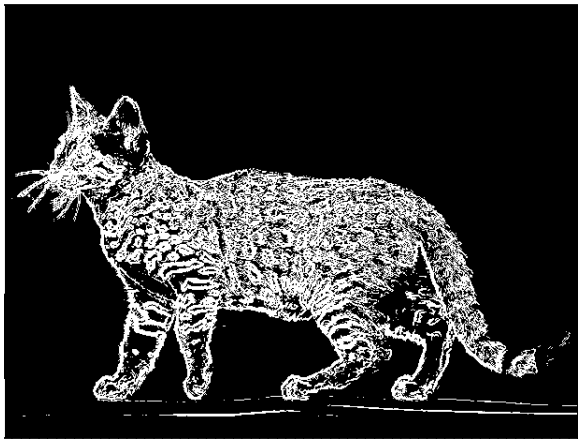


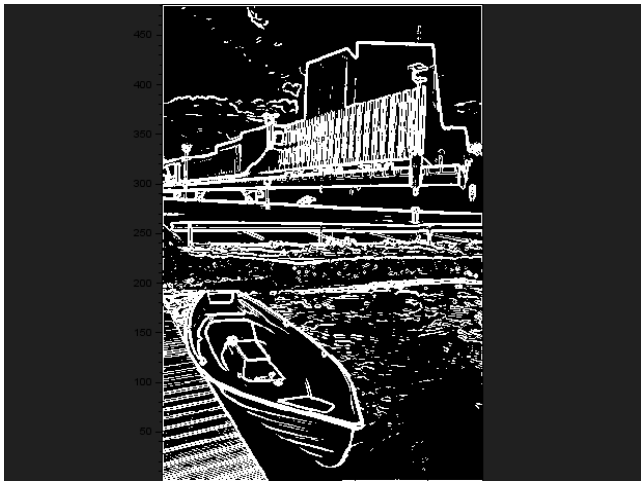
Image après transformation avec la function trouver_seuil

Image Original

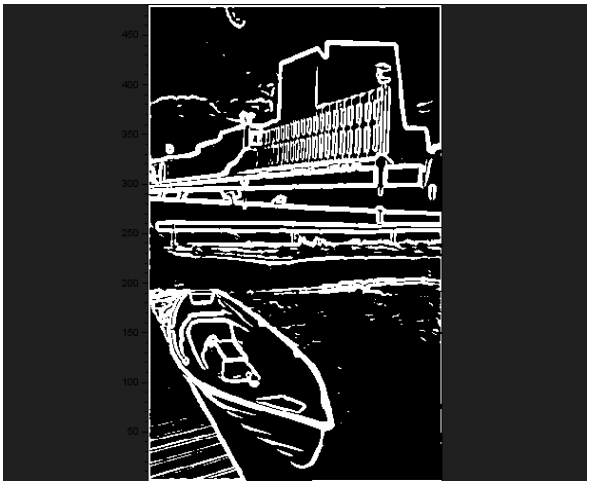
Image après transformation avec la function trouver_seuil

Exercice 4

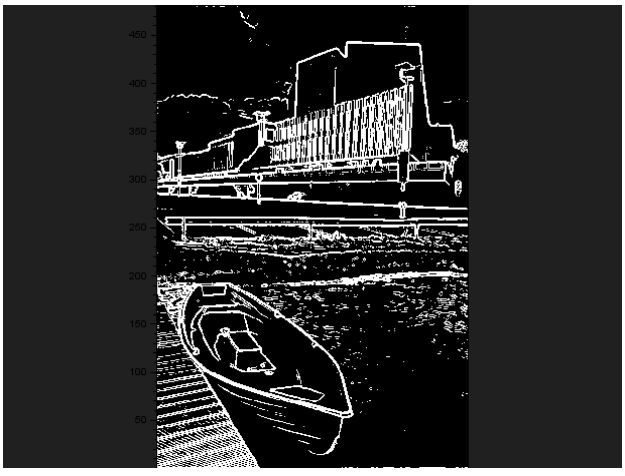
Dans l'exercice 4, on appelle deux fonctions qu'on a déjà définies dans les parties précédentes.
afficher_image(contours_p(conv2(im, W_gauss_2D(sigma), "same"), p));
Cette commande nous montre très clairement, l'utilisation des fonctions W_gauss_2D et contours_p.
On appelle d'abord la fonction W_gauss qui réduit le bruit de l'image, puis la fonction contours_p qui affiche les contours de l'image moins bruité.



p=0.7;
sigma = 0.7;

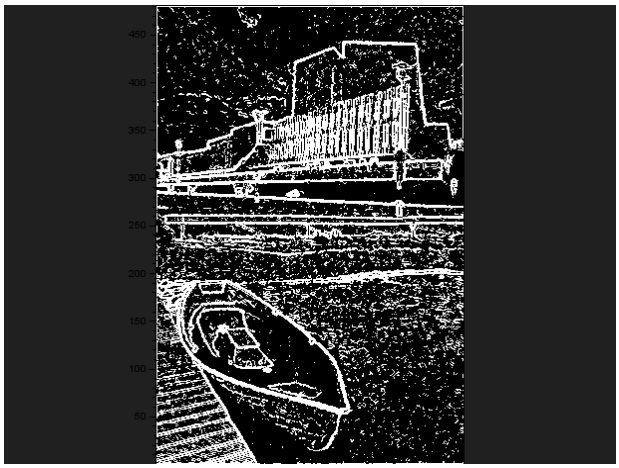


p=0.7;
sigma = 1.5;

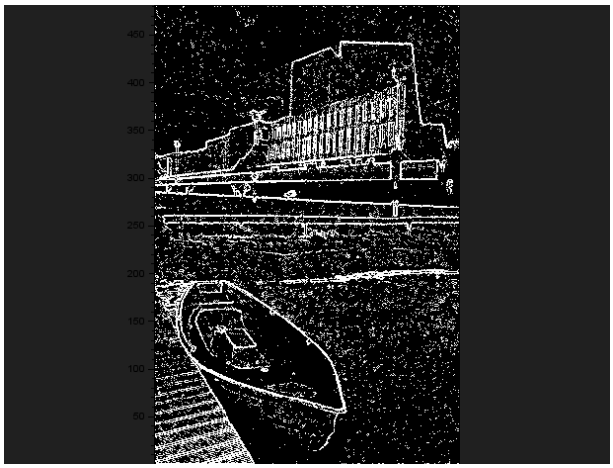


Ici, on voit qu'avec un sigma trop grand, on a une trop grande perte d'information.
p=0.8;
sigma = 0.5;

En faisant varier les valeurs de p et de sigma, on observe qu'on obtient un affichage relativement bien avec **p = 0.7** et **sigma = 0.7** . Avec **sigma = 0.7**, le filtre gaussien réduit bien le bruit de l'image (Il y a une balance entre la réduction du bruit et la réduction de contraste tel qu'on a pas trop de perte d'information) Avec **p = 0.7** , l'image affiché a une bonne affichage des contours (sans trop de perte d'information dans ce cas aussi).



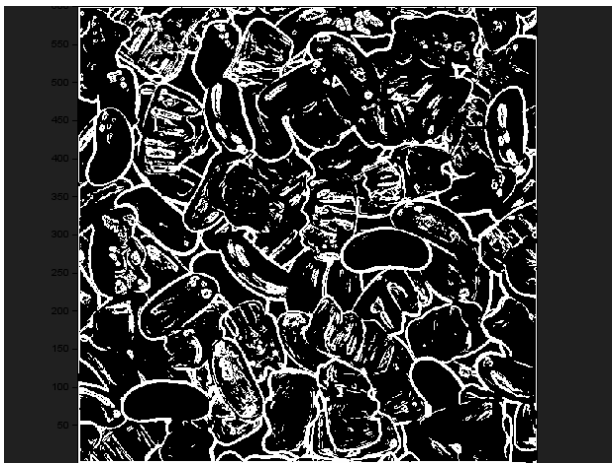
p=0.7;
sigma = 0.8;



p=0.8;
sigma = 0.5;

p=0.8;
sigma=0.55;

p=0.8;
sigma=0.8;



p=0.7;
sigma = 0.6;

p=0.8;
sigma = 0.5;

	barque.bmp	barque_bruit.bmp	plage.bmp	sweets.bmp
sigma	0.7	0.7	0.8	0.7
p	0.7	0.8	0.55	0.6

On observe la même chose avec toutes les autres images. Il nous faut une bonne combinaison de p et sigma pour avoir un bon affichage des images.
Pour l'image barque_bruit.bmp, on retrouve un affichage d'image satisfaisant avec p = 0.7 et sigma=0.8. On a aussi essayé d'afficher l'image avec sigma = 0.5 et p = 0.8. On trouve une image qui a perdu trop d'informations, comme on s'attendait. Avec un si grand sigma, le filtre gaussien, en diminuant le bruit de l'image entraîne une trop grande perte de qualité, ainsi l'image résultante avec contours.sci nous donne une image bien trop modifiée.

Filtres différentiels, détection de contours (2ème partie)

Exercice 5

Cet exercice a pour but d'illustrer les lacunes de l'opération de seuillage qu'on a défini dans la première partie du TP.
On note qu'avec cette fonction, bien qu'on ait des résultats tout à fait acceptables, on peut voir que l'épaisseur des contours qu'on affichent ne sont pas constantes et ainsi, on peut se retrouver avec un affichage problématique comme illustré ci-dessous.

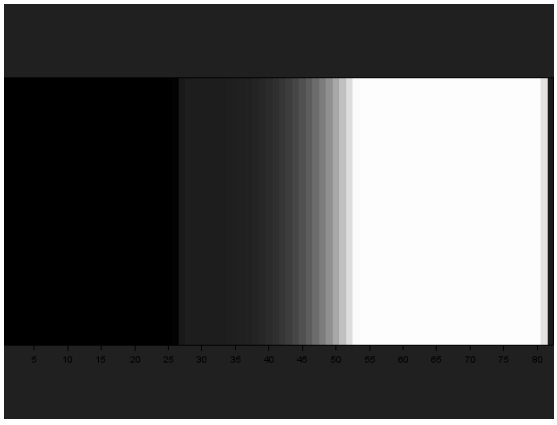
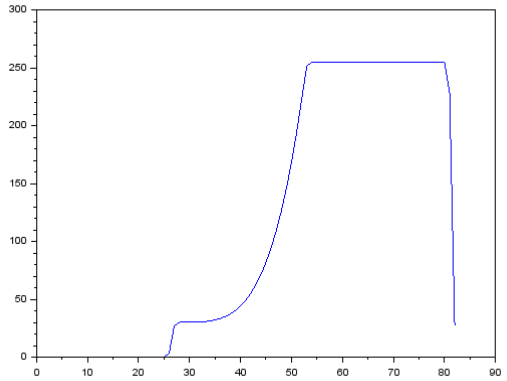
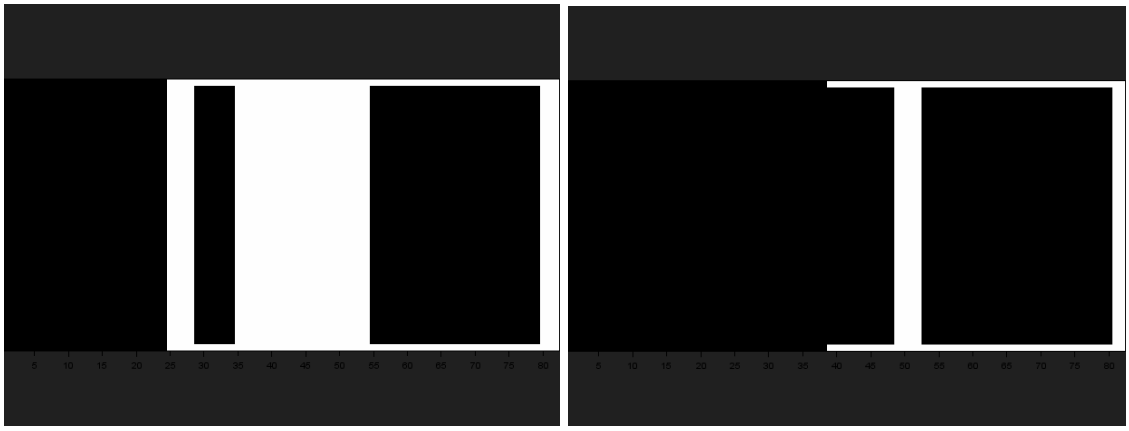


Image originale



Norme du Gradient



Comme démontré dans les images ci-dessus, on a extrait les contours de l'image originale avec la fonction contours_p, en faisant varier le paramètre p.
Cependant, on déduit qu'on ne peut pas trouver une valeur de p qui permette d'obtenir simultanément les contours en M/3 et 2M/3 avec une épaisseur comparable.

Afin de pouvoir mieux interpréter nos résultats, on visualise la norme du gradient:

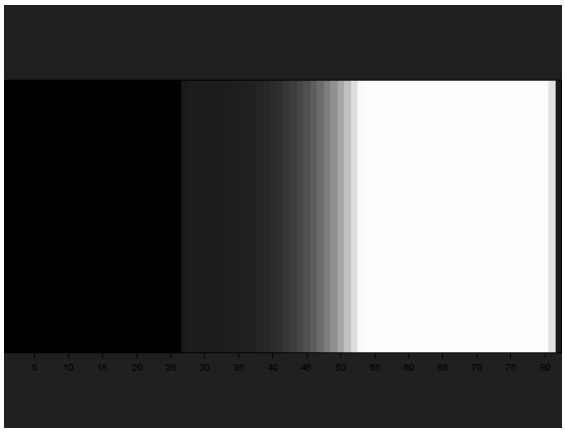
- On observe que pour la première partie du graphe, la valeur des pixels correspond à zéro. Visuellement, on peut voir que sur l'image cela représente la première fraction de l'image qui est noire (ainsi pixel de valeur 0).
- Du pixel 25 à environ 40, on voit qu'il y a une variation non linéaire des valeurs des pixels. Cela correspond au rapide variation des valeur de pixels, comme on voit dans l'image- une rapide changement de couleur de noir à gris.
- Les valeurs de pixels augmentent ensuite de manière plutôt linéaire jusqu'à atteindre la valeur 255, ce qui représente la partie blanche de l'image originale.(**pente A**)
- Puis, pour représenter la bordure de l'image, on observe une rapide chute des valeurs des pixels. (**pente B**)
- Les deux variations de pixels ont deux très différentes normes; on peut voir cela du graphe aussi. La **pente B** est bien plus raide que la **pente A.** Ainsi, pour la même valeur de p, l'épaisseur du contour 2M/3 sera toujours visiblement moins élevée que celle du contour M/3.

Exercice 6

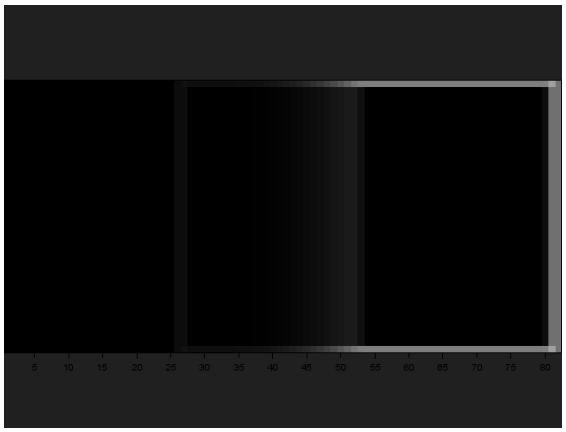
```
function [u1, v1, u2, v2]=indices_voisins(u, v, phi)
   if phi <= 0  then phi = phi + %pi
   end
   if phi <= %pi/8 || phi >=7*%pi/8 then
      u1=u
      v1=v-1
      u2=u
      v2=v+1
   elseif phi>%pi/8 & phi <=3*%pi/8 then
```

```
        u1=u+1
        v1=v-1
        u2=u-1
        v2=v+1
    elseif phi>3*%pi/8 & phi <5*%pi/8 then
        u1=u+1
        v1=v
        u2=u-1
        v2=v
    else
        if  phi>=5*%pi/8 & phi <7*%pi/8
        u1=u+1
        v1=v+1
        u2=u-1
        v2=v-1
        end
    end
endfunction
```

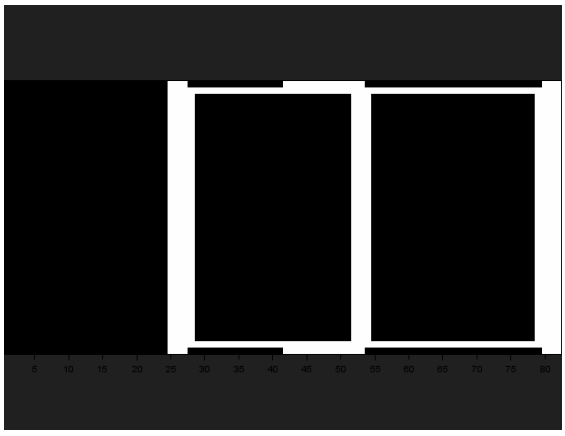
Exercice 7



Im



Imn



Imc

im_out

Seconde test avec l'image générée par la fonction disque(); :

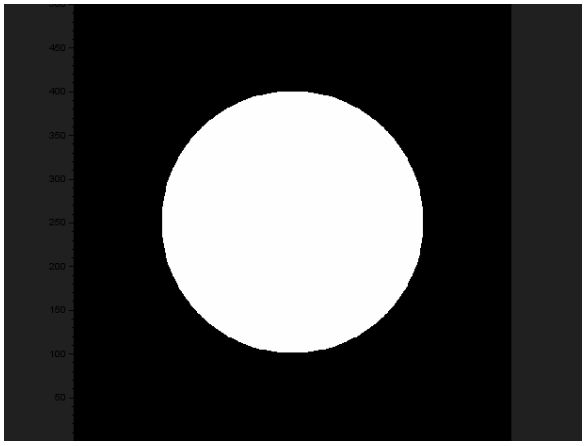
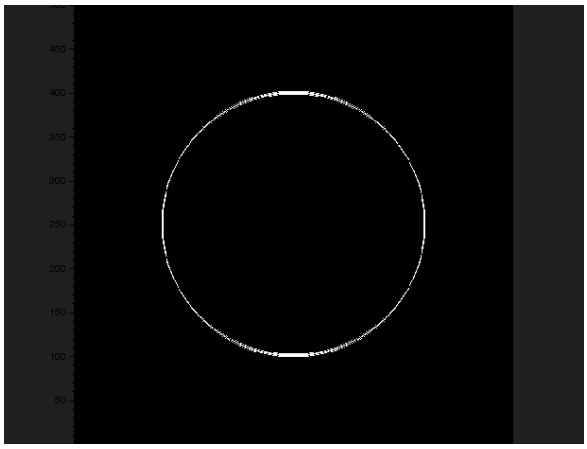


Image Initial



Im_out

Comme illustré dans les images ci-dessus, l'exercice 7 comprend d'une succession de génération d'images. Im_out correspond bien à nos attentes de la fonction contours_max, les contours ont bien la même épaisseur et les pixels insignifiant à la représentation des contours ont été éliminés.

```
function im_out=contours_max(im, p)
    Dx = 0.5*[-1, 0, 1]
    Dy = 0.5*[1; 0; -1]
    imx = conv2(im, Dx, "same");
    imy = conv2(im, Dy, "same");
    imn = norme_gradient(im);
    seuil = trouver_seuil(imn,p);
    imc = contours_seuil(imn,seuil);
    afficher_image(imc);
     [M,N] = size(im);
    for u = 1:M
        for v = 1:N
              phi = atan(imy(u,v), imx(u,v));
              if u==1 || u==M || v==1 || v==N
                 then imc(u,v) = 0
              end
              if imc(u,v) == 255
                 then [u1,v1,u2,v2] = indices_voisins(u,v,phi);
                 if imn(u1,v1) > imn(u,v) || imn(u2,v2) > imn(u,v)
                    then imc(u,v) = 0
                 end
              end

        end
```

```
    end
    im_out = imc;
endfunction
```

Conclusion :
Dans les deux exercices précédents (Exercice 6 et 7), on tente de remédier aux problèmes qu'on retrouve dans l'exercice 5. Dans l'exercice 5, on remarque qu'on arrive pas à obtenir simultanément les contours en M/3 et 2M/3 avec une épaisseur comparable.
Comme illustré dans les figures ci-dessous, les fonctions qu'on définit dans l'exercice 7 nous permettent d'afficher les contours de l'image Im avec une épaisseur comparable.

Pour commencer, on définit la fonction indices_voisins (Exercice 6) qui prend comme arguments un indice u,v et un angle phi. Cette fonction a pour but de comparer la valeur d'un pixel donné à celle de deux de ces voisins.
On récupère les pixels adjacents en fonction de la direction du gradient. En fonction de la valeur de l'angle, on récupère des voisins différents.

Explication de l'implémentation de la fonction:

- Comme énoncé, on utilise la fonction prédéfinie atan pour effectuer des calculs mathématiques avec la norme du gradient de chaque pixel de l'image filtrée.
- Utilisant les conditions données pour les angles, on se sert des structures conditionnelles (if...elseif...end) pour générer les 4 différents cas indiqués.
- Avec la fonction définie, on trouve les voisins du pixel en suivant l'angle du gradient. Ici, en examinant l'exemple donné, on voit que si l'angle est de 90° les pixels au-dessous et au-dessus du pixel courant sont considérées comme les pixels voisins.
  Leurs indices sont : (**u1**,**v1**)=(**u**+1,**v**) , (**u2**,**v2**)=(**u**-1,**v**).
- Ensuite, on vérifie la valeur des pixels voisins. Si la valeur de l'un des voisins est plus élevée que la valeur du pixel central courant, le pixel courant est affecté la valeur de 0 (Pixel représentant la couleur noire). Dans le cas contraire, on garde le pixel central à sa valeur d'origine.
- Ainsi, on peut dire que cette fonction(indice_max) permet une extraction des maxima locaux.

Pour l'exercice 7, on utilise la fonction trouver_seuille qu'on avait définie dans la première partie du TP pour effectuer la détection automatique de la valeur du seuil, avec une valeur de **p** donnée.

Premièrement, dans cette partie de la fonction contours_max,

```
 if u==1 || u==M || v==1 || v==N then
          imc(u,v) = 0
 end
```

On fait reduire tous les pixels au bord de image à 0 (noir). Dans le cas contraire, on ne verrait pas vraiment les contours de l'image car le bord de l'image serait complètement blanc, on

arriverait pas à différencier les contour et la fin de l'image en elle-même. Ainsi, cela impose un affichage bien plus cohérent du contour de bords.
Cela permet aussi que tous les contours soient de la même épaisseur (dans le cas ci-dessus, l'épaisseur des contours est de un pixel).

Puis, dans le cas où les pixels ont pour valeur 255, les indices du pixel sont passés comme argument de la fonction indice_voisins.

```
if imn(u1,v1) > imn(u,v) || imn(u2,v2) > imn(u,v) then
        imc(u,v) = 0
end
```

Dans la portion de code ci-dessus, on compare les valeurs de du pixel courant (indices:(u,v)) avec les valeurs de ces pixels voisins. Il suffit que la valeur d'un des pixels voisins soit strictement supérieure à celle du pixel courant pour que la valeur du pixel courant soit réduite à 0.

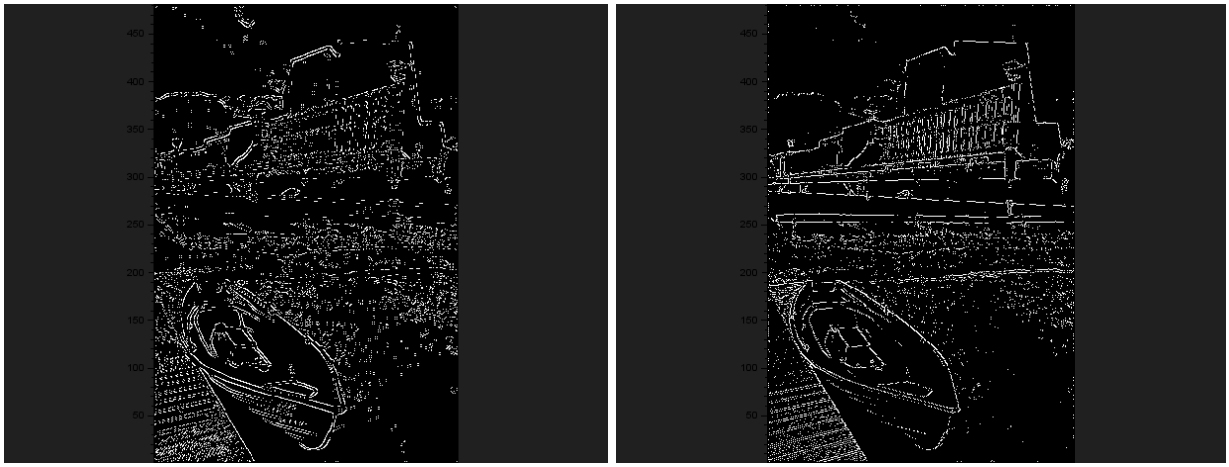
De cette façon, on arrive à éliminer tous les pixels insignifiants à l'affichage des contours pour avoir une visuelle bien plus cohérente comme illustrée ci-dessus. L'épaisseur des contours sont les mêmes, en effet c'est bien cela qu'on veut observer quand on veut afficher les contours d'une image.

Les contours superflus ont disparu et les traits sont moins marqués.

La fonction disque nous permet de générer un disque. On utilise ensuite contours_max afin d'afficher les contours du cercle. On voit bien que ce n'est que le rebord du cercle qui soit afficher comme contour. On conclut qu'on affiche que les contours d'épaisseur de 1 pixel dès qu'on rencontre un changement de valeur de pixels . (Pour le cercle, l'algorithme retrouve un brusque changement de pixel de valeur 0 à 255 (entre le fond de l'image(noir) et le cercle(blanc)).

Ce filtre nous assure que pour un contour, il n'y ait qu'une seule détection.

Exercice 8



Avec contours_p et W_gauss_2D

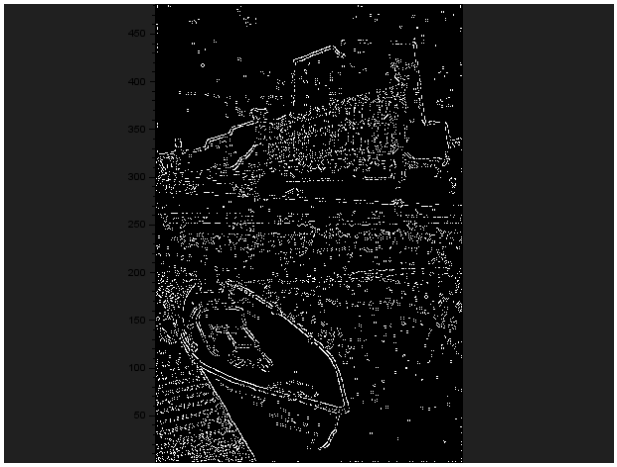
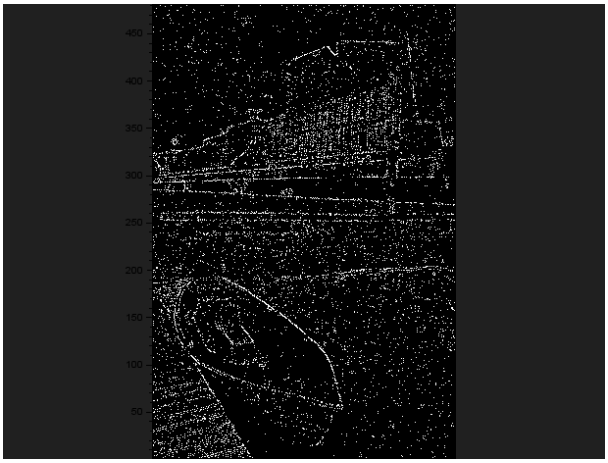
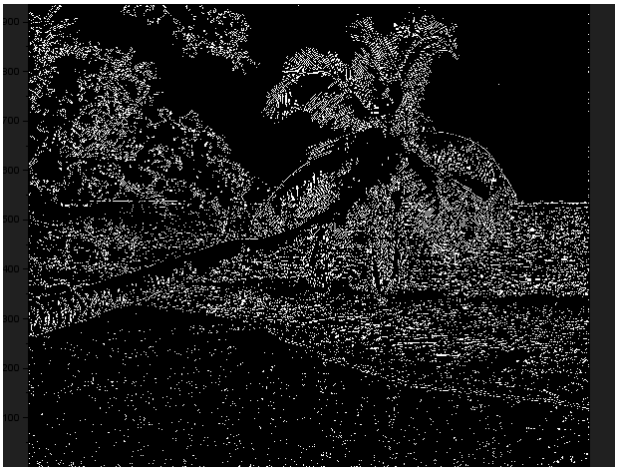


Image initial



Avec contours_p et W_gauss_2D

Image initial



Avec contours_p et W_gauss_2D

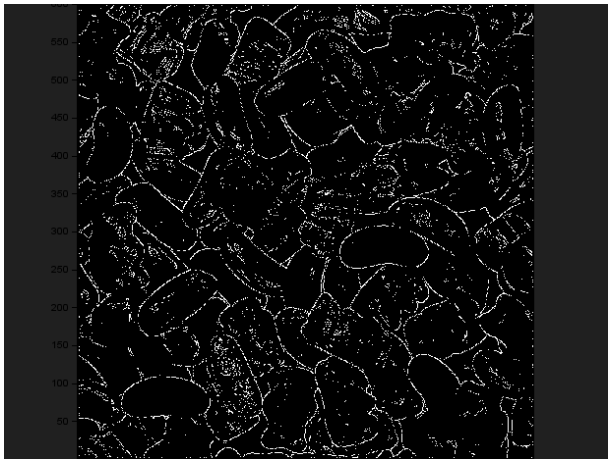
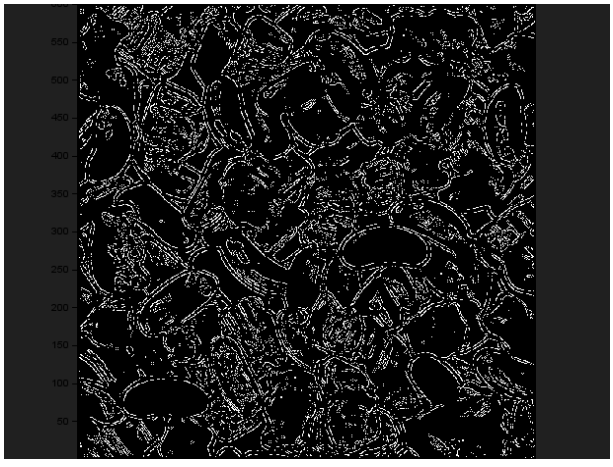


Image initial



Avec contours_p et W_gauss_2D
(Image 3)

Image initial
(Image 4)

Remarques:

- On a essayé afficher tous deux l'image initiale et l'image après avoir été appliquée aux fonctions contours_p et W_gauss_2D (exercice 4) avec la fonction contours_max.
    - Visuellement, on a un bien meilleur affichage en utilisant l'image initiale. Il y a plus de cohérence dans l'image, c'est-à-dire on arrive à mieux discerner les détails des contours de l'image initiale.
    - Par contre, suite à une application aux fonctions contours_p et W_gauss_2D, il y a une perte de la qualité d'image (comme expliqué dans l'exercice 4), ainsi, en appliquant cette image résultante à la fonction contours_max, cela entraîne une dégradation de qualité d'image qui devient subséquemment significative . On perd ainsi une grande partie des détails de l'image. Cela s'illustre plus clairement quand on compare l'image 3 et l'image 4 ci-dessus. L'image 4 comporte de bien plus de détails que l'image 3 mais est aussi visiblement bien plus bruitée. Les contours sont moins prominent dans l'image 3.

On va observer les 3 images suivantes de barque.bmp pour faire la comparaison de l'application de la fonction contours_max.
Les images ci-dessous correspondent à:

1. L'image (1) initiale après avoir été appliquée à contours_max
2. L'image (2) après avoir été appliquée aux fonctions contours_p et W_gauss_2D
3. L'image 2 appliquée à la fonction contours_max

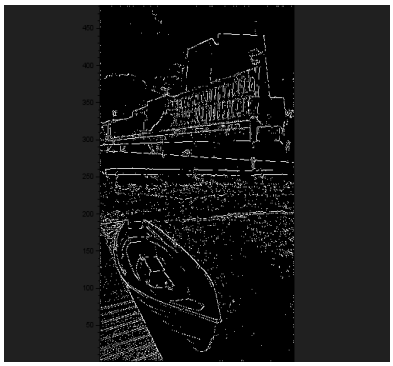


Image 1

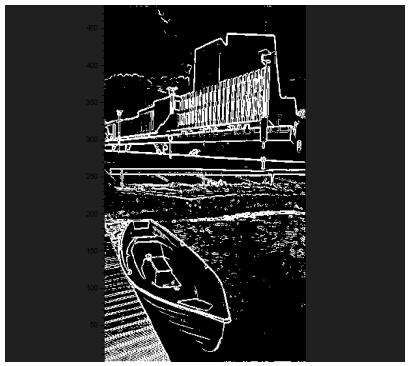


Image 2

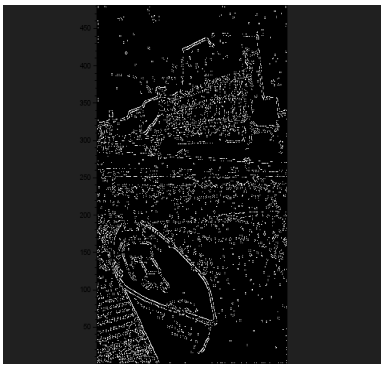


Image 3

- L'image 1 donne une bonne visualisation des contours, retenant bien les détails importants de l'image initiale.
- L'image 2 affiche bien les contours de l'image initiale et diminue aussi le bruit de l'image (comme expliqué dans l'exercice 4). Cependant on peut voir que les contours ne sont pas uniformes, c'est-à-dire qu'ils ne sont pas tous de la même épaisseur. On perd aussi une fraction des détails de l'image(bien que ce ne soit pas aussi prévalent que dans l'image 3), en raison du lissage de l'image. Cependant, on a un affichage bien plus cohérent que l'image 3.

- L’image 3 entraîne bien trop de perte d’information car premièrement, l’algorithme effectue une première dégradation de qualité avec le filtre gaussien et la fonction contours_p.
  Après contours_max, on trouve que l’affichage est bien trop inexacte par rapport à l’image initiale.

Pour conclure, on estime que l’image 1 donne une bien meilleure définition des contours :

1. On a pas trop de perte de qualité d’image.
2. On retient bien plus d’informations sur les pixels que dans l’image 3.
3. Les contours sont d’épaisseur uniforme, ainsi cela donne un meilleur affichage que l’image 2.