Séance 1 :

Exercice 1 :

- Imread('nom _de_image') : Permet de lire une image est de la transformer en matrice RGB
  Attention : cette matrice est une matrice RGB, donc composé de trois matrices :

La première décrit la valeur de tous les pixels suivant la composante rouge, la seconde suivant la composante verte et la dernière suivant la composante bleue !

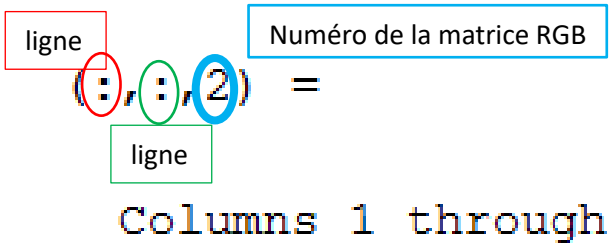
Exemple d'utilisation :

```
% test de image() et imread/ imshow:
I = imread('lena.jpg');
disp('imread nous affiche la matrice suivante:');
disp(I);
```

Et on récupère dans la console :

```
Command Window
    97    92    83    76    77    83    90    96   106    84   124   158   152   151   147   145   142   139   140   141   144   146
    97    91    82    75    76    82    89    96   103    82   123   158   153   153   150   148   144   141   142   143   146   148
    96    91    82    75    76    82    89    95   100    79   121   158   155   156   154   153   146   144   145   145   148   151
    95    88    80    74    75    81    92    97    97    75   120   155   155   158   156   156   149   148   147   147   151   155
    94    92    80    63    63    82    99   103    89    67   115   155   160   163   160   157   151   151   150   153   157   161
    94    89    78    63    64    83   102   102    88    67   114   156   158   160   160   158   154   153   152   156   159   164
    94    88    77    63    67    85   102   100    87    68   116   157   158   159   160   159   158   157   156   160   163   168
    94    87    76    64    70    88   101    96    86    69   119   159   157   157   160   162   161   161   160   163   167   171
    94    86    74    64    73    91   101    92    85    70   122   161   156   156   160   164   163   163   162   165   168   173
    94    85    73    65    76    94   101    88    83    70   124   163   156   154   160   167   163   162   161   165   168   172

  Columns 287 through 304
```

Preuve que l'on récupère 3 matrices : on peut voir dans la console :



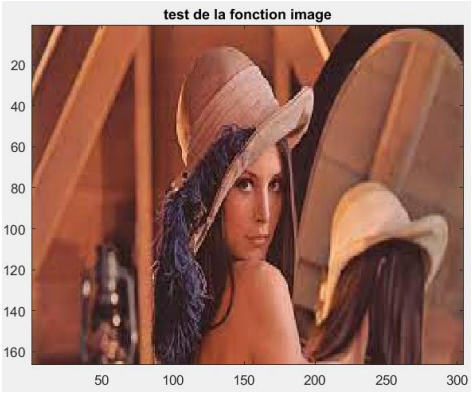
- Image(matrice) : permet d'afficher une image à partir d'une matrice

Exemple d'utilisation :

```
I = imread('lena.jpg');
```

```
figure(1)
image(I)
title ('test de la fonction image');
```

Et on récupère l'image :



- Imshow(matrice) : permet, à partir d'une matrice, d'afficher l'image associé.
  Exemple d'utilisation :

- Colormap : elle renvoie la palette de couleur utilisée par l'image sous la forme d'une matrice de 3 colonnes (RGB : donc colonne 1 : rouge, colonne 2 : vert et enfin colonne 3 : bleu). Chaque ligne de la matrice spécifie ici une couleur de la palette de couleurs.

Exemple :

```
%exercice 1:
matrice = imread('lena.jpg');
image(matrice);
colormap
```

Cela nous donne en console :

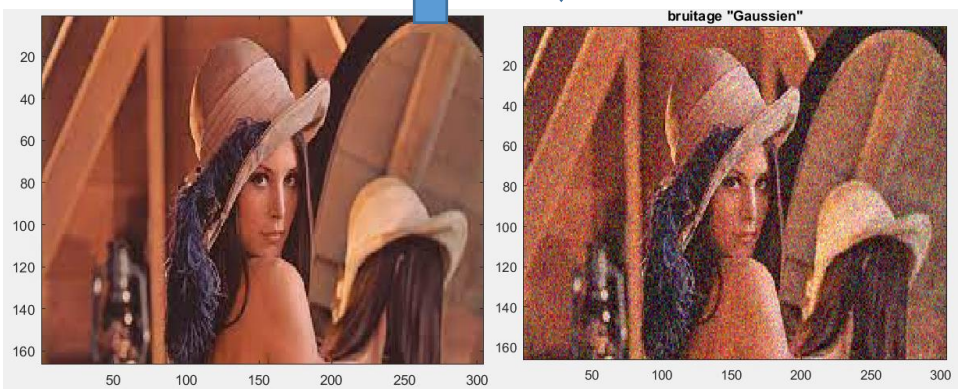
```
ans =

         0         0    1.0000
         0    0.0159    0.9921
         0    0.0317    0.9841
         0    0.0476    0.9762
         0    0.0635    0.9683
         0    0.0794    0.9603
         0    0.0952    0.9524
         0    0.1111    0.9444
```

…

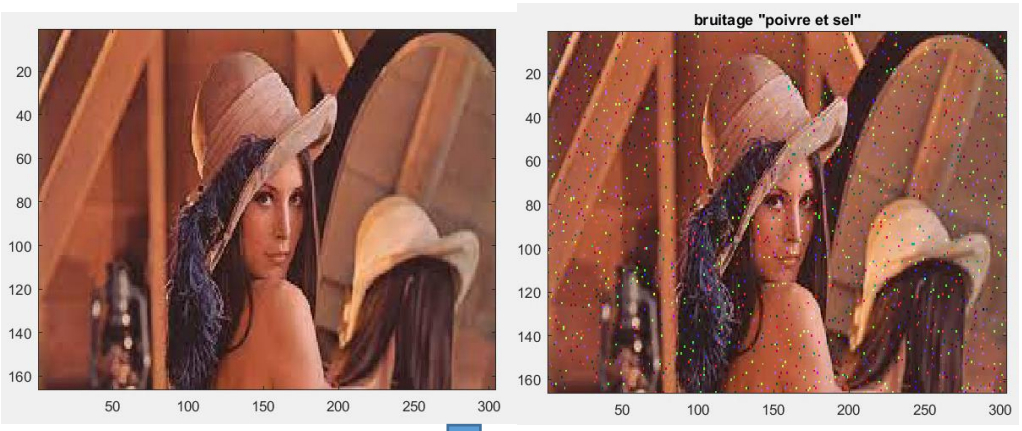
- Imnoise(matrice, méthode de bruit, pourcentage_de_bruit) : permet d'ajouter du bruit sur notre image, suivant la méthode choisit. Parmi les méthodes, on retrouve la méthode 'gaussian' qui fournit du bruit (pixel rouge, vert ou bleu) aléatoirement à l'image en suivant une probabilité Gaussienne suivant la position du pixel, ou encore la méthode 'salt & pepper', qui place des pixels RGB de manière aléatoire (cela touche par défaut 5% des pixels, d'après la datasheet de Matlab).

  Exemple :

  Méthode Gaussienne :



  Méthode poivre et sel :

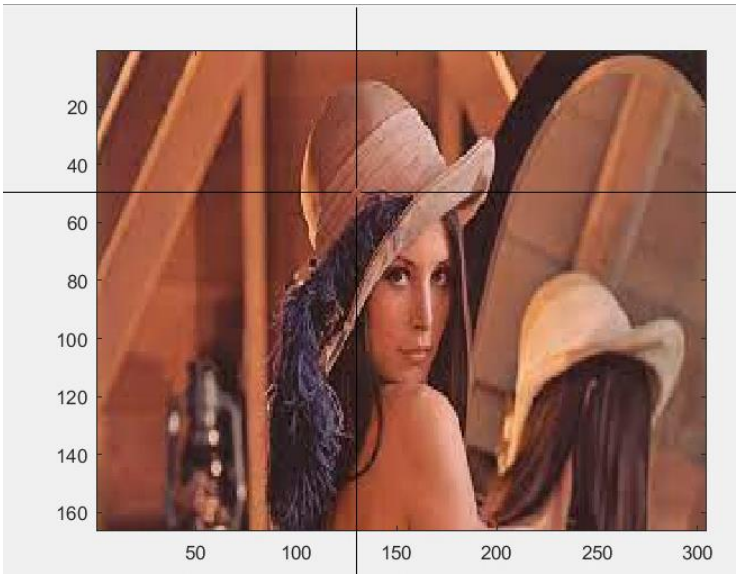


- Fft2(matrice) renvoie la transformée de Fourier de la matrice en argument. La sortie de la fonction sera également une matrice, de même dimension que X… La méthode utilisé es la méthode de transformée de Fourier rapide…
- Mean2(matrice) : renvoie la moyenne des composantes de notre matrice. Il faut donc le voir comme la valeur moyenne des pixels d'une image.
  Exemple :

```
 matrice = imread('lena.jpg');
mean2(matrice)


>> seance_1

ans =

   95.0137
```

95 est donc la valeur moyenne des pixels de l'image lena.jpg !

- Ginput permet de donner les coordonnées dans le repère de l'image, d'un point défini par le clic sur image de l'utilisateur :



- Im2bw(matrice) : prend en argument la matrice d'une image et nous donne l'image associé en noir et blanc :
  Attention ici on a une image en noir et blanc, et pas en niveau de gris, le pixel valent soit 0 soit 255 !

- Rgb2gray(matrice_type_RGB) nous renvoie une matrice qui donne une image en niveau de gris.
  Exemple :
  Grâce au code suivant :

```
figure(6)
matrice = imread('lena.jpg');
X = rgb2gray(matrice);
disp('matrice en niveau de gris');
disp(X);
imshow(X);
```

- Mat2gray(matrice) convertit une image en niveau de gris, sachant qu'il existe la variante Mat2gray(matrice, [x,y]) qui permet de choisir l'intervalle de valeurs que le pixel peut prendre.

Exercice 2 :

Tout d'abord, j'ai essayé de créer des rectangle de taille aléatoire, à des endroits aléatoires.

```
function [ Mat ] = dessin2polygones( m,n )
%définition d'un fond noir:
Mat = zeros(m,n);
% mis en place de polygone randoms:
% création d'un rectangle random dans la figure:
Zmat= randi([0,15],1,1);
for z = 0:Zmat(1,1)
    Mx = randi([1,m],1,2)
    My = randi([1,n],1,2)
    Mat(Mx(1,1):Mx(1,2),My(1,1):My(1,2))=255;
end
```

Tout d'abord, on crée une matrice de zéros, que l'on va remplir avec un nombre aléatoire de rectangle. On a donc besoin d'une boucle, qui va créer, un à un des rectangle de largeur et longueur aléatoire.

Voici le résultat :

D'ailleurs, l'exercice nous demande un fonction test qui nous dit si un point est dans un polynôme. Pour ce faire, j'ai donc regarder si suivant les coordonnées dans la matrice, la valeur du pixel est de 0 ou 1. Si on est à 255, le pixel est blanc donc on est dans un polygone…

```
function [ bool ] = appartenance( i,j, MAT )
    if MAT(i,j) == 255
        bool = true;%1 true
    else
        bool = false;%0 false
    end
end
```

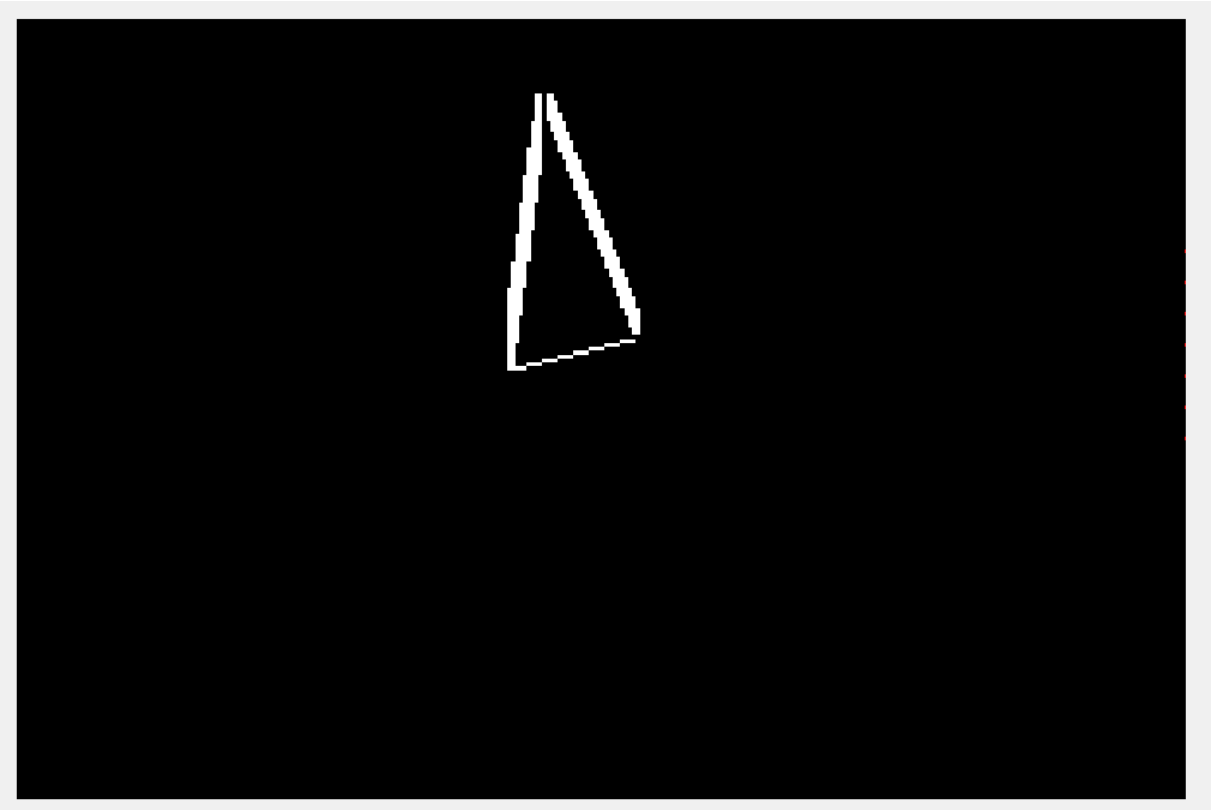
Cependant ici nous n'avons pas le contrôle du nombre de points du polygone…

Nous allons donc partir d'un cercle, dans l'image. Le centre est prit aléatoirement, le rayon également.
Ensuite il nous faut un nombre de points aléatoire sur le cercle. On va donc choisir les points suivants avec les formule trigo : j'ai besoin d'un module = rayon du cercle et d'un angle prit aléatoirement.
Pour la suite j'ai également besoin de trier les angle par ordre croissant puisque je vais relier les points entre eux. Comme je sais que 2 points peuvent former un segment, on a besoin de l'ordonnée à l'origine et de la pente et on fixe le pixel repondant à notre équation avec un pixel blanc.
Via cette méthode, j'arrive à obtenir des polygones aléatoire, dont le contour est blanc.

```
function [ mat ] = polygone_random( m,n )
%définition d'un fond noir:
mat = zeros(m,n);
%% 1) choix d'un centre:
coordonnee = randi([1,min(m,n)],2,1)
a = coordonnee(1,1);
b = coordonnee(2,1);
rayon = randi([1 min(min(coordonnee(1,1),
coordonnee(2,1)),min(m-coordonnee(1,1),n-
coordonnee(2,1)))],1,1)
r = rayon (1,1);
%% 2) choix de points aléatoires:
nb_de_pt = randi([3, 5],1,1);
angle = randi([1 100],nb_de_pt,1)* 2*pi/100;
tri_par_insertion(angle)
coordonnee_pts = zeros(nb_de_pt(1,1),2);
for i = 1:nb_de_pt(1,1)
```

```
    coordonnee_pts(i,1) = floor(a + r* cos(angle(i,1)));
    coordonnee_pts(i,2) = floor(b + r* sin(angle(i,1)));
end
disp(coordonnee_pts)
%% 3) remplissage:
    %1) on met en blanc les droites:
for i = 1:nb_de_pt(1,1)
    if i < nb_de_pt(1,1)
        coef_directeur = (coordonnee_pts(i+1,2)-
coordonnee_pts(i,2))/(coordonnee_pts(i+1,1)-
coordonnee_pts(i,1));
        ord_a_origin = coordonnee_pts(i+1,2)-
coef_directeur*coordonnee_pts(i+1,1);

        for a = min(coordonnee_pts(i,1),
coordonnee_pts(i+1,1))+1:  max(coordonnee_pts(i,1),
coordonnee_pts(i+1,1))-1
            for b = min(coordonnee_pts(i,2),
coordonnee_pts(i+1,2))+1:  max(coordonnee_pts(i,2),
coordonnee_pts(i+1,2))-1
                if abs(b - a * coef_directeur - ord_a_origin)
< 2
                    mat(a,b) = 1;
                end
            end
        end
    else
        %point1 par rapport au dernier pt!
        coef_directeur = (coordonnee_pts(1,2)-
coordonnee_pts(nb_de_pt(1,1),2))/(coordonnee_pts(1,1)-
coordonnee_pts(nb_de_pt(1,1),1));
        ord_a_origin = coordonnee_pts(1,2)-
coef_directeur*coordonnee_pts(1,1);
        for e = min(coordonnee_pts(nb_de_pt(1,1),1),
coordonnee_pts(1,1))+1:  max(coordonnee_pts(nb_de_pt(1,1),1),
coordonnee_pts(1,1))-1
            for r = min(coordonnee_pts(nb_de_pt(1,1),2),
coordonnee_pts(1,2))+1:  max(coordonnee_pts(nb_de_pt(1,1),2),
coordonnee_pts(1,2))-1
                if abs(r - e * coef_directeur - ord_a_origin)
< 2
                    mat(e,r) = 1;
                end
            end
        end
    end
end
end
```

Voici ce que ce code nous donne :



Exercice 3 :

Dans cet exercice, nous devons réaliser un programme Matlab, capable d'implémenter un filtre/ masque.

Tout d'abord, nous allons exprimer un masque sous la forme d'une matrice 3x3.

Donc il suffit de parcourir l'image, pixel après pixel et affecter le masque à tous les pixels, l'algorithme sera sous la forme d'une double boucle for.

Deux méthodes sont ici possible : je peux soit ne pas prendre en compte le contour de mon image et donc crée un contour en noir sur mon image de sortie. Ce code est donnée si dessous. L'avantage de ce code est que la taille de l'image de sortie est la même que la taille de l'image de départ !

```
function [ out ] = filtrage( image , filtre)
    M_in = imread(image);
    [m,n,z] = size(M_in);
    % on va parcourir tous les pixels
    for i = 2:m-1
        for j = 2:n-1
            % on applique le filtre:
            out(i,j)= filtre(1,1)*M_in(i-1,j-1)+...
            filtre(1,2)*M_in(i-1,j)+...
            filtre(1,3)*M_in(i-1,j+1)+...
            filtre(2,1)*M_in(i,j-1)+...
            filtre(2,2)*M_in(i,j)+...
            filtre(2,3)*M_in(i,j+1)+...
            filtre(3,1)*M_in(i+1,j-1)+...
            filtre(3,2)*M_in(i+1,j)+...
            filtre(3,3)*M_in(i+1,j+1);
        end
    end
    %je choisis de mettre les contours de l'image en noir
    % car les pixels du contours ne sont pas concerné par le filtre...
    for z = 1:m
        out(z,1) = 0;
    end
    for a = 1:n
        out(m,a) = 0;
    end
end
```

Ou alors, je peux tout simplement ignorer le contour de mon image. J'aurais donc une image de sortie de taille inférieur à celle d'origine (on retire donc 2 pixels (contour en haut et en bas ou à droite + à gauche)). Le code est celui-ci-dessous :

```
function [ out ] = filtrage( image , filtre)
    M_in = imread(image);
    [m,n,z] = size(M_in);
    % on va parcourir tous les pixels
    for i = 2:m-1
        for j = 2:n-1
            % on applique le filtre:
            %soit je retire la premiere ligne et colone : i-1
et j-1
            out(i-1,j-1)= filtre(1,1)*M_in(i-1,j-1)+...
            filtre(1,2)*M_in(i-1,j)+...
            filtre(1,3)*M_in(i-1,j+1)+...
            filtre(2,1)*M_in(i,j-1)+...
            filtre(2,2)*M_in(i,j)+...
            filtre(2,3)*M_in(i,j+1)+...
            filtre(3,1)*M_in(i+1,j-1)+...
            filtre(3,2)*M_in(i+1,j)+...
            filtre(3,3)*M_in(i+1,j+1);
        end
```

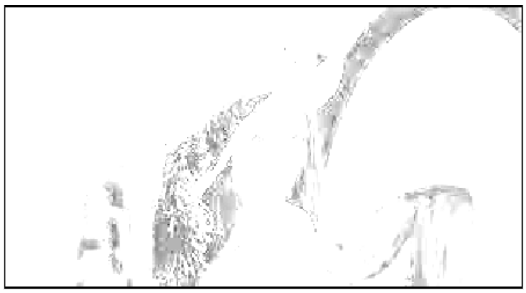
```
    end
end
```

Test de notre fonction sur l'image de Lena :

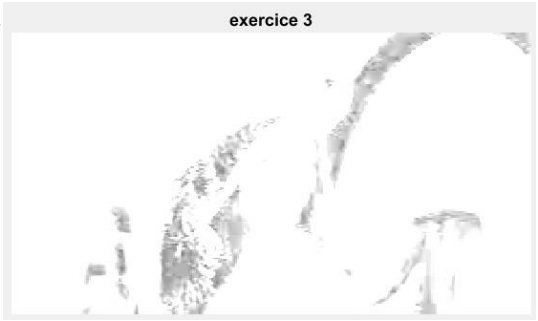
Si on applique un filtre Laplacien :

(premier code avec contour noir)

Laplacien



(second code sans contour)



Observation : Seuls les pixels noir semblent ici représenter…

Appliquons maintenant un filtre moyenneur :

Ce filtre, devant effectuer une moyenne de la valeur des pixels, sera une matrice définit ci-dessous :

$$Moyennneur = \begin{matrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{matrix}$$

On obtient ainsi :

Observation : Tout d'abord, la photo est en noir et blanc, ensuite la photo semble être plus flou (ce qui est totalement normal, car un moyenneur est un filtre passe bas)…

Exercice 4 :

- Filter2 (H,X) filtre de manière bidirectionnel une matrice d'entier X , avec un filtre H…
  ATTENTION : la matrice de l'image doit être une matrice en niveau de gris !
  (voir exemple suivant …)
- Fspecial(type) renvoie la matrice composant un filtre suivant le type rentré en argument. Par exemple : avec le filtre de Sobel :

```
Sobel =  fspecial('sobel')
```

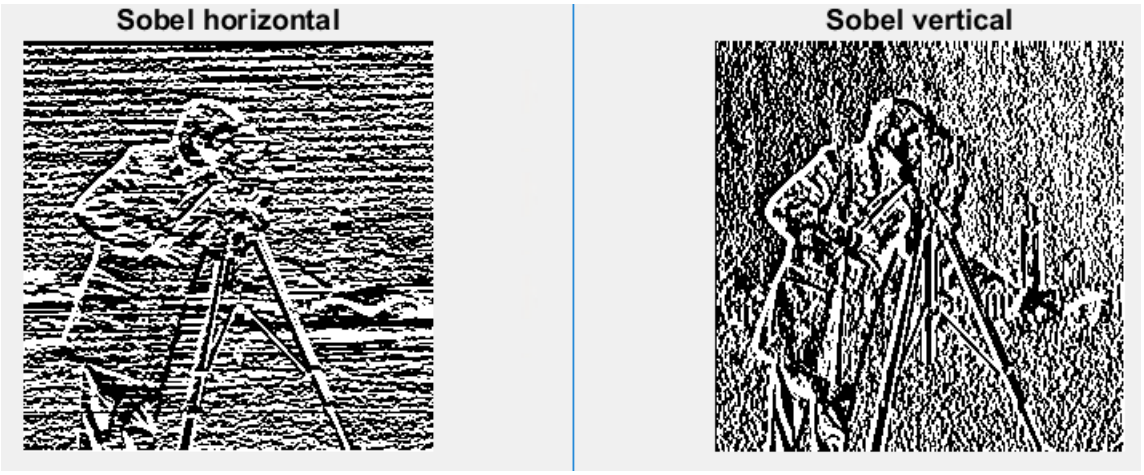
On retrouve alors dans la console :

```
Sobel =

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

Ce qui est vérifié, car nous avons la même matrice dans le cours !
(remarque, il s'agit de gradient horizontal… Mais ceci n'est pas un problème puisque pour avoir une matrice de radient vertical, il suffira de faire sa transposée !)

En appliquant filter2 sur l'image 'cameraman.tif', on obtient ceci :

```
mat = filter2(Sobel, imread('cameraman.tif'));
figure(8)
imshow(mat)
title('Sobel horizontal')
mat2 = filter2(Sobel', imread('cameraman.tif'));
figure(9)
imshow(mat2)
title('Sobel vertical')
```



Ceci nous montre alors les contours de l'image, ce qui est cohérent avec la définitions de l'utilisation des filtres de Sobel.

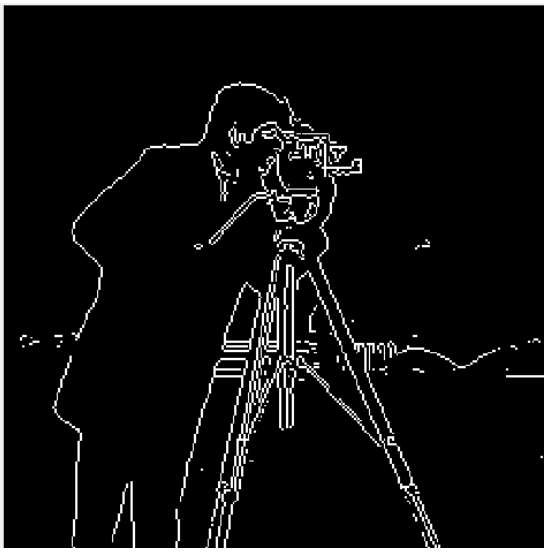
Tout d'abord, on sait que le filtre de Sobel est utilisé pour la détection de contour (filtre passe haut), le filtre de Sobel est donc un filtre qui détecte donc les bords horizontaux (via le gradient horizontal) et ceux vertical (via le gradient vertical) . filter2 filtre donc notre image (qui est en niveau de gris initialement) une fois horizontalement puis une fois verticalement suivant si on a appliqué une Sobel horizontal ou vertical !

Séance 2 :

Exercice 5 :

- Edge(matrice) : prend en argument une matrice d'une image binaire I et retourne une image ou les bords sont en niveau de gris et le reste en noir .
  Test de cette fonction :

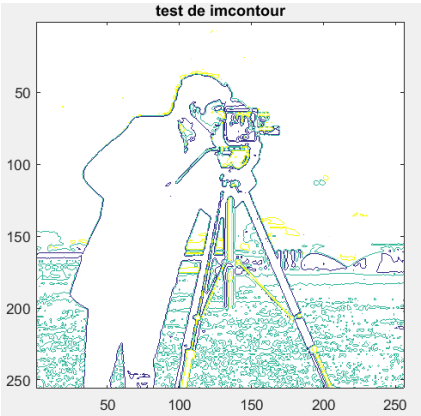
```
%% exercice 5:
% test edge:
mat = imread('cameraman.tif');
edge(mat)
```



On obtient alors ceci : on remarque que les bords de l'image sont mis en avant (blanc) et le reste est en noir, comme prévu. Cette fonction applique, par défaut un filtre de Sobel. Mais on peut choisir le filtre à appliquer en écrivant edge(mat, X) où
X = « `Prewitt » ou « Roberts » ou encore « log » !`

- imcontour(matrice, nombre_de_contour_equidistant) : elle dessine le contour de l'image en niveau de gris, le reste de l'image est en blanc.
  Par exemple :

```
figure(11)
imcontour(mat,3);
title('test de imcontour');
```



On peut également mettre un argument entier N: imcontour(matrice,N), N est le nombre de contour équidistant dans l'image.

- Grayslice(matricce, nombre_de_seuil) nous renvoie, à partir d'une matrice d'image en niveau de gris, une image indexé en utilisant une approche de seuil en plusieurs niveau.

```
mat = imread('cameraman.tif');
grayslice(mat,13)
```

Ici, on a testé la fonction grayslice avec une valeur de seuil = 13, on remarque que l'image est composé de 13 couleurs différentes. Si on augmente la valeur N, on n'observe aucun changement, mais si on la diminue fortement, on perd en information : si on ne mets que N=2, il n'y a que 2 couleurs, on a donc perdu les informations contenu dans les niveaux de gris.

Exercice 6 :

Dans cet exercice, on veut extraire les contours de la zone verte :
Nous allons donc appliquer la méthode suivante :
Tout d'abord, nous allons « supprimer » toute les composantes non vertes :
Nous allons donc créer une nouvelle image, où les pixels ayant des paramètres RGB trop loin d'un pixel vert (que j'ai choisi via impixel(…)). Pour ce faire, j'ai utilisé la distance euclidienne :

```
if (mat(i,j,1)-RGB(1))^2 + (mat(i,j,2)-RGB(2))^2 + (mat(i,j,3)-RGB(3)^2 ) > epsilon
```

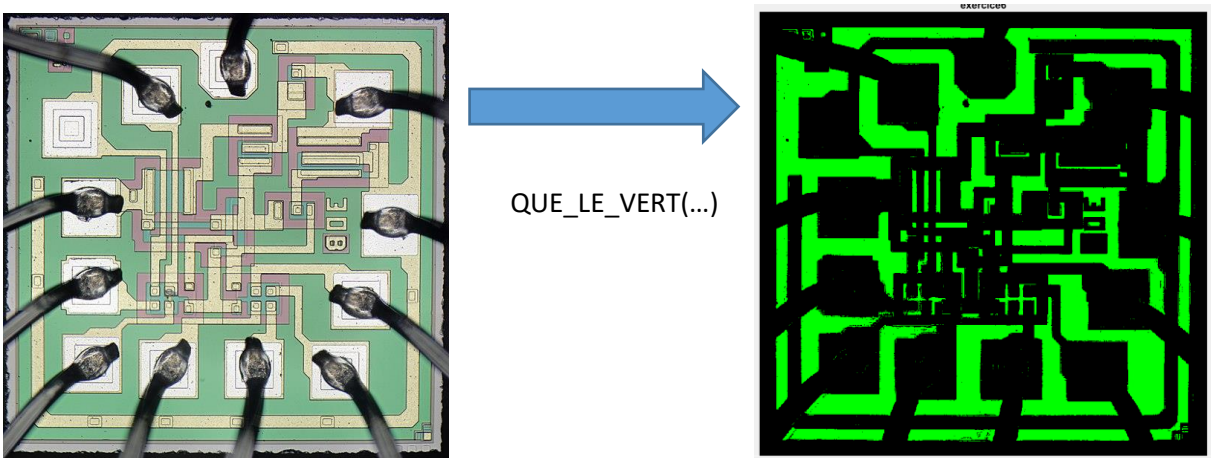
Ici si on est trop loin, on met le pixel à 0 (noir). Et on va parcourir toute l'image.
Afin de faciliter le travail avec les filtres par la suite, j'ai choisi de remplacer la couleur des pixels verts par la couleur maximal du vert :

```
matrice(i,j,1) = 0;
matrice(i,j,2) = 255;
matrice(i,j,3) = 0;
```

Voici donc le code de ma fonction :

```
function [ matrice ] = que_le_vert( RGB, image )
mat = imread(image);
[n,m] = size(mat);
m = m/3;
matrice = zeros(n,m);
epsilon = 80;
for i =22: n-22
    for j=22:m-22
        if (mat(i,j,1)-RGB(1))^2 + (mat(i,j,2)-RGB(2))^2 + (mat(i,j,3)-RGB(3)^2 ) >
epsilon
            matrice(i,j,1) = 0;
            matrice(i,j,2) = 0;
            matrice(i,j,3) = 0;
        else
            matrice(i,j,1) = 0;
            matrice(i,j,2) = 255;
            matrice(i,j,3) = 0;
        end
        if mat(i,j,2) < 150
            matrice(i,j,1) = 0;
            matrice(i,j,2) = 0;
            matrice(i,j,3) = 0;
        end
% Ajout de ligne de code afin de bien sélectionner tout le vert !
        if mat(i,j,1) >127 && mat(i,j,1) < 140 && mat(i,j,2) >180 && mat(i,j,2) <
195 && mat(i,j,3) >140 && mat(i,j,3) < 160
            matrice(i,j,1) = 0;
            matrice(i,j,2) = 255;
            matrice(i,j,3) = 0;
        end
    end
end
end
```

On obtient donc ce changement :



Maintenant que l'on a cette image, il faut appliquer des filtres de Sobel, afin d'obtenir les contours !

J'ai choisi d'utiliser Sobel via la fonction filter2 :

```
figure(14)

I = im2bw(matrice);
%imshow(I);
saveas(figure(14),'sobel_ex2.jpeg');

figure(15);

sobel = fspecial('sobel');
matrice_contour = filter2(sobel,I);
imshow(matrice_contour)
```

J'ai donc utilisé la fonction im2bw qui a passé mon image en vert, en une image en noir et blanc.

Par la suite, j' ai utilisé Sobel (gradient horizontal, qui va voir les droite horizontales) puis Sobel vertical. De cette manière, je vais pouvoir observer mon contour de la zone verte avec le plus de netteté possible.

Pour ce faire, j'ai créer une nouvelle matrice via l'opérateur logique « OU » :

```
matrice_contour = filter2(sobel,I);
imshow(matrice_contour)
hold on
matrice_contour2 = filter2(sobel',I);
[n,m] = size(matrice_contour);
matrice_final = zeros(n,m/3);
for i = 1:n
    for j = 1: m/3
        matrice_final(i,j) = matrice_contour(i,j)
|matrice_contour2(i,j) ;
    end
end
imshow(matrice_final)
```

Ainsi on obtient l'image suivante :