Analyse d'images TP 1
Bases

17 septembre 2022

Table des matières

1 Des images aux nombres et des nombres aux images

L'information essentielle relative aux images numériques, c'est à dire les couleurs ou niveaux de gris de l'image, est essentiellement contenue dans des tableaux de *nombres entiers*. Nous allons utiliser le logiciel **MATLAB** au cours de cet enseignement pour examiner puis manipuler ce contenu *numérique* des images. On va pour commencer écrire un programme qui lit le contenu d'une image (nombres) et permet de la visualiser. Ouvrez le logiciel **MATLAB**. Dans son interface, vous repèrerez une sous-fenêtre **Éditeur**, qui permet de saisir du texte. Tapez les ligne suivantes, qui correspond à une série de commandes successives qu'**MATLAB** exécutera :

```
close all
clear all
clc
A=imread('andromeda.bmp');
imshow(A)
```

L'icône représentant une flèche verte permet alors d'exécuter ces lignes de commandes.
On vous demandera lors de la première exécution d'enregistrer le fichier sous un nom de votre choix, suivi du suffixe `.m`. Choisissez par exemple `exemple_1_1.m` (premier exercice du TP1, pour vous aider à les retrouver par la suite), que vous mettrez dans un dossier **Analyse d'images** dans votre dossier **Documents** par exemple. Faites de ce dossier votre *dossier courant* (celui dans lequel **MATLAB** travaille) en le sélectionnant dans la zone en haut de l'interface **MATLAB**. Vous devez voir apparaître votre programme dans la fenêtre **Fichiers** à gauche.
Après la sauvegarde, les commandes contenues dans le fichier sont exécutées. Vous pourrez de la même manière créer et exécuter tous vos autres fichiers pour les autres exercices [1].
Commentons ce programme. Les commandes `close all` et `clear all` permettent pour la première de fermer toutes les fenêtres graphiques éventuellement déjà ouvertes dans **MATLAB** et pour la seconde d'effacer toutes les variables définies auparavant : on initialise proprement notre programme. Tant que vous n'avez pas tapé `clear all`, toutes les variables que vous définissez restent en mémoire, ce qui peut créer des confusions si vous réutilisez les mêmes noms de variables. La commande `imread` lit le contenu le plus important de l'image, c'est-à-dire les nombres qui décrivent les valeurs des pixels [2], et met le résultat dans un variable `A`. Ensuite `imshow` montre l'image correspondante au contenu de `A` dans l'interface graphique de **MATLAB**. Vous venez de créer et d'exécuter votre premier programme **MATLAB**.

1.1 Pixels et valeurs

Quelle est la structure de `A` ? Ouvrez en parallèle le fichier `andromeda.bmp` dans un logiciel de photos tel que **GIMP**. Tapez la commande `size(A)` dans **MATLAB** . Que signifient selon vous les valeurs renvoyées par cette commande au vu des informations données dans **GIMP** ? Essayez `size(A,1)` et `size(A,2)`. Interprétez. Tapez ensuite dans l'interface de commande **MATLAB** les commandes suivantes [3] :

```
>>A(100,100,1)
>>A(100,100,2);
>>A(100,100,3)
```

Ces commandes affichent trois valeurs associées au pixel d'indice (100,100) - centième ligne, centième colonne de l'image [4]. De quel type de codage des couleurs s'agit-il ? Dans **GIMP**, ouvrez la fenêtre ancrable **Pointeur**. Parcourez l'image dans la zone du pixel (100,100). Comment relier les infos données par cette fenêtre à ce que vous a donné **MATLAB** plus haut pour le pixel (100,100) ? Avez-vous repéré le décalage de numéroation des pixels entre les deux logiciels ? Quel est l'indice du pixel en haut à gauche de l'image dans **GIMP** ?
Quelle est le *type* de nombre utilisés par **MATLAB** pour représenter les intensités de l'image en mémoire. Pour le savoir taper :

```
>class(A)
```

Le type `uint8` correspond à des *entiers* codés sur *8 bits* (1 octet), *non signés* ('unsigned'). ce sont des valeurs entières entre 0 et 255.
Ces niveaux sont jugés suffisants pour obtenir des dégradés d'intensités perçus comme *continus* entre le noir et le blanc. Ce codage est encore dominant dans le monde de l'image numérique, malgré l'arrivée d'images codées sur 10 à 12 *bits* aujourd'hui.
Une remarque sur les problèmes associés au codage des images : on aura souvent besoin dans la suite de manipulations algébriques des nombres qui codent les intensités de gris ou de couleur, manipulations qui ont pour résultat qu'on ne tombe pas sur des valeurs entières [0,...,255]. Lorsque nous faisons ces manipulations, nous avons intérêt à ce que *dans les calculs intermédiaires* **MATLAB** considère toujours par défaut les nombres comme des nombres à virgule (au cas ou nous voudrions faire des additions -sur 8 bits 255+255=255 !- ou des divisions - de même 125/255 = 0 !). Pour ce faire, après avoir lu une image dans

1. Vous pouvez aussi exécuter les commandes en les tapant dans la **Fenêtre d'exécution** d'**MATLAB** les unes après les autres, et validées par **Entrée**. On reconnaît cette fenêtre au petit prompt (signe ») devant lequel vous tapez les commandes. C'est ici également que vont apparaître certains résultats numériques que nous choisirons d'afficher.
2. Et pas le contenu des *en-têtes* de l'image qui peuvent contenir d'importantes informations supplémentaires selon le format. Voir les formats d'image plus loin. .
3. Comme dit plus haut, chaque commande dans l'interface doit être exécutée en tapant **Entrée**.
4. Au passage comparez le résultat avec et sans point-virgule !

une variable `A` par exemple, faire `A=double(A)` qui a juste pour effet de lui dire de considérer les valeurs comme des nombres réels. Cependant, après les calculs, on doit retrouver, si on a pour but de créér une image, des nombres entiers *au moment de sauver l'image ou de l'afficher avec* `imshow()`, il faut *absolument* lui dire de les *reconvertir* en *entiers* entre 0 et 255. Cela se fait avant la sauvegarde ou l'affichage via la commande `B=uint8(B);` (si B est votre résultat après manipulations de A) qui dit de remettre les nombres au format d'entiers entre 0 et 255. Gardez cette remarque présente à l'esprit, c'est une source de nombreux bugs triviaux !

1.2 Modification des valeurs

On a maintenant compris que A est un tableau à trois coordonnées : le premier indice est la *ligne* du pixel dans l'image, le second sa *colonne* dans l'image, et le troisième le *canal* (pour utiliser le vocabulaire de **GIMP**). On parle dans **MATLAB** de *couche* : la première couche `A(:,:,1)` contient le *rouge*, `A(:,:,2)` le *vert* et `A(:,:,3)` le *bleu*. Tapez ensuite dans la fenêtre d'exécution d'**MATLAB** les commandes :

```
>A(100,100,1)=0;
>A(100,100,2)=0;
>A(100,100,3)=0;
```

Puis :

```
> imwrite(A,'andromeda_modifiee.bmp')};
```

Cette commande crée dans le dossier courant d'**MATLAB** un fichier image `andromeda_modifiee.bmp` de format **bitmap** (suffixe `.bmp` automatiquement reconnu [5]) contenant l'image `A` maintenant modifiée. Ouvrez ce fichier avec **GIMP**. Allez chercher le pixel que vous avez modifié. Résultat ?

1.3 Copie simple d'une image

Créez le programme suivant (que vous sauverez comme `exemple_1_2.m`) puis exécutez-le comme le précédent dans **MATLAB** :

```
close all
clear all
A=imread('andromeda.bmp');
for i=1:size(A,1)
        for j=1:size(A,2)
                B(i,j,1)=A(i,j,1);
                B(i,j,2)=A(i,j,2);
                B(i,j,3)=A(i,j,3);
        end
end
imshow(B)
imwrite(B,'andromeda\_copiee.bmp');
```

Que fait selon vous ce programme ? À l'aide des connaissances que vous venez d'acquérir, créez une copie de l'image dans laquelle vous mettrez *tous les pixels* aux valeurs (R,V,B)=(255,255,255) [Exercice 1.1].
Syntaxe simplifiée : **MATLAB** vous permet d'éviter les boucles `for i=1:size(A,1) for j=1:size(A,2) ... end end` qui permettent de parcourir explicitement tous les pixels de l'image. Il suffit d'écrire

```
close all
clear all
A=imread('andromeda.bmp');
B(1:size(A,1),1:size(A,2),1:3)=A(1:size(A,1),1:size(A,2),1:3);
imshow(B)
imwrite(B,'andromeda_copiee.bmp');
```

5. Une remarque au passage : nous allons travailler pour l'instant *uniquement au format* **bitmap** (.bmp), car c'est le plus *simple* : en particulier il ne fait aucune manipulation des valeurs de pixels (de type compression ou autre). Le même exemple n'aurait pas fonctionné avec le format `jpeg`, qui modifie toutes les valeurs de pixels lors de la compression.

La syntaxe `1:size(A,1)` signifie : pour *tous* les valeurs de l'indice `i` de 1 à `size(A,1)` (c'est à dire au nombre de lignes de `A`). On remplit en une seule ligne de commande la matrice `B` de toutes les valeurs de `A`.
Application : Essayez le code :

```
close all
clear all
A=imread('andromeda.bmp');
B(1:size(A,1),1:size(A,2),1:3)=A(1:size(A,1),1:size(A,2),1:3);
B(10:20,10:20,1:3)=255;
```

Après avoir compris ce qui se passe, créez par un programme analogue une copie de l'image dans laquelle vous avez mis à (70,100,125) un carré de pixels 100x100, dont le coin supérieur gauche est le pixel (50,50) [Exercice 1.2].
Et maintenant même exercice en mettant le même carré en *rouge* [Exercice 1.3]. On voit qu'avec **MATLAB** on peut très facilement manipuler les trois *canaux* de couleur.
Exercices d'application : Comment afficher dans **MATLAB** une image ne contenant *que le canal rouge* de `andromeda.bmp` [Exercice 1.4] ? Comment *mettre les valeurs du rouge dans le canal vert* et réciproquement [Exercice 1.5] ? Comment *permuter circulairement le trois canaux* (rouge dans vert, vert dans bleu, bleu dans rouge) [Exercice 1.6] ?

1.4 Créations de mires de couleurs

Dans la suite de notre travail, nous aurons besoin de créer des *mires de couleur*. Exécutez dans **MATLAB** le code suivant :

```
close all
clear all
for i=1:100
        for j=1:200
                B(i,j,1)=0;
                B(i,j,2)=0;
                B(i,j,3)=255;
        end
        for j=201:400
                B(i,j,1)=0;
                B(i,j,2)=255;
                B(i,j,3)=0;
        end
end
imshow(uint8(B));
```

Vous noterez le passage en `uint8` forcé dans la dernière ligne. Ici, `B` n'est *créée* à partir d'une matrice image, et donc n'est pas par défaut de type `uint8`. Il faut donc avant d'afficher transformer son type. Proposez une version plus courte de ce programme qui évite les boucles `for` [Exercice 1.7]. On utilisera les fonctions `zeros` ou `ones`, qu'on découvrira grâce à l'aide (`help zeros` en ligne de commande). Écrire un programme qui permette de créer la mire des drapeaux français l'image 1 [Exercice 1.8], puis de Papouasie-Nouvelle Guinée [Exercice 1.9], et enfin du Japon [Exercice 1.10]. Les deux derniers exemples demanderont l'utilisation de la structure `if`, à découvrir dans l'aide.

FIGURE 1 – Les drapeaux de la France, de la Papouasie-Nouvelle Guinée et du Japon

1.5 Images noir et blanc

Leur codage se fait simplement sous forme d'une seule couche de pixels, dont l'intensité varie entre 0 et 255. Chaque couche d'une image couleur est donc une image noir et blanc.
Créez une mire qui corresponde à un *dégradé uniforme de gris* entre le noir et le blanc [Exercice 1.11]. On prendra exactement *dix niveaux uniformément répartis entre le noir pur et le blanc pur*. Faites ensuite la même chose pour un *dégradé du rouge pur au blanc*. Il faudra se remettre ici en mémoire les remarques faites sur les limites de l'utilisation d'un codage sur 8 bits...

2 Manipulations simples : création et combinaison de *masques*

2.1 Notion de masque

Un *masque* est une *image de sélection* qui s'applique aux pixels de l'image originale. Dans le cas le plus simple, cette image de sélection est à deux niveaux : 0 ou 1. Les pixels à 1 sont sélectionnés, et les autres seront ignorés. Donnons tout de suite un exemple : Tapez le code suivant dans **MATLAB** :

```
close all
clear all
A=imread('andromeda.bmp');
A=double(A);
for i=1:size(A,1)
        for j=1:size(A,2)
                Maskee(i,j)=0;
        end
end
for i=100:170
        for j=150:249
                Maskee(i,j)=1;
        end
end

AMaskee(:,:,1)=A(:,:,1).*Maskee;
AMaskee(:,:,2)=A(:,:,2).*Maskee;
AMaskee(:,:,3)=A(:,:,3).*Maskee;
imshow(uint8(AMaskee))
imwrite(uint8(AMaskee),'andromeda_maskee.bmp');
```

Comprenez-vous ce que fait ce programme ?
Un masque peut servir par exemple à remplir une partie d'une image par une couleur uniforme. Dans l'exemple, suivant créez une image monochrome B de la même taille que A. Rajoutez ensuite les lignes :

```
A_Maskee_bis(:,:,1)=A(:,:,1).*Maskee(:,:)+B(:,:,1).*(1-Maskee(:,:));
A_Maskee_bis(:,:,2)=A(:,:,2).*Maskee(:,:)+B(:,:,2).*(1-Maskee(:,:));
A_Maskee_bis(:,:,3)=A(:,:,3).*Maskee(:,:)+B(:,:,3).*(1-Maskee(:,:));
A_Maskee_bis=uint8(A_Maskee_bis);
imshow(A_Maskee_bis)
imwrite(A_Maskee_bis,'andromeda_maskee_bis.bmp');
```

Que se passe-t-il [Exercice 1.12] ?
Pouvez-vous maintenant prendre deux images de même taille (exemple : les deux tableaux de Signac de même taille dont les fichiers `Signac_1.bmp` et `Signac_2.bmp` vous sont fournis, et à l'aide d'un masque construire une image qui contienne le tableau 1 *au dessus de la diagonale* et le tableau 2 *au dessous* (voir aussi l'exemple du drapeau de la Papouasie Nouvelle Guinée) [Exercice 1.13].

Pouvez-vous, en revenant sur votre travail sur le drapeau du Japon, créer un masque circulaire au centre d'une image ? En l'appliquant à un des deux tableaux de Signac, faites apparaître la partie circulaire centrale correspondante du tableau, et remplissez l'extérieur par une couleur uniforme [Exercice 1.14].

On terminera cette section sur les masques simples en obtenant à partir du fichier `picasso.bmp` (figure 2 les deux images de la figure 3 [Exercice 1.15].

FIGURE 2 – L'image de départ `picasso.bmp`.

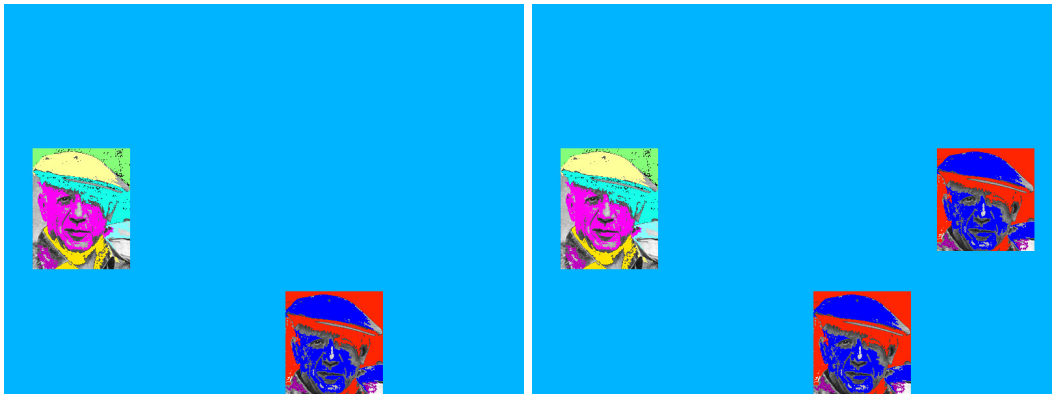

FIGURE 3 – Première et seconde variantes de `picasso.bmp`.

2.2 Les combinaisons de masques sous MATLAB

On a vu plus haut qu'un masque sous MATLAB était une image composée de zéros et de uns que l'on multiplie par l'image de départ pour masque justement une partie.
Créez sous **MATLAB** trois masques rectangulaires. Appliquez les à une image monochrome que vous créerez sous **MATLAB**.
Comment faire maintenant sous **MATLAB** des *intersections* ou des *unions* de masques [Exercice 1.16] ?

3 Analyse élémentaire d'une image

Objectif : Nous allons maintenant nous familiariser avec quelques techniques d'analyse d'image qui permettent de les modifier de façon contrôlée ou d'en extraire des informations. Il s'agit d'une première approche élémentaire de ce que nous ferons bien plus en détail dans les séances à venir.

3.1 Histogramme

Un type d'information important concerne des caractéristiques *statistiques* que l'on peut extraire des valeurs numériques des pixels de l'image. L'*histogramme* est la plus utilisée.
Pour une image en niveaux de gris, l'*histogramme* de l'image décrit la proportion des pixels possédant différentes intensités dans l'image. Dans **GIMP**, cette statistique est accessible via le menu

Couleurs→Informations→Histogramme.

On peut visualiser l'histogramme des valeurs des rouges, des verts, ou des bleus. Un histogramme global de *valeur* est également disponible. Il correspond à une transformation simple de l'image en niveaux de gris : on donne à chaque pixel le maximum des trois valeurs R, V, et B.
Ouvrez l'image `Signac_1.bmp` et observez les histogrammes.
Dans **MATLAB**, c'est la fonction `imhist` qui permet d'obtenir les mêmes résultats. Le code suivant nous donne les mêmes histogrammes que ceux observés dans **GIMP**.

```
close all
clear all
A=imread('andromeda.bmp');
R(:,:)=A(:,:,1);
V(:,:)=A(:,:,2);
B(:,:)=A(:,:,3);
Val=max(A,[],3);
[counts,x]=imhist(R,256)
plot(x,count,"linewidth",4)
[counts,x]=imhist(Val,256)
figure, plot(x,count,"linewidth",4)
```

Les histogrammes ont de nombreux usages, permettant par exemple de distinguer différentes parties de l'image, ou de corriger des couleurs déséquilibrées. Nous y reviendrons plus en détail au TP 4.

3.2 Les filtres et leurs usages

3.2.1 Les filtres de lissage et de débruitage

La présence de bruit dans une image est souvent un inconvénient que l'on cherche à éliminer. Il existe des méthodes de 'lissage' d'une image dont la plus célèbre est le filtre *gaussien*. Dans **GIMP**, ce filtre est disponible via **Filtres→Flou→Flou gaussien**. Il faut choisir la taille du noyau gaussien qui va être utilisé (prendre la valeur 5). On va voir dans un instant avec **MATLAB** ce que contient ce noyau. Observez le résultat sur l'image `eight_salt_pepper.bmp`.

Le code suivant sous **MATLAB** permet d'effectuer la même opération :

```
close all
clear all
A=imread('eight_salt_pepper.bmp');
ga=fspecial('gaussian',5,3)
B=imfilter(A,ga);
imshow(B)
```

Interprétez les valeurs contenues dans la variable `ga`. Les deux paramètres de `imfilter` décrivent respectivement la *taille du filtre*, c'est à dire la taille du carré qui va être appliqué en tout point de l'image, en pixels, et la *largeur de la gaussienne* du filtre, qui dit intuitivement la taille de la zone sur lesquelles les valeurs vont être 'moyennées'. Modifier la taille du noyau gaussien et observer le résultat. Et si on modifie la taille du filtre pour une taille de noyau donnée ?
Un des filtres les plus utiles pour éliminer le bruit est le filtre *médian*. Il remplace la valeur d'un pixel par la *médiane* (voir la définition sur wikipédia !) des valeurs des pixels dans une région carrée voisine, de taille définie (par défaut 3x3). Cette opération est *non-linéaire*, et ce filtre est donc un cas de *filtre non-linéaire*. Trouvez dans **GIMP** le filtre médian (idée : filtre Non Linéaire...). Appliquez-le à l'image `eight_salt_pepper.bmp`.
Sous **MATLAB**, on utilisera la fonction `medfilt2`, comme dans l'exemple suivant. Noter que l'on fournit à `medfilt2` la *taille* de la région carrée associée au filtre.

```
close all
clear all
A=imread('eight_salt_pepper.bmp');
C=medfilt2(A,[3 3]);
imshow(C)
```

3.2.2 Les filtres de gradient

Plusieurs filtres existent pour *détecter les contours*. Sous **GIMP** on les trouve dans **Filtres→Détection de bord→Contour**. Faire quelques essais sur l'image `eight_salt_pepper.bmp`.
Dans **MATLAB**, il va falloir préciser le filtre (et éventuellement les paramètres du filtre) qu'on veut utiliser. Le filtre le plus classique pour détecter les contours est le filtre de *Sobel*. Exécuter le code suivant :

```
close all
clear all
A=imread('eight_salt_pepper.bmp');
so=fspecial('sobel')
B=conv2(A,so);
imshow(uint8(B))
```

En examinant les valeurs de `so`, comprenez-vous comment le filtre agit ? Comparez au résultat en appliquant à A les valeurs -so, au lieu de so. Que se passe-t-il lors de l'application de la commande `uint8` comme perte d'information dans le code ci-dessus ? Comment feriez-vous pour détecter maintenant des bords *verticaux* [Exercice 1.17] ? Pour tester vos essais, vous créérez une image simple (dans **MATLAB**) composée d'un carré blanc sur fond noir. Vous devez faire apparaître uniquement le *bord* du carré [Exercice 1.18] ! Noter la commande `abs` qui fournit la *valeur absolue* d'un tableau.

3.3 Les opérations morphologiques

3.3.1 Érosion d'une image

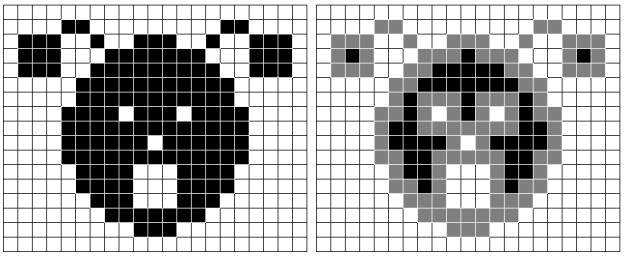

FIGURE 4 – Illustration d'un filtre d'érosion

L'*érosion* d'une image consiste à 'enlever des pixels aux bord des objets'. Pour cela, on définit un petit masque (souvent un petit carré ou un disque de rayon 2 ou 3, et on l'applique en tous les points de l'image. Dans le cas d'une image binaire, c'est le plus simple : si le petit masque est tout entier inclus dans l'image, on garde le pixel a 1, sinon on le considère comme extérieur à l'objet. L'image 4 montre le cas d'un masque carré 3x3 sur une image binaire.
Dans **MATLAB**, le petit code suivant effectue l'opération :

```
close all
clear all
A=imread('circles.png');
A=A(:,:,1); \%c'est une image NB, les trois canaux sont egaux
se=logical(ones(5,5));
B=imerode(A,se);
imshow(B,[])
```

Vous pouvez essayer avec d'autres formes de masque. Comment définiriez-vous un petit masque quasi-circulaire [Exercice 1.19] ?
Trouvez dans **GIMP** le menu permettant d'effectuer la même opération. Comparez les résultats.

3.4 Dilatation d'une image

La *dilatation* d'une image est l'opération en quelque sorte inverse : on rajoute des pixels à l'image en utilisant localement le petit masque qu'on a défini pour 'déborder' (figure 5 : Il est à noter que ces opérations

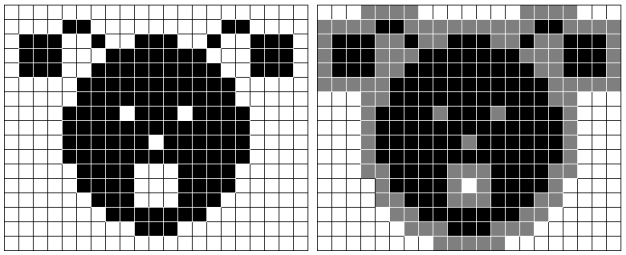

FIGURE 5 – Illustration d'un filtre de dilatation

sont *irréversibles*. On va voir plus en détail à la section suivante que dilater une image érodée ou éroder une image dilatée *ne ramène pas du tout l'image d'origine*. En utilisant la fonction `imdilate` dont l'utilisation sous **MATLAB** est identique à celle de `imerode`, faire un petit code qui effectue la dilatation à l'aide d'un masque carré 5x5 [Exercice 1.20]. Et sous **GIMP** ?

3.4.1 Ouverture et fermeture d'une image

Voyons ce qui se passe lorsqu'on érode, puis dilate une image. Cette séquence d'opération est appellée *ouverture*.La séquence inverse est appelée *fermeture*.
Intuitivement, l'ouverture permet d'éliminer des morceaux petits par rapport à la taille du masque choisi.

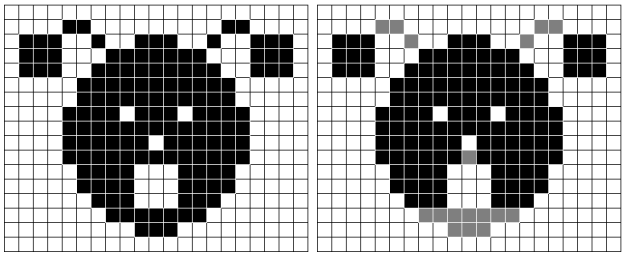

FIGURE 6 – Illustration de l'ouverture d'une image

La fermeture remplit des vides plus petits que la taille de ce même masque. Par exemple, l'ouverture permet de supprimer d'une image des objets plus petits que la taille du masque. Dans l'exemple suivant voyez ce qui se passe lorsque vous changez la taille du masque.

```
close all
clear all
A=imread('bubbles.bmp');
taille_masque=40;
se=logical(ones(taille_masque,taille_masque));
B=imopen(A,se);
imshow(B)
```

La commande `imclose` a la même syntaxe, pour l'opération de fermeture. Nous l'utiliserons dans l'exercice 1.21.

4 La segmentation

Segmenter une image, c'est séparer par une opération mathématique des *objets* d'intérêt dans cette image, en les isolant d'un fond. Le résultat de la segmentation est le plus souvent une *image binaire* dans laquelle les objets ont été mis en blanc, sur un fond noir.

4.1 Segmentation par seuillage d'intensité

Partons de l'image `holes.jpg` (voir figure 8) : il faut séparer les trous du reste de la plaque dans l'image. Les trous ont une intensité bien moins grande que l'image. En utilisant **GIMP** trouver les intensités R, V, B dans les trous.
Sous **MATLAB**, on va sélectionner les pixels dont l'intensité est plus petite qu'un *seuil* fixé par nous (on

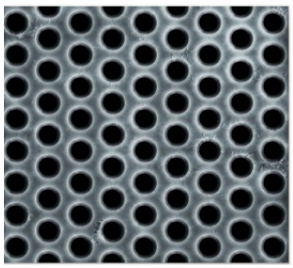

FIGURE 7 – L'image `holes.jpg`

appelle cela *segmentation par seuillage*) :

```
close all
clear all
A=imread('holes.jpg');
seuil=10;
B=A(:,:,1)<seuil;
imshow(B)
```

Vous travaillerez à trouver la bonne valeur de seuil pour bien isoler les trous. B est une image *binaire*, composée de zéros et de uns. C'est comme un masque, mais ici nous l'avons obtenue par seuillage des valeurs de A (en nous limitant à un seul canal de A, c'était le plus simple car il y avait un bon contraste entre trous et plaque sur tous les canaux).

4.2 Segmentation et opérations morphologiques

Souvent après une segmentation, il reste des petites parties dee l'image qui ne sont pas correctement segmentées. Pour le voir, faites une segmentation par seuillage sur l'image `holes_complete.bmp`. Il s'agit d'une version plus étendue de l'image précédente, avec une partie moins contrastée. Une fois la segmentation par seuillage faite avec un seuil de 5, pouvez-vous éliminer les imperfections grâce aux opérations morphologiques vues dans le paragraphe précédent ? Il vous faudra *remplir les trous avec une fermeture* puis *éliminer les pixels isolés avec une ouverture* [Exercice 1.21] !

5 Un exemple détaillé d'analyse d'image : comptage d'éléments

L'analyse d'image est le plus souvent utilisée afin d'extraire d'une image, ou d'une vidéo des *informations quantitatives*. Il s'agit souvent de reconnaître des éléments précis dans l'image (des fissures dans un matériau, des visages dans un foule, des cellules dans un tissu biologique), puis de les quantifier (connaître leur nombre, leur surface, leur densité).
Dans cet exemple, nous allons utiliser certaines des opérations que nous avons vues dans les parties précédentes pour compter automatiquement les grains de riz dans l'image suivante (`rice.png`) : Il faut d'abord *séparer les grains de riz du fond*. Essayez d'appliquer une méthode de segmentation simple par seuillage. Quel est le problème ?
Il nous faudrait donc d'abord *uniformiser le fond*. Pour cela utilisez une transformations morphologiques appropriée, suivie d'un éventuel flou gaussien. Le résultat devrait être juste le fond de l'image. Vous n'avez

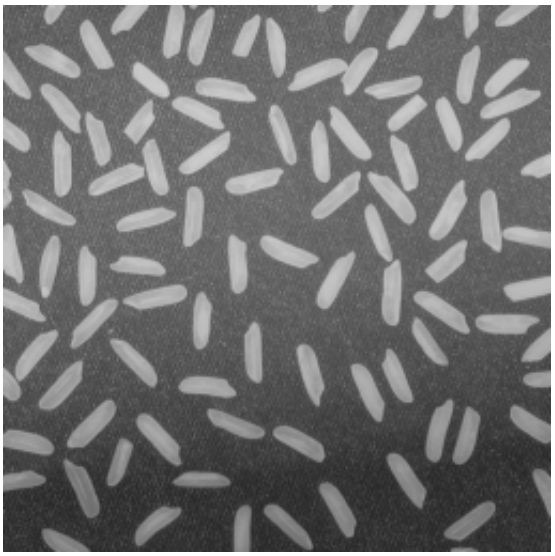

FIGURE 8 – L'image `rice.jpg`

plus alors qu'à *soustraire* ce fond à l'image de départ. Vous obtenez un fond à peu près uniforme [Exercice 1.22.a].
A ce stade, un seuillage simple devrait vous permettre de séparez les grains du fond. Si il reste quelques imperfections dans le masque obtenu, vous pouvez ensuite utiliser une opération morphologique et/ou l'application d'un filtre pour les guérir.
Ainsi, vous devez obtenir une image binaire ou les grains de riz sont en blanc et le fond en noir [Exercice 1.22.b].
Pour anlyser les objets dans une image noir et blanc, **MATLAB** vous donne accès à la commande `regionprops` (`region` `prop`*erties*`s`). En tapant la commande `data=regionprops(bw);` (si `bw` est votre image noir et blanc), vous obtenez une structure `data` qui a autant d'éléments qu'il y a de régions séparées dans votre image. Regardez combien de région cela fait dans votre cas. De plus, `data` contient aussi des informations supplémentaires quantitatives. Par exemple si vous tapez `aires=[data.Area];` le tableau `data` contient la liste des aires de vos grains de riz. Vous afficherez avec la commande `hist` l'*histogramme* des aires de ces régions. Repérez-vous sur cet histogramme les régions correspondant à des grains de riz mal séparés ? Combien y en a-t-il [Exercice 1.22.c] ? Il faudrait encore (1) retirer les grains *chevauchant le bord*, (2) revenir à l'image seuillée initiale en identifiant les objets sur celle ci pour avoir des aires correctes avant érosion !