

Objectif :

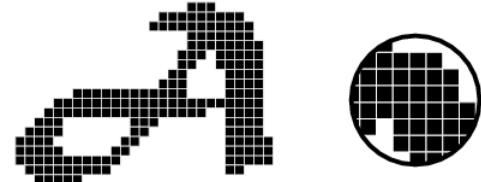
Cette partie aborde le codage et les transformations possibles opérables sur des images à partir de leur stockage sous format numérique. Il ne traite que du codage matriciel et passe sous silence le codage vectoriel.

Le codage vectoriel consiste à définir une image comme un ensemble de contours (courbes) dont on donne les propriétés mathématiques, ce type de codage rend l'image insensible au changement d'échelle, le fait de zoomer n'altère alors pas la qualité du rendu visuel. Ce codage est adapté aux images peu complexes avec des contours bien définis et sans trop de nuances de couleurs (logos, cartes, typographie, plans, etc). Les principaux formats de fichiers en stockage vectoriel sont : PostScript, PDF, SVG.

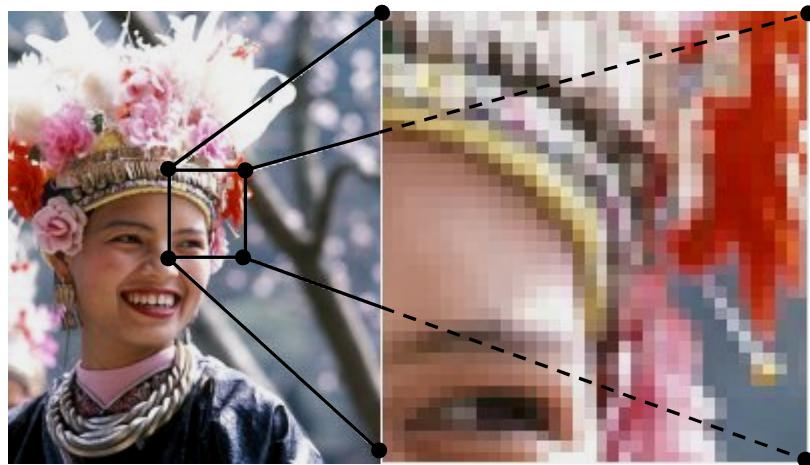
Le codage matriciel ("bitmap") consiste à définir une image comme une grille de points colorés (pixel = picture element) chaque point ayant une nuance de couleur. Ce type de stockage a pour effet de dégrader le rendu visuel lors d'un zoom important (effet de pixellisation). En revanche, il permet un stockage d'image complexe (photo) avec un grand nombre de nuance de couleur. Les principaux formats de fichiers en stockage matriciel sont : BMP(bitmap), TIFF(Tagged Image Format File), GIF, PNG, JPG(compression).



Mode vectoriel (et zoom sans pixellisation).

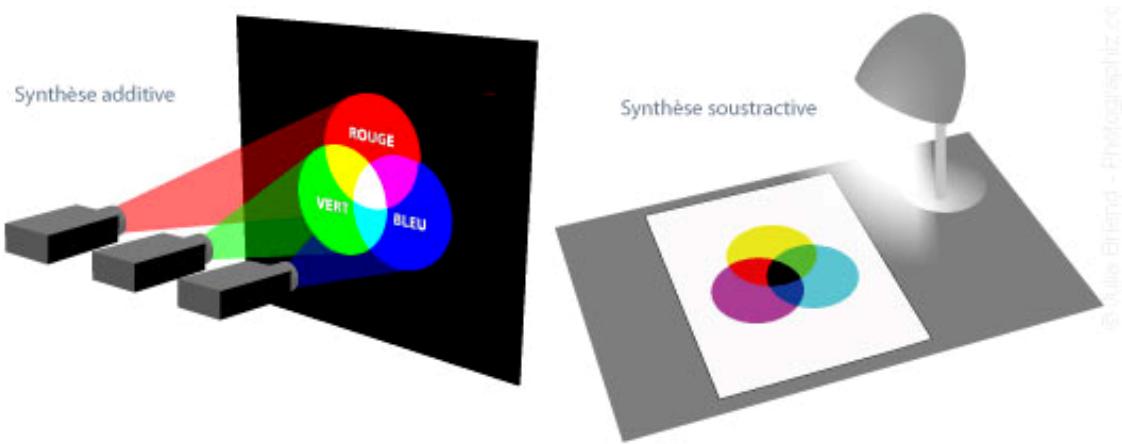


Mode matriciel (et zoom avec pixellisation).

Exemple :

Exemple d'image matricielle et pixellisation associée sur un zoom.

Colorimétrie :  
Notion de  
synthèse  
"additive" ou  
"soustractive":



Synthèse additive obtenue par superposition de faisceaux lumineux monochromatiques R, V, B

Synthèse soustractive obtenue par superposition de filtres colorés C, M, Y qui absorbent une source de lumière blanche.

## A - Vocabulaire associé aux images numériques en mode matriciel.

**Bitmap :** Une matrice de pixels ("picture element") est constituée d'un ensemble de points qui forment une image. Le pixel représente ainsi le plus petit élément constitutif d'une image numérique. A chaque pixel est associée une couleur parmi un échantillon (on parle de nuance de couleur ou de quantification ou encore d'échantillonnage couleur).

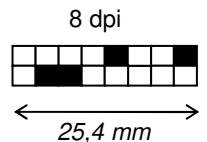
**Définition :** La définition d'une image numérique correspond au nombre de points qui la composent. Ainsi une image matricielle comportant 620 points en largeur (colonnes) par 450 point en hauteur (lignes) sera définie comme 620 x 450.  
*De même façon, la définition d'un écran est donnée par sa résolution horizontale × sa résolution verticale.*

**Résolution :** La résolution correspond au nombre de pixels par unité de longueur, les formats sont anglo-saxons :

- dpi (dot per inch) ou ppp (point par pouce), pour une image imprimée, (300 dpi équivaut à 300 points répartis sur 25.4 mm)
- ppi (point per inch) pour une image affichée sur un moniteur (écran)

La résolution perceptible à l'œil dépend de la distance de l'image.

Distance	20 cm	50 cm	1 m	3 m	10 m	20 m
Résolution perceptible (dpi)	400	150	80	25	8	4

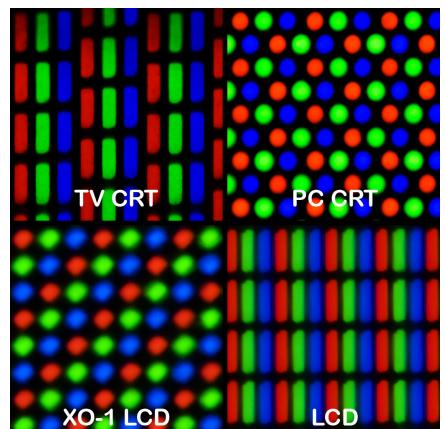


### Codage couleur :

Le codage de la couleur se fait en définissant une palette de couleur (nuancier), on code la palette sur un mot binaire (allocation mémoire de taille variable). La taille de la palette (nombre de couleur disponible) est fonction du nombre de combinaisons possibles selon la taille du mot binaire utilisé est :

- 1 bit pour le codage noir et blanc (0 = noir, 1 = blanc),
- 1 octet (8 bits) pour une palette comportant  $2^8 = 256$  nuances (couleur ou niveaux de gris), en niveaux de gris, la convention est 0 pour le noir et 255 pour le blanc,
- 3 octets (24 bits) pour une palette comportant  $2^{24}$  nuances, (soit 16 777 216 combinaisons possibles, on parle ainsi d'un échantillonnage 16 millions de couleurs, c'est l'échantillonnage classique d'un moniteur).

Dans ce cas, on code la couleur avec trois "composantes" (une par octet), de nombreuses normes co-existent, nous nous limitons ici à la norme RVB (ou RGB) : le codage RVB (Rouge-Vert-Bleu) ou son acronyme anglais RGB (Red-Green-Blue) utilise 256 nuances par composante,



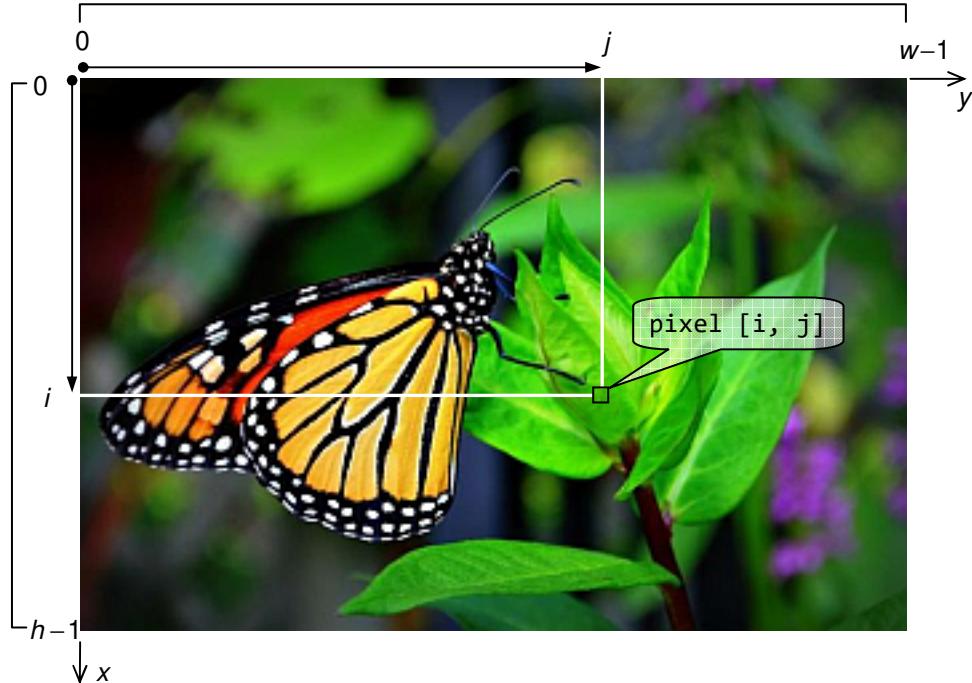
le mode RVB est très répandu pour sa facilité de gestion des couleurs et sa facilité de mise en œuvre sur les écrans puisqu'il suffit d'associer à chacune des composantes (R, V ou B) une led, l'association de trois leds formant un pixel.

## B - Stockage et décodage d'une image matricielle (bitmap) (format BMP, TIF, JPG, ...).

### Coordonnées d'une image matricielle :

Les pixels d'une image bitmap  $h \times w$  sont repérés par leurs numéros de ligne et de colonne, avec pour conventions :

- l'origine est le coin supérieur gauche,
- les abscisses correspondent aux lignes, les ordonnées aux colonnes (!),
- les lignes sont numérotées dans l'ordre descendant par rapport à la première de 0 à  $h - 1$ ,
- les colonnes dans l'ordre croissant par rapport à la première de 0 à  $w - 1$ .



### Codage RVB :

A chaque pixel est associée une liste de 3 valeurs entières de 0 à 255 (chacune codée sur un octet, soit deux caractères hexadécimaux) correspondant à la valeur de l'intensité de la composante R, V ou B.

La couleur obtenue correspond à la couleur en synthèse additive (!). On dispose ainsi de plus de 16 millions de nuances possibles, le tableau ci-dessous rappelle les 8 combinaisons de bases.

La notion de couleur inverse (ou négative) d'une autre couleur découle de cette convention, la nuance inverse d'une nuance  $n$  est  $255 - n$ , ainsi la couleur inverse (ou en négatif) d'une couleur  $[R = x, V = y, B = z]$  est  $[R = 255 - x, V = 255 - y, B = 255 - z]$

$R$	$V$	$B$	couleur
0	0	0	noir
255	0	0	rouge
0	255	0	vert
0	0	255	bleu
255	255	0	jaune
0	255	255	cyan
255	0	255	magenta
255	255	255	blanc
127	127	127	gris (50%)

## C - Lecture/Affichage d'un fichier "bitmap" - Utilisation des fonctionnalités de la bibliothèque matplotlib.pyplot

### matplotlib.pyplot

La bibliothèque `matplotlib.pyplot` comporte des fonctionnalités de lecture et d'affichage des images "bitmap" comme de simples matrices de pixels. Cela permet de s'affranchir de la lecture et du décodage spécifique de l'en-tête d'un fichier (header), qui comporte des informations d'identification du format de l'image, du nombre de lignes et de colonnes, de description éventuelle de la palette couleur, etc.

La fonction `imread('file.ext')` permet de lire le fichier `file.ext` parmi les formats `bitmap .png, .jpg, .bmp, .tif` (sous réserve que les bibliothèques `SIX` et `PIL` soient installées, sinon, il faut utiliser une distribution qui les intègre). Elle renvoie une matrice (`numpy.array`) d'éléments correspondant au codage couleur utilisé (1 élément (valeur) pour une image en niveau de gris, liste de 3 valeurs pour un codage RVB ou YUC, 4 valeurs pour un codage RVBA).



```
import matplotlib.pyplot as plt
image = plt.imread('file.ext')      # stocke dans la variable 'image' la matrice de
                                    # pixels contenue dans le fichier file.ext
```



A l'issue de cette opération de lecture, on récupère une variable `image` comportant les triplets de codage RVB.

```
>>> image
array([ [ [ 0, 61, 27], [138, 250, 23], ... , [ 14, 8, 231] ],
       ...
       [255, 0, 129], [ 0, 5, 218], ... , [ 89, 95, 103] ] ], dtype=uint8)
```

La fonction `shape` de numpy permet alors de récupérer les informations de format de l'image (dimensions de la matrice) : `hauteur(h)`, `largeur(w)`, `nb couleur(n)` utilisés pour son codage.

```
import numpy as np
h, w, n = np.shape(image)
```

La fonction `imshow(image)` permet d'afficher dans une figure le contenu de la variable `image` en tant que photo.

```
plt.figure(1)
plt.imshow(image)
plt.show()
```



Par défaut, l'affichage se fait avec une opération de lissage par interpolation (opération transparente pour l'utilisateur) qui gomme les effets de la pixellisation lorsqu'elle devient visible à l'occasion d'un zoom (lorsque la taille d'un "point image" devient supérieure à la taille du "pixel écran"). Pour désactiver cette fonctionnalité, on utilise la syntaxe :

```
plt.imshow(image, interpolation = 'none')
```

---

Intérêt de stocker le contenu d'une image dans une variable matrice (de type `np.array`)

---

image =  
matrice

Le fait de récupérer et stocker l'image dans un tableau de type "`numpy.array`" permet de la manipuler avec les fonctionnalités de "slicing" et/ou d'utiliser les fonctions de numpy avec vectorisation des opérations sur les listes (et donc sur les matrices !) ainsi que les opérations de produits matriciels (par exemple avec les filtres de convolution).



Cela améliore considérablement la rapidité de traitement et s'avère particulièrement utile lorsque l'on manipule des images avec une définition importante.

Mise en  
évidence  
sur un  
exemple.



```
def inverser_v1(image):
    lacopie = copy(image)
    h, w, n = np.shape(image)
    for i in range(h):
        for j in range(w):
            for k in range(3):
                lacopie[i, j, k] = 255 - image[i, j, k]
    return lacopie
```



```
def inverser_v2(image):
    lacopie = 255 - image
    return lacopie
```

Gain constaté:

Pour illustrer le gain de performance, on applique ces deux fonctions à une même image (définition de 414 x 620).

Les temps de traitement observés sont alors :

- pour la version 1 (3 boucles imbriquées, affectations nuance par nuance) = environ 5 s,
- pour la version 2 (vectorisation de l'opération 255 - image) = 0.001 s.

## D - Opérations simples (gestion des couleurs, niveau de gris, noir et blanc) sur une image entière.

### Trames R, V ou B

#### Filtrage des composantes [R, V, B]

#### Notion de "trame" RVB

Afin de se rendre compte des 3 trames (rouge, verte et bleue) qui par superposition constituent une image, on propose de construire une fonction qui renvoie une image d'une seule des composantes parmi  $[R, V, B]$ . Pour illustrer cette décomposition, l'image de départ (papillon) se décompose en trois trames comme ci-dessous :

Image de départ



Trame "rouge"



Trame "verte"



Trame "bleue"



#### Remarque

Les figures ci-dessus ne sont pertinentes qu'avec un affichage couleur...

**D.1** Ecrire une fonction `filtrer_RVB(image, composante)` qui renvoie une image n'ayant que la composante spécifiée.

indications :      1) composante sera une variable de type "string" parmi  $'R', 'V', 'B'$ .  
                        2) on utilisera au maximum le "slicing" de liste pour optimiser la vitesse d'exécution.

### Conversion en niveaux de gris

#### Conversion en niveau de gris

La conversion d'une image couleur en niveaux de gris revient à remplacer le triplet des nuances  $[R, V, B]$  d'un pixel par le triplet de leur valeur moyenne, soit  $[M, M, M]$  avec  $M = \frac{1}{3}(R + V + B)$ . La fonction ci-dessous `gris_brut(image)` renvoie l'image (en niveaux de gris) avec une moyenne brute "pixel par pixel".



```
def gris_brut(image):
    haut , larg, nbcoul = format(image)
    lacopie = copy(image)
    for i in range(haut):
        for j in range(larg):
            moyenne = int(sum(image[i,j])/3) # valeur moyenne arrondie à un entier !
            lacopie[i,j,:] = moyenne         # affectation des trois valeurs [R,V,B]
    return lacopie
```

## Amélioration du rendu visuel :

Cette méthode présente cependant un défaut de rendu dans la perception de la luminosité entre l'image d'origine et sa conversion en niveaux de gris car elle modifie la "luminance" de l'image globale. La luminance est la grandeur mesurable correspondant à la sensation visuelle de luminosité (ou de brillance) d'une surface. Avec cette stratégie, la moyenne calculée est une moyenne brute dans laquelle le poids de chaque composante est identique, or, à intensité égale, une lumière verte paraît plus claire qu'une lumière rouge, elle même plus claire qu'une lumière bleue car la sensibilité de l'œil dépend de la longueur d'onde de la lumière émise.

En pratique, le ressenti de la luminosité par l'œil dépend à 60% de la composante verte(V), à 30% de la rouge(R) et à seulement 10% de la bleue(B).

Pour tenir compte de cette non-linéarité entre les composantes, on corrige la moyenne par une moyenne pondérée calculée comme :  $M \approx 0.3 \times R + 0.6 \times V + 0.1 \times B$  ou plus exactement :  $M = 0.299 \times R + 0.587 \times V + 0.114 \times B$

Les deux images ci-dessous illustrent cet aspect (moyenne brute à gauche, moyenne pondérée à droite -plus claire-)



Résultat de la fonction gris\_brut()



Résultat de la fonction gris\_opti()

- D.2 Ecrire une fonction `gris_opti(image)` qui renvoie une image en niveaux de gris avec la moyenne pondérée.

indication : pour améliorer la vitesse d'exécution, on calculera la valeur moyenne comme un produit de matrice en utilisant la fonction `numpy.dot(A, B)` qui renvoie le produit matriciel de A par B , l'une des matrices étant le triplet  $[R, V, B]$  (matrice ligne !), l'autre étant la matrice des coefficients ...



```
import numpy as np  
produit = np.dot(A, B)
```

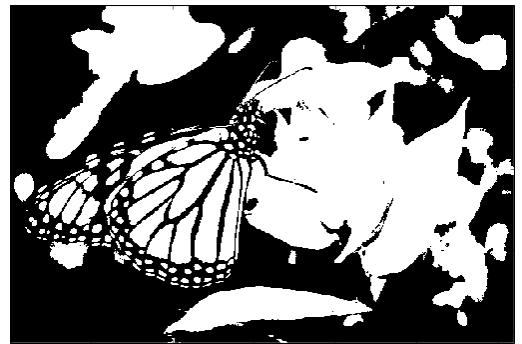
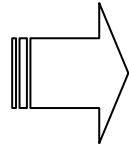
---

## Conversion en noir et blanc

---

### Conversion en noir et blanc

La conversion d'une image couleur en noir et blanc revient à convertir chaque pixel en un niveau de gris, puis à choisir un seuil (0-255) qui déterminera le choix "noir" (si gris < seuil) ou "blanc" (si gris  $\geq$  seuil).



- D.3 Ecrire une fonction `noir_et_blanç(image, seuil)` qui renvoie une image en noir et blanc en fonction du "seuil".

---

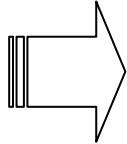
## E - Modification locale/globale des couleurs dans une image.

---

### Inversion des couleurs sur une ½ image

---

Inversion avec  
frontière  
diagonale



- E.1 Ecrire une fonction `negatif_sur_diagonale(image)` qui renvoie une image dont la couleur des pixels situés au-dessus de la diagonale (coin inférieur gauche – coin supérieur droit) aura été inversée.

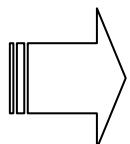
---

### Mise en évidence d'une zone rectangulaire

---

Mise en  
évidence  
d'une zone  
rectangulaire

On imagine une fonction qui permette de mettre en évidence une zone rectangulaire de l'image avec conservation des couleurs et "détourage" (tracé d'une ligne noire qui délimite la zone ainsi mise en évidence) et conversion en niveaux de gris du reste de l'image.



- E.2 Ecrire une fonction `gris_hors_rectangle(image, h1, w1, h2, w2)` qui renvoie une image dans laquelle la zone rectangulaire délimitée par les coins supérieur-gauche ( $h_1, w_1$ ) et inférieur droit ( $h_2, w_2$ ) sera mise en évidence par rapport au reste de l'image qui aura été traduit en niveaux de gris.

Pour améliorer l'impact visuel de la mise en évidence, on prévoira de rajouter un tracé noir sur une largeur de un pixel afin de visuellement mieux délimiter le pourtour rectangulaire (!).

---

### Encore plus fort, mise en évidence d'une zone circulaire ...

---

- E.3 Ecrire une fonction `gris_hors_cercle(image, hc, wc, rc)` qui renvoie une image dans laquelle la zone circulaire de rayon  $r_c$  et dont les coordonnées du centre sont  $(h_c, w_c)$  aura été mise en évidence (par un tracé noir) tandis que le reste de l'image aura été converti en niveaux de gris.

Améliorer votre fonction de manière à correctement traiter toute demande de `gris_hors_cercle()` même si le cercle déborde partiellement de l'image !



---

## F - Modification géométrique d'une image.

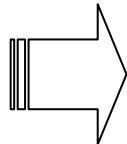
---

### Effet miroir

---

#### Effet miroir :

On imagine une fonction qui permette d'obtenir une image où la moitié "gauche" aura remplacée la moitié "droite" comme illustrée ci-dessous (effet de symétrie verticale).



- F.1 Ecrire une fonction `miroir_gauche_droite(image)` qui renvoie une image dont la moitié "gauche" aura remplacée la moitié "droite" par symétrie verticale.
- F.2 Ecrire une fonction `miroir_droite_gauche(image)` qui renvoie une image dont la moitié "droite" aura remplacée la moitié "gauche" par symétrie verticale.

---

### Effet carte de jeu...

---

- F.3 Ecrire une fonction `effet_carte(image)` qui renvoie une image dans laquelle la moitié "au-dessus" de la diagonale aura remplacée la moitié "en dessous" de la diagonale avec retournement et conversion en niveau de gris...



---

### Rotation d'une image

---

- F.4 Ecrire une fonction `pivoter(image, sens = 1)` qui renvoie une image ayant subie une rotation d'un  $\frac{1}{4}$  de tour.

(sens trigo si `sens = 1` et sens horaire si `sens = -1`).

indication : on conseille de faire au préalable un croquis sur lequel apparaît :

- l'image d'origine (rectangulaire et pas carrée !),
- l'image finale (pivotée),
- les coordonnées images sur chacune...

Pour construire une image "noire" renversée d'un quart de tour de mèmes dimensions que l'image de base, on pourra s'aider de l'instruction ci-dessous :



```
lacopie = np.zeros(haut*larg*coul) \
          .reshape(larg,haut,coul) \
          .astype(np.uint8)
```

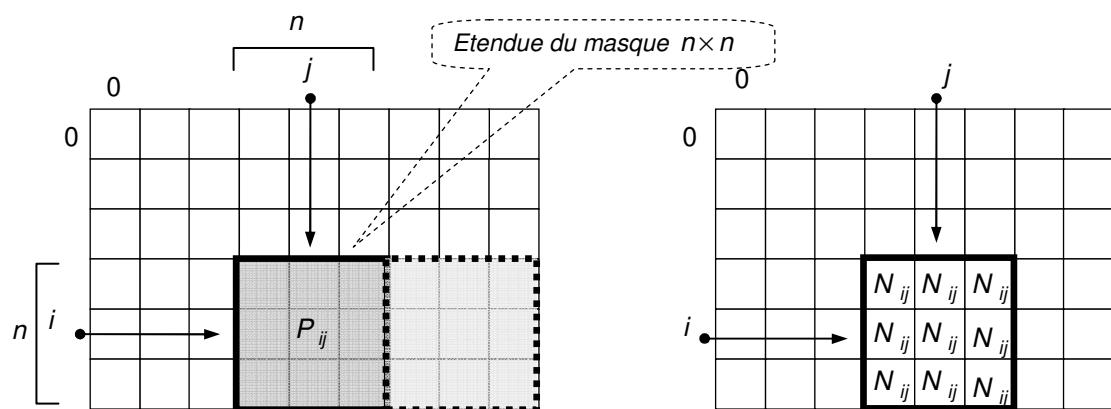


Principe de la pixellisation :

On parle de pixellisation d'une image lorsque les points (pixel) qui la composent deviennent visibles. Un traitement de pixellisation dégrade l'image en diminuant le nombre de pixel significatif (mais permet de gagner en espace mémoire), c'est typiquement le cas lors de la diminution de la définition spatiale d'une image.

Une opération de pixellisation consiste à remplacer un groupe de pixel (groupe carré  $n \times n$ ) par un pixel moyen (mais en pratique, pour garder la même dimension d'image, on remplace le groupe de  $n \times n$  pixels par  $n \times n$  fois le même pixel "moyen"). Le résultat donne une impression similaire à une opération de flouter mais l'opération de "flouter" conserve la définition et recalcule chaque pixel comme un pixel moyen compte-tenu de ses voisins, tandis que la pixellisation (que nous mettons en œuvre ici) remplace l'ensemble des pixels d'une zone par autant de fois le même pixel moyen de la zone.

Illustration :



Effets de bords.

Il faut en tenir compte du fait que la taille de l'image peut ne pas être un multiple de  $n$ , et faire en sorte de ne pas générer d'erreurs par débordement des indices.



Mise en œuvre de la pixellisation

- G.1 Ecrire une fonction `pixellisation(image,n)` qui renvoie une matrice de triplets  $[R,V,B]$  de même taille correspondant au résultat de la pixellisation de l'image (`image`) avec un masque de taille  $n \times n$ .

Résultat d'une pixellisation.

Exemple :

Les images ci-dessous montrent le résultat de la pixellisation avec un masque de  $8 \times 8$ .



Image originale (200 x 360)



Pixellisation (n = 8)

## H - Détection de contour (simple).

### Principe :

La détection de contour est une opération omniprésente dans le traitement automatisé d'image, en particulier dans le domaine de la reconnaissance de formes.

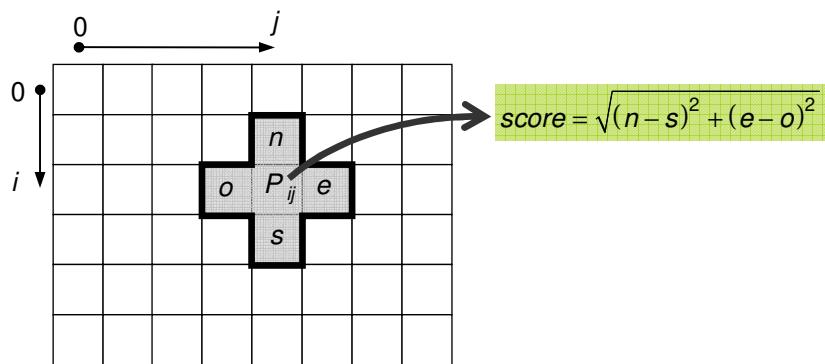
### Stratégie :

La stratégie de convolution par une matrice telle que celle du laplacien ou du gradient est une méthode classique et permet de privilégier une direction particulière pour la recherche des contours (mais plus lourde à mettre en œuvre).

### Variante :

Une variante consiste à calculer, pour chaque pixel, un score qui traduise l' "éloignement" des couleurs de ses voisins dans les directions cardinales ( $e, o, n, s$ ), puis à affecter au pixel en question une couleur en fonction de son "score". La figure ci-dessous illustre cette stratégie avec un calcul de "score" comme une distance (au sens "norme")

### Illustration :



Simplification : En théorie, pour chaque pixel  $P_{ij}$ , il faudrait calculer 3 scores relatifs à chacune des variations des nuances [R,V,B], mais pour simplifier et alléger l'écriture d'un algorithme de détection de contour, on fait le choix de ne travailler que sur des images en niveaux de gris, soit avec un codage [R,V,B] de la forme [G,G,G].

**H.1** Ecrire une fonction `detection_contour(image, seuil)` qui renvoie la matrice de pixels (noir ou blanc) de manière à ce qu'elle soit représentative des contours présents dans `image`, contours issus du calcul des scores,

`seuil` est la valeur du score au dessus de laquelle le pixel est considéré comme appartenant à un contour, un pixel dont le score est inférieur à `seuil` sera blanc, un pixel dont le score est supérieur (ou égal) à `seuil` sera noir.

Indication : utiliser la fonction `gris_opti(image)` si besoin.

### Quelques pièges sournois à éviter...



Pensez à tenir compte des effets de bords, on ne peut pas calculer le score des pixels en bordure de l'image !



Sachez que si l'on extrait les valeurs des composantes [R,V,B] ou [G,G,G], ces valeurs sont, a priori, des entiers codé sur 8 bits, puisque issues d'un codage RVB, cela a des conséquences sournoises, soient (e,o) deux valeurs:

- une opération de la forme  $e**2$  ne génère pas d'erreur même si le résultat déborde au delà de 255 (valeur maximale codable sur 8 bits) car Python modifie dynamiquement (lors du calcul) le type du résultat de `np.uint8` à `np.int32`,
- mais une opération de la forme  $(e-o)$  peut générer un avertissement (Warning à ne pas confondre avec une erreur Error !) pour dépassement de capacité (overflow encountered in ubyte\_scalars) car si le résultat n'est pas dans l'intervalle [0, 255] alors Python le décale (modulo 256) de manière à le ramener dans l'intervalle car le résultat est supposé être du même type, donc codable sur 8 bits, donc dans l'intervalle [0, 255]. Cela fausse le calcul du "score" par l'introduction d'un biais non maîtrisé dans le calcul du score et cette erreur est très délicate à détecter (cf. petit programme de démonstration à ce sujet...).

Pour palier ce biais, on pensera donc à convertir les valeurs lues (ou extraites) en flottants avant manipulations !



```
def detection_contour(image,seuil):  
    # récupération des informations de format de l'image  
    haut , larg, nbcoul = format(image)  
  
    # création d'une copie en niveaux de gris (copiegrise) et d'une image blanche  
    copiegrise = gris_opti(image)  
    lacopie = np.ones((format(image)),dtype = np.uint8) * 255  
  
    """  
    à compléter  
    """  
  
    return lacopie
```

---

Résultats d'une détection de contour.

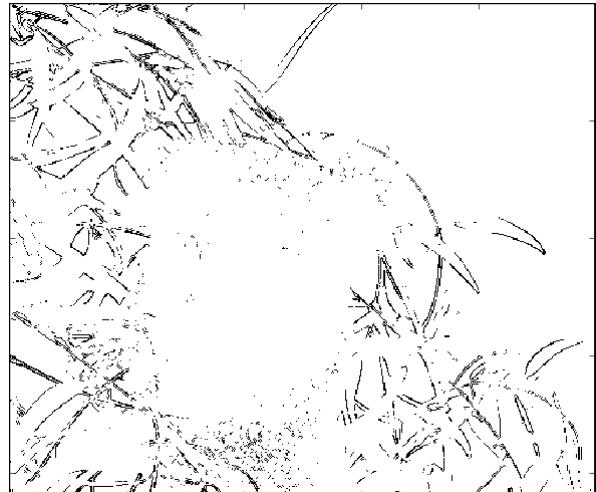
---

Exemple :

Les images ci-dessous montrent le résultat de la fonction `detection_contour(image,seuil)` à partir d'une image comportant des zones de contours nets (feuillage) et des zones floues ou bruitées (pelage et arrière-plan)



Image originale (niveaux de gris)



Détection de contour (seuil = 128)



Détection de contour (seuil = 64)



Détection de contour (seuil = 32)

## I - Histogramme d'une image.

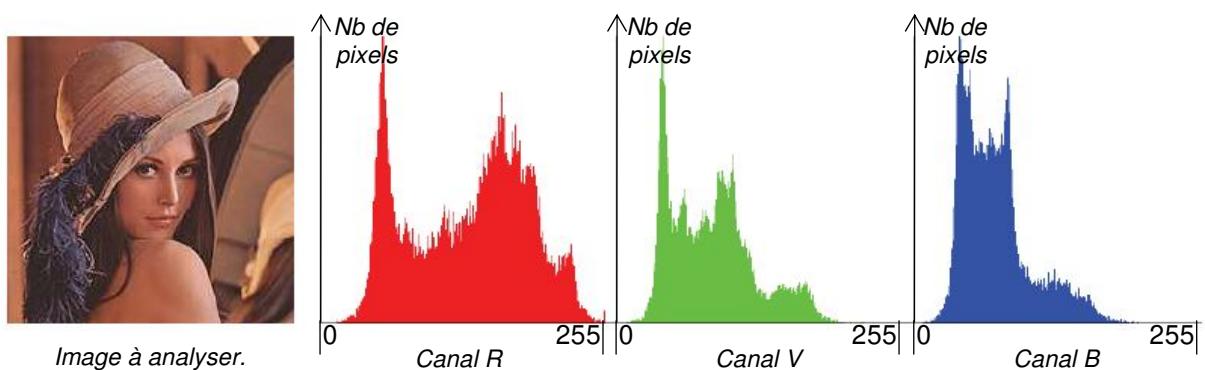
### Principe :

L'histogramme d'une image est une représentation des tonalités, elle illustre la distribution des tons (de chaque composante RVB) des pixels d'une image sous la forme d'un diagramme où on place en abscisse les nuances de tons (de 0 à 255) pour chaque canal RVB, et en ordonnée, le nombre de pixels correspondant à ce ton dans l'image.

### Définition : (wikipedia)

Pour une image monochrome, c'est-à-dire à une seule composante, l'histogramme est défini comme une fonction discrète qui associe à chaque valeur d'intensité le nombre de pixels prenant cette valeur. La détermination de l'histogramme est donc réalisée en comptant le nombre de pixel pour chaque intensité de l'image.

### Exemple :



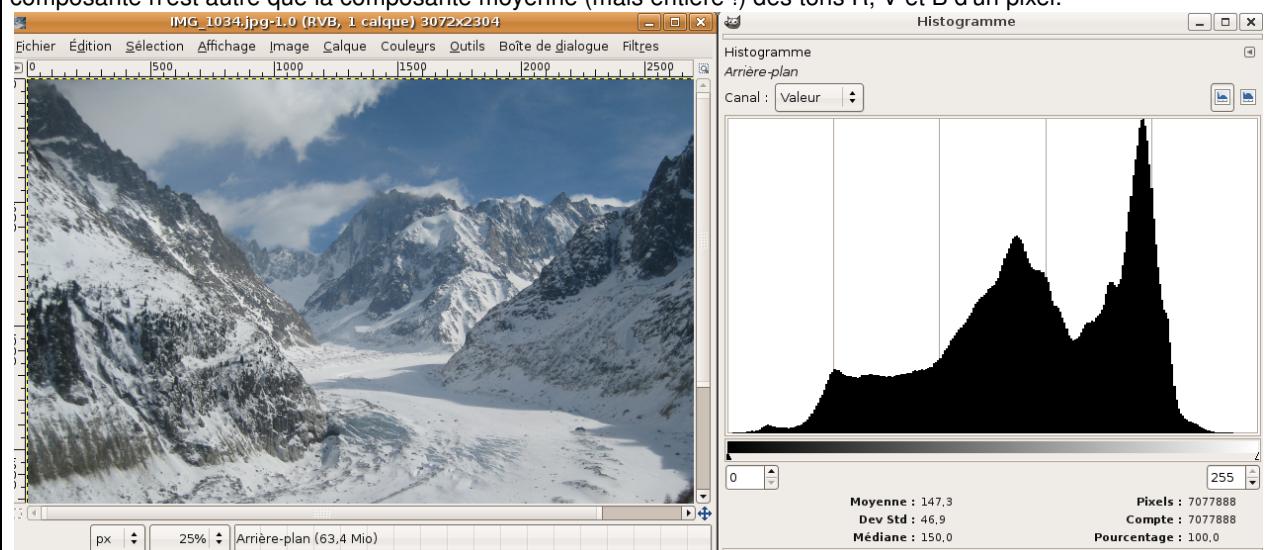
### Normalisation (wikipedia)

Les histogrammes sont en général normalisés, en divisant les valeurs de chaque classe (ton) par le nombre total de pixels de l'image. La valeur d'une classe varie alors entre 0 et 1, et peut s'interpréter comme la probabilité d'occurrence de la classe dans l'image. L'histogramme peut alors être vu comme une densité de probabilité.

### Luminosité :

L'histogramme de luminosité correspond à l'histogramme de la composante de luminosité d'une image, cette composante n'est autre que la composante moyenne (mais entière !) des tons R, V et B d'un pixel.

### Exemple :



Histogramme de luminosité d'une image.

### Utilisation :

En photographie, l'histogramme est un bon indicateur du niveau d'exposition d'une photo, une photo "surexposée" présentera un histogramme très décalé sur la droite (beaucoup de pixels ayant des intensités fortes), inversement, une photo "sous-exposée" présentera un histogramme décalé sur la gauche (nombreux pixels sombres).

- I.1 Ecrire une fonction `histogramme(image)` qui renvoie un tableau  $H$  de taille  $(4, 256)$  dont les 3 premières lignes  $H[i]_{i=0,1,2}$  sont respectivement les listes "histogrammes" des composantes R, V et B et dont la composante  $H[3]$  est la liste "histogramme" de la luminosité (valeur entière de la moyenne des composantes RVB).

Vérifier la bonne construction de votre fonction en utilisant la fonction `affiche_histogramme(image)`.