

TP : manipulation d'images

I Objectifs :

- manipuler des images en niveau de gris ;
- manipuler des images en couleur RGB ;
- réaliser des boucles imbriquées et comprendre l'ordre dans lequel les instructions sont exécutées ;
- évaluer la complexité des algorithmes et comprendre pourquoi le traitement d'image prend du temps.

Durant tout le tp, on pensera à commenter les fonctions ou les parties de code qui ne sont plus utiles par la suite, pour cela penser au raccourci Ctrl+1

Exercice n°1. Ouvrir dans spyder le fichier TP_image.py. Choisissez comme répertoire du travail le répertoire TP_image et exécutez-le TP_image.py. L'image d'un chat tapant du code doit s'afficher dans la fenêtre "Graphes" et celle d'un légo gris doivent s'afficher.

Que contiennent les variables chaton et lego ? Répondre directement dans le fichier TP_image.py.

Exercice n°2.

1. Décommentez les lignes 77 à 80 et complétez la ligne 77 pour créer une image comme l'image 1 ci-dessous.
2. Faire de même avec les lignes 84 à 88 et l'image 2 ci-dessous (vous pouvez re-commenter les lignes 77 à 80 pour éviter que l'image 1 ne s'affiche à chaque fois)
3. Re-commentez ces lignes pour éviter que les images ne s'affichent à chaque fois par la suite.

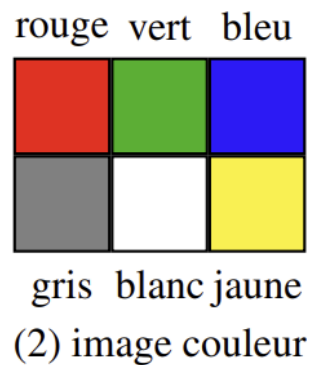
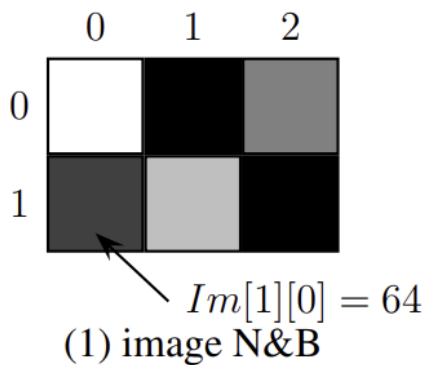


Figure 1 - Illustrations pour l'exercice n°1

Exercice n°3. Créer une fonction `dim(im:"image")->(int,int)` qui prend en argument une image en niveau de gris (sous forme d'un tableau) et qui renvoie un tuple contenant dans l'ordre le nombre de lignes et le nombre de colonnes. Cette fonction pourra être réutilisée par la suite.

Testez votre fonction avec l'une des images créées à la question 2.

II Quelques traitements élémentaires

1. Anonymat

Exercice n°4. Modifier l'image lego pour ajouter un bandeau noir devant les yeux du personnage.

Exercice n°5. Créez une fonction `copie(im)` qui renvoie une copie de l'image im.

2. Négatif

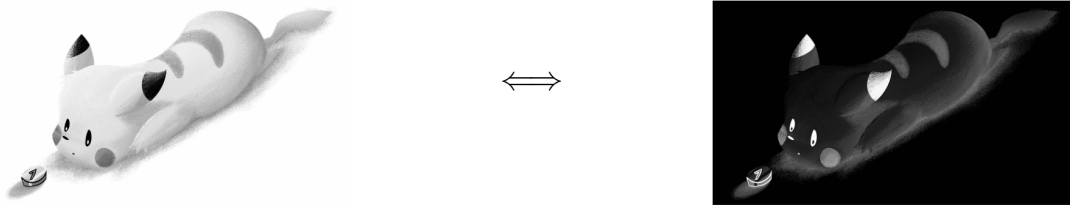


Figure 2 - Image initiale et son négatif

Exercice n°6. Écrire une fonction `negatif(im:"image")->"image"` qui prend en argument une image en niveau de gris et renvoie son négatif.

3. Miroir horizontal et rotation

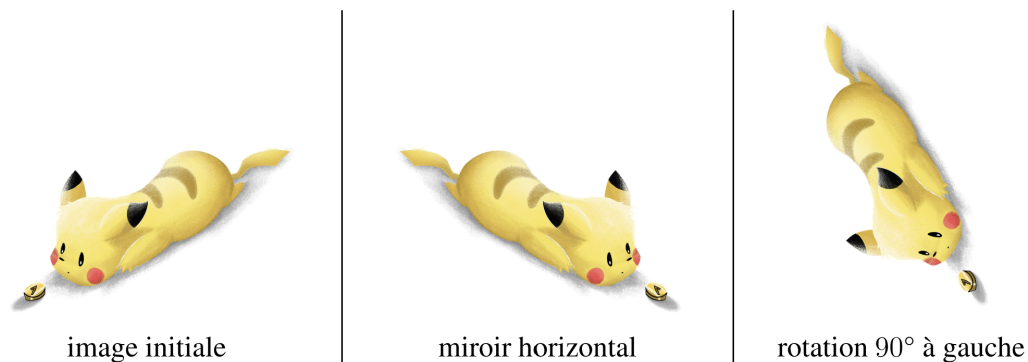


Figure 3 - Effet miroir horizontal

Exercice n°7. Écrire une fonction `miroir_horizontal(im:"image")->"image"` qui prend en argument une image et renvoie une autre image correspondant à une symétrie par rapport à un axe vertical passant par le milieu de l'image. Tester avec une des images fournies.

Exercice n°8. Écrire une fonction `rotation90gauche(im:"image")->"image"` qui réalise une rotation de 90° vers la gauche.

III La convolution pour des traitements plus évolués

La plupart des traitements automatiques d'image (floutage, accentuation des contours, réduction de bruit, etc.) reposent sur un principe fondamental : la **convolution**.

L'idée est la suivante : pour améliorer ou transformer une image, on peut **remplacer chaque pixel** par une **moyenne pondérée** de ses voisins.

Cette opération de moyenne pondérée utilise une petite matrice appelée **noyau** (ou *kernel* en anglais). Il s'agit d'un tableau de coefficients, généralement de taille 3×3 , 5×5 , etc. Pour chaque pixel de l'image, on centre le noyau sur ce pixel, on multiplie les valeurs des pixels voisins par les coefficients du noyau, puis on additionne le tout. Le résultat est la nouvelle valeur du pixel.

Exemple (cas d'un noyau de moyennage 3×3 centré sur le pixel (1, 1)) :

Extrait d'image (valeurs des pixels)

52	55	61
63	59	55
66	60	57

Noyau 3×3

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

On applique le noyau centré sur le pixel de valeur 59. Chaque coefficient du noyau est $\frac{1}{9}$, ce qui revient à faire la moyenne des 9 pixels autour du pixel central (en incluant ce pixel). Voici le détail du calcul :

$$\begin{aligned}s[1][1] &= \frac{1}{9} \times 52 + \frac{1}{9} \times 55 + \frac{1}{9} \times 61 \\&\quad + \frac{1}{9} \times 63 + \frac{1}{9} \times \mathbf{59} + \frac{1}{9} \times 55 \\&\quad + \frac{1}{9} \times 66 + \frac{1}{9} \times 60 + \frac{1}{9} \times 57 \\&= \frac{1}{9} \times (52 + 55 + 61 + 63 + \mathbf{59} + 55 + 66 + 60 + 57) \\&= \frac{1}{9} \times 528 = 58.\bar{6}\end{aligned}$$

On remplace donc le pixel central par la valeur 58,7 (arrondie).

Le choix du noyau est crucial :

- Un noyau de taille $n \times n$ contenant uniquement des $1/n^2$ permet un **floutage**.
- Un noyau avec des valeurs positives et négatives peut détecter les **changements rapides**, donc les **contours**.

En général, pour ne pas modifier la luminosité globale de l'image, on choisit un noyau dont la somme des coefficients vaut 1 (dans le cas d'un filtre de floutage), ou 0 (dans le cas d'un filtre de détection de contours).

Il est à noter que plus le noyau est grand, plus l'effet est prononcé... mais plus le calcul est long ! C'est pour cela qu'un bon traitement d'image cherche souvent un compromis entre **qualité visuelle** et **temps de calcul**.

Nous allons maintenant écrire nos propres fonctions pour appliquer différents types de noyaux à une image, et ainsi réaliser nos premiers effets de traitement d'image personnalisés.

Exercice n°9. Écrire une fonction `noyau_constant(n)` qui renvoie un noyau de floutage simple de taille $n \times n$.

Exercice n°10. Écrire une fonction :

```
calculer_pixel(im:"image", noyau:"image", i:int, j:int)->np.uint8
```

qui renvoie la valeur du pixel de coordonnée (i, j) après convolution entre l'image et un noyau 3×3 . On supposera que le pixel est suffisamment éloigné des bords.

Exercice n°11. Écrire une fonction `filtre_image(im:"image", noyau:"image")->"image"` qui applique un noyau à toute une image. Pour les pixels des bords pour lesquels la formule ne peut pas s'appliquer simplement, on renverra juste 0. Tester avec avec le noyau `moyenne(3)` et le noyau `moyenn(11)` pour observer le floutage du lego.

Exercice n°12. On utilise les deux noyaux ci-dessous (filtre de Sobel) qui sondent les variations verticales et horizontales dans l'image (la somme des coefficients vaut 0, ce qui fait que le filtre renvoie 0 dans les zones homogènes).

```
Sobel_h = [
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1]
]
Sobel_v = [
    [ 1, 2, 1],
    [ 0, 0, 0],
    [-1, -2, -1]
]
```

Pour chaque pixel de coordonnées (i, j) , on calcule le résultat par chacun des noyaux : x_h et x_v , puis on évalue $\sqrt{x_h^2 + x_v^2}$. L'ensemble des valeurs ainsi calculées pour chaque (i, j) forme une nouvelle image qui met en évidence les contours des objets présents sur l'image de départ.

On peut ensuite éventuellement appliquer une fonction de seuillage : les pixels ayant une valeur inférieure à un seuil s sont ramenés à 0 et les autres à 255. Cela n'a pas été fait sur l'image ci-dessous, mais permettrait d'améliorer le contraste.

Utiliser ces idées pour créer une fonction `detection_contours(im:"image")->"image"`



image initiale



image du contour



négatif du contour

Figure 5 - détection de contours