TP 16
Manipulation d'images

Nous allons maintenant apprendre à manipuler les images. Il s'agit d'une première approche et d'un aperçu de toutes les possibilités.
La première chose à faire est de récupérer le fichier `materiel.zip`. Il contient, en plus d'un fichier `.py` à compléter, une certaine bibliothèque de photos pour le TP. Ouvrir le fichier et l'exécuter avec la commande (sous Pyzo) « démarrer le script ». Si tout se passe bien, malgré des messages d'erreur, une image en noir et blanc s'affiche. Ensuite on peut exécuter les cellules comme d'habitude.

I Introduction

I.1 Représentation d'images

Une image en couleurs à n lignes et p colonnes est manipulée par l'ordinateur comme un tableau. Chaque case du tableau s'appelle un pixel et contient en fait trois nombres pour former une petite case de couleur : le premier indique l'intensité de la couleur rouge, le deuxième de la couleur vert, et le troisième de la couleur bleu. On parle de codage RGB (Red, Green, Blue, c'est de l'anglais *of course*). Comme cela est un peu compliqué **nous travaillons d'abord avec des images en noir et blanc** auquel cas chaque pixel est représenté par un simple nombre qui indique la luminosité du pixel.
Plus précisément, nous travaillons avec la bibliothèque `numpy`. Le code préparé charge une image dont le nom est donné dans la variable `fichier` dans un tableau `numpy` nommé `image` à deux axes, le premier correspondant aux lignes et le second aux colonnes. Chaque pixel est codé sur un octet, soit 8 bits. Cela donne 256 valeurs possibles, soit tous les entiers entre 0 et 255. Ainsi la valeur de `image[i, j]` est `0` si le pixel de la ligne i colonne j est tout noir, `255` si le pixel est tout blanc, et les valeurs intermédiaires correspondent à des niveaux de gris. Le type des données du tableau est `uint8` (entier, sans signe, sur 8 bits).
On rappelle au passage que, tout comme pour les matrices, si l'image a n lignes et p colonnes alors les lignes sont numérotées de 0 à $n-1$, les colonnes de 0 à $p-1$, le pixel $(0,0)$ est le plus en haut à gauche et $(n-1,p-1)$ le plus en bas à droite. Avec la syntaxe `numpy` le pixel d'indice (i,j) est `image[i, j]` ce qui est plus simple que `image[i][j]` que nous avons vu avec des listes de listes.

$$\begin{array}{|cccc|}\hline (0,0) & (0,1) & \cdots & (0,p-1)\\ (1,0) & (1,1) & \cdots & (1,p-1)\\ \vdots & \vdots & \ddots & \vdots\\ (n-1,0) & (n-1,1) & \cdots & (n-1,p-1)\\ \hline\end{array} \tag{1}$$

Si on travaillait avec des couleurs, alors la variable `image` serait un tableau à trois axes, `image[i, j, 0]` serait l'intensité du rouge dans le pixel (i,j), `image[i, j, 1]` l'intensité du vert et `image[i, j, 2]` du bleu. Cela complique pas mal le traitement — en fait pas tant que cela grâce à la magie `numpy`, c.f. la dernière partie de ce TP. Enfin `image.shape` est un tuple de longueur 2 (sans couleur) ou 3 (couleurs) et dans tous les cas `image.shape[0]` est le nombre de lignes et `image.shape[1]` le nombre de colonnes. La fonction `numpy.zeros((n, p))` est donc utilisée pour créer une image vierge de n lignes et p colonnes remplie de zéros, c'est à dire une image toute noire.
Vérifiez cela après avoir exécuté le code qui charge l'image :

```
>>> image
>>> image.shape
>>> image.dtype
```

I.2 Échantillon de photos

Le dossier contient également un certain échantillon d'images. Vous pouvez choisir celle que vous voulez, idéalement en gardant la même pour toute la durée du TP (on pourra facilement en prendre une autre, globale pour tout le TP). Les images sélectionnées réunissent quelques critères. Elles sont redimensionnées à une taille raisonnable (environ 300 pixels de côté) alors qu'une image directement sortie d'une caméra moderne va contenir des millions de pixels et le programme sera lent à les traiter. On apprécie aussi d'avoir un objet ou paysage qui se détache nettement du décor, y compris en noir et blanc. Même si vous pourrez utiliser aussi vos propres images, cela n'est pas conseillé au début du TP.

Remarque 1. Quelques avertissements préalables. **Utiliser une image trouvée sur internet pour un travail scolaire ou universitaire, ou pour la rediffuser, sans en avoir l'autorisation est considéré comme une faute grave**. Il est donc nécessaire de s'intéresser aux droits de l'image, et de faire un usage strictement privé des images trouvées.
Les images présentes sur Wikipedia par exemple ont souvent une licence qui autorise à les ré-utiliser, voire les modifier, et les rediffuser (licence « Creative Commons » faites pour encourager le partage) mais toujours en **citant proprement la source** de la photo. Autant que possible, dans vos travaux, utilisez vos propres photos et créez vous-même vos propres illustrations, cela sera certainement valorisé même si les illustrations ne sont pas aussi belles !
Les photos proposées ici sont des photos personnelles et **l'autorisation vous est donnée de les utiliser pour ce TP, mais pas de les rediffuser**.

I.3 Fonctionnement global

Comme dans le TP sur les matrices, les fonctions ne doivent pas modifier l'image passée en argument mais faire soit une copie soit une nouvelle image vierge. Ensuite, c'est le mécanisme de la double boucle `for` qui permet d'effectuer une opération sur chaque pixel, un par un.
Dans toutes les fonctions l'image passée en argument s'appelle `X` et l'image renvoyée s'appelle `Y`. La variable `image` est globale pour tout le fichier : ré-exécuter les fonctions si on la modifie.

II Forme de l'image

Les premières fonctions que l'on veut coder sont le miroir vertical et le pivotement de 90 degrés vers la droite.
La question qu'il faut se poser au brouillon est : si on prend une image `X` et qu'on veut en former l'image miroir `Y`, alors quel pixel de `X` va dans le pixel de coordonnées (i, j) de `Y` ? Vérifier au brouillon que c'est bien celui de coordonnées $(i, p-1-j)$ sur la même ligne, mais sur la colonne symétrique par rapport à la verticale.

Exercice 1. Écrire la fonction `miroir(X)` qui renvoie une image obtenue à partir de `X` par miroir vertical.

On poursuit avec la fonction qui pivote de 90 degrés vers la droite. Là encore il s'agit d'abord de trouver au brouillon : quel pixel de `X` va aller dans `Y[i, j]` ? On prendra garde aux dimensions de `Y` cette fois ! La réponse est la bonne combinaison des i, j, $n-1-i$, $p-1-j$.

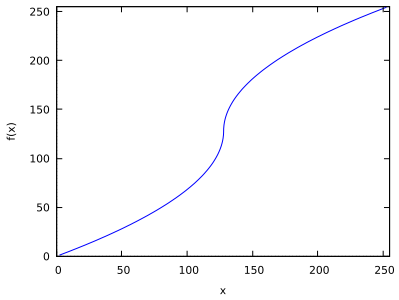
Exercice 2. Écrire la fonction `pivote(X)` qui retourne une image obtenue à partir de `X` par rotation de 90 degrés sur la droite.

III Éclairage

Pour augmenter l'éclaircissement d'une image, il suffit d'ajouter à chaque pixel une valeur fixe, par exemple 50 (l'effet sera bien visible, mais on pourra ajuster cette valeur plus tard). En effet la luminosité d'un pixel est un nombre entre 0 et 255, donc les augmenter tous de la même façon ne pourra qu'augmenter la luminosité de l'image. Attention, il ne faut pas seulement ajouter 50 : si le résultat de l'addition dépasse 255 (la luminosité maximale d'un pixel), on laissera le résultat à 255. En général, on donne un *seuil* s, correspondant à augmenter tous les pixels de la valeur s.

Exercice 3. Écrire la fonction `eclaircit(X, s)` qui éclaircit l'image `X` en augmentant tous les pixels de la valeur `s`.

On s'intéresse aussi au contraste. Augmenter le contraste, c'est augmenter la luminosité sur les pixels déjà bien lumineux, et diminuer la luminosité de ceux déjà sombres. Pour cela, on a besoin d'une fonction $f : [0, 255] \to [0, 255]$ qui « tasse » les petites valeurs vers 0 ainsi que les grandes valeurs vers 255, avec un graphe ressemblant à ceci :



On propose pour cela la fonction

$$f : [0, 255] \longrightarrow [0, 255]$$
$$x \longmapsto \begin{cases} 127 - \sqrt{127 \times (127 - x)} & \text{si } 0 \leqslant x \leqslant 127 \\ 128 + \sqrt{127 \times (x - 128)} & \text{si } 128 \leqslant x \leqslant 255 \end{cases} \tag{2}$$

Vérifiez qu'il s'agit bien d'une fonction avec ce comportement.

Exercice 4. Écrire la fonction `contraste(X)` qui augmente le contraste de l'image `X` selon le procédé décrit.

Pour obtenir le négatif d'une image, c'est comme son nom l'indique une simple inversion entre les pixels lumineux et les pixels sombres : la luminosité x d'un pixel deviendra $255 - x$ dans l'image négative.

Exercice 5. Écrire la fonction `negatif(X)` qui donne le négatif de l'image `X` selon ce procédé.

IV Flou et contours

On se propose de flouter une image. L'idée est la suivante : chaque pixel sera remplacé par une moyenne des pixels autour de lui. Mais pas n'importe comment : avec un coefficient, pour que les pixels plus proches comptent plus. La règle simple à programmer qu'on propose est : un pixel lui-même compte avec un coefficient 3, les quatre pixels sur ses côtés comptent avec un coefficient 2 et les quatre pixels qui le touchent en diagonale comptent avec un coefficient 1. Le total des coefficients fait donc 15. On représente cette opération par le tableau :

`X[i-1, j-1]`	`X[i-1, j]`	`X[i-1, j+1]`
`X[i, j-1]`	`X[i, j]`	`X[i, j+1]`
`X[i+1, j-1]`	`X[i+1, j]`	`X[i+1, j+1]`

1	2	1
2	3	2
1	2	1

Avec cette méthode on ne peut pas traiter correctement les pixels sur les bords, ceux pour lesquels il n'y a pas de pixel voisin à gauche par exemple. Dans un premier temps on n'y touche pas, l'image `Y` est initialisée à une copie de `X`. et donc les pixels sur les bords ne sont pas changés (cela ne se voit en général pas à l'œil nu). Si on est courageux, la seule façon naturelle est de « tronquer » ce tableau mais alors il faut distinguer des cas selon la position du pixel de bord. Par exemple si le pixel est sur le bord gauche mais pas dans les coins, il faut prendre la moyenne sur le pixel de droite, les deux au-dessus, et les deux diagonales haut-droite et bas-droite. Le total des coefficients est de 11. Reprendre ce procédé pour chacun des 4 bords *et* chacun des 4 coins.

Exercice 6. Écrire la fonction `flou(X)` qui floute l'image `X` selon ce procédé. On ne se préoccupera pas tout de suite des bords.

Remarque 2. De nombreuses fonctions différentes de flous peuvent être obtenues en considérant une moyenne d'un plus grand nombre de pixels autour de chacun, et en faisant varier les coefficients. Par exemple, si on prend en compte dans la moyenne seulement les pixels voisins horizontaux, mais pas les verticaux, on obtient un effet de vitesse, comme une image qui se déplace très rapidement sous nos yeux horizontalement.

$$\begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline 1 & 2 & 1 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \tag{3}$$

Il existe aussi le *flou gaussien* (appelé tel quel par les graphistes) dans lequel les poids accordé aux pixels voisins suit une loi gaussienne en fonction de la distance au pixel central. Plus la gaussienne est large, plus les pixels voisins considérés dans le calcul de la moyenne sont nombreux, et plus d'effet de flou est fort. Au contraire si la gaussienne est très resserrée alors le pixel central garde beaucoup plus d'importance que les autres et le flou est léger. Ces flous font partie des *flous linéaires*, et la donnée de tous les coefficients pour les pixels voisins est appelée *matrice de convolution*. C'est précisément l'opération de convolution qui consiste à remplacer un pixel par la moyenne des pixels voisins avec des coefficients donnés par la matrice.

Pour obtenir un flou plus fort, à défaut de ré-écrire tout avec une moyenne sur plus de pixels voisins, on peut aussi itérer plusieurs fois la fonction de flou. Attention alors au temps de calcul...
On s'intéresse maintenant au problème de la *détection de contours*. Il s'agit en quelque sorte de l'inverse du flou : quand deux pixels voisins ont des intensités différentes on veut accentuer cette différence, alors qu'en prenant la moyenne le flou va réduire cette différence. On commence donc par calculer la moyenne des pixels voisins selon la règle suivante :

$$\begin{array}{|c|c|c|}\hline -1 & -1 & -1 \\ \hline -1 & 8 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array} \tag{4}$$

qui donne un résultat qui peut très bien être négatif. En fait on s'intéresse à sa valeur absolue.
Si les pixels d'une zone ont des intensités proches, cette moyenne va être proche de zéro. Au contraire, la valeur va être très élevée si le pixel central a une intensité nettement différente de celle de ses voisins. Il faut donc se fixer un seuil au delà duquel on considère que le pixel qu'on regarde est précisément sur un contour. Ensuite, tout simplement, si la moyenne dépasse le seuil on met un pixel à 0 (noir) et sinon à 255 (blanc).

Exercice 7. Écrire la fonction `contours(X, s)` qui renvoie une image en noir et blanc (seulement les couleurs 0 et 255) détectant les contours de l'image `X` avec le seuil donné `s`.

V Mettez de la couleur dans votre vie !

Pour activer la couleur, il suffit de changer à `True` la variable `couleur` du fichier déjà préparé. Puis ré-exécuter cette cellule.
Normalement il faudrait une triple boucle pour traiter uns par uns tous les pixels sur n lignes, p colonnes et 3 composantes couleurs, n'est-ce pas ?
Mais en fait, si `image` est un tableau `numpy` de forme $(n, p, 3)$ alors pour chaque indice (i, j), `image[i, j]` est considéré comme un tableau de taille 3. Ainsi les opérations que nous effectuons sur des pixels — par exemple la fonction d'éclairage qui ajoute s à chaque pixel — se fait sur `image[i, j]` *en tant qu'opération vectorielle* et donc sur chacune de ses composantes 3 composantes, et donc elle ajoute s dans chaque couleur de chaque pixel. Un certain nombre de fonctions précédentes marchent de la même façon en couleur. Pour les contours et le seuil, la moyenne calculée est en fait un tableau (x, y, z) de longueur 3 indiquant les moyennes selon le rouge, le vert et le bleu, et on compare la valeur de $\sqrt{x^2 + y^2 + z^2}$ avec le seuil.

Exercice 8. Sur le modèle de la fonction `eclaircit`, écrire une fonction `rougir(X, s)` qui augmente l'intensité du rouge dans l'image `X`, d'une quantité `s`.
Pour éviter que cela augmente trop la luminosité, on pourra en même temps réduire l'intensité des autres couleurs. Par exemple, rougir de 20 devra diminuer le vert et le bleu de 10 chacun.

VI Pour aller plus loin

Quelques pistes d'améliorations :

1. Vérifier que les fonctions marchent bien en couleur, et les corriger s'il faut.
2. Vérifier avec plusieurs images différentes.
3. Sauvegarder son image en l'enregistrant avec l'aide des fonctions `PIL.Image.fromarray` (convertit le tableau `numpy` en image) et `PIL.Image.save` (enregistre un fichier image).
4. *Vectorialiser* les opérations en utilisant toute la puissance de `numpy` pour que l'exécution soit plus rapide, c'est à dire éliminer totalement les doubles boucles mais raisonner avec les opérations vectorielles. Cela est particulièrement facile pour le miroir et le négatif, et possible pour le flou.
5. Utiliser vos propres images. Mais attention, si elles contiennent trop de millions de pixels et si le code n'est pas optimisé alors le temps de calcul sera trop long.