TP 16
Manipulation d'images

Nous allons maintenant apprendre à manipuler les images. Les possibilités sont très larges et, en un seul TP, nous n'aurons qu'un petit aperçu du sujet. C'est aussi une très bonne idée d'utiliser ces méthodes pour le TIPE !

La première chose à faire est de récupérer le fichier `materiel.zip`. Il contient, en plus de deux versions d'un fichier `.py` à compléter, une certaine bibliothèque de photos pour le TP. Ouvrir le fichier en version niveaux de gris (pour l'instant) et exécuter au moins les deux premières cellules. Si tout se passe bien, une image en noir et blanc s'affiche.

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 niveau de gris** auquel cas chaque pixel est représenté par un simple nombre qui indique la luminosité du pixel.

Plus précisément, nous travaillons avec la bibliothèque `numpy` ainsi que `PIL`. Le code préparé charge une image avec `PIL` dont le nom est donné dans la variable `fichier` puis forme un tableau Numpy nommé `image` à deux dimensions, la première correspondant aux lignes et la seconde aux colonnes. Chaque pixel est codé sur un octet, soit 8 bits. Cela donne 256 valeurs possibles, 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).

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.

`X[0, 0]`	`X[0, 1]`	⋯	`X[0, p-1]`
`X[1, 0]`	`X[1, 1]`	⋯	`X[1, p-1]`
⋮	⋮	⋱	⋮
`X[n-1, 0]`	`X[n-1, 1]`	⋯	`X[n-1, p-1]`

Si on travaillait avec des couleurs, alors la variable `image` serait un tableau à *trois* dimensions, `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 si on sait bien utiliser `numpy`, mais passons pour l'instant. Enfin `image.shape` est un tuple de longueur 2 (sans couleurs) 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 `np.zeros((n, p))` (noir et blanc) ou `np.zeros((n, p, 3))` (couleurs) 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 à tout moment après avoir chargé l'image :

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

Remarque. Il existe aussi des images où chaque pixel contient *quatre* nombres : en plus des composantes RGB la dernière se nomme *canal alpha* et correspond à un niveau de transparence.

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 (mais on pourra à n'importe quel moment copier-coller le code de chargement d'image pour tester avec d'autres). 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.

Avertissement

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, et même si elles ne sont pas aussi belles cela sera certainement valorisé et valorisant !

Avertissement

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 librement**.

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, et si on veut changer d'image, il faut soit ré-exécuter toute la cellule qui charge `image`, soit recopier le code là où on en a besoin. Enfin quelques autres fonctions déjà prêtes permettent d'afficher l'image, voire d'en afficher deux l'une sur l'autre pour bien les comparer, et de sauvegarder le résultat. À vous de le tester.

Un dernier petit avertissement : les opérations sur les coefficients se font dans des entiers non-signés 8 bits, sur lesquels les valeurs au-delà de 255 reviennent à 0. Cela force parfois à tester avant un calcul si le résultat va dépasser ou non 255.

II Forme de l'image

Les premières fonctions que l'on veut coder sont le miroir horizontal 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 $(i, p-1-j)$, qui est 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 horizontal.

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

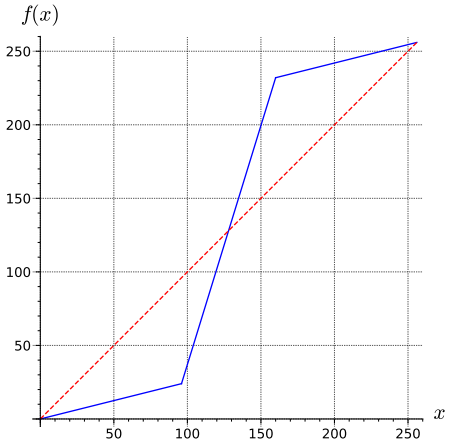
Pour augmenter l'éclaircissement d'une image, il suffit d'ajouter à chaque pixel une valeur fixe, par exemple 50 (l'effet sera bien visible, mais on pourra ajuster cette valeur plus tard). En effet la luminosité d'un pixel est un nombre entre 0 et 255, donc les augmenter tous de la même façon ne pourra 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 une valeur fixe de décalage b et on veut augmenter tous les pixels de la valeur b.

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

Une autre opération intéressante et très simple à programmer est le **seuillage**. Il s'agit de fixer une valeur s de seuil (typiquement $s = 127$ pour commencer) et de remplacer les pixels soit par 0 (une case noire) si la valeur est inférieure à s, soit par 255 (case blanche) si la valeur est supérieure à s. Cela doit plus ou moins faire apparaitre des formes, surtout sur les objets sombres se détachant bien d'un fond clair. Tester la fonction avec différentes valeurs du seuil.

Exercice 4
Écrire la fonction `seuillage(X, s)` qui effectue cette opération, avec le seuil `s`, et tester avec différentes valeurs du seuil.

Le seuillage est un cas simplifié de l'opération de changer le 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 (si x est petit alors $f(x) < x$, si x est grand alors $f(x) > x$), une fonction dont le graphe ressemble à ceci :



On propose pour cela la fonction

$$f : [0, 255] \longrightarrow [0, 255]$$
$$x \longmapsto \begin{cases} \frac{x}{4} & \text{si } 0 \leqslant x < 96 \\ 24 + \frac{13}{4}(x - 96) & \text{si } 96 \leqslant x < 160 \\ 232 + \frac{1}{4}(x - 160) & \text{si } 160 \leqslant x \leqslant 255 \end{cases}$$

Vérifiez qu'il s'agit bien d'une fonction *continue* avec le comportement voulu. Il s'agit d'une fonction *affine par morceaux*.

Remarque. Il est facile de produire de tels exemples en décidant par quels points le graphe passe, ou avec quel pente : un morceau de fonction affine est défini uniquement par deux points de passage, ou bien par un point et une pente. Ici f est la seule fonction possible découpant l'intervalle $[0, 255]$ en trois morceaux de tailles 96, 64, 96 (soit les fractions 3/8, 1/4, 3/8 de 255) avec les pentes 1/4 au début et à la fin.

Exercice 5
Écrire la fonction `contraste(X)` qui augmente le contraste de l'image `X` selon le procédé décrit.

Enfin 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 6
Écrire la fonction `negatif(X)` qui donne le négatif de l'image `X` selon ce procédé.

IV Flou

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. On utilisera 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 7
Écrire la fonction `flou(X)` qui floute l'image `X` selon ce procédé. On ne se préoccupera pas tout de suite des bords.

Remarque. De nombreuses fonctions différentes de flous peuvent être obtenues en considérant une moyenne d'un plus grand nombre de pixels autour du pixel central, et en faisant varier les coefficients. L'idée est que plus on considère une moyenne sur un grand nombre de pixels, plus l'effet de flou sera fort. Un cas connu des graphistes est le **flou gaussien** dans lequel les poids accordés aux pixels voisins suivent une loi gaussienne en fonction de la distance au pixel central. Ainsi plus la gaussienne est large, plus les pixels voisins considérés dans le calcul de la moyenne sont nombreux, et plus l'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. Tous ces flous font partie des **flous linéaires**, et la donnée de tous les coefficients pour les pixels voisins est appelée **matrice de convolution** ; l'opération de convolution consiste à remplacer un pixel par la moyenne des pixels voisins avec des coefficients donnés par la matrice.

IV.1 Contours

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. Et si deux pixels voisins ont des intensités déjà proches,

on considère qu'il n'y a pas de contour du tout. On commence donc par calculer la moyenne pondérée des pixels voisins selon la règle suivante :

-1	-1	-1
-1	8	-1
-1	-1	-1

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. On choisit donc une valeur de seuil, et si la moyenne (en valeur absolue) dépasse le seuil on mettra le pixel à 0 (tout noir) et sinon à 255 (tout blanc).

Exercice 8
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`. Il est nécessaire de tester manuellement avec différents seuils.

Remarque. En pratique, une détection automatique de contours capable de reconnaitre des formes et de les mesurer précisément se fait en de nombreuses étapes : d'abord un éventuel pré-traitement pour lisser l'image, puis une détection des contours comme ici, éventuellement avec un seuil qui s'ajuste automatiquement selon la proportion des pixels qu'on veut garder ; puis éventuellement un post-traitement pour délimiter bien précisément des contours et des zones, et enfin on peut marquer nettement les régions dessinées.

V Mettez de la couleur dans votre vie !

Le fichier à compléter se trouve dans une autre variante pour traiter les images en couleur. La variable `image` est un tableau `numpy` de forme $(n, p, 3)$.

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, dans ce cas 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 b à chaque pixel — se fait sur `image[i, j]` *en tant qu'opération vectorielle* et donc sur chacune de ses 3 composantes, et donc elle ajoute b dans chaque couleur de chaque pixel. Cette fonction (du moins sans le test pour vérifier si la valeur dépasse 255) marche donc exactement de la même façon en couleurs. C'est aussi le cas du miroir, puisque l'opération `Y[i, j] = X[i, p-1-j]` copie le tableau de taille 3 de `X` vers `Y`. Et du négatif, qu'on peut en fait écrire en une seule ligne `Y = 255 - X`. Quant au flou, l'opération de moyenne des pixels voisins est effectuée elle aussi de façon vectorielle dans chacune des 3 composantes couleurs.

Exercice 9
Tester et éventuellement adapter les fonctions précédentes en couleur.

Une fonction à adapter cette fois-ci dans les trois couleurs :

Exercice 10
Écrire une fonction `seuillage(X, a, b, c)` qui effectue un seuillage séparément sur chacune des couleurs : dans le rouge, l'intensité du la couleur sera comparée à a et le pixel sera mis soit à 0 (noir) soit à 255 (rouge pur), de même b sera le seuil de vert et c le seuil de bleu. À la fin l'image est constituée uniquement de noir, de blanc, rouge pur, vert pur, bleu pur, et des combinaisons de ces couleurs.

Enfin la dernière opération est amusante, testez-là sur diverses images !

Exercice 11
Écrire une fonction `fusion(X1, X2)` qui fusionne les deux images, c'est à dire que chaque pixel (i, j) du résultat sera la moyenne des pixels (i, j) de chacune des deux images. Cela nécessite que les deux images soient exactement de la même taille.

Puis tester avec deux images choisies parmi celles proposées, en recopiant deux fois le code qui permet d'ouvrir un fichier image.

VI Pour aller plus loin

Quelques pistes d'améliorations à tester vous-même :

1. Vérifier que toutes les fonctions marchent bien en couleur, et les corriger s'il faut.
2. Vérifier avec plusieurs images différentes, éventuellement avec vos propres images.
3. Sauvegarder son image en l'enregistrant, avec les fonctions déjà écrites dans le fichier joint qui font appel à `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. Sans vectorialisation, certaines fonctions seront bien trop lentes sur des images de grande taille.