TP 10 - Images

1 Tableaux numpy

Vous avez vu abordé en cours la bibliothèque `numpy` et son principal objet, les tableaux en une ou plusieurs dimensions qui permettent l'optimisation de nombre de calculs. Un point important est que, contrairement aux listes en python, les tableaux numpy sont constitués d'objets de même type, lesquels, le plus souvent, occupent une place identique en mémoire : des nombres en virgule flottante qui occupent 8 octets (64 bits ou chiffres binaires), ou 4 voire 2, des entiers signés ou non, sur 1, 2, 4 ou 8 octets (8, 16, 32 ou 64 bits), ce qui permet d'organiser en mémoire toutes les valeurs stockées dans le tableau de manière contiguë et qui rend les accès en mémoire plus efficaces et la vectorialisation possible.

Le type entier (`int`) en python est très différent : il n'y a guère de limite à la taille d'un entier en python, à tel point que demander la valeur de `3**3000` ne pose aucun souci (tandis que l'exécution de `3.0**3000` conduit à un dépassement de capacité : le type flottant en python est lui, bel et bien limité, puisqu'un nombre en virgule flottante est stocké sur 8 octets)

Vous vous en doutez, pour pouvoir calculer `3**3000`, le noyau python est obligé d'allouer une quantité de mémoire suffisante pour pouvoir contenir tous les chiffres binaires permettant de représenter un tel nombre, et deux entiers en python n'occuperont alors pas toujours la même place en mémoire. C'est une chose qu'on ne se permettra pas avec les tableaux numpy, où chaque valeur occupera la même place en mémoire, laquelle est décidée à l'initialisation du tableau :

```
> import numpy as np
> x = np.array([10], dtype = np.uint8) # on crée un tableau d'une seule valeur, laquelle est
# déclarée être un entier non signé codé sur 8 bits.
> x + 250 # qu'obtient-on ? comment cela s'explique-t-il ?
> x * 100 # même question
```

Voir aussi les rappels de syntaxe à la fin de l'énoncé…

Remarque 1. Les types numériques définis par numpy comprennent :

- des entiers signés : `int8, int16, int32, int64`
- des entiers non signés : `uint8, uint16, uint32, uint64`
- des nombres en virgule flottante : `float16, float32, float64` (ce dernier type est le même que le type `float` défini dans le noyau python)

Dans ce qui suit, on n'utilisera que le type `uint8` qui permet de coder un entier ayant une valeur entre 0 et 255.

2 Chargement, affichage d'une image

Une image informatique peut être de deux natures différentes : matricielle (ou bitmap) ou vectorielle.

Une image vectorielle est associée à un langage permettant de décrire des formes, des segments, des arcs de cercle, des courbes de Bézier par exemple. L'affichage d'une image vectorielle consiste à dessiner l'ensemble des objets qui la constituent. Cette opération peut être coûteuse en mémoire et en temps de calcul, mais l'avantage est qu'une telle image ne souffre pas de l'effet de pixelisation associé au images matricielles (les seules que l'on étudiera...) lorsqu'on zoome sur celle-ci.

Une image matricielle est quant à elle constituée d'un rectangle de pixels dont la couleur est codée avec une certaine précision (le plus souvent 24 bits par pixel répartis en 8 bits selon les trois composantes rouge, vert et bleu, ou pour le noir et blanc, en niveaux de gris, le plus souvent codés sur 8 bits)

On associe souvent à une image matricielle sa résolution, exprimée en points par pouce (un pouce égale 2,54cm environ), qui combinée à sa définition (i.e. le nombre de pixels en largeur et en hauteur) détermine la taille que l'image est supposée occuper sur un écran, ou à l'impression.

Les formats les plus classiques pour le stockage d'images matricielles sont :

- Bitmap (extension `bmp`) : format non compressé, sans compression
- Gif : format compressé, mais sans perte. De plus en plus rare (limité à 256 couleurs pour une image)
- Png : format compressé, également sans perte.

- Tiff : format compressé ou non (le plus souvent utilisé sans compression, conduisant à des fichiers de grosse taille)
- jpeg (extension `jpg`) format compressé, avec perte. Le facteur de compression est ici bien meilleur que les formats compressés précédents, mais l'image restituée n'est pas exactement identique à l'originale...

Avec le module `PIL`, il est facile d'importer dans une session python une image de l'un ou l'autre des formats précédents.

Allez déjà récupérer les images présentes sur `http://cahier-de-prepa.fr/pcsi-bdb/docs?rep=112` et les sauvegarder dans le répertoire de votre choix, ainsi que le script . Sauvegarder alors le code source de l'éditeur **dans ce même répertoire** en lui donnant le nom de `TP10.py`.

Introduisez dans le script les commandes suivantes (on va avoir besoin des modules numpy et de la bibliothèque matplotlib) qui importent plusieurs modules python, et exécutez celui-ci : (par l'interpréteur courant)

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as mpl
```

Puis, dans la console, chargez la photographie de la punaise :

```
>>> img = Image.open('stinkbug.png')
>>> A = np.array(img)
```

`img` est alors un objet de type `PIL` mais on va le convertir en un tableau numpy :

```
>>> A = np.array(img); A.shape
(375, 500, 3)
>>> import matplotlib.pyplot as mpl; mpl.imshow(A)
```

(375 lignes de 500 pixels, chacun ayant trois attributs de 8 bits pour les trois couleurs primaires rouge, vert et bleu)

On ne va travailler que sur des images en niveaux de gris, et en pratique on ne va garder qu'une seule des trois composantes. Le choix fait est généralement celui de la couche verte, car c'est sur celle-ci que l'on a le plus d'information (à commencer par le fait que sur un capteur cmos traditionnel, il y a autant de photosites verts que de rouges et de bleus réunis). Dans l'exemple présent de la punaise, le choix de la couche n'a aucune importance, car elles sont ici identiques, l'image n'étant dès l'origine formée que de niveaux de gris.

```
>>> punaiseGris = A[:, :, 1]; mpl.imshow(punaiseGris); mpl.colorbar()
```

A noter que le tableau `gris`, désormais constitué de 375 lignes de 500 valeurs en `uint8` est ici interprété comme une information en luminance, et par défaut de petites valeurs sont affichées avec des couleurs « froides » (le bleu) et des grandes valeurs avec des valeurs « chaudes » (le rouge).

On peut obtenir de la fonction `imshow` (de matplotlib.pyplot) un autre palette de couleurs, ou ici plutôt, en l'occurrence, en dégradé de gris :

```
>>> mpl.figure(); mpl.imshow(punaiseGris, cmap = 'gray'); mpl.colorbar()
```

Il est deux autres arguments nommés de `imshow` qui nous seront utiles : `vmin` et `vmax` car une normalisation est faite avant l'affichage que nous ne souhaitons pas, donc rajoutez au script TP10 les lignes suivantes puis exécutez le script :

```
def affiche(A):
    mpl.figure()
    mpl.imshow(A, cmap = 'gray', vmin = 0, vmax = 255)

img = Image.open('stinkbug.png')
punaiseGris = np.array(img)[:, :, 1]
img = Image.open('chat.png')
chatGris = np.array(img)[:, :, 1]
img = Image.open('poivrons.png')
poivronsGris = np.array(img)[:, :, 1]
```

(Bien sûr, si vous êtes curieux, n'hésitez pas à utiliser une autre composante primaire pour comparer : par exemple définir aussi `poivronsGris2 = np.array(img)[:, :, 2]`, afficher et comparer. Pour afficher l'image initiale, en couleurs pour le babouin et les poivrons, juste après Image.open faites `mpl.imshow(img)`, tout simplement)

3 Traitement

3.1 Courbe de couleur

3.1.1 Ajustement de l'exposition

Un histogramme nous informe sur la répartition des luminances dans l'image :

```
>>> mpl.figure(); mpl.hist(punaiseGris.flatten(), 256, range=[0,255])
```

(`flatten()` est une méthode d'un tableau numpy qui retourne un tableau constitué des mêmes coefficients, mais unidimensionnel, ce dont on a besoin avec `hist`)

Une courbe en cloche est attendue. Si celle-ci se concentre sur des petites valeurs : l'image est sous-exposée, et sera sans doute bien sombre, si elle se concentre sur de grandes valeurs, elle est sur-exposée.

Exemple :

```
>>> surexpose = 1.8*punaiseGris; affiche(surexpose)
```

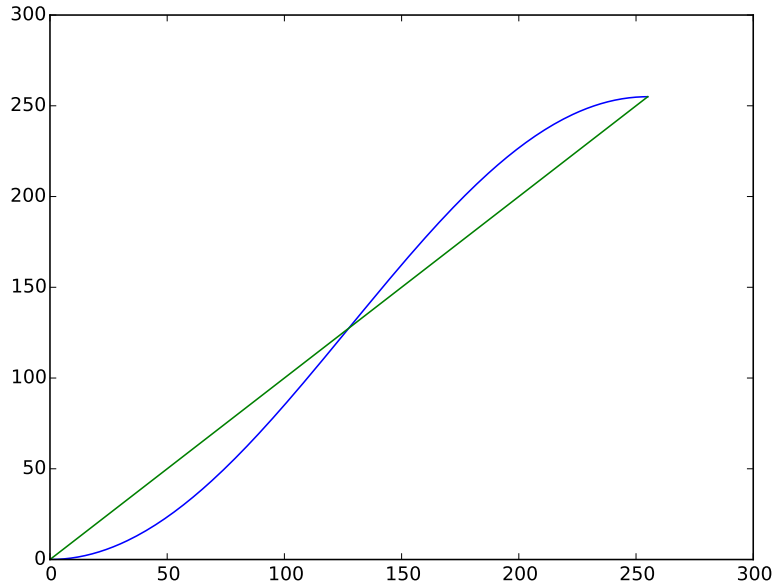
Essayer les traitements suivants :

```
>>> surexpose = 2.0*chatGris
>>> affiche(surexpose)
>>> sousexpose = 0.5*chatGris
>>> affiche(sousexpose)
```

Question 1. Devinette : l'affichage obtenu par `affiche(2*chatGris)` est très différent de celui obtenu par `affiche(2.0*chatGris)`. Quelle pourrait en être la raison ? (Allez voir l'annexe si nécessaire)

3.1.2 Ajustement non linéaire

Les photographes connaissent bien cette astuce : pour élargir la courbe en cloche dessinant la répartition des luminances dans l'image et gagner alors en contraste, on peut appliquer une fonction, dite en S :



Une bonne approximation d'une telle fonction est donnée par une sinusoïde, telle que par exemple :

$$f : x \mapsto (\sin\left(\pi\frac{x}{256} - \frac{\pi}{2}\right) + 1) \times 128 + 128$$

Question 2. Introduire une fonction d'entête `f(x)` qui prend un nombre, ou un tableau numpy `x` et qui renvoie le nombre $(\sin\left(\pi\frac{x}{256} - \frac{\pi}{2}\right) + 1) \times 128$. Bien sûr, on fera appel à la fonction `np.sin` (et le nombre π, sans surprise, est obtenu par `np.pi`). Bonne nouvelle, c'est numpy qui s'occupe de vectorialiser les calculs et de faire que la fonction s'applique à chacune des valeurs du tableau. A noter que le tableau numpy obtenu ne sera plus un tableau d'entiers, mais un tableau de flottants, mais la fonction `affiche` définie précédemment fera les conversions nécessaires…

Appliquer cette fonction aux trois images proposées, et afficher le résultat.

3.2 Un produit de convolution

On peut souhaiter, d'une image donnée :

- En réduire le « bruit »
- Flouter l'image
- En améliorer au contraire la netteté
- Détecter des contours

Un moyen simple est d'utiliser une convolution de l'image à l'aide d'un noyau bien choisi. Les noyaux considérés seront ici des matrices à coefficients réels de 3 lignes et colonnes ou bien 5 lignes et colonnes et dont la somme des coefficients vaut 1 (en fait, nous testerons un exemple où cette propriété n'est pas satisfaite)

Par exemple si le noyau est $K = \begin{pmatrix} -1 & 0 & -1 \\ 0 & 5 & 0 \\ -1 & 0 & -1 \end{pmatrix}$ et si la matrice des luminances est $A = \begin{pmatrix} 2 & 3 & 0 & 1 \\ 1 & 3 & 5 & 2 \\ 2 & 5 & 2 & 0 \end{pmatrix}$, alors le produit de convolution de A par K est la matrice $(b_{i,j})$ telle que, par exemple : $b_{2,2} = -a_{1,1} - a_{1,3} + 5\,a_{2,2} - a_{3,1} - a_{3,3} = 9$, $b_{2,3} = 5 * 5 - 3 - 1 - 5 = 16$.

(En d'autres termes, la nouvelle luminance pour le pixel obtenu est une combinaison linéaire de sa luminance initiale et de celle de ses pixels adjacents, les coefficients de cette combinaison linéaire étant ceux du noyau...)

On choisit de mettre à zéro les coefficients des première et dernière colonnes et lignes de B. Une autre approche serait de considérer que A peut être élargie avec des coefficients nuls, mais ce n'est pas celle que l'on adoptera ici.

Question 3. Ecrire alors une fonction, qu'on nommera `convolution` admettant deux arguments : un tableau bidimensionnel `A` correspondant aux luminances d'une image à traiter, et un second tableau carré comportant un nombre impair de lignes et colonnes `K` : le noyau de convolution.

On commencera par initialiser un nouveau tableau numpy, aux même dimensions que `A`, à l'aide de la fonction `zeros` de numpy (on obtiendra alors par défaut une matrice de flottants, mais ce n'est pas plus mal pour les calculs qui vont suivre) puis on ajustera, avec quatre boucles imbriquées (!) tous les coefficients intérieurs de cette nouvelle matrice (en laissant leur valeur initiale nulle aux coefficients sur les bords).

Par exemple, en reprenant les exemples numériques précédents :

```
>>> convolution(A, K)
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  9., 16.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Tester désormais sur les images importées les noyaux suivants :

$$\frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \frac{1}{10}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

A noter que les calculs sont longs (sans doute de l'ordre d'une quinzaine de secondes). Bien sûr, on affichera pour chaque exemple l'image obtenue après transformation.

Question 4. Pourquoi le dernier noyau de convolution conduit-il à une image essentiellement noire ? Comment l'obtenir en négatif ? (Donc essentiellement blanche...)

3.3 Un peu de stéganographie (à terminer à la maison)

8 bits par couleur primaire fournissent 16,7 ($(2^8)^3$) millions de couleurs différentes, mais il va sans dire qu'elles ne sont pas toutes discernables à l'oeil nu, à tel point qu'en modifiant un, deux ou trois bits de poids faible (voire 4) pour chaque luminance de chaque couleur primaire de chaque pixel, la différence n'est souvent qu'assez peu perceptible. C'est l'idée derrière la stéganographie, où on modifie des bits de poids faible des luminances d'une image pour y cacher des informations secrètes.

Par exemple, dans l'image nommée `'chat.png'` est cachée une autre image. Ce qui a été fait est la chose suivante : pour la luminance de chaque couleur primaire, les deux bits de poids faible ont été remplacés par les deux bits de poids fort d'une autre image. Bien entendu, chaque pixel de cette image cachée n'est alors connu qu'avec 4 couleurs, mais cela devrait suffire à reconnaître le sujet de cette image.

A vous de jouer pour faire apparaître cette image secrète… (En pratique, il suffit de faire des décalages bit à bit, voir le paragraphe suivant pour quelques rappels de syntaxe…)

4 Annexe : rappels de syntaxe, numpy, opérations bit à bit

- Etant donné un tableau (`nparray`) numpy, dans la variable d'instance nommée `shape`, on accède aux dimensions du tableau. Par exemple, le tableau `A` créé à partir de l'image de la punaise a pour dimensions $(375, 500, 3)$ (pour une images selon les trois couleurs primaires rouge-vert-bleu, de 375 lignes de 500 pixels)

- Une seconde variable d'instance nommée `dtype` indique le type de chaque valeur du tableau. Dans le cas présenté ici, il s'agit de `dtype('uint8')` qui est un type défini par numpy, permettant de stocker un entier non signé sur 8 bits donc pour 256 valeurs différentes.

- Avec la syntaxe `A[i,j,c]` on accède à la luminance du pixel situé ligne `i`, colonne `j` et pour la couche de couleur `c` (rouge si `c` vaut 0, verte si `c` vaut 1 et bleue si `c` vaut 2)

- Avec des slices, on accède à ce qu'on appelle une « vue » de `A`. Par exemple, `A[:, :, 0]` forme un tableau numpy bi-dimensionnel, lequel est formé de toutes les luminances selon la couche de couleur rouge (375 lignes de 500 colonnes)

  Si on pose `B = A[:, :, 0]`, alors tout accès en lecture ou en écriture à `B` correspond à un accès en lecture ou en écriture à la couche 0 du tableau `A`. Par exemple, en écrivant `B[0,1]=2`, alors on a modifié aussi la valeur de `A[0,1,0]`.

  Un avantage immédiat est que créer le tableau `B` ne vient pas dupliquer les informations de `A` et n'occupe presque pas de mémoire supplémentaire.

- On peut faire agir des opérations arithmétiques et des fonctions sur des tableaux numpy (pourvu qu'il s'agisse de fonctions définies dans le module numpy…) : par exemple `np.sin(A)` va calculer le sinus de tous les éléments du tableau `A` (les $375 \times 500 \times 3$ éléments qui le constituent) en une seule opération (et le calcul sera beaucoup plus rapide que celui que donneraient trois boucles for imbriquées) pour donner un tableau de $375 \times 500 \times 3$ nombres flottants.

- Les opérations `2*A` et `2.0*A` sont bien différentes. La première conduit à un tableau qui sera formé, tout comme `A`, d'entiers non signés codés sur 8 bits, le second conduit à un tableau de flottants (sauf mention expresse du contraire, les flottants sont codés sur 64 bits)

- En base 2, multiplier par 2 revient à décaler l'écriture d'un chiffre vers la gauche (en mettant un 0 comme chiffre des unités). Cette opération arithmétique (pour des entiers) peut s'écrire alors `x << 1`. Multiplier par 4, est décaler de deux chiffres vers la gauche et on peut écrire `x << 2` et ainsi de suite. Il existe aussi l'opération `x >> n` qui correspond à l'opération `x//(2**n)`. En pratique les chiffres qui « sortent » de l'écriture sont perdus : en supposant que `x` est un tableau numpy formé d'entiers non signés de 8 bits, multiplier par deux (décaler d'un chiffre binaire à gauche) une valeur telle que $129 = 128 + 1$ dont l'écriture binaire est `10000001` conduit à la valeur 2 (`00000010`), et diviser par deux (décaler d'un chiffre binaire à droite) conduit à la valeur 64 (`01000000`).