TP Images

Aller sur `https://cahier-de-prepa.fr/mpsi-kju/docs?rep=124` (Informatique/Documents à télécharger/Pour les MP/images) et y télécharger les trois images `stinkbug.png`, `poivrons.png` et `babouin.png`. Lancer bien sûr spyder et poursuivre votre lecture.

1 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...)

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, 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. (Sans le module `PIL`, on a seulement accès aux images `png` par le biais de fonctions de matplotlib)

Ouvrir et exécuter un script en lui donnant le nom `TP5`. Assurez-vous que le script et les images sont dans le même répertoire.

```
>>> from PIL import Image
>>> img = Image.open('stinkbug.png')
```

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

```
>>> A = 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, 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` (autrement dit des entiers de 0 à 255) 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 TP5 les lignes suivantes puis exécutez le script :

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

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('babouin.png')
babouinGris = 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 `imshow(img)`, tout simplement)

2 Traitement

2.1 Courbe de couleur

2.1.1 Ajustement de l'exposition

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

```
>>> mpl.figure(); mpl.hist(punaiseGris.flatten(), 256)
```

(`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 :

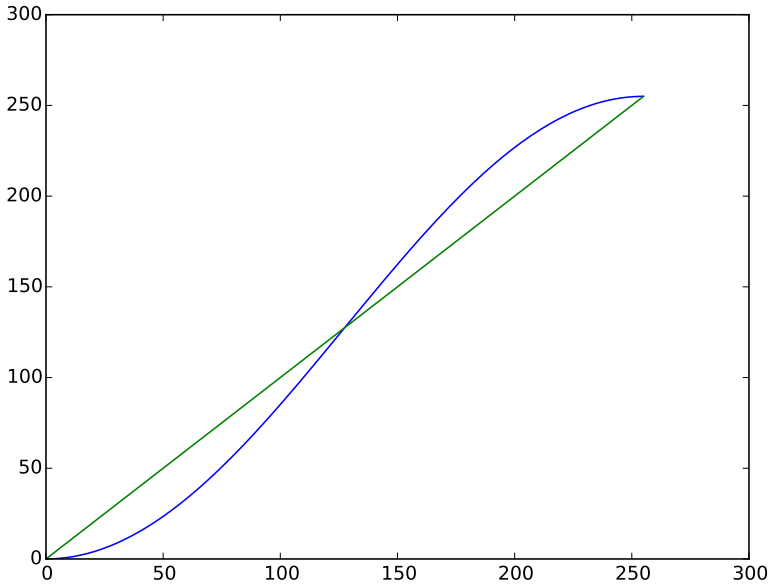
```
>>> surexpose = 1.8*punaiseGris; affiche(surexpose)
```

Essayer les traitements suivants :

```
>>> surexpose = 1.5*babouinGris
>>> affiche(surexpose)
>>> sousexpose = 0.5*babouinGris
>>> affiche(sousexpose)
```

2.1.2 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 : $x \mapsto (\sin\left(\pi \frac{x}{256.0} - \frac{\pi}{2}\right) + 1) \times 128$

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

2.2 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 : $b_{2,2} = -a_{1,1} - a_{1,3} + 5a_{2,2} - a_{3,1} - a_{3,3} = 9$, $b_{2,3} = 5 * 5 - 3 - 1 - 5 = 16$.

(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.

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.

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

3 Décomposition en valeurs singulières et compression

Comme on l'a vu, toute matrice A réelle, carrée ou non, admet une décomposition en valeurs singulières $A = U\Sigma^t V$ où U et V sont orthogonales (pas forcément de même dimensions !) et où Σ est diagonale de termes diagonaux positifs et décroissants $\sigma_1 \geqslant \sigma_2 \geq \ldots \sigma_r > 0$ où r bien sûr est le rang de A.

En d'autres termes, si A compte n lignes et p colonnes, et si on note $C_1, \ldots, C_p$ les colonnes de V, $C'_1, \ldots, C'_n$ les colonnes de U, alors $A = \sigma_1 {C'_1}^t C_1 + \cdots + \sigma_r {C'_r}^t C_r$.

Si les σ_i décroissent assez vite, on peut penser qu'on peut négliger la fin de la somme précédente et obtenir une matrice (de rang strictement inférieur à celui de A) proche de A (au sens de la norme 2, ou n'importe quelle norme bien sûr).

Obtenir cette décomposition avec numpy est très facile :

```
>>> from numpy.linalg import svd
>>> (u, s, v) = svd(A)
```

La matrice `v` obtenue correspond plutôt à la matrice transposée de V de l'expression précédente, si bien que, aux erreurs de calcul près, on retrouve la matrice initiale, si elle est carrée, par

```
>>> dot(u, dot(diag(s), v))
```

(Bien sûr, dans le script, on préfixera les fonctions `dot` et `diag` issues de numpy par `np`.)

Ecrire une fonction qu'on pourra nommer `reconstruit`, admettant quatre paramètres `u, s, v, k` où `u, s, v` sont les valeurs de retour obtenues par l'appel à `svd` et où `k` est un entier, et qui retourne la matrice constituée de la somme précédente jusqu'à la k-ième valeur singulière.

Cela revient en fait à multiplier trois matrices : la matrice formée des k premières colonnes de u, celle carrée de termes diagonaux $\sigma_1, \ldots, \sigma_k$ et enfin celle formée des k premières colonnes de v.

A noter que si la matrice initiale A compte n lignes et p colonnes, et si $k = \min(n, p)$, alors avec : `reconstruit(u, s, v, k)` on doit retrouver la matrice A... (Une valeur plus grande que ce k conduira bien sûr à une erreur)

Calculer les décompositions en valeurs singulières des matrices représentant les images importées et afficher les images obtenues en approchant celles-ci avec plusieurs nombres de valeurs singulières (de 5 en 5 peut-être, ou comme vous voulez).