Feuille de TP Python 04

Matrices/Fonctions de deux variables/Traitement d'images

L'objet de ce TP est essentiellement de mettre en place les aspects les plus élémentaires de traitement d'images N&B.

1 Qu'est ce qu'une image ?

1.1 Images bitmap, format brut et formats de fichier

Les photos, dessins et autres images complexes que l'on peut voir sur nos écrans sont, pour la plupart des images *bitmap* : une telle image est composée d'une grille rectangulaire, ayant une hauteur et une largeur, dont chaque cellule est appelée un *pixel* [1]. Chaque pixel possède une couleur, codée par trois [2] nombres entre 0 et 255 [3], chacun de ces nombres donnant la quantité d'une des couleurs fondamentales R (rouge), G (vert) ou B (bleu). La description d'une image bitmap par la donnée brute de ce tableau de couleurs de pixels prend donc au minimum hauteur $\times$ largeur $\times$ 3 octets. Sachant qu'un écran standard fait en 2015 de l'ordre de 1080×1920 pixels, une photo de fond d'écran brute pèse donc de l'ordre de 6Mo.

Les images sont un domaine où les mécanismes de compression de données sont cruciaux.

Compresser les données, c'est faire en sorte qu'elles occupent le moins de place possible en mémoire. Un exemple extrême est le cas d'une image blanche de taille 1080×1920. D'après le calcul précédent, son codage brut prend 6Mo. Un codage du type « place 1080x1920 octets blancs de suite » prend quelques octets.

Les formats d'image que l'on trouve le plus couramment sont le format JPEG (c'est ce qui sort de votre APN), le format GIF et son cousin le format PNG. Ce sont tous des formats compressés. Le fond d'écran en format JPEG prendra de l'ordre de 600ko, soit 10 fois moins que le format brut.

Nous allons manipuler des images : *i.e.* les lire du disque, où elles sont compressées, les transformer lorsqu'elles sont sous un format brut, écrire le résultat, compressé, sur le disque.

On partira donc d'une image compressée, que l'on décompresse pour pouvoir la manipuler pixel par pixel, pour la recompresser à l'écriture.

Ces considérations sont mises en avant pour que l'on se rende bien compte de la quantité de données manipulée : nos algorithmes vont devoir s'occuper de tableaux contenant de l'ordre du million de nombres.

1.2 Lecture, stockage et écriture

Un paquet d'images est disponible dans le répertoire `images/`.

Il y a beaucoup de bibliothèques permettant la manipulation d'images, notamment l'import export. En sus de `numpy` et `matplotlib.pyplot`, nous utilisons une bibliothèque importée via la commande `import matplotlib.image as mpimg`.

Les fonctions de lecture et d'écriture de cete bibliothèque ne fonctionnent (à coup sûr) qu'avec les images au format `.png`. Pour les images au format `.jpg`, il faut tester et, peut-être, installer un package supplémentaire.

Au moment de la lecture, l'image est placée dans une matrice (un `ndarray`). Suivant que l'image est en couleur ou pas, le comportement est différent :

1. Image couleur. Tapez les commandes suivantesdans un script nommé `td04-intro.py`.

```
import numpy as np # on importe la bibliothèque numpy avec son surnom standard
import matplotlib.pyplot as plt # bib matplotlib usuelle
import matplotlib.image as mpimg #bibli. matplotlib images

imgcol = mpimg.imread('images/lena_color.png')
plt.imshow(imgcol) #affiche l'image
print(imgcol) # affiche la matrice contenant l'image
print(imgcol.shape) # affiche les dimensions de la matrice
```

1. *Pict*ure *El*ement
2. parfois 4, le 4e est la transparence
3. Un *octet* code chacun de ces nombres, l'unité fondamentale de mémoire dans un ordinateur est le *bit*, valant 0 ou 1. Un *octet* est une succession de 8 bits. Le codage de la couleur d'un pixel pèse donc 3 octets. Un kilooctet (ko), c'est 2^{10} octets, un mégaoctet (Mo) c'est 2^{10} ko, *i.e.* 2^{20} octets, un gigaoctet, c'est 2^{10} Mo, *i.e.* 2^{30} octets. Ceci est à comparer à la capacité mémoire en mémoire vive (4 ou 8 Go en 2016), à la capacité d'un disque (2000 Go en 2016), d'une clé USB (64 ou 128 Go)

L'image est stockée dans une matrice 512×512 de triplets de nombres. Chaque nombre dans un triplet représente un canal de couleur. Les valeurs sont des `float` entre 0 et 1 (au lieu d'être des entiers entre 0 et 255)

2. Image Noir et Blanc. Continuez avec les commandes suivantes

```
plt.figure() #nouvelle image
imgnb = mpimg.imread('images/lena_bw.png')
imgplot=plt.imshow(imgnb) #affiche l'image
print(imgnb) # affiche la matrice contenant l'image
print(imgnb.shape) # affiche les dimensions de la matrice
```

L'image est stockée dans une matrice 512×512 de nombres. Chacun de ceux-ci représente le niveau de gris du pixel correspondant. Les valeurs sont des `float` entre 0 et 1 (au lieu d'être des entiers entre 0 et 255).

Dans cette suite de commandes, le graphisme est stocké dans l'objet `imgplot`. Il ne faut pas confondre l'image (la matrice) avec l'objet qui en fait le dessin à l'écran, le graphisme.

La représentation est colorée. Il s'agit d'une représentation en *pseudo-couleurs*. Cela signifie que `Matplotlib` traduit le nombre correspondant à chaque pixel en une couleur.

On peut changer la façon dont un nombre est transformé en couleur (la *color map*) avec la méthode `set_cmap()` de l'objet `imgplot`. Le paramètre en est une chaine de caractère, on peut trouver la liste sur

`http://matplotlib.org/examples/color/colormaps_reference.html`

Pour voir les codes couleurs, on peut ajouter `plt.colorbar()`.

Taper dans la suite du script les commandes suivantes

```
imgplot.set_cmap('gray') #redéfinit la color map pour avoir une image
                         # NB, essayer aussi 'jet', 'hot', 'spectral'
plt.colorbar() # montre la légende des codes couleurs
```

Pour clore cette introduction, comme une image NB est une matrice de nombres, on peut lui appliquer les opérations matricielles standard. Voici un exemple de somme, sauvé ensuite dans une image. Ajoutez les commandes suivantes

```
plt.figure() #nouvelle figure
imgnb2= mpimg.imread('images/boat.png')+imgnb # charge une autre image
                                              # et fait la somme des deux matrices
imgplot2=plt.imshow(imgnb2)
imgplot2.set_cmap('gray')
plt.colorbar() #montre la barre de couleurs
mpimg.imsave('lena-boat.png',imgnb2,cmap='gray') #le format se déduit du nom de fichier
```

On remarque que les codes couleurs sont affectés en utilisant toute la gamme de valeurs présentes dans la matrice. Ici les valeurs sont entre 0 et 2. On peut s'en assurer en affichant`imgnb2` dans la console. 2 est très blanc, alors que 0 est noir avec le dégradé intermédiaire.

2 Représenter graphiquement une matrice, une fonction de deux variables réelles

Le système de représentation des images peut aussi servir à représenter des matrices « normales » ou des fonctions de deux variables.

— Essayer l'exemple suivant dans un nouveau script `td04-rep-matrice.py` après avoir importé les modules `numpy` et `matplotlib.pyplot`.

```
n=5
M=np.zeros((n,n))
for i in range(n):
    for j in range(n):
        M[i,j] = 1/(i+j+1)
plt.figure()
plt.imshow(M)
plt.colorbar()
plt.show()
```

— Essayer l'exemple suivant dans un nouveau script `td04-F2VR.py` après avoir importé les modules `numpy` et `matplotlib.pyplot`.

```
x=np.linspace(-np.pi,np.pi,200)
y=np.linspace(-4,4,200)
P=np.zeros((len(x),len(y)))
for i in range(len(x)) :
```

```
    for j in range(len(y)) :
        P[i,j] = x[i]**2 + np.sin(y[j])**2
plt.figure()
plt.imshow(P)
plt.colorbar()
plt.show()
```

Comment interpréter le rendu ?
Voir le script corrigé pour les détails pour améliorer la représentation de ce type de fonctions.

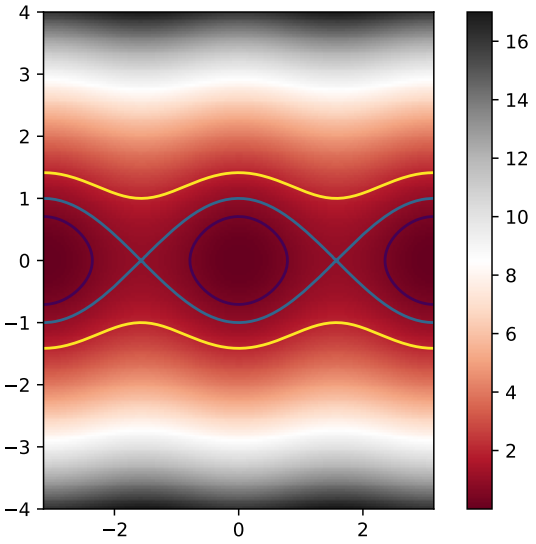

FIGURE 1 – Le graphe très amélioré.

3 Ajouter du bruit, filtrer

3.1 Le bruit blanc

Les images (ou plus généralement les signaux) issus d'un appareil de mesure [4] sont en général bruités. Cela signifie que toutes sortes de perturbations vont fausser la mesure de l'intensité lumineuse de l'image en chaque point.

Un tel bruit peut se modéliser, de façon probabiliste, par un bruit blanc. Une image, composée de bruit blanc pur, est une image dont l'intensité de chaque pixel est un nombre aléatoire tiré uniformément entre 0 et 255 (si chaque pixel est codé sur 8-bit, sachant que le zéro d'intensité est à 127) ou entre $-\frac{1}{2}$ et $+\frac{1}{2}$. (Il est important pour le bruit blanc que la moyenne soit nulle et que les pixels soient indépendants.)

3.2 Travail demandé

Dans un nouveau script `td04-bruitage.py`.

1. Ecrire une fonction `BruitBlanc(h,l)` qui retourne une matrice (un `ndarray`) comportant `h` lignes, `l` colonnes composée de nombres tirés au hasard uniformément entre $-\frac{1}{2}$ et $\frac{1}{2}$.
2. En utilisant la fonction précédemment construite, écrire une fonction `Bruite` qui prend en paramètre une image P et un intensité de bruit r et retourne l'image $P + r * B$.
3. Appliquer la fonction `Bruite` à votre image favorite en prenant plusieurs valeurs d'intensité de bruit : prendre r dans la liste 0.01, 0.05, 0.1, 0.15 et 0.2
4. Afficher et sauver (avec le nom de fichier `"nom initial"-bruit-"valeur de r".png`) l'image pour chacune de ces valeurs de r ainsi construite.

4. Un APN *est* un appareil de mesure !

3.3 Filtrer

L'opération de filtrage linéaire est en quelque sorte l'opération inverse du bruitage [5]

L'opération est la suivante : pour chaque pixel de l'image de départ, on calcule une moyenne des valeurs des pixels voisins, cette moyenne devient la valeur du pixel de la nouvelle image. L'idée de base est que

1. Dans une image normale, un pixel et ses voisins ont essentiellement la même valeur
2. En moyennant le bruit, on l'annule.

Ce n'est pas totalement exact, la première assertation est notamment mise en défaut sur les bords des objets présents dans l'image initiale. On obtient donc une version « floutée » de l'image initiale.

La moyenne à prendre, ainsi que les voisins concernés, sont codés dans un « masque ». Il s'agit d'une (petite) matrice contenant les poids à affecter à chacun des voisins. Par exemple,

1. si le masque est la matrice
$$M_1 = \begin{pmatrix} 0 & +\frac{1}{4} & 0 \\ +\frac{1}{4} & 0 & +\frac{1}{4} \\ 0 & +\frac{1}{4} & +0 \end{pmatrix},$$
cela signifie que le pixel (représenté par le centre de cette matrice) sera remplacé par le moyenne des pixels qui sont situés à ses nord, sud, est et ouest.
2. si le masque est la matrice
$$M_2 = \begin{pmatrix} 0 & 0 & +\frac{1}{36} & 0 & 0 \\ 0 & \frac{1}{18} & +\frac{1}{9} & \frac{1}{18} & 0 \\ +\frac{1}{36} & +\frac{1}{9} & \frac{2}{9} & +\frac{1}{9} & +\frac{1}{36} \\ 0 & \frac{1}{18} & +\frac{1}{9} & \frac{1}{18} & 0 \\ 0 & 0 & +\frac{1}{36} & 0 & 0 \end{pmatrix},$$
cela signifie que le pixel (représenté par le centre de cette matrice) sera remplacé par la moyenne pondérée des pixels à moins de deux pas du pixel central. Le pixel est lui même affecté du poids $\frac{2}{9}$, ses voisins nord, sud, est, ouest du poids $\frac{1}{9}$, ses voisins NE, NW, SE, SW du poids $\frac{1}{18}$ et enfin les pixels à deux pas NN, SS, EE, et WW sont affectés du poids $\frac{1}{36}$.

On voit qu'il y a un problème avec les pixels au bord de l'image, qui manquent de voisins. Pour les expériences, on ne traitera pas ces points et l'image filtrée sera donc de la même taille que l'image originelle avec des « bords » identiques.

3.4 Travail demandé

Dans un fichier `td04-filtrage.py`,

1. Ecrire une fonction `FiltreLineaire(P,M)`, qui étant donnée une image `P`, un masque `M` (tous deux des `ndarray`), retourne l'image `P` filtrée par le masque `M`
2. Appliquer à chacune des images bruitées précédemment sauvées, les deux filtres M_1 et M_2 décrits ci-dessus. Afficher et sauver les images filtrées sous le nom `"nom image bruitee"-filtre-"nom du filtre".png`.

5. Attention, ce ne peut être exactement l'opération inverse : une fois qu'on a perdu des données, on ne peut les retrouver exactement ! On cherche à obtenir un résultat raisonnable, moins bruité