TD n^o 1 : Introduction à Numpy

Nous pourrions utiliser des listes classiques de Python pour représenter des listes de nombres ou des matrices, et dans ce cas, nous pourrions écrire par exemple `t = [5, 2, 4, 1]`*. Avec cette structure de données de Python, rajouter des nombres à une liste se fait en utilisant méthode* `append` *(par exemple, après avoir exécuté* `t.append(18)`*, la liste t vaut alors* `[5, 2, 4, 1, 18]`*).*

Rappelons que les listes Python peuvent contenir des éléments de types différents, ce qui nous permet d'écrire par exemple `[4, 3.1, "cou"]` *qui contient un entier, un flottant et une chaîne de caractères. Cette souplesse de Python peut être critiquée mais elle est appréciable dans différentes situations. Cependant le fait de permettre des listes d'éléments de types différents ne permet pas de faire des calculs calculs numériques efficaces.*

La librairie `numpy` *(provient de «numeric» et «Python») propose des tableaux de type* `ndarray` *que nous appellerons tableaux «numpy» adaptés au calcul numérique.*

Exercice 1 : (**Prise en main de** `numpy`)

Il faut au prélable importer la librairie `numpy`, l'usage étant de la rebaptiser `np`.

```
>>> import numpy as np
```

On crée un tableau «numpy» avec la fonction `np.array` (en anglais «array» signifie tableau).

```
>>> t = np.array([5, 2, 4, 1])
>>> print(type(t))
<class  numpy.ndarray >
```

Le type renvoyé est `ndarray` qui signifie «n-dimensional array», c'est-à-dire «tableau à n dimensions». Ici t ne possède qu'une dimension. Une matrice sera un tableau à deux dimensions. Par exemple

```
>>> A = np.array([[1 , 2, 3], [4, 5, 6]]) # crée une matrice à 2 lignes et 3 colonnes
>>> print(A)
array([[1, 2, 3], [4, 5, 6]])
>>> A.ndim # donne la dimension du tableau A (ici: 2)
```

Remarque : une image numérique RGB sera codée par une «matrice de pixels» dont le coefficient d'indice (i, j) est un triplet de nombres (r, g, b) définissant l'intensité en rouge, vert et bleu du pixel en position (i, j). Cela pourra être représenté par un tableau `numpy` à 3 dimensions.

Accès aux éléments : comme avec les listes. Par exemple, `t[0]` vaut `5` et `A[0][1]` vaut `2`. On peut aussi extraire une partie du tableau par "tranche" (slicing). Par exemple, `t[:2]` renvoie `[5,2]` les deux premiers éléments de `t`. Signalons que des syntaxes particulières sont permises : par exemple `A[0,1]` correspond à `A[0][1]` et l'instruction `A[1,:]` renverra toute la ligne de A d'indice `1` donc `[4, 5, 6]`.

On obtient le format du tableau avec la méthode `shape` qui renvoie un n-uplet :

```
>>> t.shape
(4,) # 4 lignes (donc un vecteur colonne)
```

```
>>>> A.shape
(2, 3) # 2 lignes et 3 colonnes
```

Remarque : si l'on ne connaît pas `shape`, on peut s'en sortir avec la fonction `len`

```
>>> len(t)
4
```

```
>>> len(A) # nb de lignes
2
```

```
>>> len(A[0]) # nb de colonnes
3
```

Ce sont des Objets mutables : on modifie les éléments comme pour les listes.

```
>>> t[0] = 9
>>> t
array([9, 2, 4, 1])
```

Attention aux copies. Comme tout objet mutable, l'affectation `t2 = t` ne va pas réaliser une copie de `t`.

```
>>> t
array([5, 3, 4, 1])
```

```
>>> t2 = t
>>> t2[0] = 7
```

```
>>> t
array([7, 3, 4, 1]) # contamination de t
```

Les variables `t` et `t2` sont deux étiquettes d'un même objet (on dit qu'elles pointent vers le même objet). On utilisera `np.copy(t)` (ou encore `import copy` puis `copy.deepcopy(t)`) pour faire des copies «profondes» de tableau.

Première différence fondamentale avec les listes : les éléments d'un tableau «numpy» ont nécessairement le même type. Par exemple, l'instruction
```
for x in t:
    print(type(x))
```
affiche quatre fois `<class numpy.int32>`. Les éléments de `t` sont des entiers relatifs codés sur 32 bits. Si l'on affecte un flottant à ce tableau, il sera converti en «entier 32 bits».

```
>>> t[0] = 5.3
>>> t
array([5, 2, 4, 1])
```

On peut choisir le type des données avec l'argument `dtype`. Par exemple, l'instruction `np.array([2,1,3], dtype = 'uint8' )` permet de créer un tableau dont les éléments sont des entiers naturels codés sur 8 bits (`uint` signifie «unsigned integer»).

Attention aux dépassements de capacité dans ce cas :

```
>>> A = np.array([254,255,3], dtype='uint8')
>>> A+1 # on ajoute 1 à tous les coeff
array([255, 0, 4], dtype=uint8)
```

Le plus grand entier naturel représenté sur 8 bits est bien $2^8 - 1 = 255$.

```
>>> B= np.array([126,127,1], dtype='int8')
>>> B+1
array([ 127, -128, 2], dtype=int8)
```

Le plus grand entier relatif représenté sur 8 bits est bien $2^7 - 1 = 127$.

Deuxième différence fondamentale avec les listes : les tableaux `numpy` ont une longueur fixée. En particulier, on ne peut pas ajouter d'éléments.

```
>>> t.append(5)
[...]
AttributeError:  numpy.ndarray  object has no attribute  append
```

Quelques fonctions utiles

1. Les fonctions `np.arange` et `np.linspace` :
   ```
   >>> np.arange(1,11) # les entiers de 1 à 10
   array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
   >>> np.linspace(0,1,5) # 5 valeurs uniformément distribuées entre 0 et 1
   array([ 0. , 0.25, 0.5 , 0.75, 1. ])
   ```
2. La fonction `np.zeros` pour créer des tableaux de zéros.
   ```
   >>> np.zeros((3,2))
   array([[ 0., 0.],       [ 0., 0.],       [ 0., 0.]])
   ```
3. Les tableaux `numpy` sont adaptés à la «vectorisation des calculs» (comportement par défaut dans les logiciels de calcul numériques comme Scilab). Les opérations arithmétiques se font coefficients par coefficients.
   ```
   >>> t
   array([5, 3, 4, 1])
   >>> t+1
   array([6, 4, 5, 2])
   ```
   ```
   >>> t + t # ce n est pas de la concaténation
   array([10, 6, 8, 2])
   >>> t * t # ce n est pas le produit matriciel
   array([25, 9, 16, 1])
   ```
   On parle aussi de «mapage» :
   ```
   >>> np.sin(t) # applique sinus à tous les coefficients de t
   array([-0.95892427, 0.14112001, -0.7568025 , 0.84147098])
   >>> np.sum(t) # somme des éléments de t
   12
   >>> f = lambda x: x**3
   >>> f(t) # On peut appliquer une fonction à tous les éléments du tableau
   ```
   Remarque : la vectorisation des calculs évite l'utilisation de boucles et permet un gain de rapidité considérable.

Exercice 2 : (**Résolution numérique de systèmes linéaires**)

La bibliothèque `numpy` propose des fonctions de résolution numérique de systèmes linéaires. Comme il ne s'agit pas de calcul formel, les solutions sont données sous la forme de valeurs approchées.
Pour résoudre un système linéaire de la forme $AX = B$, on procède ainsi :

1. On crée les matrices A et B du système avec la commande `array()`
2. On résout le système avec la commande `solve()`

L'algorithme de résolution utilisé par cette fonction `solve()` est basé sur la décomposition de la matrice A sous la forme LU, produit d'une matrice triangulaire inférieure L (comme lower, inférieure) et une matrice triangulaire supérieure U (comme upper, supérieure).

Pour résoudre le systéme $\begin{cases} x + 2y + 3z = 0 \\ -x + 3y + 2z = 1 \\ y + 5z = -1 \end{cases}$, il suffit de taper les instructions suivantes :

```
>>> from numpy import array
>>> from numpy.linalg import solve
>>> A = array([[1,2,3],[-1,3,2],[0,1,5]]) # Création des matrices du système
>>> B = array([0,1,-1])
>>> solve(A,B) # Résolution du système
```

Exercice 3 : (**Quelques fonctions utiles**)

Exécutez les instructions suivantes :

```
>>> from numpy.linalg import det, eigvals, norm, solve
>>> from numpy import matrix
>>> A = matrix([[10,2,3],[4,5,6],[7,8,9]])
>>> B = matrix([[1],[2],[3]])
```

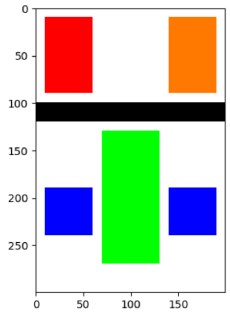
```
>>> det(A)
>>> norm(A)
>>> eigvals(A)
>>> solve(A,B)
```

Exercice 4 : (**Manipulation des images**)

Une image en niveau de gris est un tableau bidimensionnel de nombres compris entre 0 et 255. Une image couleur est un tableau tri-dimensionnel de nombres compris entre 0 et 255. Pour chaque pixel en position (y, x) (le pixel $(0, 0)$ est en haut à gauche), on associe 3 nombres, l'un pour l'intensité du rouge, l'autre pour l'intensité du vert et le troisième représente l'intensité du bleu (code RGB).
Une image blanche pourra alors être construite par :

```
>>> dessin=255*np.ones((300,200,3),dtype=np.uint8)
```

1. Créer et afficher l'image numérique ci-contre comportant 300x200 pixels en respectant les positions relatives des rectangles ainsi que leurs couleurs. Le pixel orange peut être obtenu par le triplet (255,120,0) dans le code RGB. Pour afficher le dessin, il faut écrire :
   `>>> import matplotlib.pyplot as plt`
   `>>> plt.imshow(dessin)` puis `>>> plt.show()`



2. Construire plusieurs autres images déduites de la précédente par division de l'intensité de chaque pixel par 2, 4, 8 puis 16. Observez le résultat en affichant chacune de ces images.
3. Afficher le négatif (couleurs complémentaires) de l'image construite en (1).
4. La commission internationale de l'éclairage propose d'obtenir la luminance (valeur du gris) d'un pixel à partir de ses composantes RGB par la formule :
   $$\text{Gris naturel} = 0,2125.\text{Rouge} + 0,7154.\text{Vert} + 0,0721.\text{Bleu}$$
   Écrivez une fonction `gris` qui prend un tableau représentant une image couleur RGB comme argument et renvoie un tableau représentant cette image en niveaux de gris. Appliquez cette fonction à l'image construite en (1) pour obtenir et afficher l'image en niveau de gris (appelée image B).
5. Chargez une image jpeg (`img=plt.imread("image.jpg")`) puis transformez-la en niveaux de gris.

6. Détection de contours pour l'image en niveaux de gris. Les contours d'une image correspondent aux endroits où l'intensité change brusquement, autrement, où le gradient de l'intensité est important.
   (a) Composante horizontale du gradient : Écrivez une fonction `gradientH` qui prend un tableau représentant une image en niveaux de gris comme argument et renvoie un tableau du gradient horizontal des pixels de l'image (comparaison entre pixels de 2 colonnes voisines).
   (b) Composante verticale du gradient : de même, écrivez une fonction `gradientV` qui renvoie un tableau du gradient vertical des pixels de l'image (entre 2 lignes voisines)
   (c) Appliquez ces deux fonctions à l'image B (dessin construit transformé en niveaux de gris). Affichez l'image correspondant au gradient horizontal de l'image B, puis celle correspondant au gradient vertical de B. Tous les contours ont-ils été détectés par l'un ou l'autre gradient ? Concluez.
   (d) Combinez le gradient horizontal et le gradient vertical pour obtenir une approximation de la norme du gradient global et construire le tableau rempli des valeurs des gradients normalisés (et donc des contours). Attention : la norme peut dépasser 255 ! Il faut donc normaliser la norme du gradient. Affichez l'image correspondante. Concluez.
   (e) On va maintenant utiliser un seuil pour les pixels afin d'augmenter le contraste du tableau gradient ainsi obtenu : si le gradient dépasse un certain seuil, le pixel est noirci dans une copie de l'image, sinon il est laissé en blanc.
   On fera attention au type des pixels (entier de type `np.uint8`, entier de type `int`, flottant, etc.). Choisissez une valeur seuil pour noircir les vrais contours et blanchir le reste.
   Remarque : pour choisir la valeur seuil du pixel, on peut s'aider du tracé de l'histogramme des pixels de l'image avec la syntaxe suivante :

```
>>> Nombre_pixels,valeur=np.histogram(image,bins=256)
>>> plt.plot(valeur[ :-1],Nombre_pixels)
>>> plt.show
```

7. Effectuez les mêmes opérations sur l'image en niveau de gris provenant de l'image jpeg.
8. La convolution est une nouvelle opération modifiant chacun des pixels d'une image en fonction de ses voisins. Pour cela, on défini un filtre (parfois appelé kernel) permettant de calculer la nouvelle valeur de chaque pixel en fonction de ses voisins. Cette opération peut être effectuée par :

```
>>> from scipy import ndimage
>>> k = np.array([[1,1,1],[1,1,0],[1,0,0]])
>>> B2= ndimage.convolve(B2, k, mode='constant', cval=0.0)
```

   Voici quelques filtres classiques :
   — Identité : $\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix}$ La détection de contour : $\begin{bmatrix}1&0&-1\\0&0&0\\-1&0&1\end{bmatrix}$ ou $\begin{bmatrix}0&1&0\\1&-4&1\\0&1&0\end{bmatrix}$ ou $\begin{bmatrix}-1&-1&-1\\-1&8&-1\\-1&-1&-1\end{bmatrix}$
   — Amélioration de la netteté : $\begin{bmatrix}0&-1&0\\-1&5&-1\\0&-1&0\end{bmatrix}$ Box blur : $\frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$ Flou de Gauss 3×3 : $\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$
   — Flou de Gauss 5×5 : $\frac{1}{256}\begin{bmatrix}1&4&6&4&1\\4&16&24&16&4\\6&24&36&24&6\\4&16&24&16&4\\1&4&6&4&1\end{bmatrix}$ Masque flou 5×5 : $\frac{-1}{256}\begin{bmatrix}1&4&6&4&1\\4&16&24&16&4\\6&24&-476&24&6\\4&16&24&16&4\\1&4&6&4&1\end{bmatrix}$
   Testez certains de ces filtres sur le dessin précédent et sur votre image réelle.