TP Informatique n° 4
Tableaux, matrices, images

1 Tableaux et matrices

On utilisera la bibliothèque `numpy`. Les tableaux bidimensionnels sont de type `array` (en réalité, ce sont des tableaux de tableaux unidimensionnels représentant chacune des lignes). On accède à l'élément de M situé à la ligne i et dans la colonne j par `M[i,j]`. ⚠ Si M possède n lignes et p colonnes, elles sont numérotées de 0 à $n-1$ (resp. $p-1$).

Listing 1 – Utilisation de matrices avec `numpy`

```
>>> import numpy as np
>>> M=np.array([[1,2,3],[4,5,6]])
>>> D = np.diag([1,2,3])
>>> size(M)
>>> size(M,0)
>>> size(M,1)
>>> np.transpose(M)
>>> np.dot(M,D)
>>> 2*M
>>> np.linalg.inv(D)
>>> np.linalg.matrix_rank(M)
```

■ **Exemple 1** Résoudre le système suivant avec Python :

$$\begin{cases} 2x & + & 2y & - & 3z & = & 2 \\ -2x & - & y & - & 3z & = & -5 \\ 6x & + & 4y & + & 4z & = & 16 \end{cases}$$

On pourra utiliser la fonction `linalg.solve` de la bibliothèque `numpy` et contrôler le résultat avec son propre calcul.

Bonus : Si l'on souhaitait concevoir un programme qui résout un système linéaire, quelles sont les fonctions que l'on devrait programmer ? ■

■ **Exercice 1** Pour tout $n \in \mathbb{N}^*$, on appelle matrice de Hilbert d'ordre n la matrice H_n de terme général $a_{i,j} = \frac{1}{i+j-1}$.

1. Ecrire une fonction `hilbert(n)` qui renvoie la matrice H_n.
2. On admet que la matrice H_n est inversible pour tout $n \in \mathbb{N}^*$. Etudier le rang de H_n avec Python.
3. Conclusion ?

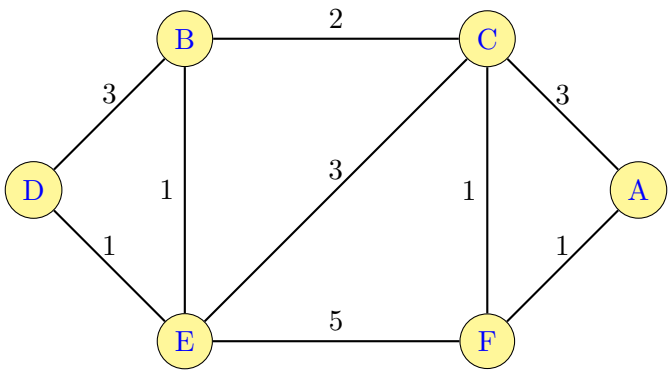
■

2 Algorithme de Dijkstra

Le but de cette partie est de parvenir à déterminer le plus court chemin entre deux villes (une ville de départ et une ville d'arrivée) en connaissant les distances entre les villes intermédiaires. Pour cela, on modélise le problème à l'aide d'un **graphe**.

Définition 1 On appelle graphe $G = (S, A)$ un couple où S est un ensemble, appelé ensemble des sommets, et A une partie de $S \times S$, appelée ensemble des arêtes. Dans la suite, on suppose G fini.

■ **Exemple 2** Considérons l'exemple suivant : l'ensemble des sommets est $\{A, B, C, D, E, F\}$ et (D, B) ou (B, C) sont des exemples d'arêtes. ■



Remarque 1 Le graphe présenté ici n'est pas orienté (si on peut aller de B à C, alors on peut aussi aller de C à B). Il est simple : il y a au plus une arête entre deux sommets. Et il est pondéré : à chaque arête on associe une valeur positive appelée poids. Elle va représenter la distance entre les deux sommets.

Définition 2 On appelle matrice d'adjacence d'un graphe la matrice $G = (g_{i,j})_{(i,j)\in S^2}$ telle que pour tout couple de sommets (i, j), $g_{i,j}$ est égal au poids de l'arrête (i, j) si elle existe, et à $+\infty$ sinon.

Pour le graphe de l'exemple, voici la matrice d'adjacence représentée sous la forme d'un tableau de type `array`.

Listing 2 – Matrice d'adjacence du graphe G

```
import numpy as np
G=np.array([[0,3,1,Inf,Inf,Inf],[3,0,1,2,Inf,Inf],[1,1,0,3,5,Inf],
        [Inf,2,3,0,1,3],[Inf,Inf,5,1,0,1],[Inf,Inf,Inf,3,1,0]])
```

■ **Exemple 3** Sur l'exemple précédent, on cherche à déterminer un plus court chemin du sommet de départ D au sommet d'arrivée A. Pour cela, on calcule les plus courts chemins partant de D, en une étape, et on obtient le tableau :

D	B	E	C	F	A
	3 (D)	1 (D)	$+\infty$	$+\infty$	$+\infty$

On fixe donc le sommet E dans notre parcours, et on met à jour le tableau, en mémorisant toujours le précédent sommet qui donne la distance minimale :

D	B	E	C	F	A
	3 (D)	1 (D)	$+\infty$	$+\infty$	$+\infty$
	2 (E)	**1 (D)**	4 (E)	6 (E)	$+\infty$

On sélectionne donc le sommet B, et on continue : à vous de jouer ! ■

■ **Exercice 2** Compléter et commenter la fonction suivante, et la tester à l'aide de l'exemple.

Listing 3 – Algorithme de Dijkstra

```
def dijkstra(G,depart,arrivee):

    N =np.size(G,0)

    parcours=list()

    for i in range(N):

        parcours.append([Inf,None,False])


    ville_select=...

    dist_interm=...

    while ...

        minimum=...

        for k in range(N):

            if parcours[k][2]==...

                dist=...

                dist_totale=...

                if dist != 0 and dist_totale < parcours[k][0]:

                    parcours[k][0]=...

                    parcours[k][1]=...

                if parcours[k][0]<minimum:

                    minimum=...

                    prochaine_ville_select=k

        ville_select=prochaine_ville_select

        parcours[ville_select][2]=...

        dist_interm=...

    chemin=list()

    ville=...

    chemin.append(ville)
```

```
51
52      while ville != ...
53
54          ville=...
55
56          chemin.append(ville)
57
58      return ...
```

■

3 Images

Le langage Python dispose de bibliothèques de fonctions adaptées au traitement des images. On se propose ici d'en faire un rapide aperçu, le but étant de réutiliser dans la mesure du possible des fonctions prédéfinies.

Une image est représentée informatiquement par un tableau T de pixels : le contenu du pixel en haut à gauche de l'image est contenu en $T[0,0]$ et celui en bas à droite est contenu en $T[n-1,p-1]$. Une image en niveaux de gris est un tableau d'entiers entre 0 et 255, et se manipule comme tel.

Listing 4 – Image en noir et blanc

```
1  import matplotlib.pyplot as plt
2  import matplotlib.image as mpimg
3  import numpy as np
4  import scipy.misc
5
6  # image en niveaux de gris
7  l = scipy.misc.lena() # Image de Léna
8  plt.gray() # Affiche l'image en niveaux de gris
9  plt.imshow(l)
10 plt.show()
```

Remarque 2 Cette image sert de référence pour tester tous les algorithmes d'imagerie numérique depuis les années 1970 !

■ **Exercice 3** Observer le résultat de la fonction suivante sur l'image de Léna.

Listing 5 – Image en noir et blanc

```
def floutage(image):
    (h, l) = image.shape
    res = image.copy()
    for i in range(1, h-2):
        for j in range(1, l-2):
            voisinage = image[i-1:i+2, j-1:j+2]
            res[i][j] = int(sum(voisinage)/9)
    return(res)
```

On peut appliquer de nombreux filtres à une image : flou gaussien, laplacien ... ■

Pour utiliser une image présente dans un fichier, on doit pouvoir se placer dans le répertoire adéquat.

Listing 6 – Manipulation de fichiers et de répertoires

```
import os # Permet d'accéder fonctions relatives au système d'exploitation

os.getcwd() # Récupère le répertoire courant
os.chdir("\user\home\") # Change le répertoire courant
os.listdir() # Liste le contenu du répertoire courant
```

On suppose que le fichier `chaton.png` est présent dans ce répertoire.

Une image en couleurs est un tableau dont la case $T[i,j]$ est un tableau contenant 3 nombres nécessaires pour coder la couleur du pixel. La première valeur correspond au rouge, la seconde au vert et la troisième au bleu. La couleur obtenue est donc la superposition de ces trois couleurs fondamentales.

Listing 7 – Chargement et utilisation d'un fichier png

```
img = mpimg.imread("chaton.png")
plt.colors()
plt.imshow(img)
plt.show()

img.shape
```

Les images se manipulent comme des tableaux usuels. On prendra garde à **copier** les tableaux avant de les modifier.

Listing 8 – Manipulation d'une image stockée dans un tableau

```
chatonbizarre=img.copy()
sauv=chatonbizarre[0:150,0:200,:].copy()
chatonbizarre[0:150,0:200,:]=chatonbizarre[150:,200:,:]
chatonbizarre[150:,200:,:]=sauv
plt.imshow(ichatonbizarre)
plt.savefig("chatonbizarre.png")
plt.show()
```

■ **Exercice 4** Ecrire une fonction qui affiche les histogrammes de rouge, vert et bleu présents dans une image (on pourra préalablement convertir les valeurs flotantes entre 0 et 1 en valeurs entières entre 0 et 255). ■

Remarque 3 Si l'on veut effectuer des manipulations plus complexes sur les images, on peut conseiller l'utilisation de la bibliothèque `Image` contenue dans Python Image Library (PIL) disponible ici : `http://www.pythonware.com/products/pil/` . Elle est installée par défaut avec la distribution Winpython `http://winpython.sourceforge.net/` .