TP 4
Simulation « générale » d'une chaîne de Markov

Dans tout ce qui suit on aura importé les packages usuels de Python par :

```
import numpy as np
import numpy.random as rd
import numpy.linalg as al
```

Le but de ce TP est de coder une fonction qui prendra en entrée la matrice de transition d'une chaîne de Markov, et permettra de simuler des trajectoires sur le graphe associé.
On utilisera ensuite cette fonction pour effectuer des statistiques et observer divers phénomènes.

1 Simulation d'une variable à valeurs dans $[\![1,n]\!]$ de loi quelconque

On considère ici un entier $n \in \mathbb{N}^*$, et une variable X à valeurs dans $[\![1,n]\!]$, telle que $\mathbb{P}(X=i)=p_i$. Les p_i vérifient donc : $\forall\, i \in [\![1,n]\!],\ p_i \geqslant 0$; et $\sum_{i=1}^{n} p_i = 1$. On cherche à simuler des tirages de X.
On procède de la manière suivante :

- On pose $q_0 = 0$, $q_k = \sum_{i=1}^{k} p_i$ jusqu'à $q_n = p_1 + \cdots + p_n = 1$.
- On découpe l'intervalle $[0,1]$ en :

$$]0,1] =]q_0,q_1] \cup]q_1,q_2] \cup \cdots \cup]q_{n-2},q_{n-1}] \cup]q_{n-1},1]$$

  On note $I_k =]q_{k-1},q_k]$ pour $i \in [\![1,n]\!]$.
- On effectue un tirage aléatoire $U \hookrightarrow \mathscr{U}(]0,1])$ (obtenu avec la commande `rd.random()`) et on regarde à quel intervalle I_k appartient la valeur tirée.

Soit k l'entier de $[\![1,n]\!]$ tel que U appartienne à I_k. On justifie dans l'exercice suivant (à faire chez soi) que la variable aléatoire X est telle que $X(\Omega) = [\![1,p]\!]$, et $\forall\, k \in [\![1,p]\!]$, $\mathbb{P}(X=k) = p_k$.

Exercice 1.

1. En se souvenant que pour tout $p \in [0,1]$, $\mathbb{P}(U \leqslant p) = p$, montrer que $\mathbb{P}(q_{k-1} < U \leqslant q_k)) = p_k$.
2. En déduire que $\forall\, k \in [\![1,p]\!]$, $\mathbb{P}(X=k) = p_k$.

On va maintenant programmer une fonction réalisant ce tirage. On observe que $X = k$ ssi k est le plus petit entier tel que $U \leqslant \sum_{i=1}^{k} p_i$.

Exercice 2.

1. Compléter la fonction Python suivante pour qu'elle prenne en argument une liste `[p1,...,pn]` de réels positifs, de somme 1 ; et renvoie un tirage de X décrite en début de partie.

```
def tirage(L):
    """L=[p1,p2,...,pn] est la loi de proba : P(X=i)=pi """
    U = rd.random()
    k = 1 # contiendra la valeur de X à renvoyer
    s = L[0] # contiendra la somme des pi de i=1 à k
    while .... :
        s = ....
        k = ....
    return k
```

2. Généraliser cette fonction pour générer des tirages d'une variable aléatoire quelconque d'univers-image fini, *ie* une variable de loi :

x	x_1	x_2	...	x_n
$P(X=x)$	p_1	p_2	...	p_n

où les x_i sont des réels quelconques, et les p_i sont positifs de somme 1. La fonction recevra comme arguments les listes $X = [x_1, \dots, x_n]$ et $P = [p_1, \dots p_n]$.
NB : cette généralisation ne servira pas dans la suite du TP.

2 Un exemple illustratif

Pour tester les codes des sections suivantes, on pourra utiliser le graphe probabiliste dont la matrice de transition est

$$M_0 = \begin{pmatrix} 1/3 & 1/6 & 1/2 \\ 1/2 & 1/2 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

On n'oubliera pas de manipuler un `np.array` :

```
M0=np.array([[1/3,1/6,1/2],[1/2,1/2,0],[1,0,0]])
```

Exercice 3.

1. Dessiner le graphe probabiliste associé à la matrice M_0.
2. À l'aide de l'outil de réduction matricielle `al.eig`, déterminer l'unique état stable de la chaîne de Markov associé à M_0.
   On doit trouver approximativement : $\begin{pmatrix} 0.5455 & 0.1818 & 0.2727 \end{pmatrix}$.
3. On prend pour état initial $V_0 = \begin{pmatrix} 1/3 & 1/3 & 1/3 \end{pmatrix}$. À l'aide de la commande `np.dot`, afficher les états $V_0, V_1, V_2, ..., V_{20}$. Quelle semble être la limite de V pour un grand nombre d'itérations ? Ce résultat semble-t-il être modifié si on prend une valeur initiale de V différente ?

3 Balade sur le graphe probabiliste

3.1 Quelques rappels sur la manipulation des matrices en Python

On définit en Python une matrice comme le `np.array` dont les composantes sont ses lignes. Ainsi la matrice $A = \begin{pmatrix} 1 & 1 \\ 2 & 4 \\ 3 & -1 \end{pmatrix}$ est encodée par

```
A=np.array([[1,1],[2,4],[3,-1]])
```

On remarque donc qu'on obtient facilement une ligne de la matrice par `A[i]` :

```
>>> A[1]
array([2, 4])
```

On peut modifier une matrice tout simplement en réaffectant des composantes ; ainsi :

```
>>> A[0,0]=888
>>> A
array([[ 888,   1],
       [ 2,   4],
       [ 3, -1]])
```

et on peut même faire ça sur une ligne entière en un coup !

```
>>> A[0]=np.array([100,200])
>>> A
array([[100, 200],
       [  2,   4],
       [  3,  -1]])
```

(ou sur une colonne mais la syntaxe est un peu moins naturelle)

```
>>> A[:,1]=np.array([10,20,30])
>>> A
array([[100,  10],
       [  2,  20],
       [  3,  30]])
```

3.2 Programmation

On considère un graphe probabiliste de matrice d'adjacence $M \in \mathcal{M}_n(\mathbb{R})$. Les sommets de ce graphe sont numérotés $1, 2, ..., n$. On rappelle que l'indexation en Python commence à 0 : ainsi, pour tous $(i, j) \in [\![0, n-1]\!]^2$, `M[i,j]` est donc la probabilité de passer de l'état `i+1` à l'état `j+1`.

Exercice 4.

1. Compléter la fonction `trajectoire(M,etat_initial,nb_etapes)` suivante qui prend en arguments :

   - la matrice de transition $M \in \mathcal{M}_n(\mathbb{R})$ de la chaîne ;
   - un état initial qui est un entier de $[\![1, n]\!]$;
   - le nombre d'étapes de temps `nb_etapes` à simuler

   et simule un parcours sur le graphe probabiliste, en partant de l'état initial spécifié.

   ```
   def trajectoire(M,etat_initial,nb_etapes):
       etat = ... # etat dans lequel se trouve le système
       traj = ... # liste des états successifs
       for k ... :
           etat = ... # calcul du nouvel état
           .... # qu'on ajoute à la trajectoire
       return np.array(traj) # manipuler des np.array pour pouvoir
                             # extraire des lignes/colonnes facilement
   ```

   On notera que si on est dans l'état i, les probabilités de transition vers les autres états de la chaîne se lisent sur la ligne $i-1$ de la matrice de transition.

2. À l'aide de la fonction `trajectoire`, programmer une fonction `trajectoires(M,etats,nb_etapes)` qui modélise les trajectoires de *plusieurs* individus se déplaçant indépendamment les uns des autres sur le graphe.
   Cette fonction prend en arguments :

   - la matrice de transition $M \in \mathcal{M}_n(\mathbb{R})$ de la chaîne ;
   - la liste des états initiaux des individus, rangée dans un `np.array` ;
   - le nombre d'étapes de temps `nb_etapes` à simuler

   Elle renverra un `np.array` avec autant de lignes que d'individus, et `nb_etapes+1` colonnes (la première colonne donne les états initiaux et les `nb_etapes` colonnes suivantes les états aux instants $1, 2, ...,$`nb_etapes`).

   Pour ce faire on créera une matrice de la taille voulue dès le départ (remplie de `0` par exemple) ; et on rangera dans chaque ligne de la matrice la trajectoire d'un individu.

   NB : une liste d'états initiaux doit être constituée d'entiers, sinon Python refusera de les considérer comme des numéros d'état. Si vous voulez passer un `np.ones()` *comme liste d'états initiaux, il faut que les « 1 » qui le composent soient reconnus comme des entiers et non comme des décimaux (affichage* `1.0`*). Pour ce faire, utiliser* `np.ones(k,dtype=int)`.

À ce stade nous sommes capables de simuler l'évolution de plusieurs individus ; donc de faire des statistiques d'évolution des individus au cours du temps.

4 Statistiques

On se place ici dans le cas de la chaîne de Markov à 3 états décrite précédemment ; on suppose que X_0 (position initiale) est la variable constante égale à 1.
Nous avons vu que l'état stable de cette chaîne de Markov est la matrice-ligne approximativement égale à :

$$\begin{pmatrix}0.5455 & 0.1818 & 0.2727\end{pmatrix}$$

Nous allons observer de manière empirique la convergence de la chaîne de Markov vers son état stable.

Si $n \in \mathbb{N}^*$ et k est un état de la chaîne, la loi faible des grands nombres montre qu'on obtient une bonne approximation de $\mathbb{P}(X_n = k)$ en faisant évoluer un grand nombre d'individus initialement dans l'état 1, pendant n étapes de temps, et en mesurant la fraction de ces individus dans l'état k à l'issue de ces n étapes.

Exercice 5 (Statistiques « spatiales » : tous les individus, à un temps fixé)**.**

1. Justifier que si `L` est un `np.array` et `a` un réel, la commande

   ```
   np.sum(L==a)
   ```

   renvoie le nombre d'éléments de `L` égaux à `a`.

2. Écrire une fonction `frequences` qui prend en argument un `np.array` de nombres, et renvoie sa fraction de composantes égales à 1 (resp à 2, à 3).
   Par exemple, `L=[1,1,2,3,1,2]` a la moitié de ses composantes égales à 1 ; un tiers de ses composantes égales à 2, et un sixième égales à 3 : `frequences(L)` doit renvoyer $[1/2, 1/3, 1/6]$.

3. À l'aide de la fonction `trajectoires`, construire une matrice `evol` encodant l'évolution sur 20 étapes de temps d'un million (10^6) d'individus initialement dans l'état 1.

4. Exécuter la commande suivante : la ligne suivante

   ```
   [frequences(evol[:,k]) for k in range(21)]
   ```

   et commenter le résultat obtenu.

5. Relancer la simulation en considérant cette fois que les 10^6 individus sont au départ dans un état choisi aléatoirement (avec un `randint`). Que dire du résultat final ?

On s'intéresse enfin à une autre observable : regarder, sur un temps d'évolution long, la fraction de temps passée par un individu dans chaque état.

Exercice 6 (Statistiques « temporelles » : un individu au cours du temps)**.**
Générer la trajectoire d'un individu pendant un grand nombre d'étapes de temps ; calculer la fraction du temps passé par l'individu dans l'état 1 (resp. l'état 2, l'état 3) à l'aide de la fonction `frequences`. Que remarque-t-on ? On parle de *propriété ergodique*.