# TP9 : Chaînes de Markov sur un ensemble d'états fini

## I. Simulation d'une chaîne de Markov

### I.1. Illustration sur un exemple

Doudou le hamster passe son temps entre ses trois activités favorites : dormir, manger et faire du sport dans sa roue. Au début de la jourée, il mange, et à chaque heure, il change d'activité selon les critères suivants.

*1)* Si, à l'heure $n$, il est en train de manger, alors il va dormir l'heure suivante avec probabilité 0.7 et faire de l'exercice avec probabilité 0.3.

*2)* Si, à l'heure $n$, il est en train de dormir, alors il continue à dormir l'heure $n+1$ avec probabilité 0.4, il va manger avec probabilité 0.3 et il va faire de l'exercice avec probabilité 0.3.

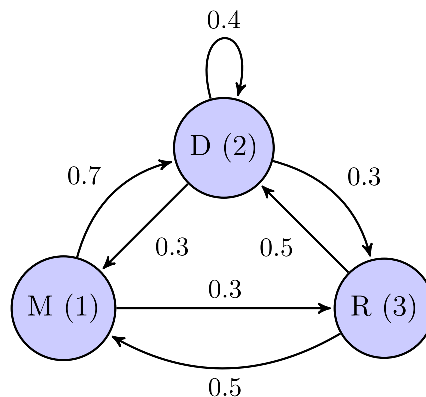
*3)* Si, à l'heure $n$, il est en train de faire de la roue, il va manger l'heure suivante avec probabilité 0.5 et il va dormir avec probabilité 0.5.

On s'intéresse ici à l'évolution du comportement de Doudou. On souhaite notamment déterminer si l'une de ses activités prendra, à terme, le dessus sur les autres.

### I.2. Modélisation mathématique

On modélise ce problème comme suit.

- On commence par numéroter les activités par un entier entre 1 et 3.
- On note $X_n$ la v.a.r. égale à l'état du hamster à l'heure $n$.
  Ainsi, la suite de v.a.r. $(X_n)$ représente l'évolution des activités du hamster.
- Cette évolution peut être modélisée par le graphe suivant.




- On définit enfin la `matrice de transition` $A$ associée au problème. Il s'agit de la matrice :

$$A = (a_{i,j}) \quad \text{où} \quad a_{i,j} = \mathbb{P}_{[X_n=i]}\big([X_{n+1}=j]\big)$$

*($a_{i,j}$ représente la probabilité de passage de l'état $i$ à l'état $j$)*

### I.3. Étude de la matrice de transition

Avant d'entamer l'étude en **Python** de ce problème, on importe les modules nécessaires.

```python
import numpy.random as nr
import numpy as np
import matplotlib.pyplot as plt
```

- Déterminer la matrice de transition du problème précédent.
  Écrire l'appel permettant de la stocker dans une variable `A`.

$$A = \begin{pmatrix} 0 & 0.7 & 0.3 \\ 0.3 & 0.4 & 0.3 \\ 0.5 & 0.5 & 0 \end{pmatrix}$$

En Python : `A = np.matrix([[0,0.7,0.3], [0.3,0.4,0.3], [0.5,0.5,0]])`

- Déterminer $\mathbb{P}_{[X_0=i]}\big([X_1 = j]\big)$. À quel coefficient de la matrice $A$ cela correspond-il ?

On a : $\mathbb{P}_{[X_0=j]}\big([X_1 = i]\big) = \mathbb{P}_{[X_n=i]}\big([X_{n+1} = j]\big) = a_{i,j}$.

Ceci permet de démontrer que la matrice de transition est indépendante de $n$.
Cette propriété est appelée **homogénéité**.

- Soit $(i_0, \dots, i_n, i_{n+1}) \in [\![1, 3]\!]^{n+2}$. Que vaut $\mathbb{P}_{[X_0=i_0] \cap \dots \cap [X_n=i_n]}\big([X_{n+1} = i_{n+1}]\big)$ ?

On a : $\mathbb{P}_{[X_0=i_0] \cap \dots \cap [X_n=i_n]}\big([X_{n+1} = i_{n+1}]\big) = \mathbb{P}_{[X_n=i_n]}\big([X_{n+1} = i_{n+1}]\big)$.

Cette propriété est appelée `propriété de Markov`.
On la retient souvent par la phrase :
*Le futur (la position $X_{n+1}$ à l'instant $n+1$) ne dépend du passé (positions $X_0$, ..., $X_n$) que par le présent (la position $X_n$ à l'instant $n$).*

Ces deux propriétés font de $(X_n)$ une chaîne de Markov homogène.

### I.4. Étude de l'évolution du comportement du hamster

Dans la suite, on considère la matrice ligne $U_n$ qui définit la loi de $X_n$ :

$$U_n = \begin{pmatrix} \mathbb{P}\big([X_n = 1]\big) & \mathbb{P}\big([X_n = 2]\big) & \mathbb{P}\big([X_n = 3]\big) \end{pmatrix}$$

- Déterminer la probabilité $\mathbb{P}\big([X_{n+1} = 1]\big)$ en fonction de $\mathbb{P}\big([X_n = 1]\big)$, $\mathbb{P}\big([X_n = 2]\big)$ et $\mathbb{P}\big([X_n = 3]\big)$ et des coefficients de la matrice $A$.

La famille $([X_n = 1], [X_n = 2], [X_n = 3])$ est un système complet d'événements. On en déduit, via la formule des probabilités totales, la probabilité $\mathbb{P}\big([X_{n+1} = 1]\big)$ :

$$\begin{aligned} \mathbb{P}\big([X_{n+1} = 1]\big) &= \mathbb{P}\big([X_n = 1]\big) \times \mathbb{P}_{[X_n=1]}\big([X_{n+1} = 1]\big) \\ &\quad + \mathbb{P}\big([X_n = 2]\big) \times \mathbb{P}_{[X_n=2]}\big([X_{n+1} = 1]\big) \\ &\quad + \mathbb{P}\big([X_n = 3]\big) \times \mathbb{P}_{[X_n=3]}\big([X_{n+1} = 1]\big) \\ &= \mathbb{P}\big([X_n = 1]\big) \times a_{1,1} + \mathbb{P}\big([X_n = 2]\big) \times a_{2,1} + \mathbb{P}\big([X_n = 3]\big) \times a_{3,1} \end{aligned}$$

► Déterminer de même $\mathbb{P}\big([X_{n+1} = 2]\big)$ et $\mathbb{P}\big([X_{n+1} = 3]\big)$.
En déduire que pour tout $n \in \mathbb{N}$, $U_{n+1} = U_n \times A$.
Exprimer enfin $U_n$ en fonction de $A$ et de $U_0$.

En procédant de la même manière, on obtient :

- $\mathbb{P}\big([X_{n+1} = 2]\big) = \mathbb{P}\big([X_n = 1]\big) \times a_{1,2} + \mathbb{P}\big([X_n = 2]\big) \times a_{2,2} + \mathbb{P}\big([X_n = 3]\big) \times a_{3,2}$
- $\mathbb{P}\big([X_{n+1} = 3]\big) = \mathbb{P}\big([X_n = 1]\big) \times a_{1,3} + \mathbb{P}\big([X_n = 2]\big) \times a_{2,3} + \mathbb{P}\big([X_n = 3]\big) \times a_{3,3}$

On en déduit que : $\forall n \in \mathbb{N},\ U_{n+1} = U_n \times A$.

Par une récurrence immédiate, on obtient : $\forall n \in \mathbb{N},\ U_n = U_0 \times A^n$.

► Que réalise l'opération `A*A` ? Et l'opération `A**2` ?
Calculer alors $\texttt{A}^5$, $\texttt{A}^{10}$ et $\texttt{A}^{20}$. Que remarque-t-on ?

Lorsqu'on évalue `A*A` ou `A**2`, on obtient $A^2$.

En évaluant `A**i` (pour $\texttt{i} \in \{5, 10, 20\}$), on remarque que la suite des puissances itérées $(A^n)$ semble converger vers une matrice proche de :

$$\begin{pmatrix} 0.27 & 0.5 & 0.23 \\ 0.27 & 0.5 & 0.23 \\ 0.27 & 0.5 & 0.23 \end{pmatrix}$$

► Écrire une fonction `simuMarkovEtape(x0,A)` qui prend en paramètre l'état initial $x_0$ et la matrice de transition $A$ et renvoie une simulation de la v.a.r. $X_1$.

```python
def simuMarkovEtape(x0, A):
    r = nr.random()
    i = 0
    # attention l'index des colonnes commence à 0
    c = A[x0-1,0]

    while c < r:
        i = i+1
        c = c + A[x0-1,i]

    return i+1
```

► Évaluer `simuMarkovEtape(2,A)` une dizaine de fois de suite.
Le résultat obtenu paraît-il cohérent ?

Partant de l'état 2, le comportement du hamster évolue vers :

× l'état 1 avec un probabilité de 0.3,

× l'état 2 avec un probabilité de 0.4,

× l'état 3 avec un probabilité de 0.3.

Les appels successifs reflètent cette distribution.

▶ Écrire une fonction `simuMarkov(X0,A,n)` qui prend en paramètre l'état initial $x_0$ et la matrice de transition $A$ et renvoie une simulation de la v.a.r. $X_n$.

> Il s'agit simplement d'itérer la fonction précédente.
>
> ```
> def simuMarkov(x0, A, n):
>     x = x0
>     for i in range(n):
>         x = simuMarkovEtape(x,A)
>     return x
> ```

## I.5. Comportement asymptotique du hamster

On souhaite conjecturer le comportement de la loi de $X_n$ lorsque $n \to +\infty$.
Pour ce faire, on compare :

× la valeur théorique de $U_n$ (loi de $X_n$) pour $n$ grand,

× une valeur approchée de $U_n$ obtenue en recueillant les effectifs de chaque état à la suite de $N$ simulations de trajectoires de taille $n$.

▶ Partant de l'état initial `x0`, par quel appel obtient-on $U_n$ ?

> Il s'agit de calculer $A^n$ et de sélectionner la ligne correspondant à l'état `x0` : `A**n[x0-1,:]`

▶ Compléter le programme suivant.

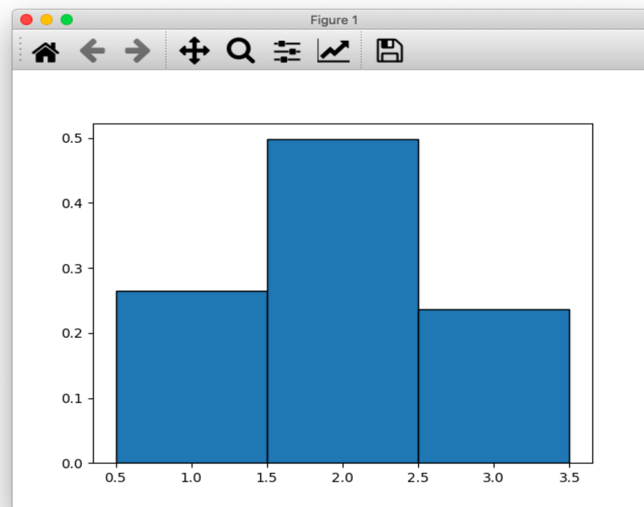
```
# Valeur des paramètres
N = 1000
n = 50
x0 = 2

# Distribution théorique
P = al.matrix_power(A,n)[x0 − 1,:]

# Valeurs observées
Obs = [simuMarkov(x0,A,n) for k in range(N)]

# Tracés des histogrammes
cl = [0.5, 1.5, 2.5, 3.5]
# Diagramme des fréquences observées
plt.hist(Obs, normed = True, bins = cl, edgecolor = ‘‘black‘‘,
        label = ‘‘Distribution approchée‘‘)
```

On obtient le diagramme suivant :




- Quel diagramme obtient-on si l'on part intialement avec un autre état `x0` ?

> On obtient les mêmes diagrammes en partant des états initiaux 1 et 3.

- Que peut-on en conclure sur le comportement à terme du hamster ?

> À terme, le hamster mange avec une probabilité d'environ 0.27, dort avec une probabilité d'environ 0.5 et tourne dans sa roue avec une probabilité d'environ 0.23.

- Si l'on est capable de démontrer que la suite $(U_n)$ converge vers une limite notée $U_\infty$, quelle relation peut-on établir entre $U_\infty$ et $A$ ?

> Si l'on est capable de montrer que $U_n$ converge (*i.e.* si tous ses coefficients convergent) et si on note $U_\infty$ sa limite, l'égalité $U_{n+1} = U_n \times A$ nous fournit, par passage à la limite :
>
> $$\boxed{U_\infty = U_\infty \times A}$$
>
> Une telle loi $U_\infty$ est dite **stationnaire** (ou **invariante**).
>
> On peut démontrer les propriétés suivantes.
>
> - Une loi invariante est un vecteur propre de la matrice de transition $A$.
> - Une chaîne de Markov homogène sur un espace d'étas fini $S$ admet au moins une loi invariante.
> - Avec une hypothèse supplémentaire (*la chaîne de Markov est irréductible*) on peut démontrer qu'une telle chaîne de Markov admet une unique loi invariante.

# II. Exercices sur les matrices

## II.1. Démontrer qu'une matrice est une matrice de transition

La somme des coefficients de chaque ligne de la matrice $A$ vaut 1 : $\forall i \in [\![1,3]\!],\ \sum\limits_{j=1}^{3} a_{i,j} \ = \ 1$.

- Démontrer cette égalité.

La famille $([X_1 = 1], [X_1 = 2], [X_1 = 3])$ forme un système complet d'événements.
On en déduit que, pour tout $i \in [\![1,3]\!]$ :

$$\begin{aligned} 1 &= \mathbb{P}_{[X_0=i]}(\Omega) = \mathbb{P}_{[X_0=i]}([X_1 = 1] \cup [X_1 = 2] \cup [X_1 = 3]) \\ &= \mathbb{P}_{[X_0=i]}([X_1 = 1]) + \mathbb{P}_{[X_0=i]}([X_1 = 2]) + \mathbb{P}_{[X_0=i]}([X_1 = 3]) \\ &= a_{1,j} + a_{2,j} + a_{3,j} = \sum_{j=1}^{3} a_{i,j} \end{aligned}$$

- Comment récupère-t-on en **Python** la **taille** d'une matrice `A` ?
  Comment récupère-t-on le nombre de lignes de `A` ? Et le nombre de colonnes ?

L'appel `(nbl, nbc) = np.shape(A)` permet de stocker :

× le nombre de lignes de `A` dans la variable `nbl`,

× le nombre de lignes de `A` dans la variable `nbc`.

On peut aussi récupérer seulement :

× le nombre de lignes : `nbl = np.size(A, 0)` (taille de « l'axe 0 »),

× le nombre de colonnes : `nbl = np.size(A, 1)` (taille de « l'axe 1 »).

- Écrire une fonction `somLigne` qui :

  × prend en paramètre une matrice `P` et un numéro de ligne `j` (arguments d'entrée),

  × renvoie la somme des coefficients de la ligne `j` de la matrice `P`,

```python
def somLigne(P,i):
    (nbl, nbc) = np.shape(P)
    S = 0
    for j in range(nbc):
        S = S + P[i,j]
    return S
```

- Tester la fonction `somLigne` sur la matrice de transition `A`.
- Écrire une fonction `verifUn` permettant de vérifier que la somme des coefficients de chaque ligne d'une matrice `P` vaut 1.

```python
def verifUn(P):
  (nbl, nbc) = np.shape(P)
  b = True
  i = 0
  while (i < nbl) & b:
    b = (somLigne(P,i)==1)
    i = i + 1
  return b
```

```python
def verifUn(P):
  (nbl, nbc) = np.shape(P)
  for i in range(nbl):
    if (somLigne(P,i) != 1):
      return(False)
  return True
```

- Vaut-il mieux utiliser une boucle while ou une boucle for pour cette fonction ?

> Il faut privilégier la boucle while ici. Ceci permet d'arrêter le calcul dès la première colonne dont la somme des coefficients ne vaut pas 1.

- La fonction `verifUn` est-elle suffisante pour démontrer que `A` est une matrice de transition ?

> Non. Il faut aussi démontrer que les coefficients de la matrice sont tous positifs.

- Écrire une fonction `tousPositifs` qui prend en paramètre une matrice `P`, et teste si tous les coefficients de la matrice `P` sont positifs.

On donne ici deux versions de la fonction. Les boucles `while` sont parfois un peu lourdes à écrire et l'utilisation d'un mécanisme à l'aide d'un `for` et d'une instruction `break` / `return` rend alors le code plus lisible.
(même si cette utilisation peut être contestée d'un point de vue pédagogique)

```
def tousPositifs(P):
    (nbl, nbc) = np.shape(P)
    b = True
    i = 0
    while (i < nbl) & b:
        j = 0
        while (j < nbc) & b:
            b = (P[i,j] >= 0)
            j = j + 1
        i = i + 1
    return b
```

```
def tousPositifs(P):
    (nbl, nbc) = np.shape(P)
    for i in range(nbL):
        for j in range(nbC):
            if P[i,j] < 0:
                return False
    return True
```

- Tester la fonction `tousPositifs` sur la matrice `A`.
- Quelle est la complexité de cette fonction ?

- Si `P` ne possède que des coefficients positifs, la fonction `tousPositifs` les teste tous. Elle s'exécute donc en $O(mn)$ (où $m$ est le nombre de ligne de `P` et $n$ son nombre de colonnes) sur une telle entrée.
- Ce cas est celui qui amène le nombre de calculs le plus élevé.
- On parle alors de **complexité dans le pire cas**.

L'intérêt de l'utilisation d'une boucle while est que le calcul s'arrête dès le premier coefficient strictement négatif rencontré. Par exemple, si le premier coefficient de la matrice vaut $-1$, un seul test est réalisé.

Il est souvent judicieux de se poser la question de la **complexité en moyenne** d'un algorithme. L'idée est de calculer la complexité comme moyenne des complexités sur chaque entrée possible (en général, on considère que chaque entrée est équiprobable).

- Écrire alors une fonction `estMTransition` qui prend en paramètre une matrice `P` et teste si `P` est une matrice de transition.

```
def estMTransition(P):
    return (tousPositifs(P) & verifUn(P))
```

- Tester la fonction `estMTransition` sur la matrice `A`.

## II.2. Démontrer qu'une matrice définit la loi d'un couple de v.a.r. discrètes

Le but de cette section est d'écrire une fonction permettant de vérifier qu'une matrice donnée en paramètre définit la loi d'un couple de v.a.r. discrètes.

Commençons par rappeler la caractérisation de la loi d'un couple de v.a.r. discrètes.

**Théorème 1.**

*Soient $I$ et $J$ deux parties de $\mathbb{N}$.*

*Notons $\{x_i \mid i \in I\}$ et $\{y_j \mid i \in J\}$ deux parties de $\mathbb{R}$ et $(p_{i,j})_{(i,j)\in I\times J}$ une famille de réels.*

*L'application $(x_i, y_j) \mapsto p_{i,j}$ est la loi d'un couple de v.a.r. discrètes* $\Leftrightarrow \begin{cases} \textit{1)}\ \forall (i,j) \in I \times J,\ p_{i,j} \geqslant 0 \\ \textit{2)}\ \sum\limits_{i\in I} \sum\limits_{j\in J} p_{i,j} = 1 \end{cases}$

▶ Écrire une fonction `sommeCoeff` qui prend en paramètre une matrice `P`, et somme l'ensemble de ses coefficients.

Nous donnons ici deux versions possibles.

- La première version est la plus classique.

```python
def sommeCoeff(P):
    (nbl, nbc) = np.shape(P)
    S = 0
    for i in range(nbl):
        for j in range(nbc):
            S = S + P[i,j]
    return S
```

- La seconde version utilise une fonctionnalité du langage : le fait que de nombreux objets (dont les `array`) sont des itérables. On appréciera la lisibilité du code produit.

```python
def estLoiCouple(P):
    S = 0
    for L in P:
        for coeff in L:
            S = S + coeff
    return S
```

On souhaite maintenant écrire une fonction `estLoiCouple` qui prend en paramètre une matrice `P`, et teste si cette matrice définit la loi d'un couple de v.a.r. discrètes.

On propose les deux versions suivantes :

```python
def estLoiCouple1(P):
    (sommeCoeff(P) == 1) & tousPositifs(P)
```

```python
def estLoiCouple2(P):
    tousPositifs(P) & (sommeCoeff(P) == 1)
```

► Comparer la complexité de ces deux fonctions.

Il s'agit ici de bien comprendre comment est évalué une expression logique de la forme `p & q`. Le processus est le suivant.

Tout d'abord, `p` est évalué. Deux cas se présentent

*1)* Soit `p` est évalué à `True`.
Dans ce cas `q` doit être évalué afin de connaître la valeur de vérité de `p & q`.

*2)* Soit `p` est évalué à `False`.
Dans ce cas `p & q` a pour valeur de vérité `False`.
L'expression `q` n'est pas évaluée.

Il est donc préférable de commencer par la fonction `tousPositifs` puisqu'elle peut s'arrêter avant d'avoir tester tous les coefficients de la matrice `P`.

Plus précisément, les fonctions `estLoiCouple1` et `estLoiCouple2` ont la même complexité dans le pire cas. Cependant :

× la fonction `estLoiCouple1` s'exécute en $O(mn)$ quelque soit son entrée,

× la fonction `estLoiCouple2` ne s'exécute en $O(mn)$ que dans le pire cas.

Ainsi, si le coefficient $P(1,1)$ est strictement négatif, `estLoiCouple2` s'arrête de suite alors que `estLoiCouple1` parcourt quand même toute la matrice pour sommer tous ses coefficients.