TP Informatique n° 6/7
Chaînes de Markov et exercices sur les matrices

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.

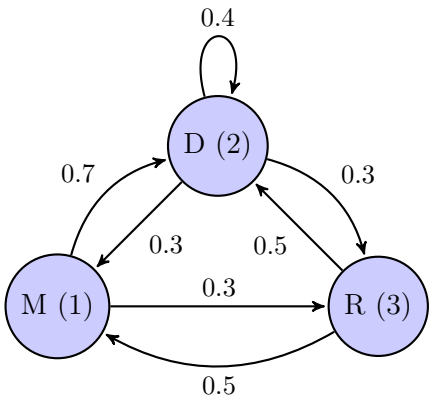
3) Si, à l'heure n, il est en train de faire de la roue, il va manger l'heure suivante avec probabilité 0.5 et il va dormir avec probabilité 0.5.

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 :

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

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

I.3. Étude de la matrice de transition

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

```
1 import numpy as np
```

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

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

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

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

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

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

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

On a : $\mathbb{P}_{(X_0=i_0,\ldots,X_n=i_n)}(X_{n+1} = i_{n+1}) = \mathbb{P}_{(X_n=i_n)}(X_{n+1} = i_{n+1})$.

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 le vecteur U_n qui définit la loi de X_n :

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

▶ Déterminer la probabilité $\mathbb{P}(X_{n+1} = 1)$ en fonction de $\mathbb{P}(X_n = 1)$, $\mathbb{P}(X_n = 2)$ et $\mathbb{P}(X_n = 3)$ et des cœfficients 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 $\mathbb{P}(X_{n+1} = 1)$:

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

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

En procédant de la même manière, on obtient :

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

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

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

▶ Que réalise l'opération `A * A` si `A` est de type `array` ?
Calculer $\texttt{A}^5$, $\texttt{A}^{10}$ et $\texttt{A}^{20}$ à l'aide de la fonction `np.linalg.matrix_power`. Que remarque-t-on ?

Lorsqu'on l'utilise entre deux objets de type `array`, l'opérateur `*` réalise une multiplication terme à terme.

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

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

▶ 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.

```
1  def simuMarkovEtape(x0, A):
2      r = random.random()
3      i = 0
4      # attention l'index des colonnes commence à 0
5      c = A[0,x0-1]
6
7      while c < r:
8          i = i+1
9          c = c + A[i,x0-1]
10
11     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.

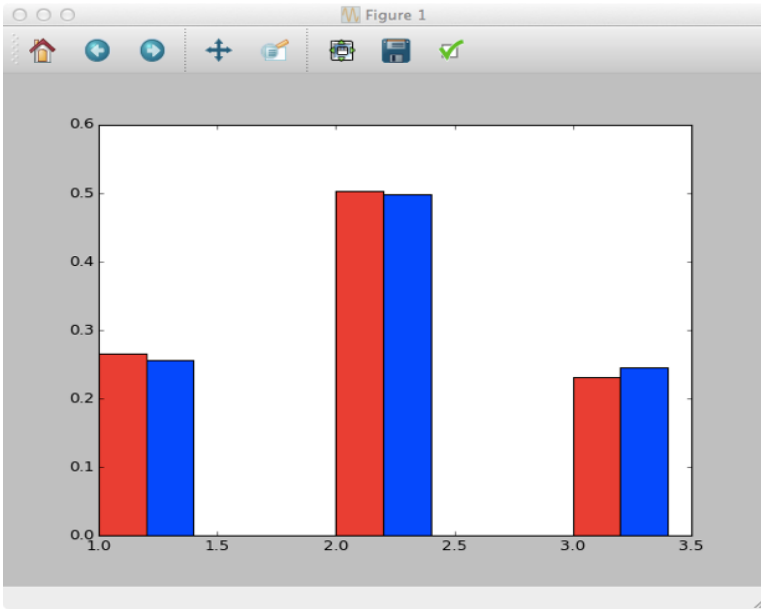
▶ Partant de l'état initial `x0`, par quel appel obtient-on U_n ?

Il s'agit de calculer A^n et de sélectionner la colonne correspondant à l'état `x0` : `np.linalg.matrix_power(A,n)[:,x0-1]`

▶ Compléter le programme suivant.

```
import matplotlib.pyplot as plt
# Valeur des paramètres
N = 1000
n = 50
x0 = 2

# Distribution théorique
P = np.linalg.matrix_power(A,n)[:,x0-1]

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

# Tableau des effectifs observés
cl = np.linspace(1,3,3)
effectif = calcEffectif(cl, Obs)

# Tracés des diagrammes
absc = np.linspace(1,3,3)
# Diagramme de la distribution théorique
plt.bar(absc, P, color = 'r', width = 0.2)
# Diagramme des fréquences observées
plt.bar(absc+0.2, effectif/N, color = 'b', width = 0.2)
plt.show()
```

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_∞, quelle relation peut-on établir entre U_∞ et A ?

> Si l'on est capable de montrer que U_n converge (*i.e.* si tous ses cœfficients convergent) et si on note U_∞ sa limite, l'égalité $U_{n+1} = A \times U_n$ nous fournit, par passage à la limite :
>
> $$U_\infty = A \times U_\infty$$
>
> Une telle loi U_∞ 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 cœfficients de chaque colonne de la matrice A vaut 1 : $\forall j \in [\![1,3]\!],\ \sum_{i=1}^{3} a_{i,j} = 1$.

▶ 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 $j \in [\![1,3]\!]$:

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

▶ 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 `somColonne` qui :
- × prend en paramètre une matrice `P` et un numéro de colonne `j` (arguments d'entrée),
- × renvoie la somme des cœfficients de la colonne `j` de la matrice `P`,

```
1  def somColonne(P,j):
2      (nbl, nbc) = np.shape(P)
3      S = 0
4      for i in range(nbl):
5          S = S + P[i,j]
6      return(S)
```

▶ Tester la fonction `somColonne` sur la matrice de transition `A`.

▶ Écrire une fonction `verifUn` permettant de vérifier que la somme des cœfficients de chaque colonne d'une matrice `P` vaut 1.

```
1  def verifUn(P):
2    (nbl, nbc) = np.shape(P)
3    b = True
4    j = 0
5    while (j < nbc) & b:
6      b = (somColonne(P,j)==1)
7      j = j + 1
8    return(b)
```

```
1  def verifUn(P):
2    (nbl, nbc) = np.shape(P)
3    for j in range(nbc):
4      if (somColonne(P,j) != 1):
5        return(False)
6    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 cœfficients 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 cœfficients de la matrice sont tous positifs.

- Écrire une fonction `tousPositifs` qui prend en paramètre une matrice `P`, et teste si tous les cœfficients 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)

```
1  def tousPositifs(P):
2      (nbl, nbc) = np.shape(P)
3      b = True
4      i = 0
5      while (i < nbl) & b:
6          j = 0
7          while (j < nbc) & b:
8              b = (P[i,j] >= 0)
9              j = j + 1
10         i = i + 1
11     return(b)
```

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

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

  - Si `P` ne possède que des cœfficients 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 cœfficient strictement négatif rencontré. Par exemple, si le premier cœfficient 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.

```
1  def estMTransition(P):
2      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.

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

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

Nous donnons ici deux versions possibles.

- La première version est la plus classique.

```
1  def sommeCoeff(P):
2      (nbl, nbc) = np.shape(P)
3      S = 0
4      for i in range(nbl):
5          for j in range(nbc):
6              S = S + P[i,j]
7      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.

```
1  def estLoiCouple(P):
2      S = 0
3      for L in P:
4          for coeff in L:
5              S = S + coeff
6      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 :

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

```
1  def estLoiCouple2(P):
2      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 cœfficients 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 cœfficient $P(1,1)$ est strictement négatif, `estLoiCouple2` s'arrête de suite alors que `estLoiCouple1` parcourt quand même toute la matrice pour sommer tous ses cœfficients.

II.3. Exercices de l'épreuve Modélisation Mathématique et Informatique (Agro-Véto 2015)

On souhaite dans ce problème écrire un programme permettant de calculer, sous certaines conditions, un vecteur propre d'une matrice.
On utilisera les notations suivantes :

- $\mathscr{M}_{m,n}(\mathbb{R})$: ensemble des matrices de taille $m \times n$ à cœfficients réels.
- $\mathscr{M}_p(\mathbb{R}) = \mathscr{M}_{p,p}(\mathbb{R})$: ensemble des matrices carrées.
- À chaque matrice $M = (m_{ij}) \in \mathscr{M}_{m,n}(\mathbb{R})$ on associe un nombre réel positif $\|M\|$, appelé **norme** de M, défini par : $\|M\| = \max_{i,j} |m_{ij}|$.

Remarquons que $\|M\|$ est toujours strictement positif, sauf lorsque la matrice M est nulle.

▶ Écrire une fonction `Norme(M)` qui étant donnée une matrice M de taille quelconque calcule et renvoie le nombre `Norme(M)`.
On interdit le recours à une quelconque fonction max *prédéfinie en Python*

```
1  def Norme(M):
2      (nbl, nbc) = np.shape(M)
3      m = 0
4      for i in range(nbl):
5          for j in range(nbc):
6              if abs(M[i,j]) > m:
7                  m = abs(M[i,j])
8      return(m)
```

Ou en utilisant le fait que les `array` sont des itérables.

```
def Norme(M):
    m = 0
    for L in M:
        for coeff in L:
            if abs(coeff) > m:
                m = abs(coeff)
    return(m)
```

- Écrire une fonction `Normalise(v)` qui étant donnée une matrice colonne $v \in \mathscr{M}_{p,1}(\mathbb{R})$ non nulle renvoie une nouvelle martice colonne $\tilde{v}$, de même forme, égale à : $\tilde{v} = \dfrac{v}{\|v\|}$.

```
def Normalise(v):
    return(v / norme(v))
```

On se donne à présent une matrice carrée $A \in \mathscr{M}_p(\mathbb{R})$. Soit v_0 un élément quelconque de $\mathscr{M}_{p,1}(\mathbb{R})$. En supposant qu'aucun des termes n'est dans le noyau de A, on peut former la suite $(u_n)_{n\geqslant 0}$ d'éléments de $\mathscr{M}_{p,1}(\mathbb{R})$ définie par la relation de récurrence :

$$\forall n \in \mathbb{N},\ v_{n+1} = \frac{Av_n}{\|v_n\|}$$

- Écrire une fonction `PuissanceIteree(A, n)` qui étant donnée une matrice carrée A et un entier naturel n, détermine la taille p de A, choisit aléatoirement une matrice colonne $v_0 \in \mathscr{M}_{p,1}(\mathbb{R})$, puis calcule et renvoie, en supposant que tous les termes de la suite ci-dessus sont bien définis, la matrice colonne v_n.

```
def PuissanceIteree(A,n):
    (nbl, nbc) = np.shape(A)
    v = np.random.rand(nbl,1)
    for i in range(n):
        v = Normalise(np.dot(A,v))
    return(v)
```

On se propose d'écrire maintenant une fonction `VecteurPropre(A, e)` qui étant donnée une matrice carrée A et un nombre $e > 0$ calcule les termes de la suite (v_n) jusqu'à ce que deux termes successifs vérifient $\|v_n - v_{n+1}\| < e$ et renvoie alors la matrice colonne v_{n+1}. Voici trois propositions de programme.

```
1  def VecteurPropre1(A, e)
2      d = A.shape
3      v = matrix(random.rand(d[0],1))
4      v = Normalise(v)
5      w = Normalise(A*v)
6      while Norme(v - w) >= e:
7          v = w
8          w = Normalise(A*v)
9      return w
```

```
1  def VecteurPropre2(A, e)
2      d = A.shape
3      v = matrix(random.rand(d[0],1))
4      v = Normalise(v)
5      w = Normalise(A*v)
6      ecart = Norme(v - w)
7      while ecart >= e:
8          v = w
9          w = Normalise(A*v)
10     return w
```

```
1  def VecteurPropre3(A, e)
2      d = A.shape
3      v = matrix(random.rand(d[0],1))
4      v = Normalise(v)
5      while Norme(v - Normalise(A*v)) >= e:
6          v = Normalise(A*v)
7      return Normalise(A*v)
```

Note sur ces différents programmes

- Il était précisé dans l'énoncé que la librairie `NumPy` était ouverte.
- Comme on peut le constater, les concepteurs ont fait ici le choix d'utiliser la classe `matrix`. L'intérêt est que l'opérateur `*` a été surchargé afin que `A * B` réalise la multiplication matricielle si `A` et `B` sont des objets de cette classe. Les lignes 3 de chacun des programmes précédents permettent de transformer un objet de type `array` en objet de la classe `matrix`.

► Parmi ces trois programmes, indiquer lequel est (ou lesquels sont) correct(s). Pour chaque programme *incorrect* on indiquera succintement ce qui ne va pas.

- Les programmes `VecteurPropre1` et `VecteurPropre3` sont corrects.
- Le programme `VecteurPropre2` est incorrect : la variable `ecart` n'est pas mise à jour dans la boucle. Elle est donc toujours évalué à sa valeur initiale. Si cette valeur est plus grande que `e`, ce programme contient donc une boucle infinie.