Projet

Aymen ECHCHALIM

Mars 2024

1 Quelques définitions

Soit $\mathbb{E}$ un espace au plus dénombrable appelé espace d'états. Les $i \in \mathbb{E}$ seront appelés états. $\mathbb{E}$ sera isomorphe à $\{1, ..., k\}$ où à $\mathbb{N}$.

1.1 Exemple

Dans le cas de la mouche qui se déplace sur un triangle équilatérale ABC, $\{A, B, C\}$ sera isomorphe à $\{1, 2, 3\}$.

1.2 Définition

Soit $(X_n)_{n\geq 0}$ une suite de variables aléatoires à valeurs dans $\mathbb{E}$. On dit que cette suite est une chaîne de Markov, si pour tout $n \geq 1$ et toute suite $(i_0, \ldots, i_n, i_{n+1})$ d'élements de $\mathbb{E}$, pour lesquelles:

$$\mathbb{P}(X_0 = i_0, \ldots, X_{n-1} = i_{n-1}, X_n = i)$$

On a la relation suivante entre probabilités conditionnelles:

$$\begin{aligned}\mathbb{P}(X_{n+1} = j | X_0 = i_0, \ldots, X_{n-1} = i_{n-1}, X_n = i) \\ = \mathbb{P}(X_{n+1} = j | X_n = i)\end{aligned}$$

On parlera **d'absence** de mémoire.

La Dynamique du processus est alors entièrement caractérisée par les $p_{i,j} = \mathbb{P}(\mathtt{X_{n+1}} = \mathtt{j} | \mathtt{X_n} = \mathtt{i})$, appelées probabilité de transition de l'état i à l'état j si la chaîne est homogène.

1.3 Définition

Toute matrice de transition $\mathcal{P} = (p_{i,j}, (i,j) \in \mathbb{E} \times \mathbb{E})$ vérifie les propriétés suivantes :

1. Pour tout couple (i, j), on a : $p_{i,j} \geq 0$
2. Pour tout $i \in \mathbb{E}$, on a :

$$\sum_{j \in \mathbb{E}} p_{i,j} = 1$$

2 Premier bloc du code

2.1 Inputs

```python
import numpy as np
import random
 #definir l'espace d'etats
 #donner le nombre d'etats
def Etats(n):
    E=[]
    for i in range(n):
        E.append(i)
    return E
def fill_matrix(n):
    P = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            element=float(input(f"inserer la valeur de l'element ({i+1},{j+1}):"))
            P[i,j]=element
    return P
```

cette partie du code, permet de définir un espace d'états $\mathbb{E}$, et d'initialiser une matrice de taille $n \times n$ telle que n correspond au nombre d'états de $\mathbb{E}$.

2.2 Fonctions de test

2.2.1 Démontrer qu'une matrice est une matrice de transition

Comme on a déja vu les propriétés d'une telle matrice dans 1.3, on procéde par l'implémentation:

```python
    def sum(P,i):
        (nbl,nbc)=np.shape(P) #nbl=nombre de lignes, nbc=nombre de colonnes
         s=0
         for j in range(nbc):
              s+=P[i,j]
         return s
    def test_Un(P):
    #Verifions si la somme sur chaque ligne vaut 1
        (nbl,nbc)=np.shape(P)
         a=True
         i=0
         while (i<nbl) & a:
             a=(sum(P,i)==1)
             i=i+1
        return (b)
```

- La fonction sum(P,i) reçoit comme argument la matrice $\mathcal{P}$ inserée par l'utilisateur, et la ligne concernée par la sommation
- Dans la fonction testUn(P), l'utilisation de la boucle while au lieu de la boucle for revient qu'avec la boucle while, on peut arrêter le calcul des la premiere ligne dont la somme ne vaut pas 1.

2.2.2 La positivité

Cette fonction sert à vérifier la condition de positivité sur les composantes de la matrice $\mathcal{P}$

```python
def Positivite(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)
```

- L'utilisation de la commande return a pour but de minimiser les calculs le plus possible, car dès qu'on affronte une valeur négative, on sort. le nombre maximal de calcul est $n \times n$.

3 Deuxieme bloc du code (plus intéressant)

Cette partie sera consacrée à conceptualiser un algorithme pour déterminer des patterns souvent cachés dans la matrice de transition $\mathcal{P}$, à savoir :

1. Classes de communication
2. Récurrence et transience
3. Période
4. Loi stationnaire

3.1 Classification des États; Decomposition en classes

Nous allons définir une classification des états et en déduire des propriétes des chaînes de Markov.
Les états d'une chaîne de Markov se répartissent en classes que l'on définit à partir de la matrice de transition.

3.1.1 Définition

Étant donnée une chaîne de Markov $(X_n)_{n\geq 0}$ de matrice de transition $\mathcal{P}$, on dira que j est atteignable depuis i si $\exists k \in \mathbb{N}, \mathbb{P}(X_{n+k} = j | X_n = i)$.
On notera $i \rightsquigarrow j$. On dire que les états i et j communiquent, noté $i \leftrightarrow j$.
En particulier, un état i est toujours accessible depuis lui même, puisque :

$$p_{i,i}^{(0)} = 1$$

3.1.2 Proposition

La relation $\leftrightarrow$ est une classe d'équivalence Considérons la relation entre états : $\mathcal{R}(i,j) \equiv i$ et j communiquent.
Cette relation est une relation d'équivalence, ceci signifie que l'on peut regrouper les élements de $\mathbb{E}$ en paquets, chaque paquet regroupant tous les élements qui communiquent entre eux.
Dans une classe, quels que soient deux élements pris dans cette classe, ils communiquent, par contre, deux élements pris dans des classes distinctes ne communiquent pas.
On dit que l'on a fait une **partition** de $\mathbb{E}$ en sous-ensembles disjoints.
Après un essai long pour classifier les états en passant par les relations d'équivalence/Relation de Chapmann-Kolmogorov, la complexité qui se trouve dans cette voie nous indique à changer la perspective !

3.1.3 Approche "Théorie des Graphes"

En Théorie des Graphes, un parcours de graphe est un algorithme consistant à explorer les sommets d'un graphe. Le mot parcours est également utilisé dans un sens différent, comme synonyme de chemin (un parcours fermé étant un circuit).

3.1.4 Algorithmes de parcours

Les algorithmes de parcours servent comme outil pour étudier une propriété globale du graphe, comme la connexité ou la forte connexité. En Théorie des Graphes, une composante fortement connexe d'un graphe orienté $\mathcal{G}$ est un sous-graphe de $\mathcal{G}$ possédant la propriété suivante, pour tout couple (u, v) de noeuds dans ce sous-graphe, il existe un chemin de u à v. On peut facilement voir que la notion de composante fortement connexe est isomorphe à celle de classe de communication.

Il existe plusieurs algorithmes qui peuvent effectuer cette tâche, mais nous, on va s'intéresser à l'algorithme de Tarjan! L'algorithme prend en entrée un graphe orienté et renvoie une partition des sommets du graphe correspondant à ses composantes fortement connexes. Le principe de l'algorithme est le suivant : on lance un parcours en profondeur depuis un sommet arbitraire. Les sommets explorés sont placés sur une pile P. Un marquage spécifique permet de distinguer certains sommets : les racines des composantes fortement connexes, c'est-à-dire les premiers sommets explorés de chaque composante (ces racines dépendent de l'ordre dans lequel on fait le parcours, elles ne sont pas fixées de façon absolue sur le graphe). Lorsqu'on termine l'exploration d'un sommet racine v, on retire de la pile tous les sommets jusqu'à v inclus. L'ensemble des sommets retirés forme une composante fortement connexe du graphe. S'il reste des sommets non atteints à la fin du parcours, on recommence à partir de l'un d'entre eux.

3.1.5 Implémentation

Voici l'implémentation de l'algorithme de Tarjan dans python.

```python
def dfs(noeud, numero_actuel, pile, numeros_noeuds, numeros_accessibles, dans_pile, P,
 ↪  composantes_fortement_connexes):
    # Input :
    # noeud : Noeud actuellement exploré.
    # numero_actuel : Le numéro actuel attribué aux nœuds pendant la traversée.
    # pile : Une pile pour suivre les nœuds dans le chemin de traversée actuel.
    # numeros_noeuds : Liste pour stocker les numéros uniques attribués à chaque nœud pendant la
     ↪  traversée.
    # numeros_accessibles : Liste pour stocker le numéro accessible minimum pour chaque nœud.
    # dans_pile : Liste pour suivre si un nœud est dans le chemin de traversée actuel.
    # P : La matrice de transition représentant le graphe.
    # composantes_fortement_connexes : Liste pour stocker des ensembles de nœuds formant des
     ↪  composantes fortement connexes.

    numeros_noeuds[noeud] = numero_actuel
    numeros_accessibles[noeud] = numero_actuel
    numero_actuel += 1
    pile.append(noeud)
    dans_pile[noeud] = True

    for voisin in range(len(P)):
        if P[noeud][voisin] > 0:
            if numeros_noeuds[voisin] is None:
                dfs(voisin, numero_actuel, pile, numeros_noeuds, numeros_accessibles, dans_pile, P,
                 ↪  composantes_fortement_connexes)
```

```
                numeros_accessibles[noeud] = min(numeros_accessibles[noeud],
                ↪  numeros_accessibles[voisin])
            elif dans_pile[voisin]:
                numeros_accessibles[noeud] = min(numeros_accessibles[noeud], numeros_noeuds[voisin])

    if numeros_accessibles[noeud] == numeros_noeuds[noeud]:
        composante = set()
        while True:
            voisin = pile.pop()
            dans_pile[voisin] = False
            composante.add(voisin)
            if voisin == noeud:
                break
        composantes_fortement_connexes.append(composante)


def tarjan_from_matrix(P):
    nb_noeuds = len(P)
    numero_actuel = 0
    pile = []
    composantes_fortement_connexes = []
    numeros_noeuds = [None] * nb_noeuds
    numeros_accessibles = [None] * nb_noeuds
    dans_pile = [False] * nb_noeuds

    for noeud in range(nb_noeuds):
        if numeros_noeuds[noeud] is None:
            dfs(noeud, numero_actuel, pile, numeros_noeuds, numeros_accessibles, dans_pile, P,
            ↪  composantes_fortement_connexes)

    return composantes_fortement_connexes

P = [
    [0.5, 0.5, 0, 0],
    [0.75, 0.25, 0, 0],
    [0.25, 0, 0, 0.75],
    [0, 0, 1, 0]
]

resultat = tarjanP(P)
print("Composantes Fortement Connexes :", resultat)
```

- Fonction dfs

La fonction dfs (Depth-First Search) effectue une traversée en profondeur à partir d'un nœud donné. Elle attribue des numéros de visite et de accessibilité aux nœuds, détecte les composantes fortement connexes, et met à jour une pile pour suivre le chemin de la traversée.

- Fonction tarjanP

La fonction principale tarjanP initialise les structures de données nécessaires et appelle la fonction dfs pour chaque nœud non visité du graphe.
Nouvelle condition de mise à jour de la low-link :
Si u et v sont des nœuds dans un graphe et que nous explorons actuellement u, alors notre nouvelle condition de mise à jour de la low-link est la suivante : Pour mettre à jour la valeur de low-link du nœud u à la valeur de low-link du nœud v, il doit y avoir un chemin d'arêtes de u à v, et le nœud v doit être sur la pile.

3.1.6 Aperçu de l'algorithme de Tarjan

- Marquer l'identifiant de chaque nœud comme non visité.
- Démarrer la recherche en profondeur (DFS). Lors de la visite d'un nœud, lui attribuer un identifiant et une valeur de low-link. Marquer également les nœuds actuels comme visités et les ajouter à une pile vue (seen stack).
- Lors de l'appel de DFS, si le nœud précédent est sur la pile, alors mettre à jour la valeur de low-link du nœud courant avec la valeur de low-link du dernier nœud visité.
- Après avoir visité tous les voisins, si le nœud courant a commencé une composante connexe, alors dépiler les nœuds de la pile jusqu'à ce que le nœud courant soit atteint.
- Un nœud a commencé une composante connexe si sa valeur d'identifiant est égale à sa valeur de low-link.

3.2 États Récurrents et Transients

On s'intéresse aux états où se trouve la chaîne de Markov après un temps long : en particulier, quels états ne seront plus visités après un certain temps, et quels états seront au contraire revisités perpétuellement ?

3.2.1 Définition

Pour tout état j, le temps d'attente τ_j de la chaîne $(X_n)_{n\geq 0}$ dans l'état à partir de l'instant 1 est le temps nécessaire pour que le système visite l'état j pour la première fois. C'est le temps de premier retour en j. Autrement dit,

$$\tau_j = \inf\{n \geq 1 : X_n = j\}$$

τ est une v.a qui prend ses valeurs dans l'espace des temps.

3.2.2 Théorème (critéres de récurrence)

Un état j est récurrent ou transient selon que :

$$\sum_{n=0}^{\infty} p_{jj}^{(n)} = +\infty \quad \text{la série diverge.}$$

$$\sum_{n=0}^{\infty} p_{jj}^{(n)} < +\infty \quad \text{la série converge.}$$

3.2.3 Exemple

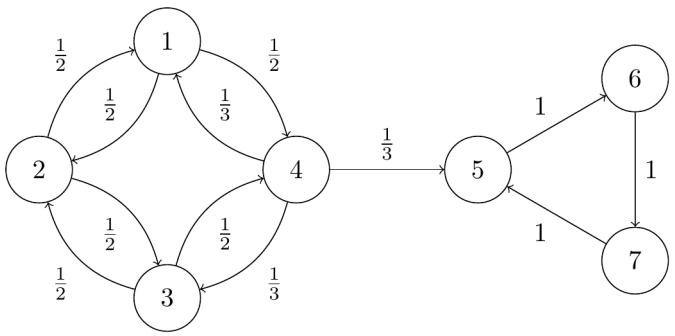

Figure 1: Graphe associé à une chaine de markov

Pour les états 5, 6 et 7, il est clair que la probabilité de retour est de 1, puisque la chaîne de Markov tourne autour du triangle, donc ces états sont récurrents. Les états 1, 2, 3 et 4 sont transitoires. Le passage de l'état 4 à l'état 5 est irréversible, donc $f_{j,j} \leq 1 - p_{4,5} = \frac{2}{3}$, et l'état 4 est transient. De même, si nous passons de 1 à 4 à 5, nous ne reviendrons certainement pas à 1,donc $f_{1,1} \leq 1 - p_{1,4}p_{4,5} = \frac{5}{6}$, et l'état 1 est transient. Par des arguments similaires, $f_{3,3} \leq 1 - p_{3,4}p_{4,5} = \frac{5}{6}$ et $f_{2,2} \leq 1 - p_{2,1}p_{1,4}p_{4,5} = \frac{11}{12}$

3.2.4 Remarque

Tous les états de la classe {1,2,3,4} sont transients, tandis que les états de la classe {5,6,7} sont récurrents.

3.2.5 Théorème

Soit C une classe de communication. Alors les états dans C sont soit tous récurrents, soit tous transients. La récurrence est donc une propriété de classe.

Preuve: Soient $(i,j) \in C$, on suppose que i est récurrent. Alors $\exists (n,m)$ t.q $p_{i,j}^{(n)} > 0$ et $p_{j,i}^{(m)} > 0$. Alors, d'après l'équation de Chapman-Kolmogorov:

$$\sum_{r=1}^{\infty} p_{jj}^{(n+m+r)} \geq \sum_{r=1}^{\infty} p_{jj}^{(n)} p_{i,i}^{(r)} p_{i,j}^{(m)} = p_{j,i}^{(m)} (\sum_{r=1}^{\infty} p_{i,i}^{(r)}) p_{i,j}^{(n)}$$

Si i est récurrent, alors $\sum_{r=1}^{\infty} p_{i,i}^{(r)} = +\infty$, par conséquent, $\sum_{r=1}^{\infty} p_{jj}^{(n+m+r)} = +\infty$, j est récurrent.

3.2.6 Théoreme

- Toute classe de communication non close est transient.
- Toute classe de communication fini et clôt est récurrente.

Preuve:
Soit C une classe non close, il existe donc $i \in C$ et $j \notin C$ tel que $p_{i,j} \geq 0$. Or i et j ne communiquent pas, puisque i n'est pas accessible à partir de j.
Donc pour tout n, $\mathbb{P}(X_n = i | X_1 = j, X_0 = i) = 0$, On en déduit que $\mathbb{P}(\tau_i < +\infty | X_1 = j) = 0$, finalement,

$$\mathbb{P}(\tau_j = +\infty | X_1 = j) \geq \mathbb{P}(\tau_j = +\infty | X_1 = j)\mathbb{P}(X_1 = j | X_0 = i) = p_{i,j} > 0$$

i est donc transient, et de même, C est transiente. On considère maintenant une classe C close finie. Procédons par l'absurde, on supposons que tous les états de C sont transients. Soit $i \in C$. Partant de i, la chaîne reste dans C. Donc $\mathbb{P}_i(\sum_{j \in C} N_j = +\infty) = 1$, cette somme étant finie, on en déduit que, $\mathbb{P}_i(\exists j \in C, N_j = +\infty) = 1$. Or, $\forall j \in C$, j est transient, alors $\mathbb{P}_i(N_j = +\infty) = 0$. Contradiction!
Ainsi, toute classe close d'une chaîne à espace d'états finis est récurrente.

3.2.7 Implémentation

On vise maintenant à déterminer la nature des classes en s'inspirant du théorème 3.2.6

```python
def Nature_class(matrice_transition):
    classes = tarjan_from_matrix(matrice_transition)
    for class_i in classes:
        is_recurrent = True
        for state in class_i:
            for j in range(len(matrice_transition)):
                if j not in class_i and matrice_transition[state][j] != 0:
                    print(f"class {class_i} is transient")
                    is_recurrent = False
                    break
            if not is_recurrent:
                break
        if is_recurrent:
            print(f"class {class_i} is recurrent")
Nature_class(matrice_transition)

```

3.2.8 Analyse de Complexité

1. Algorithme de Tarjan : La fonction tarjan, appelée dans 'NatureClass' a une complexité temporelle de $\mathcal{O}(n+m)$, où n est le nombre de noeuds et m est le nombre d'arêtes dans le graphe associé á la matrice de transition.
2. Itération sur les classes : Après l'éxecution de l'algorithme de Tarjan, 'classes' contient les CFC. Alors cette partie contribue $\mathcal{O}(m)$ à la complexite totale, avec m=le nombre des classes totales.
3. Boucles imbriquées : À l'intérieur de chaque classe, la fonction itère sur chaque état de cette classe, puis itère sur tous états du graphe pour vérifier les transitions à l'extérieur du classe. Supposons que k soit la taille moyenne des classes et n soit le nombre des noeuds(états) dans le graphe. Les deux boucles contribuent $\mathcal{O}(k \times n)$

La complexité totale vaut : $\mathcal{O}(n+m+m \times k \times n)$. (complexité polynomiale)
Nous pourrions réduire cette complexité en intégrant des algorithmes plus sophistiqués exploitant la théorie des graphes pour parcourir les nœuds avec une complexité temporelle linéaire. Cependant, nous ne sommes pas intéressés par cette tâche pour le moment.

3.3 Périodicité

Il s'agit d'étudier dans quelles conditions le temps qui sépare deux retours au même état j est ou n'est pas multiple d'un temps minimum. Pour ce faire, on introduit la notion de période.

3.3.1 Definition:

Soit $j \in E$. On appelle période de j, et on note $d(j)$, le PGCD de tous les entiers $n \geq 1$ pour lesquels $p_{i,j} > 0$.

$$d(j) = pgcd(n \geq 1, p_{j,j}^{(n)} > 0)$$

Si $d(j) \geq 2$, on dit que j est périodique de période $d(j)$; si $d(j) = 1$, on dit que j est apériodique. Si j est un état de non-retour, on pose $d(j) = \infty$.

En particulier, si $p_{i,i} > 0$ (le graphe possède alors une boucle),i est apériodique.

3.3.2 Théorème:

Si i est périodique de période d finie et si $i \leftrightarrow j (j \neq i)$, alors j est aussi périodique de période d. La périodicité est une propriété de classe.

Démonstration: Si $i \leftrightarrow j$, alors il existe deux entiers n et m tels que $p_{i,j}^{(n)} > 0$ et $p_{j,i}^{(m)} > 0$. Comme i est de période $d(i) = d$, il existe un entier $s \geq 1$ (s est un multiple de $d \geq 1$) tel que $p_{i,i}^{(s)} > 0$. On a donc: $p_{j,j}^{(m+s+n)} \geq p_{j,i}^{(m)} p_{i,i}^{(s)} p_{i,j}^{(n)} > 0$. Comme $p_{i,i}^{(s)} > 0$ entraîne $p_{i,i}^{(2s)} > 0$, on a aussi $p_{j,j}^{(m+2s+n)} > 0$. La période de j divise donc à la fois $m + s + n$ et $m + 2s + n$ donc aussi leur différence s. En particulier elle divise la période $d(i)$ de i. De la même façon on montre que $d(i)$ divise $d(j)$ et donc $d(i) = d(j)$.

3.3.3 Implémentation

Dans ce programme, on pourrait utiliser des algorithmes de détection de cycles, mais nous préférons exploiter la définition du cours.

```python
from math import gcd

def periode_etat(transition_matrix, j, borne_max):
    P = np.array(transition_matrix)

    values = []

    for n in range(1, borne_max + 1):
        P_power = np.linalg.matrix_power(P, n)
        p_jj_n = P_power[j][j]
        print(f"p_{j}_{j}^{({n})}: {p_jj_n}")  # Print p_jj_n for each value of n

        if 0 < p_jj_n <= 1:
            values.append(n)

    if values:
        period = gcd(*values)
        return period, P[j][j] ** period
    else:
        return None, None


etat = 0
borne_max_values = [10, 20, 50, 100, 200]
for borne_max in borne_max_values:
    print(f"Testing with borne_max = {borne_max}")
    periode, p_jj_n = periode_etat(matrice, etat, borne_max)
    if periode is not None:
        print(f"Période de l'état {etat}: {periode}")
        print(f"p_{etat,etat}^(n): {p_jj_n}")
    else:
        print(f"Période de l'état {etat}: Indéfinie")
    print("=" * 50)

```

3.3.4 Apreçu de la fonction PeriodeEtat

- À l'intérieur de la fonction, la matrice de transition P est convertie en un tableau NumPy.
- Pour chaque valeur entière n de 1 à 'borneMax', la fonction calcule la puissance $'n'$ de la matrice de transition.
- Si la probabilité $p_{j,j}^{(n)}$ est dans la plage $(0, 1]$, la valeur n est ajoutée à la liste 'values'.
- Après avoir itéré sur toutes les valeurs de n, la fonction calcule le plus grand commun diviseur (pgcd) des valeurs de la liste values. Ce pgcd représente la période de l'état j.
- Enfin, la fonction renvoie la période calculée ainsi que la valeur $p_{j,j}^{(m)}$, où m est la période calculée.

La raison pour laquelle on a introduit 'n' valeurs pour borneMax est pour comparer les résultats obtenus afin de garantir une consistance. Généralement, on aura pas de problème avec des matrices de taille comme celles traitées dans le cours.

3.4 Loi stationnaire

Dans ce paragraphe, on note les lois de probabilités μ sur l'ensemble des états $\mathbb{E}$ comme des vecteurs lignes, $\mu = (\mu_0, \mu_1, \ldots)$, oú $0 \leq \mu_i \leq 1$ et $\sum_{i \in \mathbb{E}} \mu_i = 1$.

3.4.1 Definition:

Soit ν une loi de probabilité sur l'ensemble des états. Elle est dite 'stationnaire', ou invariante si $\nu = \nu \cdot P$ ou, de façon équivalente, si pour tout $j \in \mathbb{E}$ on a :

$$\nu_j = \sum_{i \in \mathbb{E}} \nu_i \cdot p_{i,j}$$

On peut aussi caractériser ν^T comme un vecteur propre de P^T associé à la valeur propre 1.

Supposons qu'il existe une loi de probabilité stationnaire ν pour tout $n \geq 0$ on a aussi :

$$\nu = \nu \cdot P^{(n)}$$

Il en résulte que si on prend ν comme loi de probabilité initiale, ie $\mu_i^{(0)} = P(X_0 = i) = \nu_i$, alors la loi de probabilité de X_n qui est donnée par:

$$\mu^{(n)} = \mu^{(0)} \cdot P^{(n)}$$

est indépendante de n et est égale à $\mu^{(0)}$, soit $P(X_n = i) = P(X_0 = i) = \nu_i$. De façon générale, si l'on prend ν comme loi de probabilité initiale,la chaine de markov devient un processus stationnaire.

Il existe plusieurs méthodes pour calculer la loi stationnaire. On s'intéresse à la méthode de la puissance itérée.

3.4.2 Méthode de la puissance itérée:

La méthode des puissances est un algorithme pour calculer la valeur propre dominante d'une matrice. Bien que cet algorithme soit simple à mettre en oeuvre, il ne converge pas très vite.

Calcul de valeurs propres:

Étant donné une matrice A, on cherche une valeur propre de plus grand module et un vecteur propre associé, le calcul des valeurs propres n'est en général pas possible directement (par une expression analytique): On utilise alors des méthodes itératives, et la méthode des puissances est la plus simple d'entre elles. La méthode repose sur un théorème s'appuyant sur la réduction de Jordan.

3.4.3 Implémentation

```python
def power_iteration(matrice_transition, tol=1e-6, max_iter=1000):
    n = len(matrice_transition)
    pi = np.ones(n) / n
    for _ in range(max_iter):
        pi_new = np.dot(pi, matrice_transition)
        if np.linalg.norm(pi_new - pi, ord=1) < tol:
            return pi_new
        pi = pi_new

    return None

stationary_distribution = power_iteration(matrice_transition)
print("Stationary Distribution:", stationary_distribution)
```

3.4.4 Aperçu de l'algorithme

Pour détérminer la distribution stationnaire d'une Chaine de Markov, voici un bref apreçu de l'algorithme de l'itération de puissance :

1. Initialisation : Commencez avec une estimation initiale de vecteur de distribution stationnaire.
2. Itération :
   - Multipliez le vecteur de distribution actuel par P.
   - Normalisez le vecteur résultant pour garantir qu'il somme à 1.
   - Repétez ce processus jusqu'à la convergence, où le vecteur de distribution cesse de changer de manière significative entre les itérations.