3I005 Projet Chaînes de Markov

BEROUKHIM Keyvan MONGENOT Nicolas

1 Préambule technique

Le projet a été développé en python 3 et utilise les modules suivant :
-scipy.sparse.linalg
-pyAgrum
-numpy, matplotlib.pyplot, math, time, random, ...

2 Comportement asymptotique

Le but de cette partie est de tester différents algorithmes, qui, étant donné une chaîne de Markov, déterminent sa distribution stationnaire.

2.1 Méthodes itératives

2.1.1 Simulation

Le principe de la simulation est de choisir aléatoirement le prochain état selon l'état actuel. À chaque étape, l'algorithme s'arrête si la distribution de probabilité obtenue est semblable à la distribution précédente.

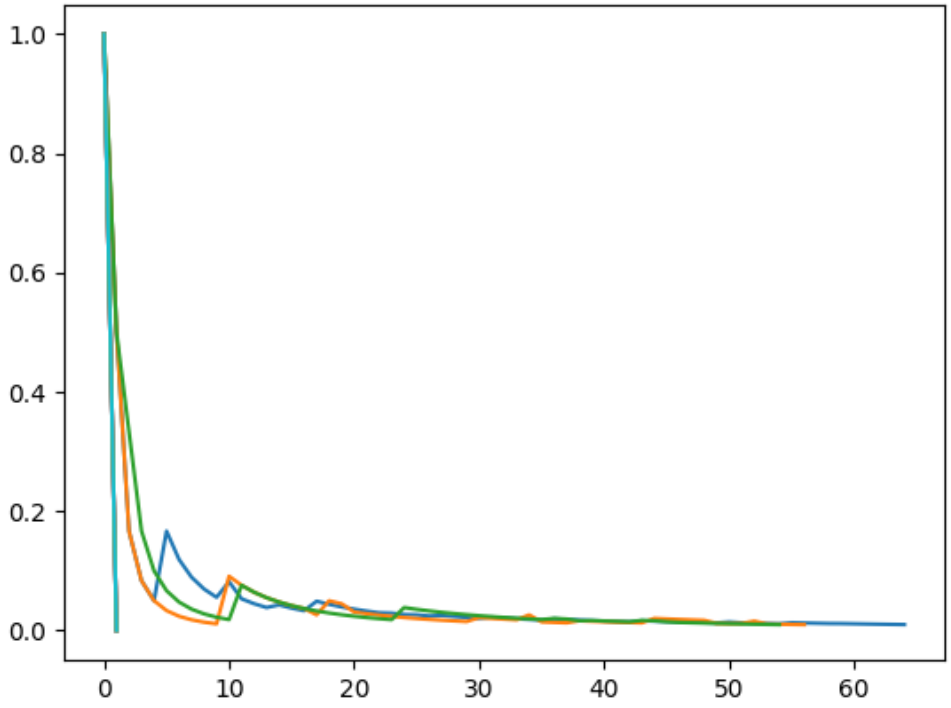

Figure 1: Évolution de l'erreur de 10 instances de Feu
(7 instances s'arrêtent dès la deuxième itération)

Le premier problème que l'on peut remarquer est que si les premier et deuxième nœuds parcourus sont les mêmes, l'algorithme va terminer dès la deuxième itération en retournant un résultat erroné (une probabilité de 1 pour ce nœud et 0 pour les autres). Afin de résoudre ce problème, on continue à **itérer tant qu'un seul nœud a été visité**.

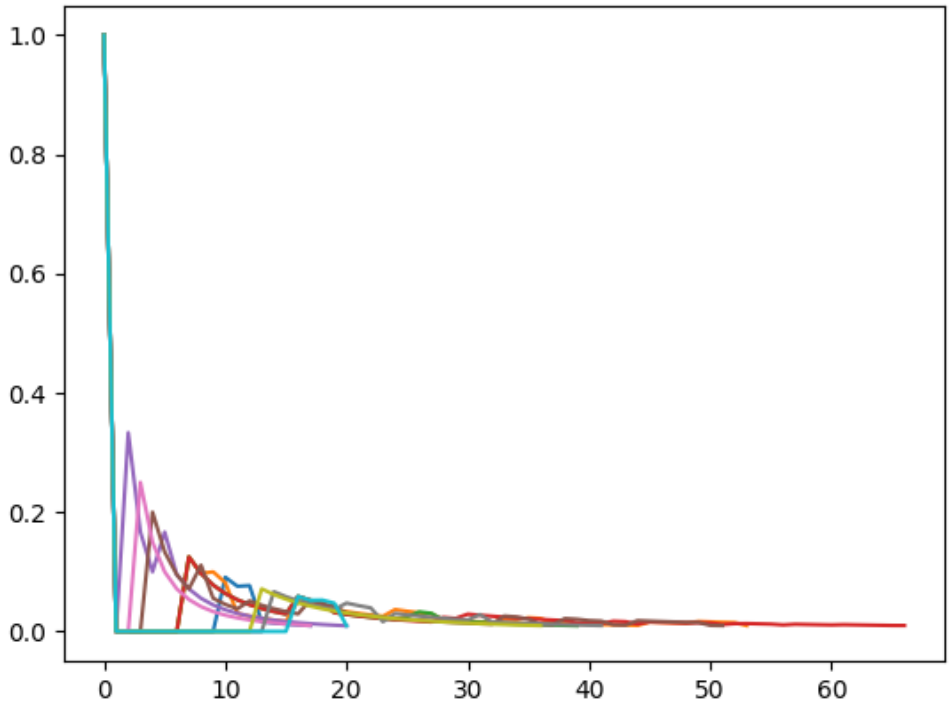

Figure 2: Évolution de l'erreur de 10 instances de Feu ($err = 10^{-2}$)

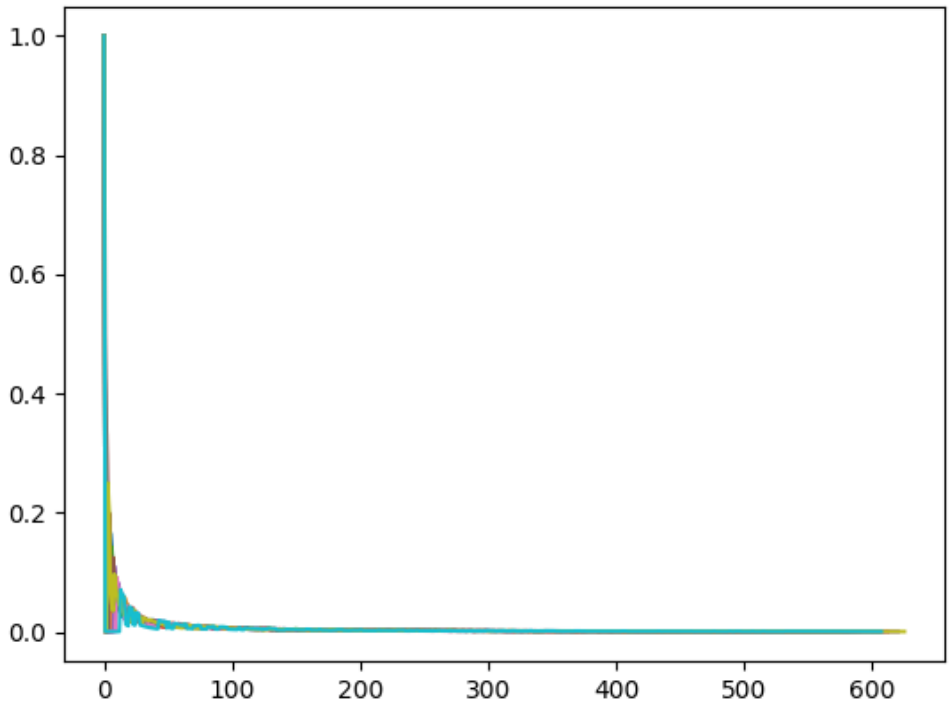

Figure 3: Évolution de l'erreur de 10 instances de Feu ($err = 10^{-3}$)

On remarque que **la courbe est une hyperbole**.

La distribution asymptotique n'étant théoriquement pas disponible, on se sert de l'erreur relative pour le critère d'arrêt. **Pour diviser l'erreur par un facteur k, le nombre d'itérations requises est multiplié par k**. On peut se demander ce qu'il en est pour l'erreur réelle.

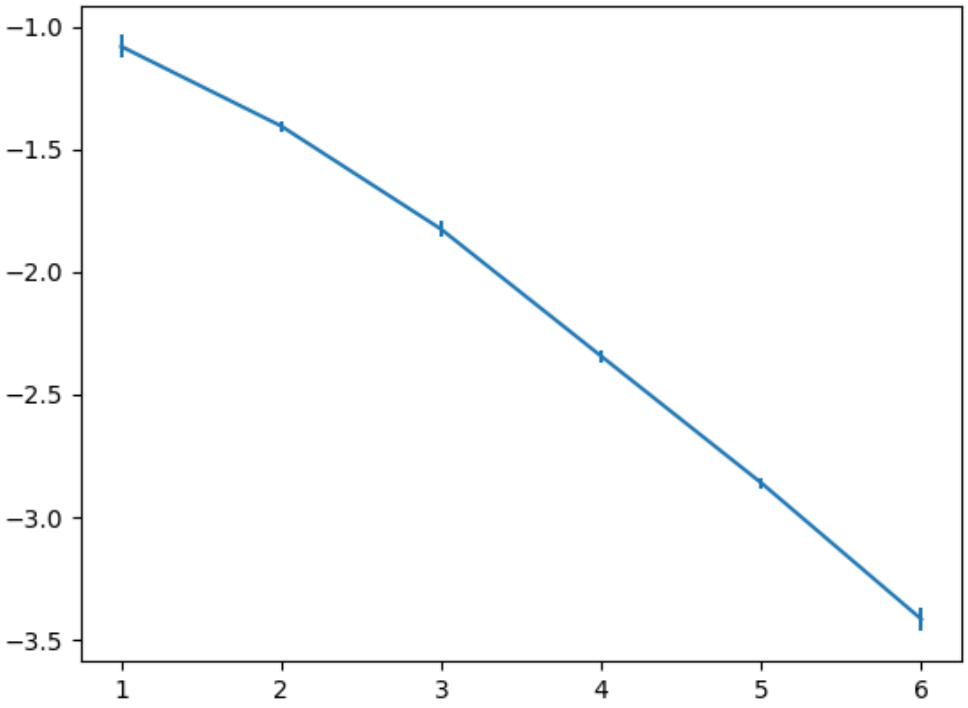

Figure 4: log_{10}(erreur réelle) en fonction de log_{10}(1/erreur relative)

Pour obtenir ces valeurs, on mesure à chaque itération la distance entre la distribution actuelle et la distribution asymptotique théorique : [3/8, 2/8, 3/8].

La droite a un coefficient directeur de -0,5 (avec MonoBestiole aussi). L'erreur réelle converge donc plus lentement que l'erreur relative: pour diviser l'erreur réelle par k il faut diviser l'erreur relative par k^2, c'est à dire multiplier le nombre d'itérations par k^2. L'**erreur réelle est en $1/\sqrt{n}$**. Cette technique semble atteindre sa limite vers **3 ou 4 décimales de précision**: abaisser le seuil d'erreur relative sous 10^{-6} ne change pas l'erreur réelle.

Si la chaîne de Markov n'est pas irréductible (comme "Mouse In Maze"), itérer sur une seule instance de la chaîne n'est pas suffisant. On réalise donc la moyenne des distributions obtenues sur différentes instances. Pour obtenir ce schéma, l'écart type des erreurs obtenues étant relativement grand, on a fait la moyenne sur 1000 instances. L'erreur réelle est obtenue en comparant les distributions avec [0, 0, 0, 0, 0.5, 0.5].

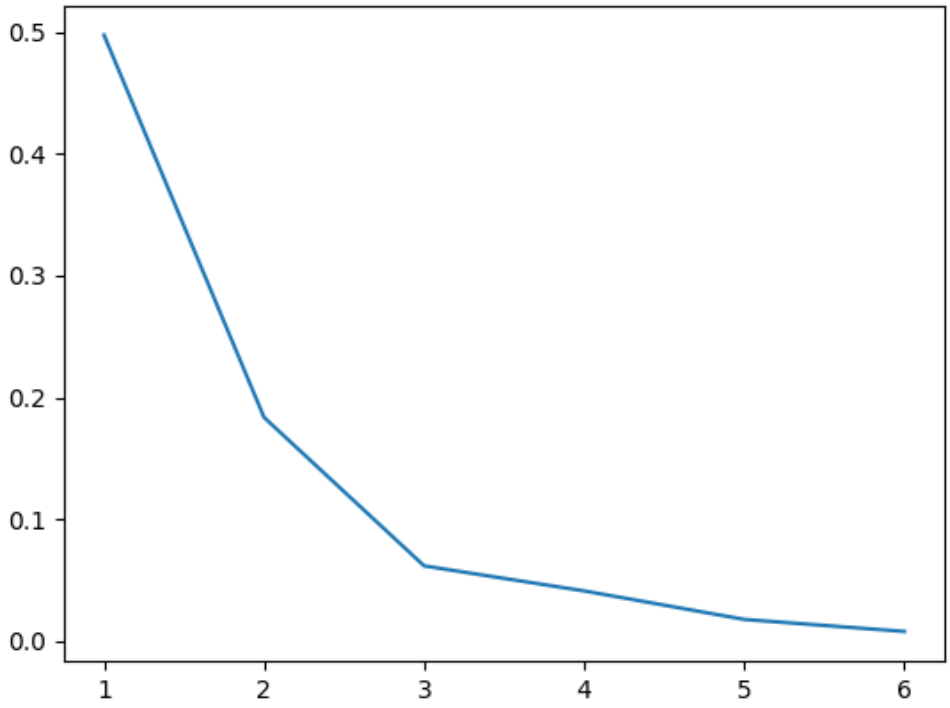

Figure 5: Erreur réelle en fonction de $log_{10}(1/\text{erreur relative})$

Au coût d'un **nombre d'instances plus grand**, on parvient à faire **converger la chaîne de Markov non irréductible** vers la distribution de probabilité asymptotique moyenne.

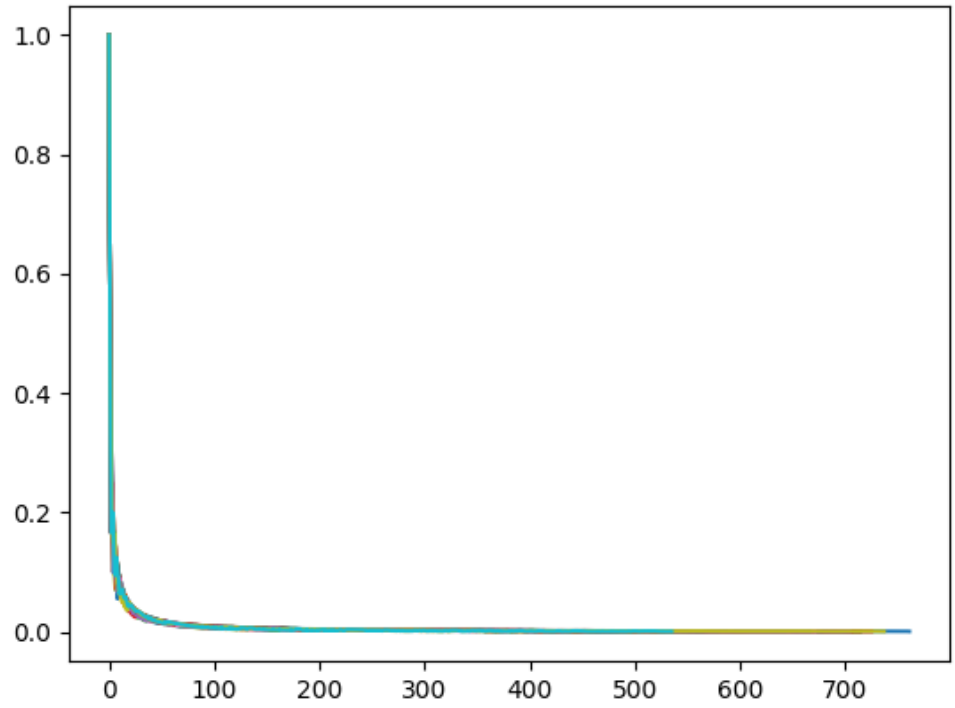

Figure 6: Évolution de l'erreur de 10 instances de Mono20 ($err = 10^{-3}$)

Le problème de la "Mono bestiole" est équivalent à celui du "Feu" mais il permet de faire varier le nombre d'états (on note "Mono10" le problème de la Mono bestiole avec 10 états).
Pour une erreur relative fixée (à 10^{-6} ici), **augmenter le nombre d'états** de la chaîne **n'augmente presque pas le nombre d'itérations** effectuées : 661 000 itérations pour Mono10 et 668 000 itérations pour Mono1000 ($\sigma = 1000$). Le temps d'exécution reste aussi constant : 2,7 secondes pour Mono10 et 2,8 secondes pour Mono1000 ($\sigma = 0,1$). En effet, l'implémentation (cf CollGetDistributionVerbose.py) est telle que chaque itération est en $O(1)$

En conclusion, pour une erreur réelle err, quelque soit le nombre d'états, le nombre d'itération et le temps d'exécution sont en $\boldsymbol{O(err^{-2})}$

2.1.2 Convergence de π_n

Le principe de cette méthode est de multiplier le vecteur de distribution actuel (π_i) par la matrice de transition tant que π_i n'a pas convergé.

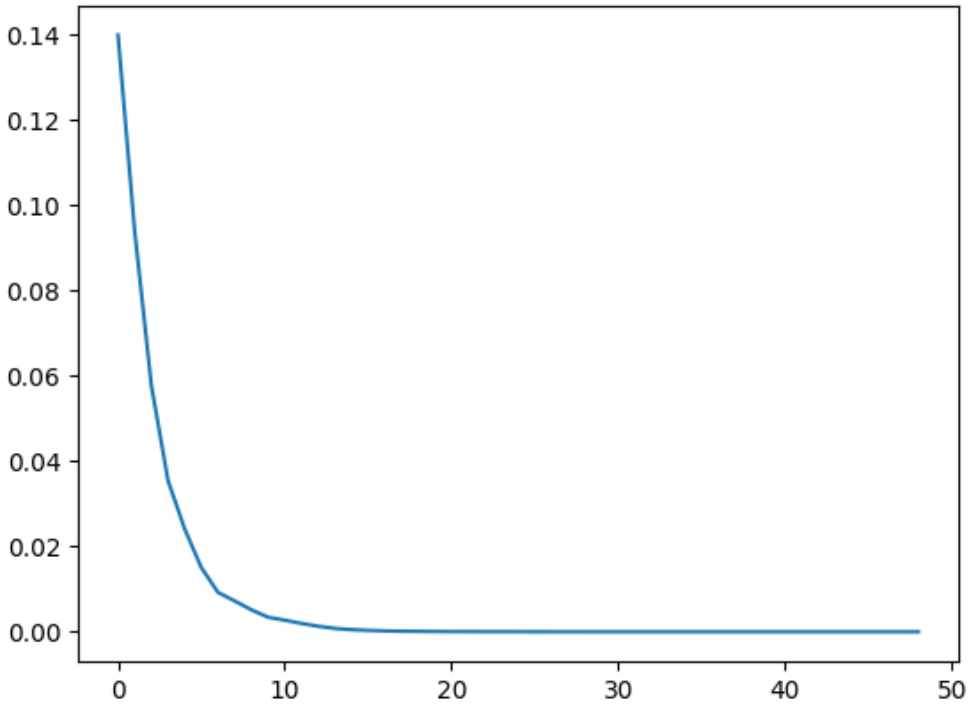

Figure 7: Évolution de l'erreur relative sur "Feu" en fonction de l'itération (jusqu'à 10^{-9} d'erreur)

En passant au logarithme, on obtient une droite de coefficient directeur négatif, l'erreur relative est donc exponentielle décroissante en le nombre d'itérations.

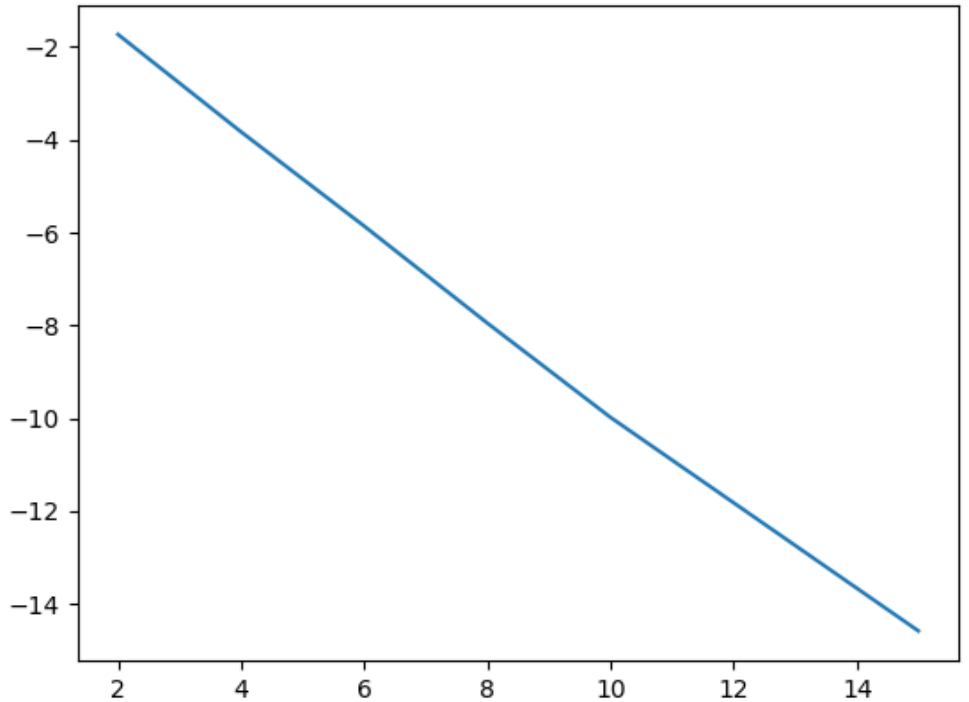

Figure 8: log_{10}(erreur réelle) en fonction de log_{10}(erreur rela)
(la précision n'augmente plus après 15 décimales)

En passant au logarithme, on obtient une droite de coefficient directeur -1. L'erreur réelle est donc proportionnelle à l'erreur relative. Au final, pour une chaîne fixée, le nombre d'itération est en $log(1/err)$

Par ailleurs, pour "Mouse In Maze", **une seule instance** de la chaîne suffit pour obtenir la distribution asymptotique moyenne.

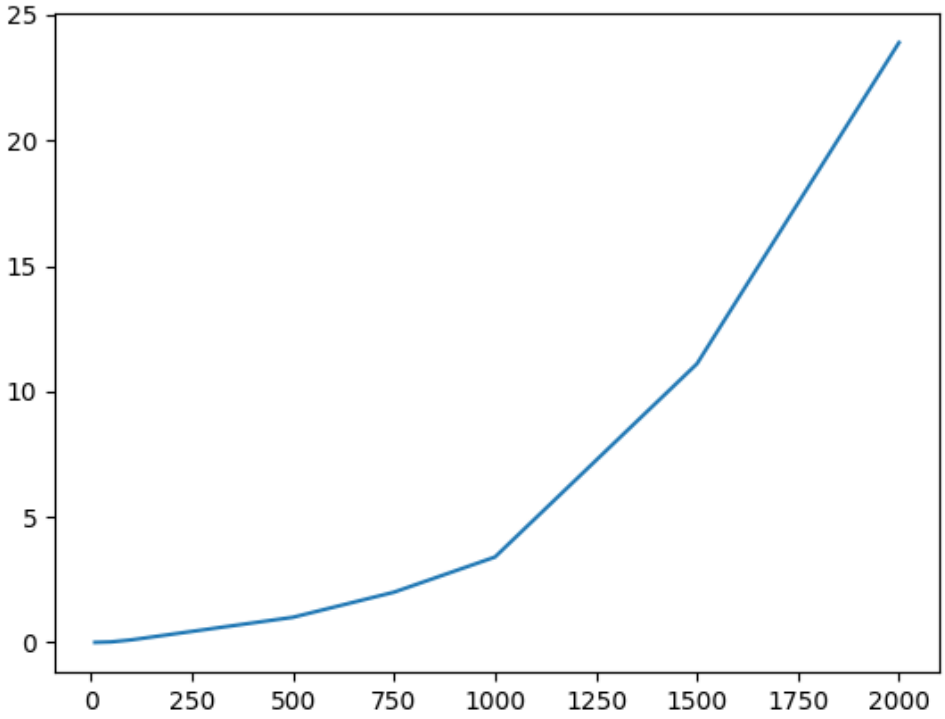

Figure 9: temps d'exécution en fonction du nombre d'états ($err = 10^{-9}$)

Le temps d'exécution d'une itération augmente quadratiquement avec le nombre d'états.
On remarque que le nombre d'itérations varie avec le nombre d'états. Ici, il semble être en $\sqrt[3]{m}$.
En conclusion, on a un temps d'exécution en

$$\boldsymbol{nb\_iter * m^2} \approx log(1/err) * m^{7/3}$$

2.1.3 Convergence de M^n

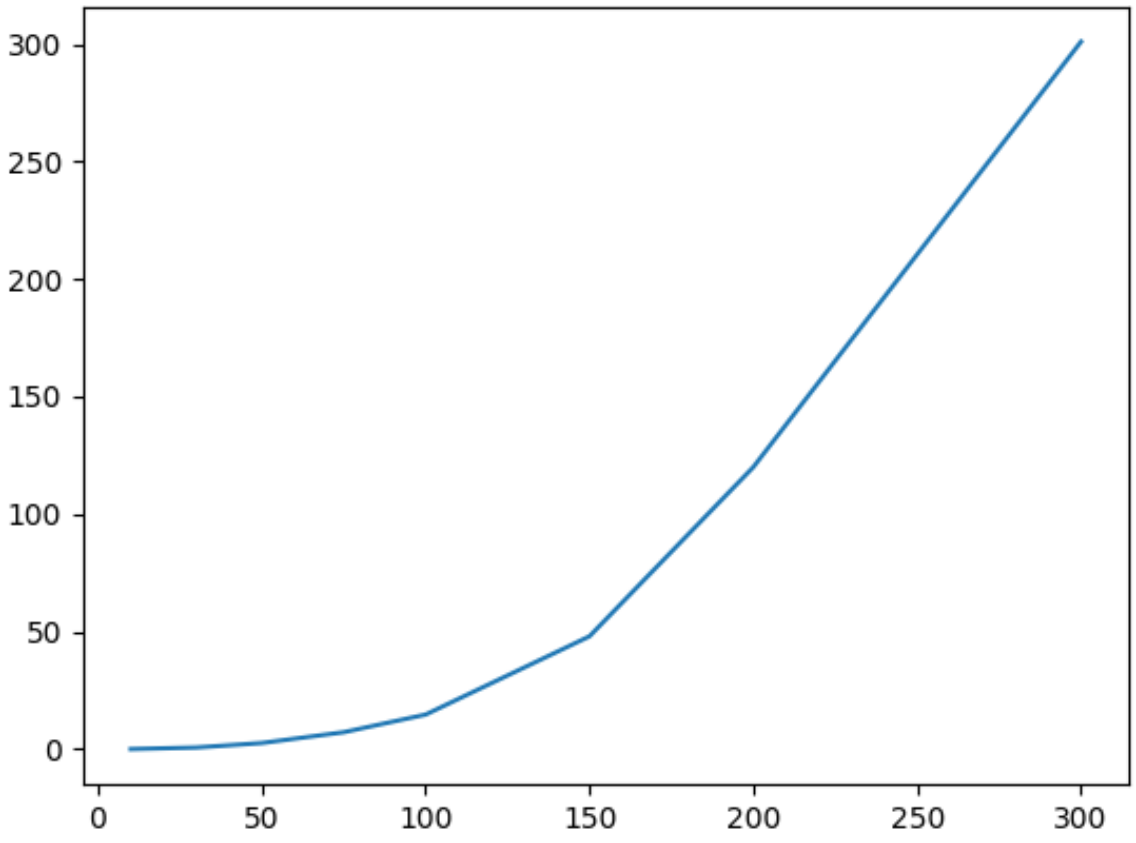

Figure 10: temps d'exécution en fonction du nombre d'états

En notant n_i le nombre d'itérations avec la méthode i et m le nombre d'états on a une complexité de $\boldsymbol{n_2 * m^3}$. Cette méthode est **plus lente** : pour 300 états, le temps d'exécution est environ 300 secondes alors que la méthode de convergence de π_n dure environ 1 seconde.

Si la chaîne de Markov est ergodique alors la matrice converge. Dans ce cas, à chaque itération, on peut calculer $M^i * M^i = M^{2i}$ au lieu de $M^i * M = M^{i+1}$ sans perdre d'information. On a alors une complexité de $log(n_2) * m^3$

Si la chaîne converge et n'est pas ergodique, une exécution de l'algorithme suffit pour obtenir la matrice, qui, à toute distribution initiale associe sa distribution finale. On a donc une complexité de $n_i m^3$ avec les deux méthodes.

Notons que **la suite des M^i peut ne pas converger**. Auquel cas, l'algorithme ne termine pas.

En conclusion, cette méthode ne semble pas être viable face à la méthode de convergence de π_n.

2.2 Méthode non itérative : méthode du point fixe

Pour trouver le point fixe de la matrice, on cherche le vecteur propre associé à la valeur propre 1 et dont les coefficients correspondent à une distribution de probabilité: c'est la "distribution stationnaire". Notre implémentation consiste à calculer l'ensemble des valeurs propres et des vecteurs propres associés puis de chercher le vecteur qui convient. Il existe sûrement des implémentations plus efficaces.

Si la chaîne de Markov est irréductible, il existe une unique probabilité stationnaire. La précision obtenue est de 15 décimales environ, c'est la **meilleure précision atteignable** avec les méthodes 1 et 2.

Avec 'Mouse In Maze", il y a plusieurs probabilités stationnaires et l'algorithme n'est **pas utilisable**.

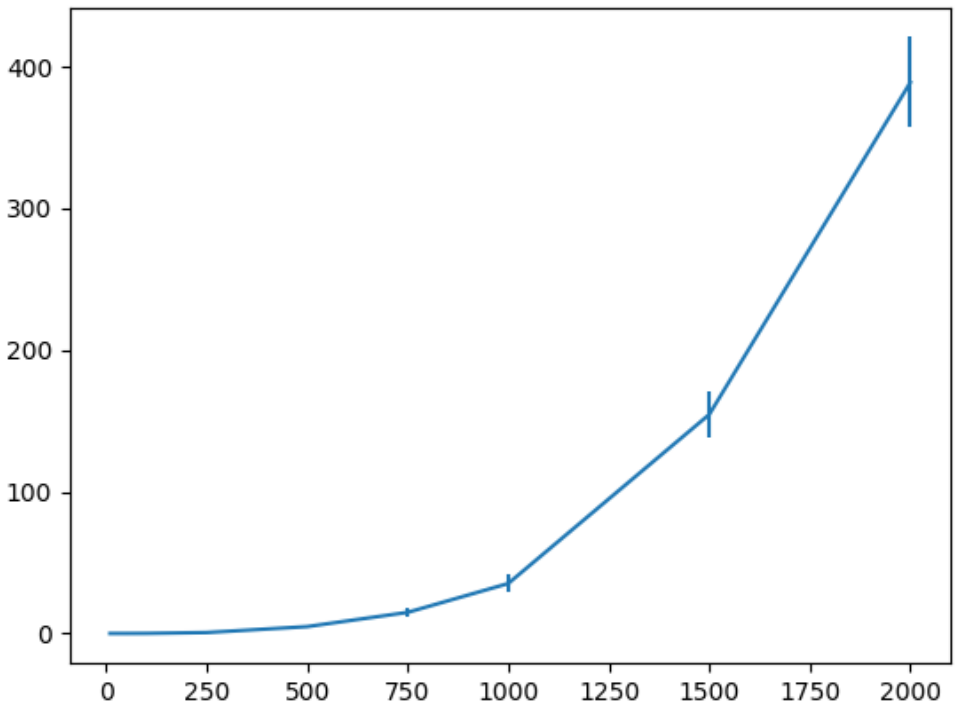

Figure 11: temps d'exécution en fonction du nombre d'états

Pour une même précision, la durée d'exécution de cet algorithme est **entre celles des méthodes de convergence de π_n et M^n**.

3 Le jeu de l'oie

Le problème est similaire à celui de "MonoBestiole"

Avec une instance de Oie possédant 20 cases normales, on a un temps moyen de 1.10^{-4}s ($\sigma = 7.10^{-05}$) et un nombre d'itérations moyen de 10.8 ($\sigma = 5.4$). Rajouter des "sauts" augmente l'écart-type, rajouter des puits augmente la moyenne et l'écart-type, avoir un très petit dé augmente la moyenne et baisse l'écart-type, un dé très grand ne baisse pas la moyenne (car il diminue les chances de tomber sur la dernière cases).

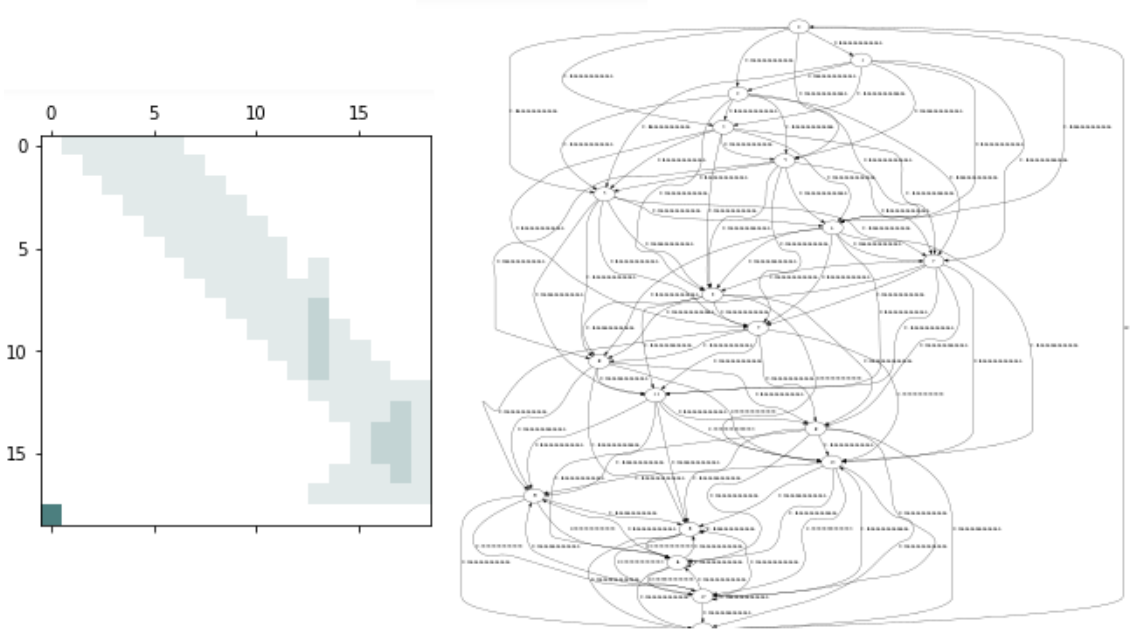

Figure 12: matrice et graphe de transition d'une instance de Oie20 avec un saut

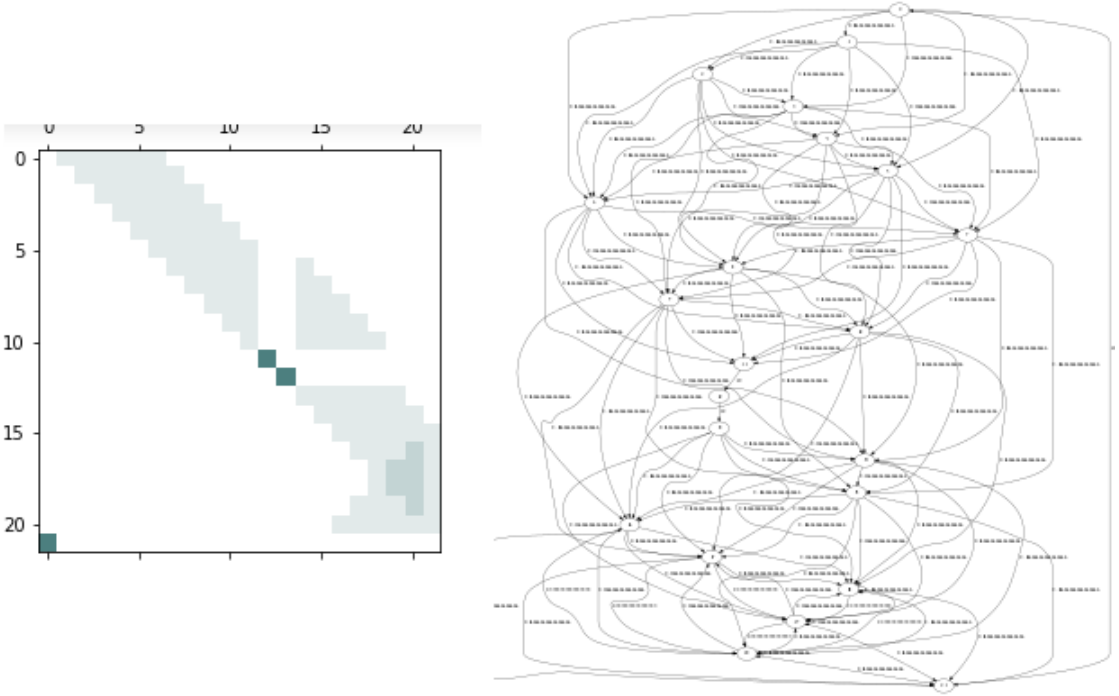

Figure 13: matrice et graphe de transition d'une instance de Oie20 avec un puits

Le graphe de transition devient rapidement illisible avec le nombre d'états.
Un saut pourrait être représenté par un état avec une probabilité de 1 vers sa destination. Cependant, un saut se faisant durant la même itération qu'un déplacement, on ne représente pas cet état transitoire.
Pour un puits, il y a 3 états possibles ("vient d'arriver", "là depuis 1 tour" et "là depuis 2 tours").
Au final, on a donc un nombre d'états égal au **nombre de cases moins le nombre de sauts plus deux fois le nombre de puits**.

Pour mesurer la complexité expérimentale des algorithmes, on fait varier le nombre d'états de l'instance. Les "sauts" et "puits" n'étant pas pertinents pour mesurer la complexité des algorithmes de façon générique, on ne fait que varier le nombre de case.
Ensuite, on trace les courbes en utilisant une **échelle logarithmique pour les deux axes, si la courbe obtenue est une droite, alors la complexité est en** $O(x^{coefdirecteur})$.

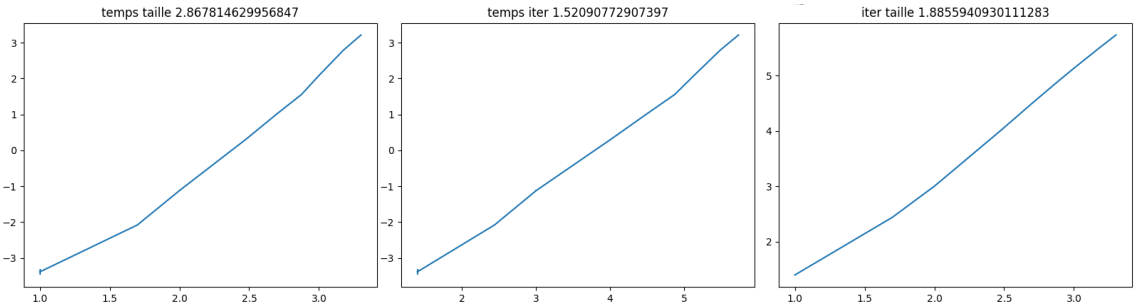

Figure 14: courbes obtenues pour vecteur

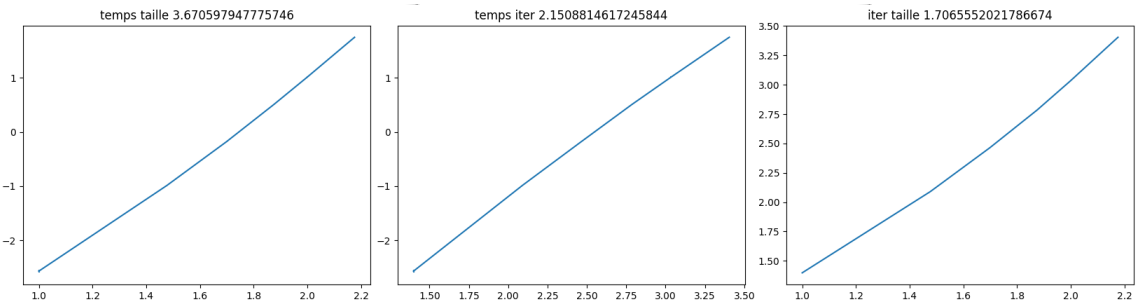

Figure 15: courbes obtenues pour matrice

On mesure le temps d'exécution et le nombre d'itération en fonction du nombre d'états et en fonction du nombre d'itérations. (les valeurs sont données en annexe). On obtient les coefficients directeurs suivants :

	temps/nb_états	temps/nb_iter	nb_iter/nb_états	Oie1500 durée(s)
simulation	**0,0**	–	0,0	13
vecteur	2,87	**1,52**	1,89	621
matrice	3,67	**2,15**	1,71	∞
pointfixe	**2,29**	–	–	96

La simulation demande une erreur relative de 10^{-6} et vecteur et matrice une précision de 10^{-9}. Pour les grandes instances, seulement une ou deux exécutions de l'algorithme ont été effectuées, mais les valeurs obtenues restent fiables car l'écart type des mesures est toujours faible. Le tableau se lit : pour la méthode de convergence de π_n , on a mesuré un temps d'exécution proportionnel au nombre d'itération à la puissance 1,52.

Avec le problème de l'oie, le nombre d'itération des méthodes du vecteur ou de la matrice augmente presque quadratiquement avec le nombre d'états. La méthode du point fixe n'a pas ce problème. Il est résulte une meilleure complexité pour la méthode du point fixe.

La méthode de la simulation : Oie50 met 11s à s'exécuter et Oie2000 13s : le temps n'augmente pas avec le nombre d'états. Cette méthode fournit une précision inférieure mais son temps d'exécution est plus faible que celui du point fixe.

Pour les méthodes du vecteur et de la matrice, les complexités mesurées ne correspondent pas à la complexité théorique mais restent cohérentes entre elles. Cela est peut-être dû au parallélisme mis en place par les méthodes de numpy.

4 Conclusion

Étant donné une chaîne de Markov, nous avons le choix entre plusieurs méthodes. Le temps d'exécution de la simulation ne dépendant pas du nombre d'états de la chaîne, un critère pour choisir cette méthode est donc que le nombre d'états soit élevé. Cependant avec sa complexité en $1/err^2$, cette méthode est peu efficace pour avoir une grande précision.

La méthode de convergence de π_n a une complexité théorique quadratique avec le nombre d'itérations et varie peu avec la précision souhaitée. Cependant, le nombre d'itérations nécessaires pour obtenir la précision souhaitée varie selon les chaînes (pour un même nombre d'états).

La technique de la convergence de la matrice est similaire à celle de π_n sauf en terme efficacité: elle varie théoriquement avec le cube du nombre d'états.

La méthode du point fixe donne une précision élevée (10^{-15}), de plus, selon les chaînes, elle peut être plus rapide que celle de convergence de π_n.

Rajoutons que ces deux dernières méthodes peuvent ne pas être utilisables dans certains cas et par ailleurs que des améliorations de leur complexité sont possibles. On pourrait par exemple multiplier les matrices avec l'algorithme de Strassen.