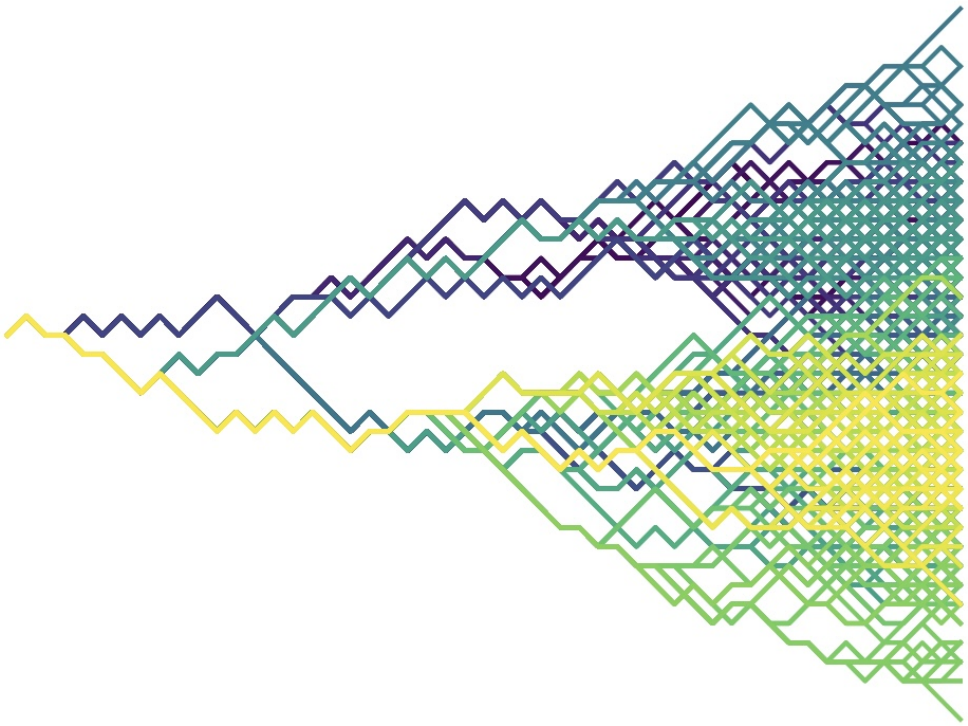
Projet S8 : Marche Aléatoire Branchante



Auteurs : YAKOUBI Elies

Date : 20 juin 2023

Remerciement

Je tiens à exprimer mes sincères remerciements à M. Bastien Mallein pour son soutien et son accompagnement tout au long de la réalisation de ce projet. Sa guidance précieuse, son expertise et son enthousiasme ont été d'une grande valeur pour la réussite de ce travail.

M. Mallein m' a non seulement fourni des conseils avisés sur la méthodologie de recherche, mais a également consacré du temps et de l'énergie pour discuter et échanger sur les différentes idées et les approches à suivre. Sa patience, son expertise et sa passion pour le sujet ont été une source d'inspiration.

Sommaire

Introduction

0.1 La marche aléatoire branchante

Les marches aléatoires branchantes sont des modèles appartenant au domaine des probabilités et des statistiques qui decrivent des particules se déplaçant de manière aléatoire dans un espace et ce divisant de manière régulière.
Ces marches présentent un comportement complexe et sont un sujet de recherche fascinant dans le domaine des mathématiques et de la modélisation stochastique. Dans une marche aléatoire branchante, une particule peut effectuer différents types de mouvements à chaque étape. Elle peut se déplacer vers la gauche avec une probabilité pl, vers la droite avec une probabilité pr, ou se diviser en 2 particules distinctes avec une probabilité r.Les deux nouvelles particules suivront ensuite la même loi de déplacement que la particule d'origine, c'est-à-dire qu'elles auront les mêmes probabilités pr, pl et r pour leurs mouvements futurs. Ces probabilités obéissent à la formule des probabilités totales, c'est-à-dire que :

$$pl + pr + r = 1$$

.

Ces marches aléatoires branchantes sont couramment utilisées pour modéliser divers phénomènes du monde réel. Comprendre comment générer efficacement et précisément ces processus est essentiel pour de nombreuses applications pratiques, telles que la modélisation de la volatilité des prix sur les marchés financiers ou la prévision de la croissance de structures biologiques (les arbres, les réseaux neuronaux ou les vaisseaux sanguins).
Dans ce rapport, nous explorerons en détail le concept des marches aléatoires branchantes. Nous aborderons les méthodes efficaces pour générer et simuler ces processus, en mettant en évidence les outils et les techniques utilisés dans le domaine des mathématiques et de la modélisation stochastique. À travers cette étude approfondie des marches aléatoires branchantes, nous espérons fournir une compréhension claire de ce concept complexe et de son potentiel dans divers domaines de recherche et d'application pratique.

0.2 Objectifs et méthodologie

L'objectif de ce projet est de simuler un algorithme complexe basé sur l'équation de FKPP (Fisher-Kolmogorov-Petrovsky-Piskunov) afin de réaliser une marche aléatoire branchante et d'observer la particule ayant la position la plus à droite. Cette approche, qui repose sur des calculs de probabilités, offre une solution efficace pour effectuer des simulations à long terme, tout en évitant les problèmes de mémoire et les temps de calcul prohibitifs rencontrés avec d'autres méthodes.

Il est crucial d'utiliser cette méthode basée sur l'équation de FKPP pour mener des simulations à grande échelle sur une longue période de temps. En effet, les approches traditionnelles peuvent rapidement devenir impraticables en raison de contraintes de mémoire et de temps de calcul. Grâce à cette approche efficace, nous serons en mesure d'observer la particule la plus à droite de manière précise et réaliste, tout en maintenant des performances computationnelles acceptables.
La simulation de marche aléatoire, même sur une machine puissante, peut rapidement saturer la mémoire et entraîner des problèmes de performance. Il est donc crucial de trouver une approche astucieuse pour modéliser efficacement les quantités d'intérêt liées à ce processus. Bien que très utile pour de nombreuses applications, elle peut devenir un défi si elle est mal implémentée.
Le premier chapitre de ce rapport présentera une méthode volontairement simpliste qui soulignera un problème majeur : le temps de calcul et la saturation de la mémoire. Pour résoudre ce problème, le chapitre 2 présentera une méthode avec un temps de calcul raisonnable, mais qui présentera un inconvénient non présent dans le premier algorithme : la perte de la généalogie des particules. Ces deux premiers exemples de simulation nous permettront donc, dans le chapitre 3, de présenter l'algorithme de simulation basé sur l'équation de FKPP de manière plus détaillée et complète.

Chapitre 1

Modélisation naïve d' une marche aléatoire branchante

1.1 Description de la methode naïve

Tout d'abord, nous allons présenter une approche dite "naïve" de simulation de la marche aléatoire branchante. Cette approche peut être tentante à première vue, car elle est simple à mettre en œuvre. Cependant, nous allons montrer qu'elle présente des limites importantes en termes de performance et de gestion de la mémoire.
Pour illustrer cela, nous vous proposerons un exemple de script Python qui simule la marche aléatoire branchante de la manière la plus simple possible.
En comprenant les défauts de cette approche naïve, nous pourrons ensuite explorer des solutions plus sophistiquées et optimisées pour modéliser la marche aléatoire branchante de manière efficace.
Nous aborderons les différentes stratégies et techniques qui peuvent être utilisées pour améliorer les performances de génération de la marche aléatoire branchante tout en évitant les problèmes de saturation de la mémoire.
voici le pseudo-code de l' algorithme que nous avons mis en place

Algorithm 1 Simulation des trajectoires des particules

Variables: p, q, r : les probabilités ; Nbr : le nombre d' itérations de la simulation
Sortie : Listes de particules, trajectoires et positions

Initialiser les listes pour les particules, les trajectoires et les positions

for *chaque itération jusqu'à Nbr* **do**
  Initialiser les listes pour les nouvelles particules et les nouvelles trajectoires Récupérer les dernières valeurs des positions maximales enregistrées
  for *chaque particule* **do**
    Récupérer la position de la particule actuelle et sa trajectoire Déterminer un déplacement dx basé sur un tirage aléatoire **if** *le tirage aléatoire est dans la plage pour la division* **then**
      Ajouter une particule enfant
    end
    Mettre à jour la position de la particule et conserver les nouvelles positions maximales Ajouter la nouvelle particule et sa trajectoire aux nouvelles listes
  end
  Mettre à jour les listes de particules et de trajectoires avec les nouvelles listes Enregistrer les nouvelles positions maximales et la position de la particule la plus à droite
end

Le script Python correspondant est fourni en annexe pour votre référence.

1.1.1 Resultat de l algorithme

Nous traçons différents graphes pour mieux comprendre le comportement des particules de la marche aléatoire branchante.

Dans la figure 1.1, nous traçons sur un même graphique la position de la particule la plus haute à chaque étape (en vert) ainsi que la trajectoire (en orange) de la particule qui sera la plus à droite au temps terminal. On observe que la trajectoire orange peut s'éloigner de façon importante de la trajectoire verte. Enfin, on trace en bleu une trajectoire typique de la marche branchante. Cette représentation nous permet de visualiser la différence de comportement entre ces particules et d'observer comment leur trajectoire évolue au fil du temps.

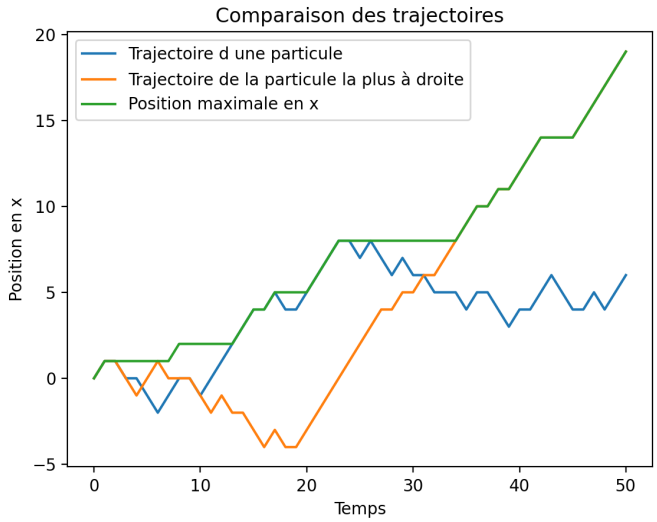


FIGURE 1.1 – Comparaison de différentes trajectoires d'une marche aléatoire branchante.

La figure 1.2 représente l'ensemble des trajectoires des particules. Nous avons observé une forme de cône qui se dessine à mesure que le nombre de particules augmente. Cette représentation nous a permis de mieux comprendre l'évolution de la distribution des particules dans l'espace au fil du temps.

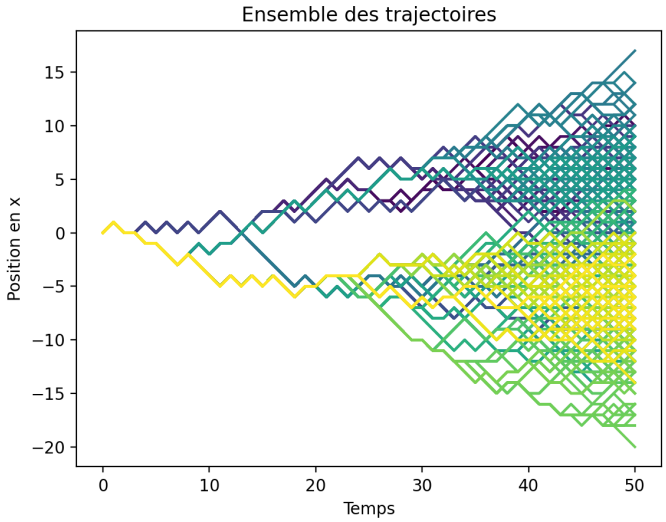


FIGURE 1.2 – ensemble des trajectoires

Nous avons également effectué une mesure en échelle logarithmique du temps d'exécution de notre algorithme. Sur la Figure 1.3, il est clairement visible que le temps d'exécution présente une croissance exponentielle. Cette tendance s'explique par le fait que lorsque les particules se divisent, nous avons deux nouvelles particules indépendantes à suivre, ce qui devient rapidement problématique en termes de mémoire et de temps de calcul. Le temps de calcul commence à devenir significatif après environ 50 itérations.

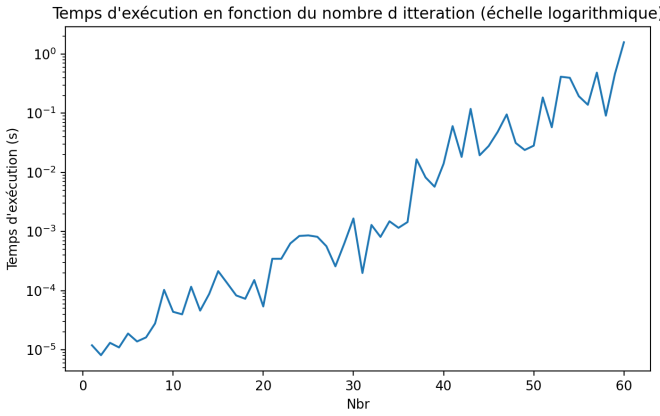


FIGURE 1.3 – Temps de calcul en fonction du nombre d itteration.

1.1.2 Conclusion

Les différents graphes que nous avons produits nous ont permis de visualiser clairement et avec précision les résultats de notre simulation, et nous ont aidé à mieux comprendre le comportement des particules dans le cadre de notre étude sur la marche aléatoire branchante.

Cependant, comme mentionné précédemment, cet algorithme présente des limitations. En effet, lorsque le nombre d'itérations devient trop élevé, c'est-à-dire lorsque

$$(1+r)^n > 10^5$$

(où n représente le nombre d'itérations), le temps de calcul de l'algorithme devient considérablement plus important.

Il est donc impératif de trouver une méthode alternative pour simuler le processus sur une plus longue période de temps.

Dans le prochain chapitre, nous présenterons une méthode de simulation alternative que nous avons nommée la méthode de l'urne.

Chapitre 2

Méthode Alternative de l'Urne

2.1 Description de la méthode

La méthode de l'urne est une approche utilisée pour modéliser des processus stochastiques. Son principe est simple : une urne contient des balles de différentes couleurs, chaque couleur représentant un événement possible.

Dans notre simulation , nous adaptons cette méthode en utilisant des balles pour représenter les particules et leurs actions potentielles. Chaque couleur de balle correspond à une action spécifque : se déplacer vers la droite, se déplacer vers la gauche ou se diviser en deux nouvelles particules.

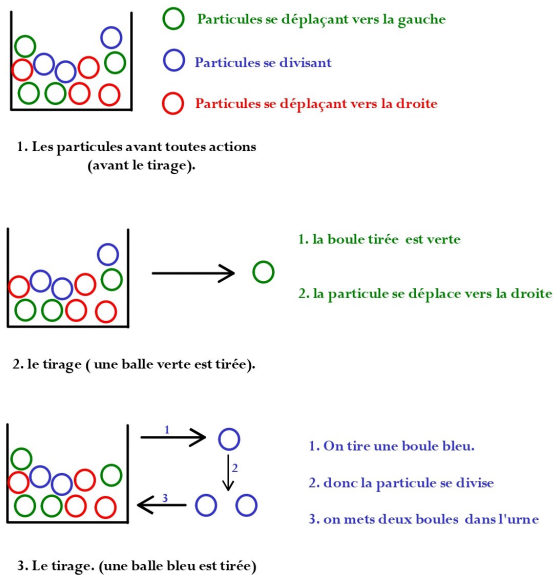


FIGURE 2.1 – Représentation schématique de la methode de l'urne

2.2 Mise en place de l'algorithme

Voici l'algorithme de simulation du mouvement en utilisant la méthode de l'urne :

procedure BIN(n, p, S) ▷ Fonction pour générer un nombre aléatoire **if** $n > 100$ *and* $n * p > S$ *and* $n * (1 - p) > S$ **then**

return Un nombre aléatoire de la distribution normale avec $\mu = n * p$ et $\sigma = \sqrt{n * p * (1 - p)}$ **else**

return Un nombre aléatoire de la distribution binomiale avec n et p

procedure BRANCHINGPROCESSSIMULATION(*nombre_etapes*, *p_branch*, *dt*)

Initialise *particules* ▷ Liste de dictionnaires pour stocker le nombre de particules à chaque position **for** $t \leftarrow 1$ *to nombre_etapes* **do**

$x \in particules[t - 1]$ ▷ Parcours de toutes les positions actuelles

Obtient $nb\_particules \leftarrow particules[t - 1][x]$ **if** $nb\_particules > 0$ **then**

$n\_branch \leftarrow$ BIN($nb\_particules$, p_branch)

$n\_right \leftarrow$ BIN($nb\_particulesn\_branch$,1 - p_branch)

$n\_left \leftarrow nb\_particules - n\_branch - n\_right$

Met à jour $particules[t][x] \leftarrow 2 * n\_branch$

Met à jour $particules[t][x + 1] \leftarrow n\_right$

Met à jour $particules[t][x - 1] \leftarrow n\_left$

return *particules*

end procedure=0

Le script Python correspondant est fourni en annexe.

2.3 Resulats

Nous avons exécuté une simulation sur une durée de 1000 générations avec S = 10. Voici les résultats obtenus :

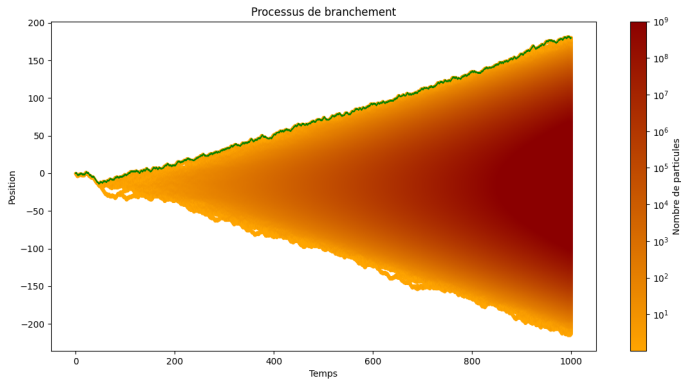


FIGURE 2.2 – Les positions occupées par les particules et leurs densités

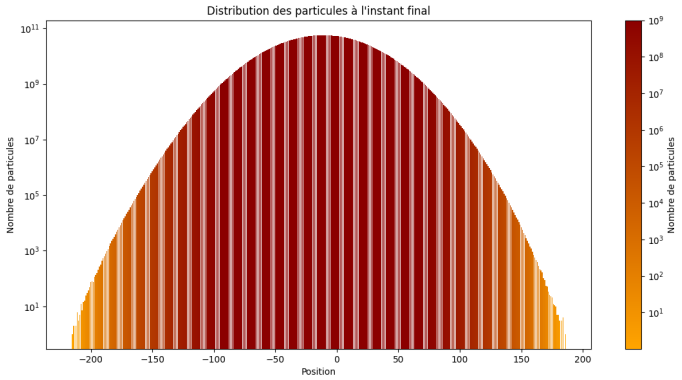


FIGURE 2.3 – Distribution normale des particules par la méthode de l'urne

Lorsque les probabilités de déplacement sont équilibrées, c'est-à-dire que la probabilité de se déplacer vers la droite est égale à la probabilité de se déplacer vers la gauche, nous observons une distribution symétrique des particules autour de la position initiale. Donc les premieres générations ressemblent bien à celle de l'algorithme naif.

L'ajustement des probabilités de division a un impact significatif sur l'évolution de la population de particules. Lorsque la probabilité de division est élevée, nous observons une augmentation rapide du nombre de particules au fil du temps.

Pour mieux visualiser la distribution des particules, nous avons representé la concentration des particules par une nuance de couleur . Les couleurs plus intenses correspondent à des densités de particules plus élevées, tandis que les couleurs plus claires indiquent des densités plus faibles. Cette représentation visuelle facilite l'analyse et la compréhension des résultats.

Bien que la méthode de l'urne offre une approche intuitive et efficace pour simuler le mouvement des particules , elle présente certaines limites. L'une des principales limitations est que cette méthode ne conserve pas les liens entre les particules parentes et leurs descendants. Par conséquent, nous perdons des informations sur la généalogie des particules, ce qui peut être crucial pour certaines analyses.

De plus, la méthode de l'urne peut être sujette à des erreurs d'échantillonnage si les probabilités de déplacement et de division sont très faibles. Dans de tels cas, la taille de l'échantillon peut être insuffisante pour représenter avec précision la distribution des particules

2.4 Conclusion

En conclusion, nous avons présenté une simulation du mouvement branchant des particules en utilisant la méthode de l'urne. Cette méthode nous permet de modéliser les déplacements et les divisions des particules de manière efficace et intuitive. Cependant, nous devons garder à l'esprit les limites de cette méthode, notamment la perte d'informations sur la généalogie des particules. Pour cette raison on introduit dans la partie suivant un simulation qui s'appuie sur les équations différentielle partiels (EDP).

Chapitre 3

Méthode de simulation à partir de l' équation de Fisher, Kolmogorov, Petrovsky, Piscounov

3.1 Présentation de l'équation de Fisher, Kolmogorov, Petrovsky, Piscounov

L'équation de FKPP, ou équation de Fisher-Kolmogorov-Petrovsky-Piscounov, a été introduite en 1937 [0] pour décrire la propagation d'une onde de densité de population dans un milieu homogène. Cette équation non linéaire est couramment utilisée pour modéliser la diffusion de populations biologiques, la combustion et la réaction-diffusion.

L'équation FKPP dans le cadre d'une marche aléatoire branchante s'écrit :

$$\frac{\partial u}{\partial t} = \frac{1}{2}\frac{\partial^2 u}{\partial x^2} + u - u^2 \text{ avec } u(0,x) = 1 \text{ si } x < 0, \text{ 0 sinon.} \tag{3.1.1}$$

où $u(t,x)$ est la probabilité que le plus grand déplacement dépasse la position x au temps t. Cette équation est une équation de réaction-diffusion non linéaire, qui décrit la diffusion de particules ainsi que leur interaction locale.

La solution de l'équation FKPP peut être obtenue en utilisant des méthodes numériques telles que la méthode des éléments finis ou la méthode des différences finies. Dans notre cas, nous avons utilisé une discrétisation particulière pour résoudre l'équation par la méthode des éléments finis :

$$u(t+\Delta t, x) = p_r u(t, x-\Delta x) + p_l u(t, x+\Delta x) + ru(t,x)[2-u(t,x)] \tag{3.1.2}$$

où Δt et Δx sont les pas de temps et d'espace, respectivement, et p_r, p_l, et r représentent respectivement les probabilités que la particule se déplace vers la droite, la gauche ou se divise.

Enfin, la condition initiale pour cette équation est donnée par :

$$u(0,x) = \mathbb{1}_{x\leq 0} \tag{3.1.3}$$

où $\mathbb{1}_{x\leq 0}$ est la fonction indicatrice qui vaut 1 si $x \leq 0$ et 0 sinon. Cette condition initiale modélise la présence initiale des particules à gauche de la position 0.

Le lien entre l'équation de FKPP et la marche aléatoire branchante réside dans le fait que les deux modèles peuvent être utilisés pour étudier des phénomènes similaires de propagation de front. En particulier, il a été démontré que dans certaines conditions, la solution de l'équation de FKPP peut être approximée par un processus de marche aléatoire branchante.

Plus précisément, si l'on considère une échelle discrète dans l'espace, où chaque site est occupé par une particule, alors la densité de population dans l'équation de FKPP peut être reliée au nombre moyen de particules dans la marche aléatoire branchante. Dans cette analogie, la diffusion correspond au mouvement aléatoire des particules, et la croissance de la population est associée à la possibilité de division de la particule.

Notre objectif sera donc de calculer la probabilité que la particule la plus à droite se trouve à droite d'une position X prédéfinie. Nous souhaitons calculer la probabilité $P(R_t \geq x) = u(t,x)$(avec u la solution de l'équation FKPP), où R_t représente la position de la particule la plus à droite à l'instant t. Cette probabilité

est donnée par la solution de l'équation. Nous allons choisir un horizon temporel T et un objectif X pour pouvoir faire des observations. Nous avons également besoin de définir trois ensemble de particules, codés par leur couleur.

Définition 1 : Une particule est rouge si sa descendance la plus à droite à l'instant T se trouve dans l'intervalle [X,+∞].

Définition 2 : Une particule est orange si sa descendance la plus à droite à l'instant T se trouve dans l'intervalle [X - Δ, X].

Définition 3 : Une particule est bleue si sa descendance la plus à droite à l'instant T se trouve dans l'intervalle [-∞, $X - \Delta$].

Remarque 1. Notons egalement que si une particule n'est ni rouge ni orange elle est obligatoirement bleu.

3.2 Résolution de l'equation de F-KPP

Comme énoncé dans la partie précédente, nous allons résoudre l'équation aux dérivées partielles (3.1.1) en utilisant la méthode des différences finies (3.1.2).

Ci-dessous se trouve un script Python qui implémente cette méthode :

```
dx = 0.1  # Pas d'espace
dt = 0.01  # Pas de temps
pr = 0.3  # Valeur de pr
pl = 0.3  # Valeur de pl
r = 0.4  # Valeur de r

# Param tres du domaine spatial et temporel
x_min = -10.0
x_max = 10.0
t_max = 1.0

# Nombre de points de discr tisation
Nx = int((x_max - x_min) / dx) + 1
Nt = int(t_max / dt) + 1

# Cr ation du tableau pour stocker les valeurs de u
u = np.zeros((Nt, Nx))

# Condition initiale
u[0, :] = np.where(np.linspace(x_min, x_max, Nx) <= 0, 1, 0)

# Boucle temporelle pour r soudre l'EDP
for i in range(1, Nt):
    for j in range(1, Nx - 1):
        u[i, j] = pr*u[i-1, j-1] + pl*u[i-1, j+1] + r*u[i-1, j]*(2 - u[i-1, j])
```

Nous avons également décidé de tracer les solutions de cette équation à différents moments. Ci-dessous, le graphique qui illustre les solutions à different temps de notre équation.

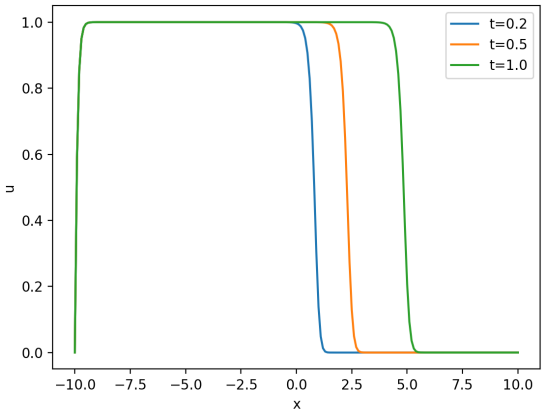


FIGURE 3.1 – Solution de l'équation F-KPP avec condition initiale $1_x \leq 0, tracéeauxtemps0.2{,}0.5et1$.

3.3 Modélisation de la marche aleatoire grace a l equation de FKPP

Maintenant que nous avons résolu l'équation de FKPP grâce à la méthode des éléments finis, nous sommes en mesure de fournir des probabilités permettant de prédire la couleur d'une particule. Les probabilités obtenues sont les suivantes :

Après avoir défini $U(t,x)$ et $V_\Delta(t,x)$ comme les probabilités suivantes :

$$U(t,x) = \mathbf{P}(\text{ particule est rouge}) = u(T-t, X-x)$$
$$V_\Delta(t,x) = \mathbf{P}(\text{la particule est orange}) = U(t,x+\Delta) - U(t,x)$$

nous pouvons calculer les 5 probabilités conditionnelles suivantes en utilisant que $P(R_t > x) = u(t,x)$:

$\mathbf{P}(\text{ particule va à droite} \mid \text{particule est rouge}) = p_r \ \frac{U(t+\delta t, x+\delta x)}{U(t,x)}$
$\mathbf{P}(\text{ particule va à gauche} \mid \text{particule est rouge}) = p_l \ \frac{U(t+\delta t, x-\delta x)}{U(t,x)}$
$\mathbf{P}(\text{ particule se divise en 2 particules rouges} \mid \text{particule est rouge}) = r \cdot \frac{U(t+\delta t,x)^2}{U(t,x)}$
$\mathbf{P}(\text{ particule se divise en 1 particule rouge et 1 orange} \mid \text{ particule est rouge}) = r \cdot \frac{2U(t+\delta t,x)V\Delta(t+\delta t,x)}{U(t,x)}$
$\mathbf{P}(\text{ particule se divise en 1 particule rouge et 1 bleu} \mid \text{particule est rouge}) = r \cdot \frac{2U(t+\delta t,x)(1-U(t+\delta t,x+\Delta))}{U(t,x)}$

Remarque 2. On note que la formule des probabilités totales est vérifiée pour ces 5 probabilités.

En utilisant ces probabilités conditionnées à la particule rouge, nous pouvons déduire les probabilités conditionnées à la particule orange en remplaçant $U(t,x)$ par $V_\Delta(t,x)$. Ainsi, les probabilités conditionelles sachant que la particule est orange sont les suivantes :

$\mathbf{P}(\text{particule va à droite} \mid \text{particule est orange}) = p_r \left(\frac{V\Delta(t+\delta t,x+\delta x)}{V\Delta(t,x)}\right)$
$\mathbf{P}(\text{particule va à gauche} \mid \text{particule est orange}) = p_l \left(\frac{V\Delta(t+\delta t,x-\delta x)}{V\Delta(t,x)}\right)$
$\mathbf{P}(\text{particule se divise en 2 particules oranges} \mid \text{particule est orange}) = r \cdot \left(\frac{V\Delta(t+\delta t,x)^2}{V\Delta(t,x)}\right)$
$\mathbf{P}(\text{particule se divise en 1 particule orange et 1 bleu} \mid \text{particule est orange}) = r \cdot \left(\frac{2V\Delta(t+\delta t,x)(1-U(t+\delta t,x+\Delta))}{V\Delta(t,x)}\right)$

Grâce à ces différentes probabilités, nous sommes maintenant en mesure d'effectuer une simulation en partant d'une seule particule rouge. Cette simulation nous permettra de modéliser une marche aléatoire branchante et de tracer l'ensemble des particules rouges et oranges. Nous négligeons les particules bleues qui ne sont pas pertinentes, car elles ne font pas partie des particules les plus à droite. Cela nous permet d'optimiser à la fois le temps de calcul et l'utilisation de la mémoire.

voici le pseudo-code de l' algorithme que nous avons mis en place :

Algorithm 2 Simulation des particules en utilisant l' equation de F-KPP

Paramètres: Paramètres de la simulation

`// Résolution de l'équation FKPP`
Résoudre l'équation FKPP pour obtenir $u(x, t)$ en utilisant la méthode des éléments finis

`// Initialisation des particules`
Initialiser la liste des particules avec une particule rouge à l'origine (0, 0)

`// Simulation`
while *Le temps de simulation n'a pas atteint la valeur finale T* **do**
  foreach *particule dans la liste des particules* **do**
    if *La particule est rouge* **then**
      Calculer les probabilités conditionnelles correspondantes en utilisant les formules fournies
      Effectuer un choix en utilisant les probabilités conditionnelles pour déterminer le mouvement de la particule
      Ajouter de nouvelles particules rouges et oranges en fonction du choix effectué
    end
    if *La particule est orange* **then**
      Calculer les probabilités conditionnelles correspondantes en utilisant les formules fournies
      Effectuer un choix en utilisant les probabilités conditionnelles pour déterminer le mouvement de la particule
      Ajouter de nouvelles particules rouges et oranges en fonction du choix effectué
    end
  end
end

`// Tracer les positions des particules`
Tracer les positions des particules rouges et oranges obtenues lors de la simulation

Le script Python correspondant est fourni en annexe pour votre référence.

3.4 Resulats

Nous traçons différents graphes pour mieux comprendre le comportement des particules de la marche aléatoire branchante.

La Figure 3.2 présente les trajectoires complètes des particules rouges et oranges à l'instant T=8, avec une valeur de X égale à 4 et un delta de 0.5. Une observation attentive révèle que les particules rouges occupent l'intervalle $[X, +\infty]$, tandis que les particules oranges se trouvent dans l'intervalle $[X - \Delta, X]$. Cette observation offre une précision accrue et couvre une échelle de temps bien plus étendue que celle de la partie 1 de l'étude.

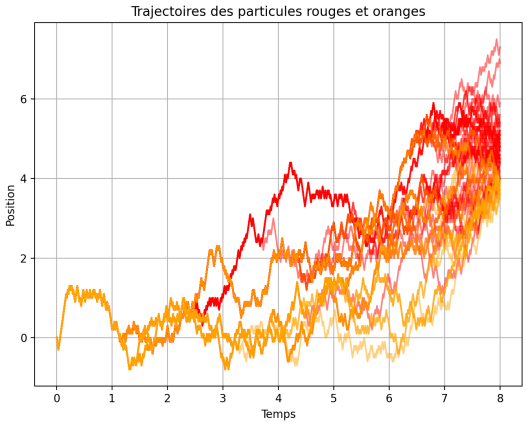


FIGURE 3.2 – ensemble des trajectoires rouge et orange

La Figure 3.3 présente une représentation visuelle des trajectoires distinctes des particules en rouge et en orange, illustrées sur des graphiques individuels. Cette visualisation permet d'observer clairement les comportements et les tendances spécifiques de chaque ensemble de trajectoires.

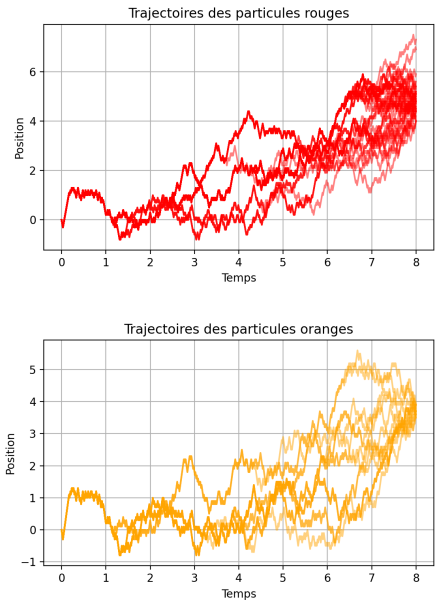


FIGURE 3.3 – ensemble des trajectoires rouge et orange sur des graphiques séparés

La Figure 3.4 met en évidence la trajectoire de la particule atteignant la position la plus à droite à la fin de la simulation. Cette représentation graphique nous permet de visualiser clairement le parcours parcouru par cette particule spécifique tout au long de l'expérience.

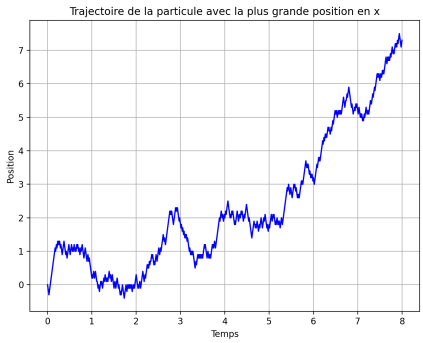


FIGURE 3.4 – trajectoire de la particule la plus a droite

La Figure 3.5 présente une représentation du temps de calcul de la simulation, mesuré à l'échelle logarithmique. Une observation remarquable est que le temps de calcul dans la Figure 3.5 est nettement inférieur à celui de la Figure 1.3. Cette différence notable met en évidence une amélioration significative de l'efficacité du processus de simulation. En utilisant une échelle logarithmique, nous pouvons clairement visualiser l'écart important entre les deux temps de calcul. Ce graphique démontre les progrès réalisés dans l'optimisation des algorithmes. L'observation de ces résultats encourageants renforce la confiance dans l'utilisation de cette méthode de simulation et ouvre de nouvelles perspectives pour des applications plus complexes et des analyses approfondies.

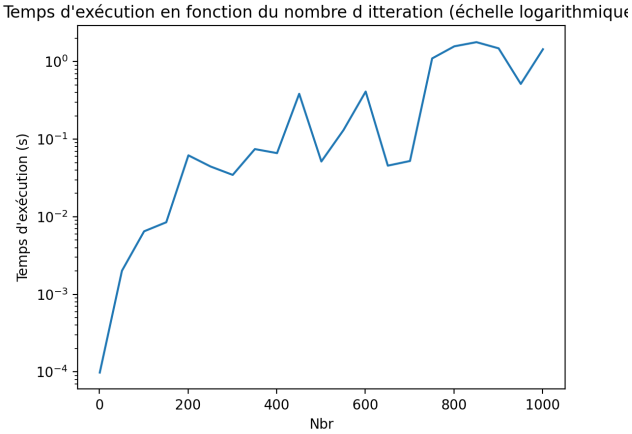


FIGURE 3.5 – temps de calcul de la simulation grace à l' equation de F-KPP

Chapitre 4

Conclusion

Dans ce rapport, nous avons examiné différentes méthodes pour simuler une marche aleatoire branchante. La première méthode présentée a révélé certaines limites, notamment en termes de consommation de mémoire et de temps de calcul. Afin d'optimiser le code, nous avons développé une seconde méthode qui a considérablement accéléré le temps de calcul. Cependant, cette approche a également entraîné une perte de généalogie des particules.

Pour surmonter ces limitations, nous avons proposé une troisième méthode qui combine les avantages des deux premières. Cette méthode se révèle à la fois rapide et permet d'accéder à la généalogie des particules. De plus, en classant les particules en rouge, orange et bleu, nous avons pu éliminer un grand nombre de particules qui ne nous intéressaient pas, ce qui a contribué à une meilleure efficacité du processus.

En conclusion, la troisième méthode que nous avons mise en œuvre s'est avérée plus performante que les deux premières. Elle a permis d'obtenir des résultats plus rapidement tout en conservant la généalogie des particules. Cette approche représente donc une solution efficace pour résoudre le problème étudié. Cependant, il convient de noter que des améliorations supplémentaires pourraient être envisagées pour optimiser davantage le code et explorer d'autres aspects pertinents de cette problématique et pouvoir par exemple faire la simulation sur un temps continue et non discret.

Chapitre 5

Annexe

5.1 Algorithme naif

Voici l'algorithme de simulation "naïf" que nous avons implémenté :

```
import random
import matplotlib.pyplot as plt
import time

p = 0.4  # probabilit  d'aller vers la droite
q = 0.4  # probabilit  d'aller vers la gauche
r = 0.2  # probabilit  de se diviser en deux
Nbr=50  # nombre d iteration dans la boucle de simulation
# initialisation des variables
particules = [0]
trajectoires = [[0]]
positions_max_x = [0]
positions_x_premiere_particule = [0]
positions_x_particule_plus_droite = [0]
debut=time.time()
# boucle de simulation
for i in range(Nbr):
    nouvelles_particules = []
    nouvelles_trajectoires = []
    max_x = positions_max_x[-1]
    particule_plus_droite_x = positions_x_particule_plus_droite[-1]
    for j in range(len(particules)):
        x = particules[j]
        trajectoire = trajectoires[j]
        dx = 0
        rdn = random.random()
        if rdn < p:
            dx = 1
        elif rdn < p + q:
            dx = -1
        elif rdn < p + q + r:
            # ajouter une particule enfant sans changement de position
            nouvelles_particules.append(x)
            nouvelle_trajectoire = trajectoire.copy()
            nouvelle_trajectoire.append(x)
            nouvelles_trajectoires.append(nouvelle_trajectoire)

        x += dx
        max_x = max(max_x, x)  # mettre    jour la position maximale en x
        # mettre    jour la particule la plus    droite
        particule_plus_droite_x = max(particule_plus_droite_x, x)
        nouvelles_particules.append(x)
        nouvelle_trajectoire = trajectoire.copy()
        nouvelle_trajectoire.append(x)
        nouvelles_trajectoires.append(nouvelle_trajectoire)
```

```
46
47      particules = nouvelles_particules
48      trajectoires = nouvelles_trajectoires
49      # ajouter la position maximale en x pour cet instant
50      positions_max_x.append(max_x)
51      # ajouter la position en x de la premi re particule pour cet instant
52      positions_x_premiere_particule.append(trajectoires[0][-1])
53      # ajouter la position en x de la particule la plus    droite pour cet instant
54      positions_x_particule_plus_droite.append(particule_plus_droite_x)
55  fin=time.time()
56  dure= fin - debut
57
58  print(f"Le code a pris {dure} secondes pour  {Nbr} it rations.")
59  '''
60  Dans cette partie de notre  tude  sur la marche al atoire branchante, nous allons
        afficher plusieurs   lments  . Le premier graphe nous permettra de comparer la
        trajectoire de la premi re particule, celle du maximum en tout temps et celle
        de la particule la plus    droite.
61  Le deuxi me graphe, quant    lui, nous permettra de visualiser l'ensemble des
        trajectoires et de mettre en  vidence  la forme de "c ne" qui se dessine au
        fil du temps.
62  En outre, nous avons d cid  d'afficher individuellement chaque   lment   du
        premier graphe pour obtenir des affichages plus pr cis et faciliter la
        compr hension de notre  tude . Ces graphes nous permettront ainsi d'analyser
        les r sultats de notre simulation de mani re claire et approfondie.
63  '''
64  #pour le premier graphe
65
66  # Obtenir la position maximale en x    l'instant final
67  position_max_x = max(particules)
68
69  # Trouver la trajectoire de la particule la plus    droite    l'instant final
70  trajectoire_particule_plus_droite = []
71  for trajectoire in trajectoires:
72      if trajectoire[-1] == position_max_x:
73          trajectoire_particule_plus_droite = trajectoire
74          break
75
76  # Cr er une figure pour les graphes individuels
77  plt.figure()
78
79  # Tracer la trajectoire de la premi re particule
80  plt.subplot(311)
81  plt.plot(range(len(positions_x_premiere_particule)),
82           positions_x_premiere_particule)
83  plt.xlabel('Temps')
84  plt.ylabel('Position en x')
85  plt.title('Trajectoire d une particule')
86
87  # Tracer la position en x de la particule la plus    droite
88  plt.subplot(312)
89  plt.plot(range(len(trajectoire_particule_plus_droite)),
90           trajectoire_particule_plus_droite)
91  plt.xlabel('Instant')
92  plt.ylabel('position en x ')
93  plt.title('Trajectoire de la particule la plus    droite')
94
95  # Tracer la position maximale en x
96  plt.subplot(313)
97  plt.plot(range(len(positions_max_x)), positions_max_x)
98  plt.xlabel('Temps')
99  plt.ylabel('Position en x')
100 plt.title('Position maximale en x')
101
102 # Ajuster l'espacement entre les graphes
```

```
plt.subplots_adjust(hspace=1)


# tracer des graphe individuels pour etre plus precis :

    # graph1
plt.figure()
plt.plot(range(len(positions_x_premiere_particule)),
         positions_x_premiere_particule)
plt.xlabel('Temps')
plt.ylabel('Position en x')
plt.title('Trajectoire d une  particule')

    # graph2:
plt.figure()
plt.plot(range(len(trajectoire_particule_plus_droite)),
         trajectoire_particule_plus_droite)
plt.xlabel('Temps')
plt.ylabel('position de la particule la plus    droite')
plt.title('Trajectoire de la particule la plus    droite')

    # graph3:
plt.figure()
plt.plot(range(len(positions_max_x)), positions_max_x)
plt.xlabel('Temps')
plt.ylabel('Position en x')
plt.title('Position maximale en x')

# Cr er une figure pour le trac  de l'ensemble des trajectoires
plt.figure()

# Parcourir toutes les trajectoires et les tracer
for trajectoire in trajectoires:
    plt.plot(range(len(trajectoire)), trajectoire)

plt.xlabel('Temps')
plt.ylabel('Position en x')
plt.title('Ensemble des trajectoires')
plt.show()


# ici on affiche les 3 sur le meme graph comme demmander :
import matplotlib.pyplot as plt

plt.figure()

# Tracer la trajectoire de la premi re particule
plt.plot(range(len(positions_x_premiere_particule)), positions_x_premiere_particule
    , label='Trajectoire d une particule')

# Tracer la position en x de la particule la plus    droite
plt.plot(range(len(trajectoire_particule_plus_droite)),
    trajectoire_particule_plus_droite, label='Trajectoire de la particule la plus
       droite')

# Tracer la position maximale en x
plt.plot(range(len(positions_max_x)), positions_max_x, label='Position maximale en
    x')

plt.xlabel('Temps')
plt.ylabel('Position en x')
plt.title('Comparaison des trajectoires')
plt.legend()

# Afficher le graphique
```

```
plt.show()


# ensembel des trajectoire avce la palette de couleur :
import matplotlib.cm as cm

cmap = cm.get_cmap('viridis')  # Choisissez la palette de couleurs souhait e

for i, trajectoire in enumerate(trajectoires):
    plt.plot(range(len(trajectoire)), trajectoire, color=cmap(i/len(trajectoires)))

plt.xlabel('Temps')
plt.ylabel('Position en x')
plt.title('Ensemble des trajectoires')
plt.show()
```

5.2 Algorithme de l urne

Voici l'algorithme de la méthode de l'Urne que nous avons implémenté :

```
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict
from scipy.stats import norm
import matplotlib.colors as mcolors
from math import log10

def bin(n, p, S=10):
    if n > 100 and n*p > S and n*(1-p) > S:
        return int(norm.rvs(loc=n*p, scale=np.sqrt(n*p*(1-p))))
    else:
        return np.random.binomial(n, p)

def processus_branchement(nombre_etapes, proba_droite, proba_branchement, dt):
    particules = [defaultdict(int) for _ in range(-nombre_etapes, nombre_etapes+1)]
    particules[0][0] = 1

    for t in range(1, nombre_etapes+1):
        for x in list(particules[t-1].keys()):
            nb_particules = particules[t-1][x]
            if nb_particules > 0:
                n_branch = bin(nb_particules, proba_branchement)
                n_right = bin(nb_particules - n_branch, proba_droite)
                n_left = nb_particules - n_branch - n_right
                particules[t][x] += 2 * n_branch
                particules[t][x + 1] += n_right
                particules[t][x - 1] += n_left

    return particules

nombre_etapes = 1000
proba_droite = 0.5
proba_branchement = 0.03
dt = 1

particules = processus_branchement(nombre_etapes, proba_droite, proba_branchement,
    dt)

colors = ["orange", "darkred"]
n_bins = [10 ** i for i in range(1, 10)]
cmap_name = 'my_list'
cm = mcolors.LinearSegmentedColormap.from_list(cmap_name, colors, N=1000)
norm = mcolors.LogNorm(vmin=1, vmax=max(n_bins))

plt.figure(figsize=(10, 6))
particule_droite = []

for t in range(nombre_etapes+1):
    x_values = list(particules[t].keys())
    s_values = [particules[t][x] for x in x_values]
    colors = [cm(norm(value)) for value in s_values]
    plt.scatter([t]*len(x_values), x_values, c=colors, s=10)
    if len(x_values) > 0:
        particule_droite.append(max(x_values))

plt.plot(range(nombre_etapes+1), particule_droite, color='green')

plt.xlabel("Temps")
plt.ylabel("Position")
plt.title("Processus de branchement")

sm = plt.cm.ScalarMappable(cmap=cm, norm=norm)
```

```
sm.set_array([])
plt.colorbar(sm, ticks=n_bins, label='Nombre de particules')

plt.show()

# Ajout de l'histogramme

plt.figure(figsize=(10, 6))
x_values = list(particules[nombre_etapes].keys())
y_values = [particules[nombre_etapes][x] for x in x_values]
plt.bar(x_values, y_values, color=colors)
plt.yscale('log')
plt.xlabel("Position")
plt.ylabel("Nombre de particules")
plt.title("Distribution des particules    l'instant final")

sm = plt.cm.ScalarMappable(cmap=cm, norm=norm)
sm.set_array([])
plt.colorbar(sm, ticks=n_bins, label='Nombre de particules')

plt.show()
```

5.3 Algorithme de la simulation par F-KPP

Voici l'algorithme permettant d'effectuer la simulation en utilisant l'équation de F-KPP :

```
import numpy as np
import matplotlib.pyplot as plt

# Param tres du domaine spatial et temporel
T = 10
X = 4
Delta = 0.5
delta_t = 0.01
delta_x = 0.1
r = delta_t

pr = 0.5 * (1 - r)
pl = 0.5 * (1 - r)

# Nombre de points de discr tisation
Nx = int(3*X / delta_x) + 1
Nt = int(T / delta_t) + 1

# Cr ation du tableau pour stocker les valeurs de u
u = np.zeros((Nt, Nx))

# Condition initiale
u[0, :] = np.where(np.linspace(-X, 2*X, Nx) <= 0, 1, 0)

# Calcul des indices pour la boucle temporelle
indices = np.arange(1, Nx - 1)

# Boucle temporelle pour r soudre l'EDP
for i in range(1, Nt):
    u[i, indices] = pr * u[i - 1, indices - 1] + pl * u[i - 1, indices + 1] + r * u
        [i - 1, indices] * (2 - u[i - 1, indices])
# Fonction U(t, x) utilisant u
def U(t, x):
    i = int(round((T-t)/ delta_t))
    j = int(round((X-x) / delta_x)) + int(X / delta_x)
    return u[i, j]


def VDelta(t, x):
    a = U(t, x + Delta) - U(t, x)
    if a > 0:
        return a
    else:
        return 0

# Initialisation des compteurs de particules orange et bleu
compteur_orange = 0
compteur_bleu = 0
compteur_rouge = 1
temps=1
# Liste pour stocker les positions de toutes les particules
particules = [(0, 0, "r",[(0,0)])]  # La premi re particule

# Boucle pour les 1000 it rations
for _ in range(800):
    # Liste temporaire pour stocker les nouvelles particules
    nouvelles_particules = []

    # Parcours des particules existantes
    for particule in particules:
        t, x,color,trajectoire = particule
        if color == "r":
```

```
            # Calcul des probabilit s de d placement et de division pour la
                particule courante
            p_right = 1 if U(t, x) == 0 else pr * (U(t + delta_t, x + delta_x)
                / U(t, x))
            p_left = 1 if U(t, x) == 0 else pl * (U(t + delta_t, x - delta_x) /
                 U(t, x))
            p_divide_red = 1 if U(t, x) == 0 else r * ((U(t + delta_t, x) ** 2)
                 / U(t, x))
            p_divide_red_orange = 1 if U(t, x) == 0 else r * (2 * U(t + delta_t
                , x) * VDelta(t + delta_t, x) / U(t, x))
            p_divide_red_blue = 1 if U(t, x) == 0 else r * (2 * U(t + delta_t,
                x) * (1 - U(t + delta_t, x + Delta)) / U(t, x))

            a = np.random.rand()

            # D placement de la particule
            if a < p_right:
                nouvelles_particules.append((t + delta_t, x+delta_x, "r",
                    trajectoire + [(t + delta_t, x + delta_x)]))
            elif a < p_left + p_right:
                nouvelles_particules.append((t + delta_t, x-delta_x, "r",
                    trajectoire + [(t + delta_t, x - delta_x)]))

            elif a < p_divide_red + p_left + p_right:
                nouvelles_particules.append((t + delta_t, x, "r", trajectoire +
                     [(t + delta_t, x)]))
                nouvelles_particules.append((t + delta_t, x, "r", trajectoire +
                     [(t + delta_t, x )]))
                compteur_rouge += 1
            elif a < p_divide_red + p_left + p_right + p_divide_red_orange:
                nouvelles_particules.append((t + delta_t, x, "r", trajectoire +
                     [(t + delta_t, x )]))
                nouvelles_particules.append((t + delta_t, x, "o", trajectoire +
                     [(t + delta_t, x )]))
                compteur_orange += 1
            else:
                nouvelles_particules.append((t + delta_t, x, "r", trajectoire +
                     [(t + delta_t, x )]))
                compteur_bleu += 1
        elif color == "o":
            p_right =1 if VDelta(t, x)==0 else   pr * (VDelta(t + delta_t, x +
                delta_x) / VDelta(t, x))
            p_left = 1 if VDelta(t, x)==0 else pl * (VDelta(t + delta_t, x -
                delta_x) / VDelta(t, x))
            p_divide_orange =1 if VDelta(t, x)==0 else r * ((VDelta(t + delta_t, x)
                 ** 2) / VDelta(t, x))
            p_divide_orange_blue =1 if VDelta(t, x)==0 else r * (2 * VDelta(t +
                delta_t, x) * (1 - U(t + delta_t, x + Delta)) / VDelta(t, x))
            #test
            a = np.random.rand()
            if a < p_right:
                nouvelles_particules.append((t + delta_t, x+delta_x, "o",
                    trajectoire + [(t + delta_t, x +delta_x)]))
            elif a < p_left + p_right:
                nouvelles_particules.append((t + delta_t, x-delta_x, "o",
                    trajectoire + [(t + delta_t, x -delta_x)]))

            elif a < p_divide_orange + p_left + p_right:
                nouvelles_particules.append((t + delta_t, x, "o", trajectoire + [(t
                     + delta_t, x )]))
                nouvelles_particules.append((t + delta_t, x, "o", trajectoire + [(t
                     + delta_t, x )]))
                compteur_orange += 1
            else:
                nouvelles_particules.append((t + delta_t, x, "o", trajectoire + [(t
```

```
                    + delta_t, x )]))
106                 compteur_bleu += 1
107
108
109     # Mise    jour des particules
110     particules = nouvelles_particules
111
112     print("rouge orange bleu temps ")
113     print(compteur_rouge, compteur_orange, compteur_bleu,temps)
114     temps+=1
115
116  # ici on verifie que les particules ont la bonne couleur
117  # Suppression des particules dont la position finale est inf rieure    X - Delta
118 particules = [(t, x, color, trajectoire) for t, x, color, trajectoire in particules
        if x >= X - Delta]
119
120 # Mise    jour de la couleur des particules restantes
121 particules = [(t, x, "r" if x > X else "o", trajectoire) for t, x, color,
        trajectoire in particules]
122
123 # Collecte des trajectoires des particules rouges et oranges    la fin de la
        simulation
124 trajectoires_rouges = [p[3] for p in particules if p[2] == 'r']
125 trajectoires_oranges = [p[3] for p in particules if p[2] == 'o']
126
127 # Affichage des trajectoires des particules rouges et oranges sur un graphique
128 plt.figure(figsize=(8, 6))
129 for trajectoire in trajectoires_rouges:
130     temps, positions = zip(*trajectoire)
131     plt.plot(temps, positions, color='red', alpha=0.5)
132 for trajectoire in trajectoires_oranges:
133     temps, positions = zip(*trajectoire)
134     plt.plot(temps, positions, color='orange', alpha=0.5)
135 plt.xlabel('Temps')
136 plt.ylabel('Position')
137 plt.title('Trajectoires des particules rouges et oranges')
138 plt.grid(True)
139 plt.show()
140
141
142 #ici on veut la trajectoire du maximal
143
144 # Recherche de la particule avec la plus grande position en x au temps final
145 particule_max_x = max(particules, key=lambda p: p[1])
146
147 # Affichage de la trajectoire de la particule avec la plus grande position en x
148 trajectoire_max_x = particule_max_x[3]
149 temps_max_x, positions_max_x = zip(*trajectoire_max_x)
150
151 # Affichage de la trajectoire de la particule avec la plus grande position en x sur
        un graphique
152 plt.figure(figsize=(8, 6))
153 plt.plot(temps_max_x, positions_max_x, color='blue')
154 plt.xlabel('Temps')
155 plt.ylabel('Position')
156 plt.title('Trajectoire de la particule avec la plus grande position en x')
157 plt.grid(True)
158 plt.show()
159
160
161
162 fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 12))
163
164 # Graphique des trajectoires des particules rouges
165 ax1.set_xlabel('Temps')
```

```
ax1.set_ylabel('Position')
ax1.set_title('Trajectoires des particules rouges')
ax1.grid(True)
for trajectoire in trajectoires_rouges:
    temps, positions = zip(*trajectoire)
    ax1.plot(temps, positions, color='red', alpha=0.5)

# Graphique des trajectoires des particules oranges
ax2.set_xlabel('Temps')
ax2.set_ylabel('Position')
ax2.set_title('Trajectoires des particules oranges')
ax2.grid(True)
for trajectoire in trajectoires_oranges:
    temps, positions = zip(*trajectoire)
    ax2.plot(temps, positions, color='orange', alpha=0.5)

# Ajustement des espacements entre les sous-graphiques
plt.subplots_adjust(hspace=0.4)

# Affichage de la figure
plt.show()
```

Chapitre 6

Bibliographie

E. Brunet, A. D. Le, A. H. Mueller et S. Munier. *How to generate the tip of branching random walks evolved to large times.*

A. Kolmogorov, I. Petrovsky, N. Piskunov. *Fisher-KPP equation. Comptes rendus de l'Académie des Sciences*, 208 :347–349, 1937.