Projet de programmation
Comment modéliser un mouvement brownien de manière optimale ?

Baptiste Pasquier, Alix Plamont, Eric Vong
21 mai 2020

1 Introduction

Le mouvement brownien a des applications multiples en tant qu'outil stochastique notamment pour la finance, comme le modèle de Black-Scholes régulièrement utilisé pour déterminer le prix d'une action.

À première vue, un tel processus peut sembler difficile à modéliser. Cependant, à l'origine, le mouvement brownien est d'abord un processus physique, celui du déplacement aléatoire d'une grosse particule plongée dans un champ de petites particules, la grosse particule peut alors rentrer en collision avec d'autres particules, ce qui va modifier sa trajectoire. On appelle mouvement brownien son mouvement au cours du temps. Nous avons mis en œuvre cette approche pour modéliser le mouvement brownien.

FIGURE 1 – *Exemple de mouvement brownien*

2 Synthèse

Pour comparer nos différents modèles, nous implémentons le modèle initial et déterminons les paramètres pertinents pour définir effectivement un mouvement brownien : nous prenons en compte le libre parcours moyen (lpm, distance moyenne que la particule suivie effectue avant de rentrer en collision avec une autre particule), on peut en déduire la fréquence de collision $f = \frac{v}{lpm}$, la distance moyenne par rapport à la position initiale ($\bar{d}$) et la distance maximale atteinte par rapport à la position initiale ($d_{\max}$).

Dans le modèle de référence, nous générons un nuage de particules autour de la particule suivie. La densité surfacique (n^*) des particules sera constante sur les simulations. Nous faisons ensuite avancer toutes les particules et nous traitons les collisions inter-particules au fur et à mesure de leur apparition.

Si, dans nos modèles, les particules n'ont pas à proprement parler de rayon et apparaissent ponctuelles, celles-ci sont en fait considérés comme des disques lors des collisions ; en effet, nous avons introduit un paramètre $\varepsilon > 0$ servant de « tolérance » dans la détection des collisions : cette tolérance est en fait directement reliée aux rayons des particules.

Pour avoir moins de calculs, nous avons envisagé de seulement prendre en compte les collisions qui se font avec la particule suivie afin d'avoir une complexité plus faible.

Nous nous sommes ensuite intéressés à la différence entre un environnement ouvert et un environnement clos. L'avantage de l'environnement clos est qu'il nous permet de générer particules d'un coup sans jamais avoir à vérifier leur pertinence. Ainsi, cela permettrait d'abaisser à nouveau la complexité.

Enfin, nous avons étudié la génération des environnements : est-ce que la génération locale de petites particules autour de notre particule suivie va influer sur les résultats, en comparaison avec la génération globale. En effet, il parait pertinent d'essayer de faire une génération locale des particules à chaque itération pour éviter d'avoir un nombre trop élevé de particules, ce qui ralentirait considérablement l'exécution de la simulation.

Nous avons ainsi défini trois modèles :

Modèle n° 1. À chaque collision de la grosse particule, nous définissions un nouvel environnement et de nouvelles petites particules dans un disque de rayon fixé.

Modèle n° 1.1. Il s'agit d'une amélioration du modèle précédent (divisant par deux le temps de calcul). Le temps est découpé en intervalles réguliers et dans chacun d'eux l'algorithme recherche des collisions dans un disque de plus en plus petit (le nombre de petites particules susceptibles de rencontrer la grosse est réduit au fur et à mesure que le temps au sein de l'étape s'écoule).

Modèle n° 2. Nous définissons un unique environnement de dimension finie dans lequel vont se dérouler les collisions de la grosse particule. Lorsqu'une petite particule sort de cet environnement, nous générons une nouvelle particule aléatoire à l'intérieur de l'environnement.

Modèle n° 3. Nous reprenons le modèle n° 2 en considérant à présent les collisions des petites particules entre-elles.

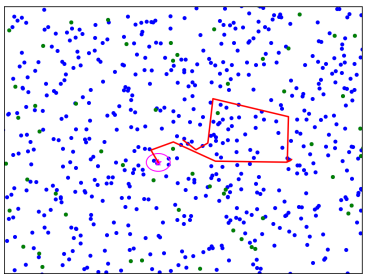

FIGURE 2 – *Exemple de simulation avec le modèle n° 2*

3 Difficultés

Génération aléatoire dans un disque. Dans nos premiers essais, nous avons remarqué que la répartition des collisions n'était pas celle attendue. Nous avons alors constaté que notre fonction de génération aléatoire de points dans un disque était mathématiquement fausse.

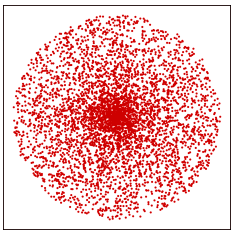

FIGURE 3 – *Génération incorrecte*

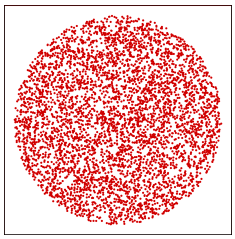

FIGURE 4 – *Génération correcte*

Complexité temporelle. L'utilisation d'une méthode de *code profiling* nous a permis d'optimiser certaines parties de notre code, avec un gain temporel notable de 25% pour le modèle n° 1. Pour permettre la réalisation de nombreux calculs dans le but d'obtenir des résultats statiquement fiables, nous avons aussi utilisé une méthode de *multiprocessing* qui nous a permis de diviser par 4 le temps d'exécution des simulations.

4 Résultats

Le temps de calcul étant important pour les simulations issues des modèles 2 et 3, nous nous sommes résignés à étudier les statistiques de nos modèles sur un échantillon de 32 simulations (par modèle).

Les paramètres choisis sont les suivants : $n^* = 10^4$, $\varepsilon = 10^{-4}$, $v = 10$ et $V = 0.1$ (v et V sont les vitesses des petites particules et de la grosse particule).

Le tableau suivant récapitule les résultats moyens pour chaque modèle.

	lpm	$\bar{d}$	$d_{\max}$	Nb collisions
1	$1.6\,10^{-3}$	$1.1\,10^{-2}$	$2.0\,10^{-2}$	67
1.1	$1.7\,10^{-3}$	$1.0\,10^{-2}$	$1.9\,10^{-2}$	62
2	$1.3\,10^{-3}$	$9.6\,10^{-3}$	$1.8\,10^{-2}$	84
3	$1.5\,10^{-3}$	$9.9\,10^{-3}$	$2.0\,10^{-2}$	67

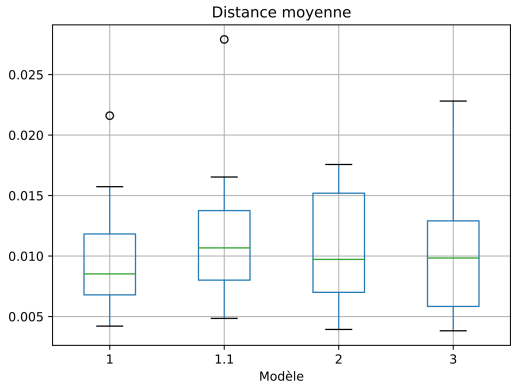

FIGURE 5 – *Boxplots de la distance moyenne selon les modèles*

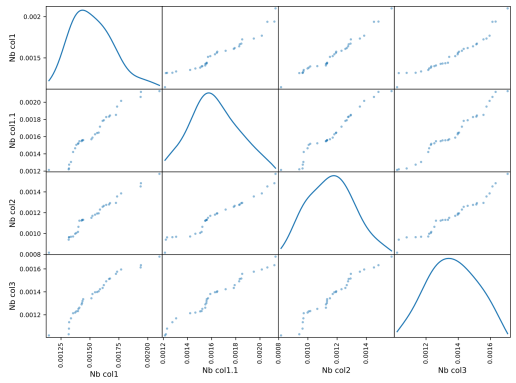

FIGURE 6 – *Matrice de scatter plots de la lpm selon les modèles*

5 Conclusion

Bien que le modèle 2 semble se démarquer par un nombre plus important de collisions, et ainsi un lpm plus petit, les différences entre les quatre modèles implémentés apparaissent négligeables. En particulier, cette expérience nous aura appris les vertus de l'aléatoire pour simplifier énormément une modélisation, à la fois en termes de complexité temporelle et spatiale et en termes de facilité de mise en œuvre.

Les (faibles) écarts observés sur le modèle 2 peuvent s'expliquer par le fait que les petites particules ne réapparaissent pas directement après être sorties de l'environnement ; elles sont régénérées à chaque collision de la grosse particule. Ce phénomène n'est pas observé au sein du modèle 3 car des petites collisions surviennent fréquemment, les petites particules sont ainsi souvent régénérées.

A Instructions d'exécution

Le code est fourni sous la forme d'un package `brownian` disponible sur Github via le lien suivant : https://github.com/baptiste-pasquier/brownian.

Pour installer le package, il est possible de télécharger le répertoire puis exécuter la commande suivante :

```
python setup.py install
```

Ou directement exécuter la commande suivante :

```
pip install git+https://github.com/baptiste-pasquier/
    brownian
```

Exemple d'utilisation :

```
from brownian.simulation1 import Simulation1
from brownian.outils import stats

# Initialisation d une simulation de type 1
a = Simulation1(nb_max_collisions=10, density=0.02,
    epsilon_time=0.5, time_interval=2, speed=10,
    speed_BP_init=10)
# Calcul de la simulation
a.calcul(show=True, coeff_affichage=2, pause=0.5)
# Affichage des statistiques
stats(a, show=True)
# Affichage de la trajectoire finale
a.traj_image(coeff_affichage=2)
```

Le dossier examples contient d'autres exemples d'utilisation.

Pour obtenir des informations plus détaillées, il est possible de consulter le fichier README.md.

B Description d'un point de design du code

Pour les modèles n° 1, n° 2 et n° 3, nous définissons une classe `Particle` qui contient toutes les informations nécessaires pour définir une particule : sa position `self.x`, `self.y` et son vecteur vitesse `self.vx`, `self.vy`. L'attribut `self.epsilon_time` définit la précision pour la détection des collisions. La méthode `update_time(self, delta_time)` permet de mettre à jour la position de la particule après un déplacement rectiligne pendant la durée `delta_time`. La méthode `collision(self, particle2)` permet la détection d'une collision avec une autre particule. On calcule ainsi les dates de collision t_x et t_y selon x et y et on considère qu'une collision est possible si $|t_x - t_y| < \varepsilon$ et si cette collision a lieu dans le futur ($t_x > 0$ et $t_y > 0$). La méthode `change_theta(self, new_theta)` permet de changer l'angle du vecteur vitesse de la particule, cette méthode sera utilisée à chaque collision.

C Coût d'une simulation de type 2

Notons N le nombre de particules dans l'environnement et $nb\_max\_collisions$ le nombre de collisions à simuler.

La fonction `random_particle` utilise les fonctions `random`, `sqrt` (en temps constant) puis fait des opérations élémentaires en temps constant, d'où une complexité en $\Theta(1)$.

C.1 Classe `Workzone_square`

La méthode `__init__` de la classe `Workzone_square` effectue une affectation en temps constant puis définit une liste de taille N, en faisant appel à la fonction `random_particle` (temps constant) à chaque itération de la boucle, nous avons ainsi une complexité en $\Theta(N)$.

La méthode `workzone_update_time` contient une boucle pour modifier une liste de taille N : à chaque itération on accède à l'élément d'indice i dans la boucle (en temps constant sur Python) et on effectue deux opérations élémentaires pour mettre à jour les positions (en temps constant),d'où une complexité en $\Theta(N)$.

La méthode `delete_outside` effectue des opérations en temps constant sur une liste de taille N, d'où une complexité en $\Theta(N)$.

C.2 Classe `Simulation2`

La méthode `__init__` de la classe `Simulation2` effectue des comparaisons et des affectations qui sont en temps constant, nous avons donc une complexité en $\Theta(1)$.

La méthode `calcul` effectue des affectations en temps constant et initialise l'environnement `Workzone_square` (complexité $\Theta(N)$). Vient ensuite une boucle `while` exécutée $nb\_max\_collisions$ fois. A chaque itération on regarde le temps nécessaire pour une collision entre la particule suivie et les autres particules (qu'on détermine en $\Theta(1)$ puisque la solution est analytique) et on sélectionne ensuite le minimum qui demande de

comparer N collisions, ce qui donne une complexité totale en $\Theta(N)$.

On se place dans le cas où une collision est possible : on met à jour les positions des petites particules via `workzone_update_time` (complexité $\Theta(N)$), et on donne à la particule suivie une nouvelle orientation en temps constant. On supprime ensuite les particules en dehors de l'environnement avec `delete_outside` (complexité $\Theta(N)$).

A chaque itération on effectue donc des appels à des fonctions de complexité $\Theta(N)$ et des appels à des fonctions de complexité constante, on a donc une complexité en $\Theta(N)$ à chaque itération. On rentre dans la boucle `while` $nb\_max\_collision$ fois et on a précisé auparavant que chaque itération de la boucle a une complexité en $\Theta(N)$. Ainsi, nous avons une complexité totale de $\Theta(N \times nb\_max\_collision)$.

C.3 Récapitulatif du coût

Classe `Simulation2` : 1 exécution

Initialisation : $\Theta(1)$

Méthode `calcul` : 1 exécution

Initialisation de la grosse particule : $\Theta(1)$

Initialisation de l'environnement (`Workzone_square`) : $\Theta(N)$

Boucle `while` : $nb\_max\_collisions$ exécutions

Opérations en temps constant : $\Theta(1)$

Calcul de la première collision : $\Theta(N)$

Mise à jour de l'environnement (`workzone_update_time`) : $\Theta(N)$

Suppression et regénération de particules : $\Theta(N)$

Total : $\Theta(1) + \Theta(N) + \Theta(N) + \Theta(N) = \Theta(N)$

Total : $\Theta(1) + \Theta(N) + \Theta(nb\_max\_collisions \times N) = \Theta(nb\_max\_collisions \times N)$

Total : $\Theta(1) + \Theta(nb\_max\_collisions \times N) = \Theta(nb\_max\_collisions \times N)$