Correction sujet Centrale 2018
Simulation cinétique d'un gaz parfait

Marine Saint-Mézard

Introduction

On s'intéresse dans ce sujet à la simulation de chocs entre particules d'un gaz parfait. On peut se trouver dans un espace de dimension 1, 2, ou 3. Les bibliothèques numpy, random et math sont autorisées pour traiter le sujet.

Attention bien que python soit un langage non typé, il est ici clairement demandé de faire apparaître les types des arguments lors de la définition d'une fonction ainsi que le type de retour.

1 Initialisation

1.1 Placement en dimension 1

On cherche dans cette sous-partie à écrire une fonction qui permet de générer une situation initiale convenable (pas de problème de condition aux limites ni de chevauchement) dans le cas 1D. On cherche donc à placer N particules de rayon R sur un segment de longueur L.

1 La ligne 9 permet de choisir aléatoirement un flottant compris entre 0 et L (exclu).

2 Le paramètre c de la fonction possible est un tableau de dimension 1 qui correspond à l'abscisse du centre de la particule que l'on cherche à placer sur le segment.

3 La ligne 3 permet de s'assurer que l'on ne rencontre pas de problème de type condition aux limites. En effet on ne peut pas placer une particule de rayon R sur $[0, R]$ ou sur $[L - R, L]$. Si on place le centre de la particule dans un de ces 2 intervalles alors la sphère dépassera du segment.

4 Les lignes 4-5 permettent de s'assurer que les particules déjà placées ne se chevauchent pas avec la particule que l'on cherche à placer.

5 La fonction possible prend en argument une position, qui correspond aux coordonnées du centre de la sphère. Elle détermine si oui ou non il est possible de placer cette sphère à cette position sans rencontre de problème de chevauchement

avec une autre sphère ou de problème de conditions aux limites.

6 Pour éviter de tester les problèmes de conditions aux limites dans la fonction possible, on peut directement choisir la position p dans le segment adéquat c'est à dire $[R; L-R]$. Ce qui est possible avec l'instruction : `p = R +(L-2R) * np.random.rand(1)` Attention la fonction np.random.rand() ne prend que des entiers en argument.

7 On cherche à placer 4 sphères de rayon 0.5 cm sur un segment de longueur 5. On considère qu'on a déjà placé 3 particules aux positions $[1; 2.5; 4]$. Comme le montre le schéma de l'énoncé il n'est pas possible d'insérer une 4ème sphère sans chevauchement ou dépassement des bornes du segment. Il y a donc une boucle infinie, le programme ne sort jamais de la boucle while, l'appel à la fonction ne se termine jamais.

8 Dans le cas où $N \ll N_{max}$ l'appel à la fonction possible renverra *true* à chaque appel (la probabilité de chevauchement étant très faible), La boucle while est donc appelée N fois, dans chaque appel la fonction possible est éxécutée et cette dernière a une compléxité linéaire en le nombre d'éléments déjà placés sur le segment (à cause de la boucle for qui parcourt une liste de longueur (len(res)) qui tend vers N). On a donc une compléxité quadratique en $O(N^2)$.

9 Nous avons vu dans la question 7 que le programme actuel ne se termine pas forcément. On peut en effet se retrouver dans une configuration où le placement des premières particules empêche de venir placer les autres sans chevauchement. L'instruction while ne se termine jamais si il n'y a pas de solution de placement possible. Pour remédier à cela on propose de recommencer à zéro le placement des particules dès qu'une particule est rejetée par la fonction possible.

On a donc les instructions suivantes :

```
res = []
while len(res)<N:
    p = L * np.random.rand(1)
    if possible(p):
        res.append(p)
    else:
        res = []
return res
```

On notera que cette fonction ne se termine que s'il existe une solution, c'est à dire si le segment est suffisamment long pour que l'on puisse effectivement placer N particules de rayon R, ce que la fonction ne vérifie pas actuellement.

1.2 Optimisation du placement en dimension 1

Dans la première sous-partie nous avons créé une fonction qui permet de placer les particules sur un segment de longueur L, cependant elle n'est pas optimale et ne

vérifie pas qu'il est effectivement possible de placer N particules sur le segment, ce qui peut entrainer une boucle infinie. On cherche donc dans cette partie à optimiser la fonction placement1D.

10 La fonction peut prendre la forme suivante :

```
def placement1DRapide(N:int, R:float, L:float) -> [np.ndarray]:
    l = L - N*2*R
    if l<0:
        return [] #il n'y a pas de configuration possible
    #on place N particules de façon aléatoire sur [0;l]
    res = [l*np.random.rand(1) for i in range(N)]
    for i in range(N):
        for j in range(N): # Déplacement des particules à droite
            if res[j] > res[i]:
                res[j] = res[j]+2*R
            # Transformation en particule réelle
            res[i] = res[i]+R
    return res
```

Cette fonction n'est pas optimale puisqu'elle contient une double boucle for dont on peut se passer si on trie la liste avant de décaler les particules à droite.

On peut optimiser la fonction ainsi :

```
def placement1DRapide(N:int, R:float, L:float) -> [np.ndarray]:
    l = L - N*2*R
    if l<0:
        return [] #il n'y a pas de configuration possible
    #on place N particules de façon aléatoire sur [0;l]
    res = [l*np.random.rand(1) for i in range(N)]
    res.sort()
    for i in range(N):
        res[i] = res[i]+(2*i+1)*R
    return res
```

Attention à ne pas écrire `np.random.rand(l)` car la fonction random.rand prend en argument un entier qui correspond au nombre d'entiers à retourner. Voir la documentation à la fin du sujet pour plus de précisions.

11 La fonction *Placement1DRapide* appelle la fonction native *sort* qui a une compléxité en $N * log(N)$, puis effectue une boucle for sur une liste de longueur N. Nous avons donc une compléxité en $N * log(N)$ ce qui est meilleur que la compléxité en N^2 que nous avions dans la partie I.

1.3 Analyse statistique

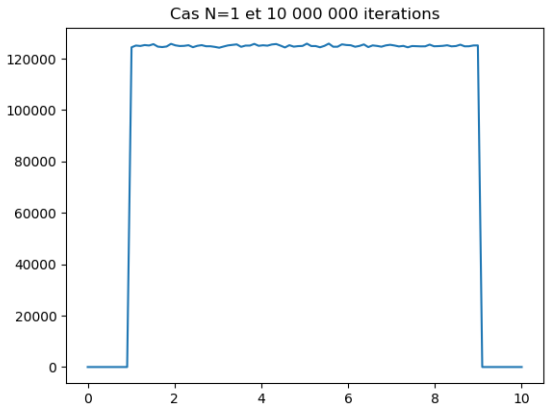
12 Le terme histogramme de la question est très mal choisi, en effet les particules peuvent occuper toutes les positions dans l'intervalle [0,L], on a donc une fonction continue et on représente donc ici une densité de probabilité et non un histogramme.

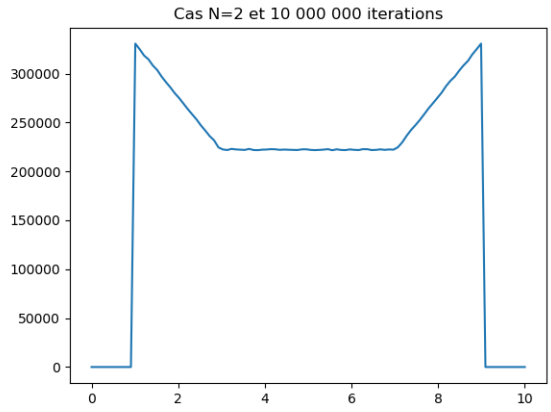
— Pour le cas $N = 1$, on place une particule de façon aléatoire sur le segment $[r; L - R]$ on obtient donc une répartition uniforme de probabilité $\frac{1}{L-2R}$, car $\mathcal{U} \sim ([1, 9])$.

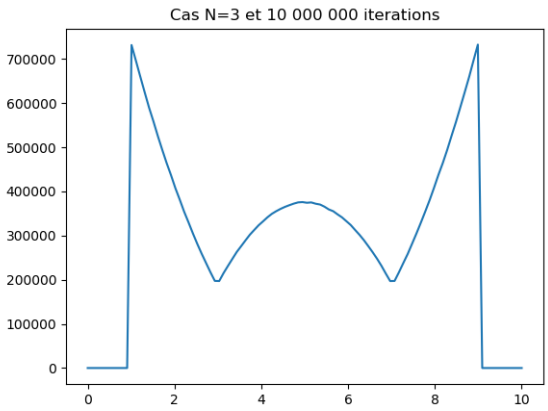
— Pour le cas $N = 2$, nommons X et Y les deux boules que l'on souhaite placer et x et y leur position observée. Plaçons X en premier, alors on se retrouve dans le cas de la question 1 et $\mathcal{U}([1, 9])$. On a alors $Y \sim \mathcal{U}([1, max(3, x-2)] \cup [min(x+2, 7), 9])$.

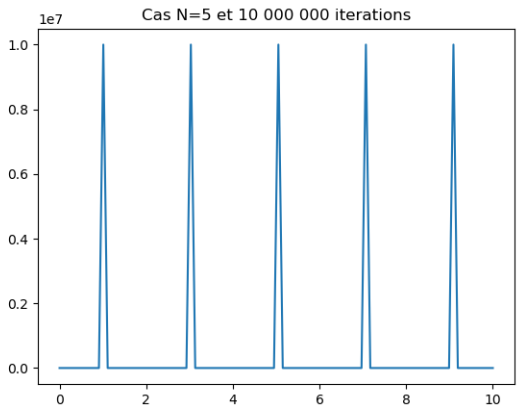
— Pour le cas $N = 5$ il n'y a qu'une seule configuration possible. En effet le rayon vaut 1, on veut placer 5 sphères donc les sphères occupent l'intégralité du segment.

On effectue 10 000 000 simulations et on affiche la fréquence obtenue pour chaque abscisse. Voici les résultats obtenus :









1.4 Dimension quelconque

13 On se place maitenant en dimension quelconque. La condition pour que 2 particules ne se chevauchent pas est donc maintenant que la distance (norme euclidienne) entre les centres des particules soit supérieure à $2 * R$. En dimension quelconque on peut ecrire la fonction suivante :

```
def placement(D:int, N:int, R:float, L:int)->[np.ndarray]
    def possible(c):
        for p in res:
            dist2 = 0
            for i in range(D):
                dist2 += (c[i]-p[i])*(c[i]-p[i])
            if np.sqrt(dist2)<2*R:
                return False
        return True
    res = []
    while len(res) < N:
        p = [(R + (L-2*R)*np.random.rand(1)) for i in range(D)]
        print(len(res))
        if possible(p):
            res.append(p)
        else:
            res = []
return res
```

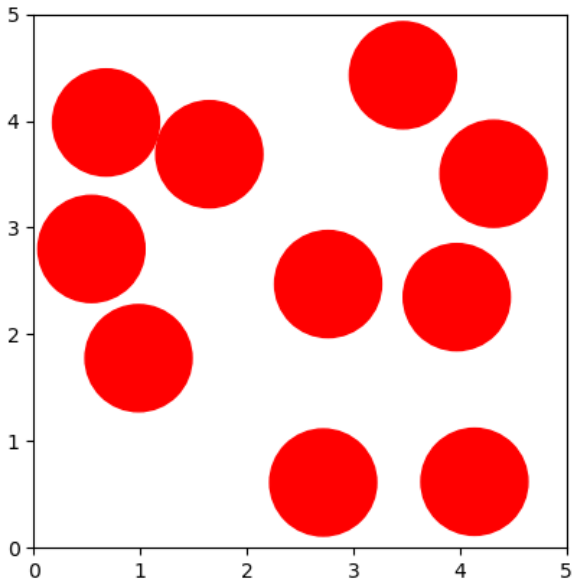


FIGURE 1 – Exemple avec D=2, N=10, L=5 et R=0.5

2 Mouvement des particules

2.1 Analyse physique

14 Entre 2 événements, par définiton il n'y a ni choc ni rebond, les particules ont une trajectoire rectiligne à vitesse constante.

15 Si on se place dans le cas où $m_1 = m_2$ les particules ont des vitesses après le choc égales à

$$\begin{cases} v_1^{'} = v_2 \\ v_2^{'} = v_1 \end{cases}$$

Les 2 particules échangent donc leur vitesse.

16 Dans le cas où $m_1 \ll m_2$, on obtient

$$\begin{cases} v_1^{'} = -v_1 + 2v_2 \\ v_2^{'} = v_2 \end{cases}$$

Cette situation correspond au cas d'un choc entre une particule de gaz et la paroi.

2.2 Evolution des particules

17 On cherche à calculer la position et la vitesse d'une particule dans un espace de dimension quelconque au bout d'un vol de t secondes sans choc ni rebond. S'il n'y a aucun choc ni rebond cela signifie qu'on ne rencontre pas la paroi et qu'il n'y a pas de modification de la vitesse, il suffit donc de mettre à jour la position.

```
def vol(p:[np.ndarray, np.ndarray],t:float) -> None:
    for d in range(len(p[0])):
        p[0][d] += p[1][d]*t
```

18 Dans le cas d'un rebond contre la paroi, on se trouve dans le cas $m_1 \ll m_2$ avec $v_2 = 0$. D'après la question 16, il suffit d'inverser le signe de la $d^{\text{ème}}$ composante de la vitesse.

```
def rebond(p:[np.ndarray, np.ndarray],d:int) -> None:
    p[1][d] = -p[1][d]
```

19 On s'interesse maintenant au choc entre deux particules. Il suffit alors d'échanger les vitesses des 2 particules d'après la question 15 puisque les particules sont identiques on a $m_1 = m_2$

```
def choc(p1:[np.ndarray, np.ndarray], p2:[np.ndarray, np.ndarray]) ->None:
    p1[1], p2[1] = p2[1], p1[1]
```

3 Inventaire des évènements

3.1 Prochains événements dans un espace à une dimension

20 On calcule le temps avant que la particule rebondisse sur une paroi. Attention à renvoyer un nombre positif.

```
def tr(p:[np.ndarray, np.ndarray] ,R: int , L: int) -> float:
    if p[1][0] == 0:
        return None
    if p[1][0] > 0:
        return (((L-R)-p[0][0])/p[1][0],0)
    else:
        return (abs(p[0][0]-R)/p[1][0],0)
```

21 On calcule le temps de collision entre deux particules si il existe un instant t au bout duquel elles se rencontrent. Pour cela on utilise les lois de physique pour déduire $x_1(t)$ et $x_2(t)$. On a donc dans le cas d'une particule ponctuelle :

$$\begin{cases} x_1(t) = x_1^0 + v_1 * t \\ x_2(t) = x_2^0 + v_2 * t \end{cases}$$

On résout ensuite $x_1(t) = x_2(t)$.
On en déduit $t = \frac{x_1 - x_2}{v_2 - v_1}$.
Il reste a prendre en compte le fait que la particule à un rayon R.

```
def tc(p1: [np.ndarray, np.ndarray], p2: [np.ndarray, np.ndarray], R: int) -> int o
    x1 = p1[0][0]
    x2 = p2[0][0]
    v1 = p1[1][0]
    v2 = p2[1][0]

    if abs(x1 - x2) <= 2*R:
        return 0
    if v1 == v2:
        return None

    x1 = x1 + R*((x2 > x1)*2-1)
    x2 = x2 + R*((x2 < x1)*2-1)
    t = -(x2-x1)/(v2-v1)
    if t < 0:
        return None
    return t
```

3.2 Catalogue d'événements

22

```
def ajoutEv(catalogue,e):
    for i in range(len(catalogue)):
        if catalogue[i][1] <= e[1] :
            catalogue.insert(i,e)
            return catalogue
    catalogue.append(e)
    return catalogue
```

23

```
def ajout1p(catalogue, i, R, L, particules):
    temps_paroi,d = tr(particules[i], R, L)
    catalogue = ajoutEv(catalogue, [True, temps_paroi, i, None, d])
    for j in range(len(particules)):
        if i != j:
            temps_collision = tc(particules[i], particules[j], R)
            if temps_collision >0 :
                catalogue = ajoutEv(catalogue, [True, temps_collision, i, j,None])
    return catalogue
```

24

```
def initCat(particules, R, L):
    catalogue = []
    for i in range(len(particules)):
        ajout1p(catalogue, i, R, L, particules)
    return catalogue
```

25 La fonction initCat ajoute plusieurs fois les mêmes éléments puisqu'il considère les chocs entre les particules i et j mais aussi le choc entre les particules j et i.

26 On cherche à déterminer la compléxité de la fonction initCat. Cette fonction comprend :

- un appel à ajout1p
- celle-ci appelle une fois la fonction *tr* et N fois la fonction *tc* et N fois la fonction ajoutEv
- tc et tr sont en O(1) mais ajoutEv est en $O(n)$

On a donc une complexité globale en $O(n^3)$

27 Pour optimiser la fonction initCat, il y a plusieurs choses à faire :

- il ne faut pas ajouter 2 fois chaque collision.
- il faut retravailler la fonction ajout1p, pour cela on crée une liste désordonnée comprenant tous les événements à rajouter liés à la particule i, puis on ajoute cette liste dans le catalogue existant. On ne trie ainsi qu'une fois la liste lors de l'ajout au catalogue. On peut utiliser un tri fusion entre la liste à ajouter et le catalogue existant.

4 Simulation

28 Si toutes les particules n'ont pas une vitesse nulle, il existe un indice i tel que la particule i ait une composante de vitesse non nulle. Cette particule rentrera soit en contact avec une paroi soit avec une autre particule au bout d'un certain temps t.

29

```
def etape(particules, e):
    #vol de la particule jusqu'à ce que se produise l'événement
    for i in range(len(particules)):
        vol(particules[i], e[1])
    # cas d'un choc
    if e[3]:
        choc(particules[e[2]],particules[e[3]])
    # cas d'un rebond
    else:
        rebond(particules[e[2]], e[4])
```

30

```
def majCat(catalogue, particules, e, R, L):
    temps = e[1]
    for event in catalogue:
        event[1] = event[1] - temps
        if event[3] == e[2] or event[3] == e[3]:
            event[0] = False
    ajout1p(catalogue, e[2], R, L, particules)
    if e[3]:
        ajout1p(catalogue, e[3], R, L, particules)
```

31

```
def simulation(bdd, d, N, R, L, T):
    #definition du probleme
    particules = placement(d, N, R, L)
    catalogue = initCat(particules, R, L)

    temps = 0
    op = 0
    while temps < T:
        #on regarde le dernier evenement
        event = catalogue[-1]
        etape(particules, event)
        #on met à jour les compteurs
        temps += event[1]
        op += 1
        #on supprime le dernier event de la liste catalogue
```

```
            catalogue.pop()
            #on met à jour le catalogue
            majCat(catalogue, particules, event, R,L)
            #on enregistre l'opération en bdd
            enregistrer(bdd, temps, event, particules)
    return op
```

32 Les événements présents en double comme évoqué à la question 25 ne perturbent pas l'évolution de la simulation car leur premier paramètre devient False lors de l'appel à la fonction *majCat*.

5 Exploitation des résultats

Cette partie permet de tester les connaissances en SQL, il est intéressant de noter qu'elle est parfaitement indépendante des parties précédentes et peut donc être traitée même si les questions précédentes ne l'ont pas été. Le candidat devrait prendre le temps de le faire, de plus elles sont d'une difficulté tout à fait raisonnable.

33 On cherche à compter le nombre de simulations effectuées pour chaque dimension de l'espace.

```
SELECT SI_DIM, count(SI_NUM)
FROM SIMULATION
GROUP BY SI_DIM
```

34 On cherche pour chaque simulation, le nombre de rebonds enregistrés et la vitesse moyenne des particules qui frappent une paroi.

```
SELECT si_num, count(*), avg(re_vit)
FROM rebond AS rb
GROUP BY rb.si_num;
```

35 On cherche pour une simulation n donnée, calcule, pour chaque paroi, la variation de quantité de mouvement due aux chocs des particules sur cette paroi tout au long de la simulation.

```
SELECT REBOND.RE_DIR, SUM(2 * PARTICULE.PA_M * REBOND.RE_VP)
FROM REBOND, PARTICULE
WHERE PARTICULE.PA_NUM = REBOND.PA_NUM
AND SI_NUM = 1
GROUP BY RE_DIR
```