Centrale 2018 - Corrigé

Simulation de la cinétique d'un gaz parfait

I Initialisation

I.A - Placement en dimension 1

1. `np.random.rand(1)` renvoie un vecteur à une seule coordonnée.
   La seule valeur de ce vecteur est un nombre aléatoire de $[0, 1[$ (selon une distribution uniforme).
   En multipliant cette valeur par L, on obtient une valeur aléatoire de $[0, L[$ (selon une distribution uniforme).
2. Le paramètre `c` est un tableau à un seul élément, qui contient l'abscisse à tester.
3. La ligne 3 permet de vérifier si l'abscisse choisie n'est pas trop près des bords du récipient.
4. Les lignes 4 et 5 testent, pour chaque particule déjà présente dans le récipient, si l'abscisse de la nouvelle particule ne va pas créer un "conflit" (deux particules qui se "chevauchent")
5. La fonction `possible` teste si l'abscisse contenue dans `c` peut être choisie comme position d'une nouvelle particule à rajouter dans le récipient.
6. Pour éviter d'avoir à faire le test en ligne 3, on peut choisir une position aléatoirement dans l'intervalle $[R, L-R]$.
   Pour cela, on remplace la ligne 9 par :
   `p = R + (L-2R) * np.random.rand(1).`
7. Il faut un espace libre de longueur 1 pour pouvoir placer une nouvelle particule.
   Or, la première particule déjà posée occupe l'intervalle $[0.5, 1.5]$, la deuxième occupe $[2, 3]$ et la troisième occupe $[3.5, 4.5]$.
   Les seuls espaces libres sont donc $[0, 0.5]$, $[1.5, 2]$, $[3, 3.5]$ et $[4.5, 5]$.
   Tous ces espaces sont de longueur 0.5, et il n'y aura donc pas de place suffisant pour insérer la quatrième particule.
   De fait, la fonction `possible` va renvoyer `False` quelle que soit la valeur de `p` choisie.
   On n'ajoutera donc jamais de nouvelle valeur à la liste `res` (ligne 10), qui restera donc de longueur 3, ce qui empêchera de sortir de la boucle `while` initiée à la ligne 8.
   La fonction `placement1D` ne terminera donc jamais !
8. Si $N \ll N_{max}$, on peut considérer que la fonction `possible` renverra toujours `True` (avec la modification proposée en question 6
   Dans ce cas, la boucle `while` sera exécutée exactement `N` fois.
   A chaque passage dans cette boucle,
   - on choisit un nombre aléatoirement et on le multiplie par $(L-2R)$ et on ajoute R (version modifiée), opérations en $O(1)$
   - On teste : `possible(p).` Cette opération est en $O(k)$ (où k est le nombre de particules déjà présentes)
   - On ajoute enfin 1 élément à une liste, opération en $O(1)$

   La complexité globale est donc en $O\left(\sum_{k=0}^{N-1} k\right) = O(n^2)$.
9. Les lignes 7 à 11 de la fonction `placement1D` deviennent :

```
res = []
while len(res) < N:
   p = R+(L+2R)*np.random.rand(1)
   if possible(p):
      res.append(p)
   else:
      res = []
```

I.B - Optimisation du placement en dimension 1

10. Chaque particule occupe un espace de longueur $2R$.
    L'espace libre laissé par les N particules est donc $\ell = L - 2NR$.
    On écrit donc :

```
def placement1Drapide(N:int, R:float, L:float) -> [np.ndarray]:
   l = L - 2 * N * R
   res = l * np.random.rand(N) # choix des N emplacements des particules virtuelles
   res.sort() # On ordonne les abscisses choisies
   for i in range(N) :
      for j in range(i,N)
         res[j] = res[j] + 2 * R
   return res
```

L'utilisation de la fonction `sort`, en $O(N\ln(N))$ permet de simplifier l'écriture de la suite, et est tout à fait raisonnable car le reste de l'algorithme est en $O(N^2)$.

11. • La fonction `np.random.rand(N)` crée le tableau en $O(N)$
La fonction `sort` effectue le tri en $O(N\ln(N))$.
La double boucle effectue $\sum_{i=1}^{N}(i-1) = \frac{(N-1)N}{2}$ opérations
La complexité globale est donc en $O(N^2)$.
• On a la même complexité affichée que la fonction `placement1D`.
Cependant, pour la fonction `placement1D`, il s'agit d'une complexité dans le meilleur des cas (où on a n'a pas besoin de recommencer un ou des tirages). La complexité moyenne sera donc en général sensiblement supérieure, surtout quand N se rapproche de N_{max}.
Dans le cas de la fonction `placement1Drapide`, la complexité sera toujours la même.
On peut donc espérer en moyenne un gain plus ou moins important en utilisant la fonction `placement1Drapide` :
– ce gain sera faible si $N \ll N_{max}$
– ce gain sera très important si $N \simeq N_{max}$

I.C - Analyse statistique

12. Cette question me semble très discutable vis-à-vis du programme...
On considérera que l'on construira ces histogrammes sur les valeurs $[1, 9]$.
• Si $N = 1$, quelle que soit le nombre de classes choisies, le diagramme aura des rectangles de hauteur sensiblement égale, car le choix de la seule position suit une distribution uniforme.
• Si $N = 2$, on aura aussi un diagramme avec des rectangles de hauteur sensiblement égale.
• Si $N = 5$, l'espace libre est nul, car $L - 2NR = 0$.
Les particules auront donc nécessairement les positions $[1, 3, 5, 7, 9]$. Suivant le nombre de classes choisies, l'histogramme variera, mais il n'y a rien d'aléatoire ici !

I.D - Dimension quelconque

13. Pour écrire cette fonction, on écrit d'abord une fonction `dist`, qui calcule la distance entre 2 points de $\mathbb{R}^n$ (où n est donné par la taille commune des deux tableaux contenant les coordonnées des points).

```
def dist(p1,p2):
    n = len(p1)
    s = 0
    for k in range(n):
        s = s + (p1[k]-p2[k]) **2
    return math.sqrt(s)
```

On peut alors écrire la fonction demandée :

```
def placement(D:int, N:int, R:float, L:float) -> [np.ndarray]:
    def possible(c:np.ndarray) -> bool:
        for p in res:
            if dist(c,p) < R: return False
        return True

    res = []
    while len(res) < N:
        p = R + (L-2R) * np.random.rand(D)
        if possible(p): res.append(p)
    return res
```

II Mouvement des particules

II.A - Analyse physique

14. Entre deux événements, une particule évolue en ligne droite (car on néglige les forces d'interaction et autre), dirigée par le vecteur vitesse de la particule.

15. Si $m_1 = m_2$, ces formules deviennent : $\begin{cases} \vec{v_1}' = \vec{v_2} \\ \vec{v_2}' = \vec{v_1} \end{cases}$.
Cela revient donc à échanger les vecteurs vitesse des deux particules.
Peut-être que le concepteur attendait une justification physique ?

16. Si $m_1 \ll m_2$, alors $m_1 + m_2 \simeq m_2$ et $m_1 - m_2 \simeq -m_2$.
Ces formules deviennent $\begin{cases} \vec{v_1}' = -\vec{v_1} + 2\vec{v_2} \\ \vec{v_2}' = \vec{v_2} \end{cases}$

Dans le problème qui nous occupe, cette situation correspond aux rebonds sur une paroi du récipient, qui a une masse très supérieure à celle des particules.

II.B - Evolution des particules

17. On a $M(t) = M(0) + t\vec{v}$, donc on peut écrire :

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

18. En suivant les instructions (seule une coordonnée de la vitesse est modifiée), on écrit :

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

19. Comme on a supposé les particules de même masse, on applique le résultat de la question 15 et on écrit :

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

Remarque. Ainsi écrite, la fonction `choc` est opérationnelle quelle que soit la dimension.

III Inventaire des évènements

III.A - Prochains événements dans un espace à une dimension

20. En notant $M(t) = (x_1(t), \dots, x_n(t))$ la position de la particule à l'instant t, il s'agit, pour chaque coordonnée, de calculer $t_i = \sup\{t \geq 0 : x_i(t) \in [R, L-R]\}$, puis de prendre le minimum des t_i.
Or, $x_i(t) = x_i(0) + tv_i$, donc si $v_i \neq 0$,

$$x_i(t) = L - R \Leftrightarrow t = \frac{L - R - x_i(0)}{v_i} \quad \text{et} \quad x_i(t) = R \Leftrightarrow t = \frac{R - x_i(0)}{v_i},$$

donc $t_i = \max\left(\frac{L - R - x_i(0)}{v_i}, \frac{R - x_i(0)}{v_i}\right)$, (car comme $x_i(0) \in [R, L-R]$, une des deux valeurs est négative, et c'est la positive qui nous intéresse).
Si $v_i = 0$, la particule "ne bouge pas dans la direction i", donc elle n'atteindra jamais un bord dans cette direction... et t_i n'a alors aucun intérêt ($t_i = +\infty$).
On écrit donc :

```
def tr(p:[np.ndarray, np.ndarray], R:float, L:float) -> None or (float, int):
   def ti(pi,L,R) :
      if pi[1] != 0:
         t1 = (L-R-pi[0])/pi[1]
         t2 = (R-pi[0])/pi[1]
         return max([t1,t2])
      return -1
   n = len(p[0])
   l=[]
   for i in range(n):
      t=ti([p[0][i],p[1][i]],L,R)
      if t != -1 : l.append(t)
   return min(l)
```

La fonction `ti` introduite prend en entrée une liste composée de deux éléments : x_i et v_i, ainsi que les valeurs R et L, et calcule le t_i comme indiqué en préambule de la réponse.
Si $v_i = 0$, cette fonction renvoie la valeur -1.

Remarque. Et je viens de me rendre compte que j'ai répondu en dimension quelconque, alors que l'énoncé demande en dimension 1... et ne demande pas que le temps en retour... Bref, lisez l'énoncé, ça aide !

En dimension 1, c'est beaucoup plus simple. On écrit :

```
def tr(p:[np.ndarray, np.ndarray], R:float, L:float) -> None or (float, int):
   if p[1] != 0:
      t1 = (L-R-p[0])/(p[1])
      t2 = (R-p[0])/p[1]
      return (max([t1,t2]),0)
```

21. On est toujours en dimension 1.
On a $x_1(t) = x_1(0) + tv_1$ et $x_2(t) = x_2(0) + tv_2$, donc $x_1(t) - x_2(t) = x_1(0) - x_2(0) + t(v_1 - v_2)$.
Les particules entrent en contact au moment où $|x_1(t) - x_2(t)| = 2R$ (s'il existe).
• La situation initiale est supposée sans chevauchement, donc $|x_1(0) - x_2(0)| \geq 2R$.
• Si $v_1 - v_2 = 0$, les particules n'entreront jamais en contact.
• Si $v_1 - v_2 > 0$, alors,
si $x_1(0) - x_2(0) \geq 2R$, alors l'écart entre les particules croit au cours du temps, donc elle ne se rencontreront jamais ;
et si $x_1(0) - x_2(0) \leq -2R$, les particules se rencontreront à l'instant t tel que

$$x_1(t) - x_2(t) = -2R \Leftrightarrow t = \frac{-2R - x_1(0) + x_2(0)}{v_1 - v_2}$$

• Enfin, si $v_1 - v_2 < 0$, alors,
si $x_1(0) - x_2(0) \leq -2R$, alors l'écart entre les particules croit au cours du temps, donc elle ne se rencontreront jamais ;
et, si $x_1(0) - x_2(0) \geq 2R$, les particules se rencontreront à l'instant t tel que

$$x_1(t) - x_2(t) = 2R \Leftrightarrow t = \frac{2R - x_1(0) + x_2(0)}{v_1 - v_2}.$$

Il ne reste donc plus qu'à écrire la fonction en tenant compte de tous ces cas :

```
def tc(p1:[np.ndarray, np.ndarray], p2:[np.ndarray, np.ndarray], R:float) -> None or float:
   x1 = p1[0]
   x2 = p2[0]
   v1 = p1[1]
   v2 = p2[1]
   if v1-v2 > 0:
      if x1-x2 <= -2*R:
         return (-2*R-x1+x2)/(v1-v2)
   if v1-v2 < 0:
      if x1-x2 >= 2R:
         return (2R-x1+x2)/(v1-v2)
```

III.B - Catalogue d'événements

22.
```
def ajoutEv(catalogue, e) :
   n = len(catalogue)
   i=0
   while i<n & catalogue[i][1] > e[1] : # calcul de l'indice où il faut ajouter e
      i=i+1
   catalogue.insert(i,e)
```

23.
```
def ajout1p(catalogue, i, R, L, particules):
   N=len(particules)
   tparoi=tr(particules[i],R,L)
   if tparoi != None:
      t,d = tparoi
      ajoutEv(catalogue, [True,t,i,None,d])
   for j in range(N):
      if j != i:
         tchoc = tc(particule[i],particule[j])
         if tchoc != None
            ajoutEv(catalogue,[True,tchoc,i,j,None])
```

24.
```
def initCat(particules, R, L) :
   N=len(particules)
   catalogue = []
   for i in range(N):
      ajout1p(catalogue, i , R, L, particules)
   return catalogue
```

25. On a calculé à quel moment la particule i peut rencontrer la particule j, mais on a aussi calculé à quel instant la particule j peut rencontrer la particule i.
Si elles peuvent effectivement se rencontrer, on aura alors inscrit deux fois leur rencontre dans le catalogue des événements.

26. `initCat` lance N fois la fonction `ajout1p`. Sa complexité est donc N fois celle de `ajout1p`.
`ajout1p` lance :
– une fois `tr` qui a une compexité en $O(1)$
– $N-1$ fois `tc` qui a une complexité en $O(1)$
– au maximum N fois `ajoutEv` qui a une complexité maximale en $O($`len(catalogue)`$)$ (en cas d'ajout à la fin)
Dans le pire des cas, où on ajoute chaque fois (donc N^2 fois) un événement en dernière position de `catalogue` (qui grossit au cours du temps), ces ajouts auront une complexité totale en :

$$\sum_{k=0}^{N^2-1} k = O(N^4).$$

Comme le reste du travail se fait en $O(N^2)$, on a une complexité globale en $O(N^4)$ dans le pire des cas.

27. Le plus gros du coût vient de la fonction `ajoutEv`. Une façon de gagner beaucoup en complexité consisterait à
– mettre dans un premier temps les éléments dans n'importe quel ordre dans `catalogue`, en utilisant un `.append`. Ces ajouts se font en temps constant ($O(1)$), et la complexité globale pour avoir un catalogue en désordre est alors en $O(N^2)$.
– puis, après avoir ajouté tous les éléments dans le désordre, on effectue un tri fusion sur le `catalogue` pour obtenir un `catalogue` trié.
Comme le `catalogue` contient au maximum N^2 éléments, le tri fusion aura une complexité en $O(N^2\ln(N^2)) = O(N^2\ln(N))$.
La fonction `initCat` ainsi écrite aura une complexité globale en $O(N^2\ln(N))$, ce qui est bien meilleur que la complexité initiale en $O(N^4)$.

IV Simulation

28. • Comme on a supposé la conservation de l'énergie globale (qui n'est que cinétique ici), si l'energie au départ est non nulle, elle ne sera jamais nulle, ce qui assure qu'il y aura toujours au moins une particule en mouvement. Or, une particule en mouvement rectiligne finira toujours par rencontrer une paroi, ce qui assure le caractère non vide du `catalogue`.
• Par contre, si l'energie est nulle au départ (toutes les particules sont immobiles), elle restera nulle au cours du temps, et donc il n'y aura jamais ni collision, ni rebond, et donc le `catalogue` restera vide.

29.
```
def etape(particules, e):
    N = len(particules)
    t = e[1] # t est le temps avant le premier événement
    for k in range(N): # on calcule d'abord la nouvelle position des particules
        vol(particules[k],t)
    i = e[2] # i est le numéro de la première particule de l'événement
    if e[3] == None: # On a alors un rebond sur la paroi
        rebond(particules[i],e[4])
    else : # on a un choc entre les particules i et e[3] de même poids
        choc(particules[i], particules[e[3]])
```

30.
```
def majCat(catalogue, particules, e, R, L) :
    i = e[2]
    for x in catalogue:   # On enlève les événements dépendant de i
        if x[2]==i or x[3]==i:
            catalogue.remove(x)
    if e[3] != None :
        j=e[3]
        for x in catalogue: # on enlève les événements dépendants de la particule j
            if x[2]==j or x[3]==j:
                catalogue.remove(x)
    t=e[1]
    for x in catalogue : # On met à jour le temps
        x[1]=x[1]-t
    ajout1p(catalogue, i, R, L, particules) #on ajoute les événements avec la nouvelle trajectoire de i
    if e[3] != None :
        j=e[3]
        ajout1p(catalogue, j, R, L, particules) # pareil pour j, si cette particule existe
```

Remarque. Ainsi écrite, on a une fonction `majCat` en $O(N^3)$, car on ajoute $2N$ fois un élément dans `catalogue`, qui est de longueur $\simeq N^2$.
On aurait donc encore intérêt à mettre les nouveaux événements en vrac à la fin, puis à trier le `catalogue`...

31.
```
def simulation(bdd, d:int, N:int, R:float, L:float, T:float) -> int:
```

```
particules = situationInitiale(d, N, R, L)
t=0
n=0
catalogue = initCat(particules, R, L)
while t < T:
   e = catalogue.pop()
   etape(particules,e)
   majCat(catalogue, particules, e)
   t=t+e[1]
   enregistrer(bdd, t, e, particules)
   n=n+1
return n
```

32. Ces doublons disparaissent lors de la mise à jour, car, en cas de choc entre deux particules i et j, on enlève tous les événements dépendant de i ou de j, donc, en particulier, le doublon non traité.

33. Changer le repère temporel à chaque événement
    - oblige à effectuer $O(N^2)$ soustractions pour mettre à jour le temps dans le `catalogue`, et ce à chaque nouvel événement
    - évite une addition à chaque appel des fonctions `tr` et ou `tc` après un événement.
      Or, ces fonctions sont appelées $2N$ fois au maximum par événement, donc on évite globalement $2N$ additions.
    - oblige à faire une addition pour chaque événement, afin de connaitre le temps global (`t = t + e[1]`)

    On aurait donc été gagnant à garder un temps global dans les événements, que ce soit en complexité ou en source d'erreurs d'arrondi, car, dans la situation actuelle, les Δt à ajouter risquent d'être petits, et donc source d'erreurs d'arrondi !

V. Exploitation des résultats

31.
```
SELECT COMPT(SI_NUM), SI_DIM FROM SIMULATION
   GROUP BY SI_DIM
```

32.
```
SELECT SI_NUM, COMPT(RE_NUM), AVG(RE_VIT) FROM REBOND
   GROUP_BY SI_NUM
```

33.
```
SELECT RE_DIR, SUM(2*PA_M * RE_VP) FROM REBOND
   JOIN PARTICULE ON REBOND.PA_NUM = PARTICULE.PA_NUM
   WHERE SI_NUM = n
   GROUP BY RE_DIR
```