Centrale 2018
Simulation de la cinétique d'un gaz parfait
Un corrigé

1 Initialisation

1.A Placement en dimension 1

Q.1. En ligne 9, on affecte à `p` une valeur flottante aléatoire de $[0, L[$.

Q.2. Le paramètre `c` correspond à la position d'une éventuelle nouvelle particule.

Q.3. En ligne 3, on vérifie que la position est compatible avec les paramètres du récipient, c'est à dire que le centre de la particule est dans $[R, L-R[$.

Q.4. Dans la boucle des lignes 4 et 5, on vérifie que l'éventuelle nouvelle particule n'empiète pas sur celles déjà présentes.

Q.5. `possible(c)` indique si `c` est une position admissible pour une nouvelle particule, compte-tenu de celles déjà enregistrées.

Q.6. On peut s'arranger pour que la variable `p` soit dans $[R, L-R[$ en écrivant

```
p=R+np.random.rand(1)*(L-2*R)
```

Q.7. Avec les trois premières particules placées, il ne reste plus d'espace de diamètre 1 disponible. La boucle conditionnelle de la ligne 8 ne se terminera donc pas.

Q.8. Si N est très petit devant $N_{\max}$, il est très peu probable que les particules choisies soient incompatibles. Il est aussi très peu probable que l'on choisisse aléatoirement un élément `p` dans $[0, R[$ ou $[L-R, L[$. Il est donc raisonnable de faire le calcul quand la boucle conditionnelle sera effectuée N fois.
Lors du tour numéro k (en commençant la numérotation à 1), `res` contient déjà $k-1$ particules et l'appel à `possible` a un coût $O(k)$.
La somme des coûts est donc $O(N^2)$ et la complexité temporelle est quadratique en fonction du nombre des particules.

Q.9. Le nouveau code suggéré est

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

On vide la liste des particules dès qu'une particule est rejétée.

1.B Optimisation du placement en dimension 1

Q.10. Les étapes 1 et 2 ne posent pas de problème.
Pour la troisième, on doit faire "gonfler" chaque particule virtuelle à son tour, ce qui revient à décaler son centre de R vers la droite. Il faut aussi décaler de $2R$ (espace pris par la nouvelle particule réelle) vers la droite toutes les particules (réelles ou virtuelles) qui sont à droite de la particule virtuelle traitée.

```
def placement1Drapide(N,R,L):
    #Etape 1
    l=L-2*N*R
    #Etape 2
    res=[]
```

```
    for i in range(N):
        res.append(l*np.random.rand(1))
    #Etape 3
    for i in range(N):
        pos=res[i][0]
        for j in range(N):
            if res[j][0]>=pos and j!=i:res[j][0]+=2*R
        res[i][0]+=R
    return res
```

On propose, ci-dessous, une amélioration de cette fonction.

Q.11. Les deux premières étapes se font en temps linéaire en fonction de N.
Pour la troisième étape, les deux boucles imbriquées induisent un nombre d'opérations de l'ordre de N^2.
On a finalement encore une complexité $O(N^2)$. Cette version n'est pas meilleure que ce que l'on a vu en question 8 mais on n'a plus besoin d'hypothèse supplémentaire.

Pour améliorer la fonction, on peut commencer par trier la liste initiale (des particules virtuelles). On sait alors quelles seront les particules à droite de la particule traitée (ce sont celles de plus grand numéro).

```
def placement1Drapide(N,R,L):
    #Etape 1
    l=L-2*N*R
    #Etape 2
    res=[]
    for i in range(N):
        res.append(l*np.random.rand(1))
    #tri des particules
    res.sort()
    #Etape 3
    for i in range(N):
        res[i][0]+=R
        for j in range(i+1,N):
            res[j][0]+=2*R
    return res
```

Cela n'améliore cependant pas la complexité asymptotique (la double boucle induit encore $O(N^2)$ opérations).
Pour faire vraiment mieux, il faut supprimer la double boucle. Pour cela, je gère simplement une variable d me permettant de mémoriser la somme cumulée des déplacements. Quand on transforme ou déplace une particule, il suffit de tenir compte de cette somme cumulée.

```
def placement1Drapide2(N,R,L):
    #Etape 1
    l=L-2*N*R
    #Etape 2
    res=[]
    for i in range(N):
        res.append(l*np.random.rand(1))
    #tri des particules
    res.sort()
    #Etape 3
    d=0
    for i in range(N):
```

```
        res[i][0]+=R+d
        d=d+2*R
    return res
```

La complexité est alors $O(N)$ pour les boucles plus le coût du tri en $O(N\log(N))$. On atteint ainsi une complexité quasi-linéaire $O(N\log(N))$.

1.C Analyse statistique

Q.12. Dans les trois cas, $R=1$ et une particule prend un espace de 2. L'histogramme à dessiner n'est pas très clair (puisque les particules peuvent prendre une infinité de positions). Je propose ainsi de tenter de dessiner un graphique où on regroupe sur l'abscisse i la proportion de particules dont le centre est situé dans l'intervalle $[i,i+1[$.

- Si $N=1$, l'espace libre est $\ell=10-2=8$. On place une unique particule virtuelle en $p=[0,8[$. Le centre de la particule réelle est alors $p+1$. On obtient alors histogramme correspondant à une loi uniforme sur $\{1,\dots,8\}$.
- Si $N=2$, l'espace libre est $\ell=10-4=6$. Notre histogramme comportera encore huits barres. On choisit une première particule fictive dont le cetre est dans $[0,6[$; notons i l'unique entier de $\{0,\dots,5\}$ tel que ce centre soit dans $[i,i+1[$. Le choix de la deuxième particule fictive donne de même un entier j. Ci-dessous, j'indique le couple correspondant aux entiers de $\{1,\dots,8\}$ associés pour un couple (i,j) donné aux particules réelles.

	0	1	2	3	4	5
0	1,3	1,4	1,5	1,6	1,7	1,8
1	1,4	2,4	2,5	2,6	2,7	2,8
2	1,5	2,5	3,5	3,6	3,7	3,8
3	1,6	2,6	3,6	4,6	4,7	4,8
4	1,7	2,7	3,7	4,7	5,7	5,8
5	1,8	2,8	3,8	4,8	5,8	6,8

  Chacun des intervelles associé à i ou j a la même probabilité, c'est à dire que toutes les cases du tableau ci-dessus sont équiprobable. Sur 72 particues choisies, en moyenne, il y en aura $11,9,8,8,8,8,8,9,11$ dans les intervalles associés à $1,2,3,4,5,6,7,8$ (on a compté de fois où un des entiers k apparaît dans le tableau). Ce n'est pas tout à fait uniforme puisque les "bords" sont un peu plus représentés.
- Si $N=5$ alors il n'y a qu'une façon de placer les particules : avec des centres en positions $1,3,5,7,9$. Il n'y a plus que 5 barres de même hauteur.

1.D. Dimension quelconque

Q.13. On écrit une fonction `norme` qui donne la norme d'un vecteur et permettra d'obtenir la distance entre deux particules.

```
def norme(t):
    res=0
    for i in range(t.ndim):
        res=res+t[i]**2
    return res**(1/2)
```

Il reste à adapter la fonction `placement` en créant des particules de bonnes dimensions et en utilisant notre norme.

```
def placement(D,N,R,L):
    def possible(c):
        for p in res:
            if norme(c-p) < 2*R: return False
```

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

2 Mouvement des particules

2.A Analyse physique

Q.14. Entre deux événements, une particule se déplace en ligne droite.

Q.15. Si $m_1 = m_2$, les formules deviennent

$$\vec{v_1}' = \vec{v_2} \quad \text{et} \quad \vec{v_2}' = \vec{v_1}$$

Les particules échangent donc leurs vitesses.

Q.16. Si $m_1 \ll m_2$ on a

$$\vec{v_1}' \approx -\vec{v_1} + 2\vec{v_2} \quad \text{et} \quad \vec{v_2}' \approx \vec{v_2}$$

Il s'agit ici du cas d'un rebond sur la paroi : la vitesse de la particule est changée en son opposé.

2.B Evolution des particules

Q.17.
```
def vol(p,t):
    p[0]=p[0]+t*p[1]
```

Q.18.
```
def rebond(p,d):
    p[1][d]=-p[1][d]
```

Q.19. Les vitesses sont échangées (question 15).

```
def choc(p1,p2):
    p1[1],p2[1]=p2[1],p1[1]
```

3 Inventaire des événements

3.A Prochains événements dans un espace à une dimension

Q.20. Selon le signe de l'unique composante de la vitesse, on va heurter la partie droite (composante positive) ou gauche (composante négative).
Dans le premier cas, il faut calculer le temps où la position du centre plus R vaut L. L'équation est

$$p[0][0] + R + t \times p[1][0] = L$$

Dans le second cas, il faut calculer le temps où la position du centre moins R vaut 0. L'équation est

$$p[0][0] - R + t \times p[1][0] = 0$$

Ces équations se résolvent simplement.

```
def tr(p,R,L):
    if p[1][0]>0:
        return ((L-p[0][0]-R)/p[1][0],0)
    elif p[1][0]<0:
        return ((R-p[0][0])/p[1][0],0)
    else:
        return None
```

Q.21. On cherche s'il existe un $t > 0$ en lequel les centres des particules seront à distance $2R$. L'équation est ainsi

$$2R = |(p_1[0][0] + tp_1[1][0]) - (p_2[0][0] + tp_2[1][0])|$$

ou encore

$$(p_1[1][0] - p_2[1][0])t = \pm 2R - p_1[0][0] + p_2[0][0]$$

Si $p_1[1][0] - p_2[1][0] = 0$, il n'y a pas de solution. Sinon, on a deux temps t_1 et t_2 possibles. On renvoie celui qui est positif s'il y en a un (physiquement, on ne peut avoir deux solutions positives).

```
def tc(p1,p2,R):
    if p1[1][0]==p2[1][0]:
        return None
    else:
        t1=(2*R-p1[0][0]+p2[0][0])/(p1[1][0]-p2[1][0])
        t2=(-2*R-p1[0][0]+p2[0][0])/(p1[1][0]-p2[1][0])
        if t1>0:
            return t1
        elif t2>0:
            return t2
        else:
            return None
```

3.B Catalogue d'événements

Q.22. On cherche la position où insérer le nouvel événement par un simple parcours de liste (on pourrait envisager une dichotomie mais comme on n'impose pas de complexité, je vais au plus simple). Il reste alors à faire l'insertion.

```
def ajoutEv(catalogue,e):
    i=0
    while i<len(catalogue) and catalogue[i][1]>e[1]:
        i=i+1
    catalogue.insert(i,e)
```

Q.23. Pour chaque particule de numéro $\neq i$, il y a au plus un événement à ajouter (une collision). Il faut de plus ajouter une éventuelle collision avec la paroi pour la particule i.

```
def ajout1p(catalogue,i,R,L,particules):
    for j in range(len(particules)):
        if j!=i:
            t=tc(particules[i],particules[j],R)
            if t!=None:
                ajoutEv(catalogue,[True,t,i,j,None])
        else:
            t=tr(particules[i],R,L)
            if t!=None:
                ajoutEv(catalogue,[True,t[0],i,None,t[1]])
```

Q.24. On part d'un catalogue vide et on le fait grossir en ajoutant les événements à venir liés à chaque particule.

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

Q.25. La liste renvoyé contiendra des doublons puisque la collision de deux particules est obtenue deux fois.

Q.26. Dans l'appel `ajoutEv(catalogue,e)`, le nombre des opérations est au pire de l'ordre de la taille de la liste.
Un appel à `ajout1` avec un catalogue de taille c va effectuer successivement des appels à `ajoutEv` avec un catalogue de taille c puis $\leqslant c+1$ puis $\leqslant c+2$ etc. Les appels à `ajoutEv` ont donc globalement un coût $\sum_{i=0}^{N-1}(c+i) = O(Nc+N^2)$. Les autres opérations ont un coût $O(N)$ qui est négligeable.
Dans l'appel `initCat`, on appelle `ajout1` avec un catalogue de taille 0 puis $\leqslant N$ puis $\leqslant 2N$ etc. Les appels ont donc un coût

$$\sum_{i=1}^{N}(iN^2+N^2) = O(N^4)$$

Q.27. La fonction à optimiser est `ajoutEv`. La recherche de la position d'insertion peut être obtenue en $O(\log(c))$ où c est la taille du catlogue. Cela sera utile si la fonction d'insertion sur les listes est de complexité constante (ou logarithmique) ce qu'il est difficile de savoir.

4 Simulation

Q.28. Si une particule est de vitesse non nulle alors, comme l'énergie cinétique est conservée, il y aura toujours une particule de vitesse non nulle. Il y aura alors toujours au moins un évenement de collision avec une paroi qui sera envisageable.

Q.29. On fait se déplacer toutes les particules jusqu'à l'événement (appels à `vol`).
Il reste à traiter l'événement avec `choc` ou `rebond` selon la nature de l'événement.

```
def etape(particules,e):
    for i in range(len(particules)):
        vol(particules[i],e[1])
    if e[4]!=None:
        rebond(particules[e[2]],e[4])
    else:
        choc(particules[e[2]],particules[e[3]])
```

Q.30. On commence par lister les événements du catalogue : décompte du temps et mise à `False` des événements mettant en jeu les particules de l'événement ayant eu lieu.
Il reste à ajouter les événements impliquant les particules ayant été modifiées (une ou deux selon la nature de l'événement).

```
def majCat(catalogue,particules,e,R,L):
    for i in range(catalogue):
        catalogue[i][1]=catalogue[i][1]-e[1]
        if catalogue[i][2] in [e[i][2],e[i][3]]:
            catalogue[i][0]=False
        if catalogue[i][3]!=None and catalogue[i][3] in [e[i][2],e[i][3]]:
            catalogue[i][0]=False
    ajout1p(catalogue,e[2],R,L,particules)
    if e[3]!=None:
        ajout1p(catalogue,e[3],R,L,particules)
```

Q.31. Après l'initialisation du tableau des particules et de la liste des événements, on gère deux variables : l'une pour la durée écoulée depuis le début de la simulation et l'autre pour le nombre des événements enregistrés.
Tant que la simulation n'est pas terminée, on cherche l'événement valide suivant (suite de `pop` jusqu'à tomber sur `e[0]==True`). On prend en compte l'événement en augmentant le temps de simulation. Si ce nouveau temps est convenable ($< T$), on enregistre l'événement (et on met le compte d'événements à jour).

```
def simulation(bdd,D,N,R,L,T):
    particules=situationInitiale(D,N,R,L)
    catalogue=initCat(particules,R,L)
    temps=0
    nbev=0
    while temps<T:
        e=catalogue.pop()
        if e[0]:
            temps=temps+e[1]
            etape(particules,e)
            majCat(catalogue,particules,e,R,L)
            if temps<T:
                enregistrer(bdd,temps,e,particules)
                nbev=nbev+1
    return nbev
```

Q.32. Quand le premier événement d'un doublon est executé, l'autre est déclaré non valide.

Q.33. Sur une longue simulation, et si on utilise un temps fixe, on va devoir ajouter de très petites durées (temps entre deux événements) à des temps beaucoup. plus grand et un phénomène de cancellation paeut intervenir. Cependant, ces sommes sont quand même faites dans la fonction `simulation` et tout ceci ne me semble pas lumineux. En termes de complexité asymptotiques, cela ne changera pas grand chose.

5 Explotation des résultats

Q.34. On fait un compte du nombre de tuples présents dans `SIMULATION` par dimension d'espace (groupement).

```
SELECT SI_DIM,COUNT(*)
FROM SIMULATION
GROUP BY SI_DIM
```

Q.35. On fait un groupe par simulation. Le nombre de rebonds est le nombre de tuples par groupe et, pour chaque groupe, on peut calculer la moyenne des vitesses.

```
SELECT SI_NUM,COUNT(*),AVG(RE_VIT)
FROM REBOND
GROUP BY S.SI_NUM
```

Q.36. On a ici besoin de la tables des rebonds (qui permettra de sélectionner le numéro de simulation) et de la table des particules (on a besoin des masses). Comme on veut un résultat par paroi, on groupe selon l'attribut `RE_DIR`.

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